SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Formal and Incremental Verification of SysML Specifications for the Design of Component-Based Systems

Oscar Alberto Carrillo Rozo

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Nº | X | X | X

THÈSE présentée par

Oscar Alberto Carrillo Rozo

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Formal and Incremental Verification of SysML Specifications for the Design of Component-Based Systems

Unité de Recherche :
Institut Femto-ST/DISC

Soutenue publiquement le 17 décembre 2015 devant le Jury composé de :

Christian Attiogbé	Rapporteur	Professeur, LINA, Université de Nantes
Frédéric Boniol	Rapporteur	Professeur, ONERA, Université de Toulouse
Jacques Julliand	Examinateur	Professeur, DISC, Université de Franche-Comté
François Vernadat	Examinateur	Professeur, LAAS - CNRS, INSA de Toulouse
Hassan Mountassir	Directeur de thèse	Professeur, DISC, Université de Franche-Comté
Samir Chouali	Co-Directeur de thèse	MDC, DISC, Université de Franche-Comté

Acknowledgements

There are a lot of people who was involved during the development of this thesis, professionally and personally and whom I would like to say THANK YOU.

First I would like to thank the people who helped and encouraged me to continue my post-graduate studies here in France, Michel Riveill who accepted my candidature in the École Polytechnique de Nice - Sophia Antipolis to do my master studies and Carlos Barrios who helped and supported me to organize my post-graduate studies.

I really enjoyed my time in the Femto-ST/DISC laboratory and I'm very grateful with my supervisors Hassan Mountassir et Samir Chouali for have given me the oportunity to work in this thesis with them. Hassan for his help and guidance during this four years, and Samir for been always there to support me, help me and guide my research during all that time, he has greatly enhanced my knowledge and comprehension of computer science.

I would like to thank Professor Christian Attiogbé and Professor Frédéric Boniol for accepting being my referees and giving me valuable remarks to enhance my work. Also, I would like to thank Professor Jacques Julliand and Professor François Vernadat for accepting being my examiners.

Furthermore, I would like to thank the people from the laboratory in Besançon who were there supporting me in one way or another: Aloïs, Fouad, Adrien, Elizabeta, Aydée, Elena, Kalou, Roméo, Hamida, Pierre-Cyrille, Olga, Pierre-Alain, François, Françoise, Jacques, Frédéric, Ivan, Jean-Marie, Dominique...

Additionnally, I would like to thank my colleagues from the Université de Savoie for giving me the opportunity to work with them as a ATER and supporting me during the writing of this thesis: Jean-Charles, Pierre, Xavier, Tom, Jacques-Olivier, Christophe, Rodolphe...

Finally, I would like to give special thanks to my beloved wife Ruby for her support and patience during all this time we had lived apart, I would never have been able to finish my dissertation without her support and confidence.

Contents

List of Figures

List of Tables

List of Definitions

Acronyms

BDD Block Definition Diagram.

CBD Component-Based Development.

CBS Component-Based Systems.

CSP Communicating Sequential Processes.

EMF Eclipse Modeling Framework.

IBD Internal Block Diagram.

IDL Interface Description Language.

INCOSE International Council on Systems Engineering.

OMG Object Management Group.

OOD Object-Oriented Design.

RFP Request for Proposals.

SD Sequence Diagram.

SysML Systems Modeling Language.

UML Unified Modeling Language.

XML Extensible Markup Language.

1

Introduction

1.1/ Context and Challenges

The systems become increasingly complex and their implementation asks for more rigorous conception approaches. To develop reliable systems, some software engineering approaches have been proposed and particularly top-down approach, which allows building a system, step by step from high abstract specifications, like in [Wir71, CL02, AaDB+02, vL03, LNRT10]. This approach has been used to design Component-Based Systems (CBS) [Szy02] constituted by communicating entities. It allows effectively to enhance development process reliability and reduce development costs.

CBS are widely used in the industrial field, and they are built by assembling various reusable components. Its success is due, generally, to the development of complex systems by assembling smaller and simpler components and its reduced development cost. However, this approach of development leads to construct CBS generally even bigger, therefore more complex. Consequently the question of their reliability is not always guaranteed. Hence the need to integrate more formal approaches in the development process of the CBS.

To ease the communication between the various stakeholders in a CBS development project, one of the widely used modeling language is SysML [OMG12], which besides allowing modeling of structure and behavior, it has capabilities to model requirements. It offers a standard for modeling, specifying and documenting systems. Therefore, in this thesis, we propose to exploit it. The improvements, brought by SysML, have allowed increasing its popularity in the industrial and academic environment. A SysML specification of a system is described by structural diagrams and behavioral diagrams. The architecture refinement of a system is an important concept in SysML, and it is based in a development process that can start from an abstract level and evoluates towards more detailed levels which can end in an implementation.

In this context, we have identified two main challenges:

The first one concerns the development by refinement of a CBS, which is modeled only by its SysML interfaces and behavior protocols. In this case, it is a question of replacing an abstract specification by a composition of blocks preserving its structural properties and its behavioral properties. Structural diagrams of SysML describe the system in static mode, and behavioral diagrams describe the dynamic operation of the system. The blocks are modeled by two diagrams, the Block Definition Diagram (BDD), which defines the architecture of the blocks and their performed operations, and the internal block diagram used to define the ports of each block and connectors between them linking their ports.

During the refinement process, these two diagrams can be checked to decide whether the proposed architecture satisfies or it is inappropriate compared to the abstract specification.

The second one concerns the difficulty to decide what to build and how to build it, by considering only system requirements and reusable components. Then, the question that arises is: how to specify a CBS architecture, which satisfies all system requirements? There is a subsequent subject in this issue, and it concerns compatibility between the set of reusable components that will compose the system, which must be guaranteed. Indeed, we, generally, exploit reusable components from a component library to construct CBS, so it is necessary to guarantee component compatibility. Our goal, here, is to guide, by the requirements, the CBS specifier to build a consistent system architecture that fulfills all requirements.

1.2/ CONTRIBUTIONS

In this section we present an abstract of the contributions proposed in this thesis. These contributions are oriented to give an answer to the challenges presented in the above section. We present in Figure 1.1 a diagram representing our contributions (numbered as 1,2, and 3) to the specification of CBS in SysML.

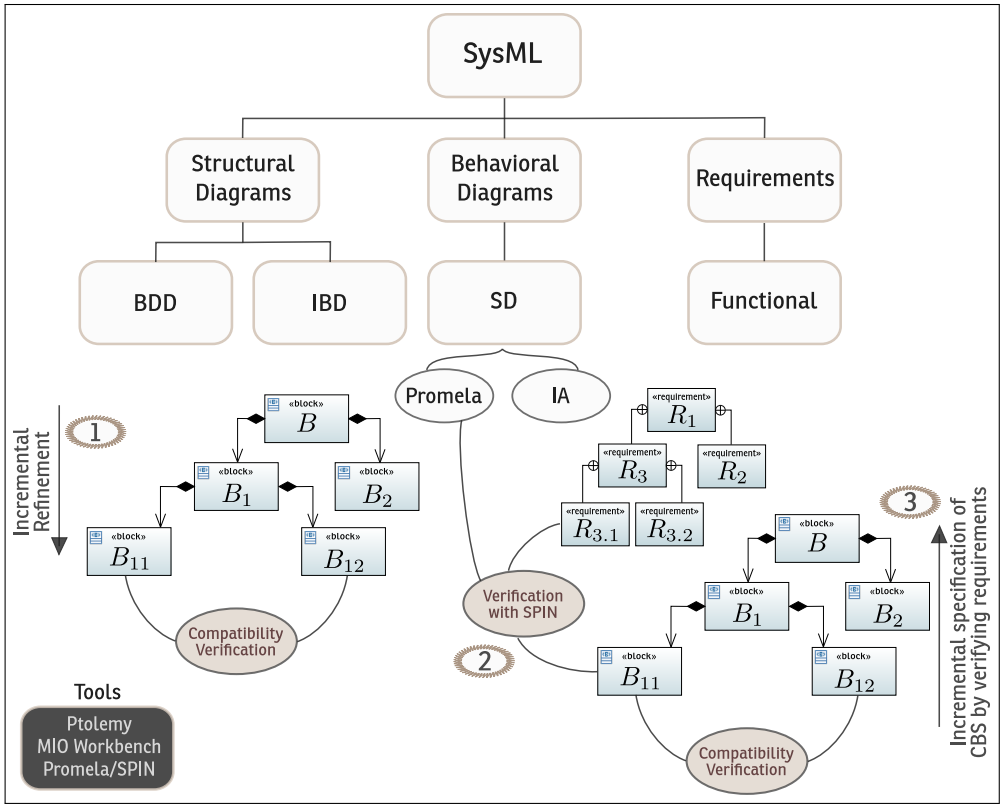

Figure 1.1 – Thesis contributions

Firstly, we focus on the fulfillment of an abstract SysML specification by assembling several concrete blocks according to a refinement process (see Contribution 1 on Figure 1.1). Block behaviors are described by interface automata which can be derived from behavioral

diagrams as proposed in [CH11]. This approach aims to formalize the decomposition process, by defining refinement relations between blocks, and by focusing on the verification of architectural and behavioral aspects of SysML blocks. In this contribution we exploit the tools: Ptolemy II for verifying compatibility between the assembled components, and the MIO Workbench tool to verify refinement.

Secondly, based on SysML requirement diagram and component interfaces, specified with SD, we propose a formal and methodological approach to specify incrementally the system architecture that preserves all the system requirements (see Contribution 3 on Figure 1.1). Therefore, we propose to treat, one by one, atomic requirements, extracted from the requirement diagram (provided by the specifier), to construct a partial architecture, of the system, composed of atomic components and composite components. At each step, we propose to select an atomic requirement from a SysML requirement diagram, and choose a component from a library that should satisfy the selected requirement. Then we verify whether the component satisfies the requirement thanks to the LTL formula which specifies the requirement and the Promela program which specifies the component SD (see Contribution 2 on Figure 1.1). After that, we verify the compatibility between the selected components, and the selected one in the precedent step, and we verify also the preservation of the requirements treated in the precedent steps. This process ends when all atomic requirements are treated, or when we detect incompatibility between components, or the non preservation of the requirements by component composition. When the process ends correctly, we guarantee the architecture consistency of the final CBS which then fulfills all the requirements.

In this context, this thesis presents new contributions which are:

- The exploitation of SysML requirement diagram to specify the requirements of CBS [CCM13, CCM14a].
- The specification of SysML requirements with LTL (Linear Temporal Logic) formulae for their verification on components [CCM14a], thanks to their SD which are translated to Promela by adapting the approach proposed in [LTM+09].
- The verification of components compatibility by exploiting the interface automata formalism [dAH01], obtained from SD of components, thanks to the approach proposed in [CH11]. In this work we adapted the compatibility verification algorithm to handle SysML requirements and to verify also their preservation by the composition [CCM12a, CCM12b, CCM15].
- The verification of components behavior refinement by applying alternating simulation in interface automata [CCM15].
- The proposition of an incremental approach to construct CBS and to verify their requirements in order to avoid the problem of the combinatorial explosion of the number of states of the verified components. Indeed, the requirement verification is performed on elementary (generally small) components, so we avoid the verification on composite components thanks to the requirements preservation by the composition. This contribution allows obtaining the CBS architecture that fulfills all the requirements. Indeed, this architecture is constructed incrementally and also validated incrementally against to SysML requirements at each step [CCM13, CCM14a].

1.3/ Tools

In this section, we describe the tools that we used to model and verify the CBS in this thesis.

Papyrus This plugin made by Lanusse et al. [LTE+09, GDTS10], it is designed to work in the Eclipse Modeling Framework (EMF) [SBMP08] to model any kind of EMF model, one of them is SysML. We used this modeling tool to design the SysML models in this thesis. Besides of being a free and open source software, it has a growing community of users in the industrial and academic field [LST+11].

MIO Workbench This tool is a plugin made by Bauer et al. [BMSH10], it is an Eclipse-based editor and verification tool for modal I/O automata. We used it in this thesis, to verify behavioral refinement of CBS.

Ptolemy II This is a tool for modeling, simulate and design concurrent, real-time and embedded systems [BLL+05]. It integrates a module to work with interface automata [LX04]. It represents the interaction between components as actors and evaluates their compatibility by various means, and one of those is the compatibility verification by interface automata. We used this tool to validate the compatibility between components in a CBS.

SPIN This is a popular open-source model-checker that supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal procedures [Hol97]. SPIN verifies design specification written in the verification language *Promela* (Process Meta Language) [Hol91], and it accepts correctness claims (properties) specified in the syntax of standard Linear Temporal Logic (LTL) [Pnu77]. We used this model-checker to verify if a component satisfies a given requirement.

1.4/ Case Study

In this section, we present a safety vehicle system as the case study, which we will use as the pivot case study during this thesis.

The system presented here is a case study inspired from the study of Barnes et al. [BMFN01] and the safety requirements dictated by the American Department of Transportation to improve protection of vehicle occupants [Adm98]. There, the authors define the conditions to activate automatically the safety devices in a car.

Figure 1.2 shows a safety system that consists of several sensors all around a vehicle (accelerometers, impact sensors, pressure sensors, tachometers, brake pressure sensors, gyroscopes, etc.) that detect whether a collision occurred. When a car collides with a barrier or the breaks are pressed, there will be a speed deceleration. The sensors will detect the acceleration/deceleration values and will send them to a central unit, which is marked with number 1 in the image. This center unit must decide whether to inflate the airbags (front, side, knee, etc.), which are marked with number 2 in the figure, and/or lock the seat-belts

(marked with number 3 in the figure). From the directives in [Adm98], we assume that the maximum deceleration that the chest can accept is 60G, and therefore, the airbag must be deployed. In the same way we assume that every time the vehicle decelerates at more than 3G, the seat-belts must be locked.

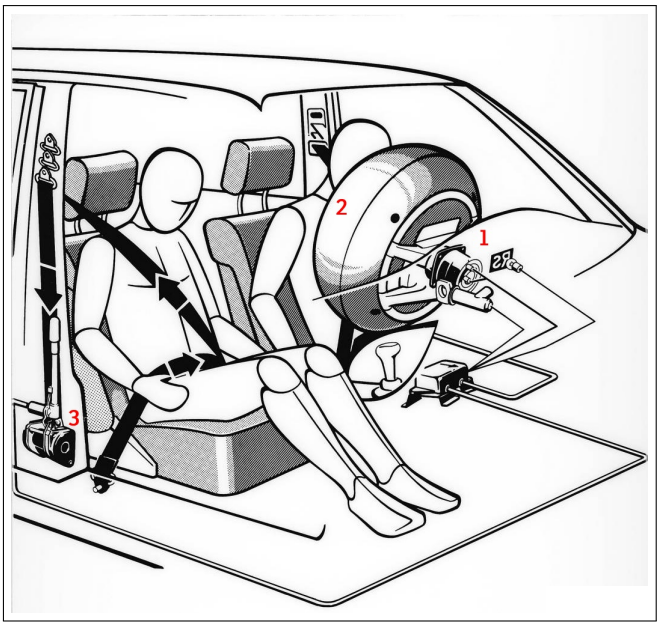

Figure 1.2 – A safety vehicle system[1]

1.5/ PUBLICATIONS

The work presented in this thesis has been already published in national and international conferences, in the following, we list the references for the published articles:

- Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Formalizing and verifying compatibility and consistency of SysML blocks. In *ACM SIGSOFT Software Engineering Notes (UML-FM 2012)*, volume 37, pages 1–8, Paris, France, 2012. ACM
- Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Vérification de la consistence et de la compatibilité entre blocs SysML. In *Conférence en Architectures Logicielles (CAL 2012)*, Montpellier, France, 2012
- Samir Chouali, Oscar Carrillo, and Hassan Mountassir. Specifying System Architecture from SysML Requirements and Component Interfaces. In *Software Architecture (ECSA 2013)*, pages 348–352, Montpellier, France, 2013
- Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Incremental Modeling of System Architecture Satisfying SysML Functional Requirements. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software (FACS 2013)*, Lecture Notes in Computer Science, pages 79–99. Springer International Publishing, Nanchang, China, 2014

[1] Image taken from [Dav10]

- Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Modélisation Incrémentale d'une Architecture de Système Satisfaisant des Exigences Fonctionnelles SysML. In *Conférence en Architectures Logicielles (CAL 2014)*, Paris, France, 2014
- Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Verification of a SysML Block Decomposition in a Refinement Process. In *Under consideration for publication in Software & Systems Modeling (SOSYM)*. Springer Berlin Heidelberg, 2015

1.6/ Document Outline

In this section we give a summary of the content of this thesis, which is structured in three parts and composed of nine chapters as follows:

In Part I, we introduce the scientific context of this work, there, we first give, in Chapter 2, an overview of Component-Based Systems and the key concepts to describe them. Then in Chapter 3, we present the SysML modeling language and the diagrams that we will use to model Component-Based Systems. In Chapter 4, we introduce the concept of Interface Automata, which we will use later in our approach to describe behavior protocols and verify component compatibility. Finally, in Chapter 5, we present some works that are related to ours in one way or another.

In Part II, we present the contributions of this thesis regrouped in three chapters. Chapter 6 handles the first identified issue, which is how to model, by refinement, a CBS that is initially described only by its interfaces, and guarantee that the proposed system refines the initial abstract definition. Chapter 7 explains our proposition to formally verify SysML requirements on system components by translating the initial requirements into LTL properties that we later verify over a Promela model. This Promela model is obtained from the component behavior protocol described in its sequence diagram. Finally, we use the model-checker SPIN to verify the properties. Chapter 8 deals with the second identified issue, which is how to build a CBS from a list of functional requirements, represented by a SysML requirement diagram and a library of reusable components.

In Part III, we conclude our work with Chapter 9, where we present the conclusions and perspectives of this thesis.

I

Scientific Context

2

Component-Based Systems

In this chapter, we define the main concepts of Component-Based Systems. In Section 2.1 we define software components, later in Section 2.2 we describe the difference between objects and components. We continue then describing the properties of abstraction and reusability of components in Section 2.3. Section 2.5 introduces behavior protocols, which are used to describe the order of arrival for events accepted or invoked by a component.

2.1/ Software Component Definition

Over the years, software developing field has evolved through different paradigms. Structured programming changed over time to class paradigm and the revolutionary object oriented programming. Objects of nowadays have grown, and are identified as *software components*. In this section, we define and describe properties of the later to better understand the differences between objects and components that we will expose in Section 2.2.

Several definitions have been proposed to define software components and one of the more complete was proposed in [SP97] during the European Conference of Object-Oriented Computing (ECOOP 1996). This definition is:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

From this definition, a software component is a unit of composition with other peers, a

Contents

component must encapsulate its implementation and interact with its environment on account of only well defined interfaces. Specifically, these interfaces must give information about what the component requires from other components and what it can offer as services. Nevertheless, to use a component correctly, we must fulfill a *contract*. This contract lists a series of constraints about the way of use to make the component execute its functionalities [Szy02]. It is also required to define what the environment of composition and deployment must provide to make the components interact properly. This environment is composed of a component model with composition rules and a framework that defines deployment, installation, and activation rules of components. Therefore, software systems designed to be an assembly of components with a predefined architecture are called Component Bases Systems (CBS).

Components can be refined and improved by subsequent versions. A company that sells third party components can propose different improved versions of the same component. Traditional version management would assume that the version of a component evolves at a single source. Nevertheless, in a free market, version evolution is more complex and version management can become a problem in its own right, mostly because versions can also change at interface level.

2.2/ Components vs. Objects

In this section, we discuss the key points that differentiate a component from an object.

From [Szy02], we can define an object as a symbolic container, which integrates information and mechanisms that represents a physic or moral identity from the real world. The concept of object leads to the concept of instance, encapsulation, and identity.

Objects and components are frequently considered as synonyms, or very similar. A component can be seen as a collection of objects that communicate between them. The borderline between a component and other components or objects is very clear. The interaction with a component and therefore with its objects must pass through its borderline, which for components is at interface level. In this way, granularity in a component remains hidden, and one cannot use its objects directly. A component cannot be used to identify its composing objects. Moreover, objects inside a component have access to their implementations, but accessing an object implementation from outside the component must be avoided [SP97].

A component may contain one or several classes, but one class should not be part of several components. Although, a component may depend on other components by import relations like the inheritance relationships between classes and objects. Therefore, the parent class and its subclass do not have to be on the same component. If the parent class resides in another component, the inheritance relationship between the two classes traverse the component limits and demands an import relationship between them [Szy02]. More over, instead of classes or objects, a component may contain traditional procedures to manage global variables or it can be completely developed in a functional programming language.

The following properties list other differences between objects and components [DW98]:

- Components lead to other means of intercommunication more extensive than objects, i.e. distributed *SmallTalk* components support the interaction of several re-

mote users over several machines [Ben87].

- Granularity level of components is more extensive than the one of objects and their communication means through interfaces is widely more complex.
- Components are deployed in different configurations with no need of reconfiguration for each host infrastructure.

In summary, components are very close to objects and therefore Component-Based Development (CBD) is greatly linked to Object-Oriented Design (OOD). However, many factors like granularity, the concept of composition and deployment, or even the process of development, distinguish components from objects.

2.3/ Abstraction and Reusability

In the following section, we present the concept of abstraction and reuse of components, which represent the two main features in the CBD approaches.

Component abstraction is the level of visibility of its implementation through its interfaces. In an ideal component abstraction, the environment should not know any detail of the interface specification, hence the idea of *Blackboxes*. To understand the working of a component, it is only a matter of interpretation of its outputs as a function of its inputs. Tests on a component are performed following this principle. Generated outputs, after the component execution, are verified to decide whether the component works as expected.

Nevertheless, blackbox components, in which implementation information are strictly hidden, are not the best suitable for a more interactive communication with their software environments. Outputs in a component may not depend only from their inputs, but also from the responses of a set of services provided by their neighbors (see Figure 2.1).

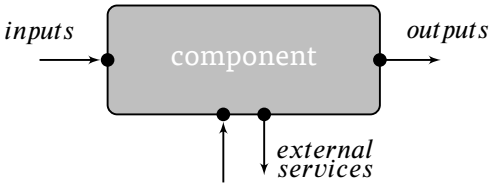

Figure 2.1 – A blackbox component

The external services, activated by the component, must also be specified. Such services often depend on the calling component state. For example, the design pattern *observer* [GHJV94] needs to ask information about the state of the observed object. If the observed object is a bank account and the observer is a balance check component, we need to know if the observer component is called before or after a balance change. Operation specifications, like withdraws or deposits, in a blackbox component do not specify when the observer component is called and the calling component state during the operation. This intermediate state in the blackbox component may be described informally but in more complex cases this approach may often fail.

The question now is how to design more interactive components keeping their implementations hidden? The answer is to use *greybox* components, these are components that reveal some internal operations. The component may give more details if needed, i.e., information about the conditions under which external services are called. For instance, in

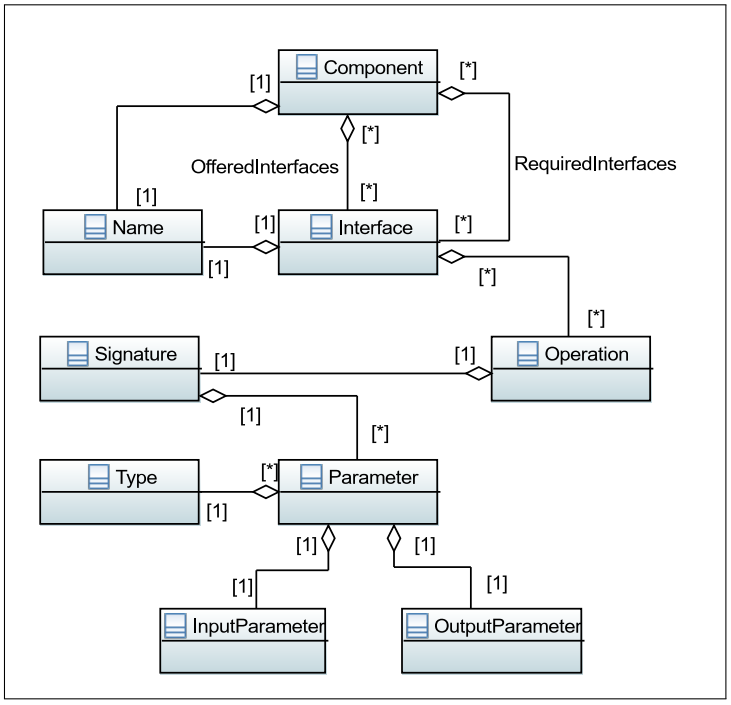

Figure 2.2 – UML metamodel for the specification syntax of a component

our balance check observer component, the greybox specification will clearly specify that the updated balance notification is called after its change [BW97, Szy02].

Hence, the abstraction of a component still remains as the best solution, that allows a user to quickly understand its essential working. One specification explaining that a component is used to sort some data is better understood than a specification giving the inner workings of the algorithm used to sort that data. In this way, authors of the component can later modify the internal code of the component without needing to redeploy its environment. Thus, in a component sorting data, we could change a bubble sorting algorithm for a quick-sort one.

A component is a reusable unit, that can be used in different contexts or be replaced for another one, which refines its original implementation with new functionalities without affecting its clients (other components). A component may also be adapted to new environments, without modifying its source code, even if its interfaces are not intended to communicate with directly.

2.4/ Component Interfaces

Component interfaces are the way by which components communicate. Interface descriptions must be explicitly enough to properly describe a component role. Interfaces and their description must be separate from the component implementation [CL02, Szy02]. This separation allows firstly to replace an implementation without changing its interfaces, secondly to improve system performance without rebuilding it all, and thirdly to add new interfaces without changing the existing ones.

There are two types of interfaces: required and offered interfaces. A component can ex-

port interfaces to the environment and can import interfaces from other components. An offered interface describes one or more services offered by the component to its clients, whereas a required interface describes a required service by the component from its environment.

Specifically, an interface is a set of operations that may be called by clients (mostly other components). An operation may have zero or more input and output parameters. Generally, an operation has at least one output parameter (return value). These concepts are exposed in the UML metamodel shown in Figure 2.2.

A component may provide an interface directly through procedural interfaces from traditional libraries, or indirectly, through objects. Most of component-based approaches use object interfaces, rather that procedural interfaces, being components an evolution of object-based development.

2.5/ Behavior Protocols

In this section, we define the concepts about behavior protocols for components and some languages used to describe these protocols.

2.5.1/ Definition

A component behavior protocol defines the way in which a component must be used, establishing a timing order for calling and/or receiving a response from exposed services. Yellin and Storm described behavior protocols for the first time in [YS97].

A protocol describes the interactions between a component and its environment (clients). An interaction can be an event for answering a call to one of the exposed services or an event for asking an offered service provided by the environment. A beavior protocol aims to establish an order for the use of services of the component and reduce the number of possible combinations while using these services. Another goal of behavior protocols is to give a more detailed description than the services one, and to facilitate the abstraction, reusability and modularity of components.

Behavior protocols allow also to verify composability and compatibility properties in a component assembly. Therefore, it is not enough to call a component service properly, but we must also ensure that the assembly is well composed and that there will not be a blocking state or indesirable situations during the events timeline [AAA07].

For instance, in a printer component, a user should not send a document to print before asking to reserve the resources and validate his identity, otherwise the component will respond unexpectedly. In this same case, if the user does not release the printer, then the component will stay in a blocked state waiting for the user to send an event to unlock it.

2.5.2/ Description Protocol Languages

There are several languages to describe component protocols as exposed in [Mou11]. There is no language that will fit all behavior specifications, as there are languages for different contexts. In the following, we present some of them according to the formalism

used to express the protocol.

- *Formal semantics*: in this category we found protocols that provide an explicit semantic to describe the interactions with a component. Here we meet approaches like [AG97] where the authors propose a formal protocol to describe the connections between components using Communicating Sequential Processes (CSP) [Hoa78]. They propose to formally describe the semantic definitions of connectors independently of component interfaces. The use of CSP allows for consistency and compatibility checking of architectural descriptions.

- *Process algebra*: in this category we found the proposal of Carlos Canal et al. [CFTV00]. They propose to extend the descriptions of CORBA Interface Description Language (IDL) [McH07] to specify component protocols using polyadic Π-calculus, a process algebra specially well suited for the specification of dynamic and evolving systems [Mil99]. This language aims to enrich CORBA IDL with information about the way components expect their services to be called, how they use other component services, and even some semantic aspects of user interest.

- *Statechart*: Some approaches like [MRR03a, MRR03b], propose to model component behavior using statecharts. The use of finite state machines relies on mathematical models and it allows simulation of resulting applications and even generation of code. Another interest of using finite state machines is that we can enhance UML component descriptions by adding statechart diagrams that later we can verify for compatibility and safety properties.

- *Automata*: In this category we group languages that use automata to describe component behavior. We found here approaches like [ZVB+08] that uses component-interaction automata, [dAH01, dAH05] that proposes interface automata, or [LNW07] who proposes an extension of interface automata named modal automata. In these languages, component behaviors are described through labeled transition systems. The labels have semantics of input, output, or internal actions that represent the internal functions, and required and offered services of a component. In this category, we place the description language that we will use in this thesis to define component behavior. A detailed description of interface automata is presented in Chapter 4.

We decided to present some of these languages by their formalism. Nevertheless, we could have also classified them by their content (i.e., services, messages, channels, etc.), the unit hosting the protocol (i.e., the component itself, its interfaces, etc.), or even their formal verification capabilities.

2.6/ Software Architecture

Software architecture can be seen as the art of designing systems. In the same manner that a building architect proposes new designs to his customers by means of a number of different views. A software architect will present several views in which some particular aspect of the system is emphasized to expose his understanding of the system to a client [PW92].

Bass et al. [BCK03] give the following definition for software architecture:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural.

We can define a software architecture as an abstract description of the system, of its refinement in components, the interfaces of its components and their interactions. Components provide a way to isolate specific sets of functionality within units that we can distribute and install separately from other functionality [MHH+09].

2.7/ Architectural Patterns

Software architectures can be categorized according to the pattern used to assemble its components. Patterns like [Wir71] started to propose to gradually refine a system from upper levels to small functionalities following a desired degree of granularity. Years later, approaches like [TB85] wanted to formalise a separate layer for user interfaces and make applications more interactive. Nowadays, most of the systems have an architecture composed of several layers like the one in Figure 2.3 [MHH+09]. Components of the same layer share the same concern and they do not know much about the internal operation of one another. The main layers that we can identify are:

- *Presentation layer*: Components in this layer implement the functionalities that allow users to interact with the system. There are UI components that present the information to users and get input from them, and UI Process components that present the information in a logical representation that is independent of any specific user interface implementation.
- *Service layer*: Besides, user interfaces, the system can interact with its clients or other systems through services. This layer is charged of presenting to clients the interface for getting information from the business layer in a structured manner and wrap information to be exchanged in structured messages.
- *Business layer*: Components in this layer implement the core functionalities of the system, and encapsulate the relevant business logic. In this layer, there will be an application façade that will provide a simplified interface to business components, so the clients will not need to know any details about the internal business components and their relations.
- *Data layer*: Components in this layer provide access to data that is hosted within the boundaries of the system or network resources. They will hide all the logical definitions needed to connect to this data sources and will ease the treatement and maintenance of data sources. Another class of components in this layer are service agents that will provide information obtained from other component services in a format understandable by the system.
- *Crosscutting layer*: In this layer we find components that will perform their functionalities for more that one layer. For instance, security components are required in other layers to perform authentication, authorization, and validation.

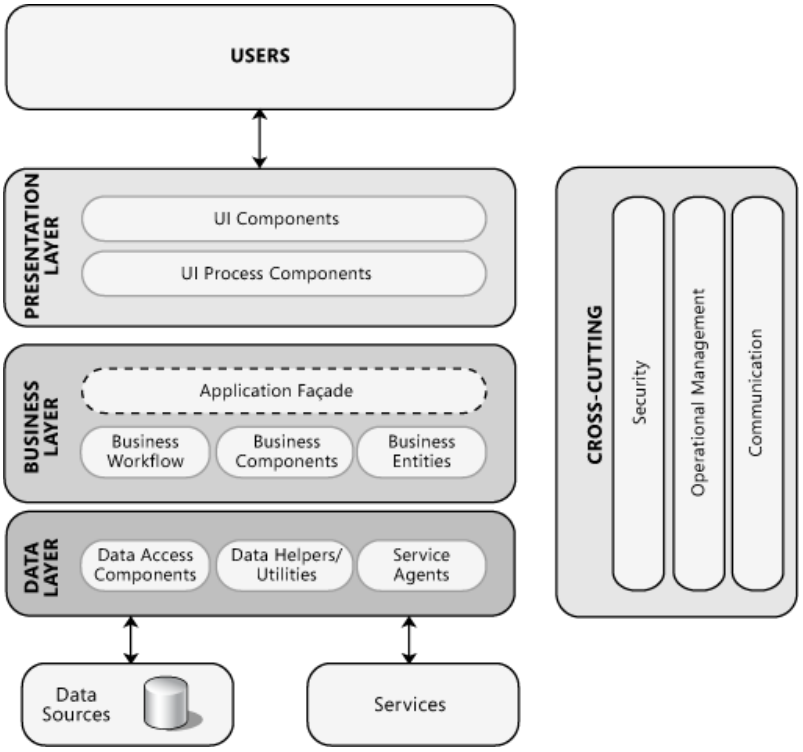

Figure 2.3 – Common layers in a software architecture[1]

From this layered style, we can derive other ones like the message bus, n-tier/3-tier, SOA, client/server, etc, among others. For example in the client/server model, the system is segregated into two applications and the client makes request to the server, most of the times the server is a database with application logic represented by stored procedures.

[1]Image taken from [MHH+09]

3

SysML Language

The Systems Modeling Language (SysML) [OMG12] is a modeling language definition designed in response to the need for unifying the wide range of modeling languages, tools, and techniques used in the systems engineering field. For this goal, the Object Management Group (OMG) [OMG15] and the International Council on Systems Engineering (INCOSE) made in March 2003 a Request for Proposals (RFP) [OMG03] for using the Unified Modeling Language (UML) in Systems Engineering. Then, in May 2006, they proposed a SysML standard and released the version 1.0 in September 2007. Since then, they have been integrating suggestions from the systems engineering community and nowadays we are in version 1.3 presented in April 2012.

The basis of SysML are in UML version 2.0 [OMG05], which allows defining new profiles by modifying, adding and excluding diagrams from its standard set. SysML reuses a subset of its diagrams and adds new features to better fit the needs in the RFP, so that it allows the specification, analysis, design, verification, and validation of a wide range of complex systems. These systems may include software, hardware, data, processes, people, and facilities.

SysML is defined by nine diagrams, classified in three subgroups: *Structural*, *Behavioral*

Contents

and *Cross-Cutting* diagrams. Figure 3.1 shows this categorization and the modification degree of the diagrams with respect to their UML counterparts. Diagrams in orange color are taken from UML without modifications, those in orange/yellow are modified versions, and yellow ones are the new additions. We will describe these diagrams in the rest of this chapter, with an emphasis on those used in our work.

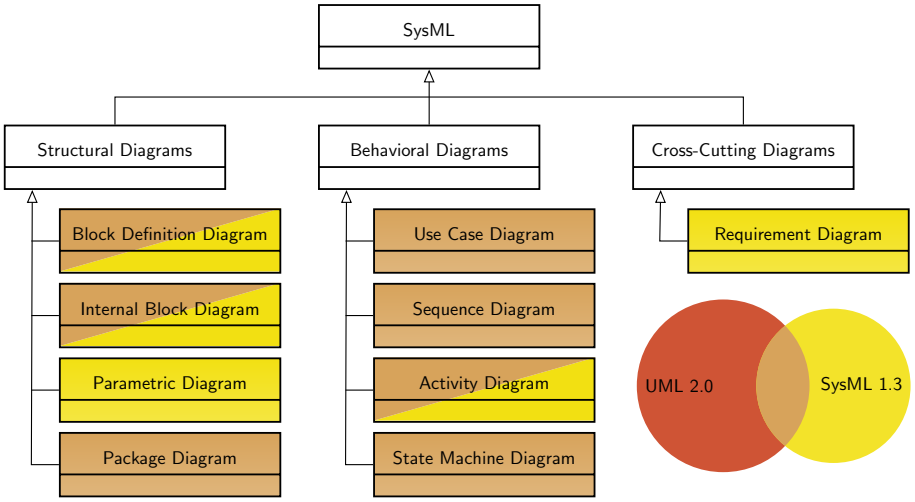

Figure 3.1 – Relation between UML and SysML

This chapter is organized as follows: Sections 3.1, 3.2, and 3.3 will be consecrated to describe each SysML group and their diagrams, Section 3.4 presents some of the tools widely used in the SysML community to edit the diagrams and we end in Section 3.5 with the conclusion.

3.1/ Structural Diagrams

Structural diagrams are intended to describe the system architecture showing its constitutive parts, communication links, internal composition, and initial values. The basic elements in structural diagrams are blocks. A block is a modular unit that represents the structure of a system or one of its elements. It may list structural or behavioral features by properties and actions respectively. Some of the properties may hold other parts of the system (that may be also described by blocks) and be linked through connectors to indicate how they relate to one another.

This set of SysML diagrams are meant to organize blocks and allow us to represent a system hierarchy, in which a system at one level is composed of systems at a more basic level. In the following we present the diagrams grouped in this category:

3.1.1/ The Block Definition Diagram

Block Definition Diagrams (BDDs) are based on the UML class diagram with some capabilities excluded, such as more specialized forms of associations. They describe the architectural structure of the system as components with their properties, operations, and relationships. Every component is represented by a block, no matter whether they are

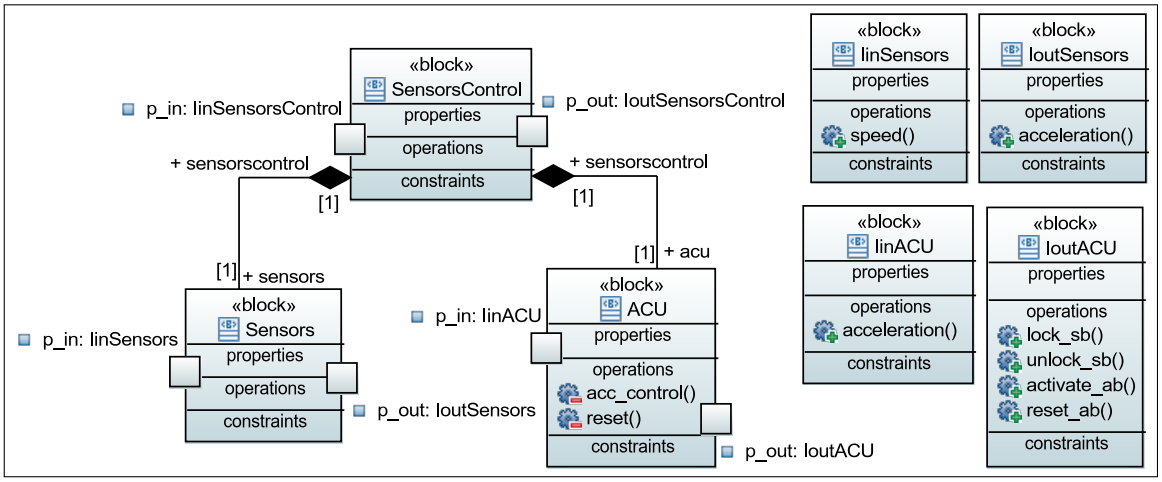

Figure 3.2 – A block definition diagram

logical (software) or physical (hardware) elements, and they are organized in the diagram to show a system hierarchy or a system classification tree describing their associations, generalizations, and dependencies.

Figure 3.2 presents a preliminary BDD diagram for our case study where the main block *SensorsControl* is associated to two composing parts, the block *Sensors* and *ACU*. Since the association link starts with a black diamond symbol then this association is a composing one. Inside each block box we can list its constraints, operations, parts, references, values and properties, and around it we can put the ports allowing it to communicate with other blocks. These ports are typed by interface blocks that can be presented in the same diagram or in another one. They list as operations the actions that we can demand or should provide to the corresponding block through their typed ports.

3.1.2/ The Internal Block Diagram

The Internal Block Definition Diagram (*IBD*) is based on the UML composite structure diagram, with restrictions and extensions as defined by SysML. It captures the internal structure of a block in terms of properties and connectors between properties. Each property is represented as a part and ports are used to specify allowable types of interaction. They are linked by the interactions between them, such as software operations, discrete state transitions, flows of inputs and outputs, or continuous interactions.

In Figure 3.3 we show an IBD describing the internal structure of the block *SensorsControl* from our case study. It presents the block as the main frame and inside we see the two composing blocks, *Sensors* and *ACU*, instantiated as part properties, . They are linked through a connector that goes from the *p_out* port of *Sensors* part to the *p_in* port of *ACU* part.

3.1.3/ The Parametric Diagram

The Parametric Diagram is a new diagram introduced in SysML and it aims to describe the parametric constraints between the structural elements. Here we can define initial, minimum, and maximum values for the variables specified in the blocks. This diagram includes notations that constraint the values of an IBD and define the relations between

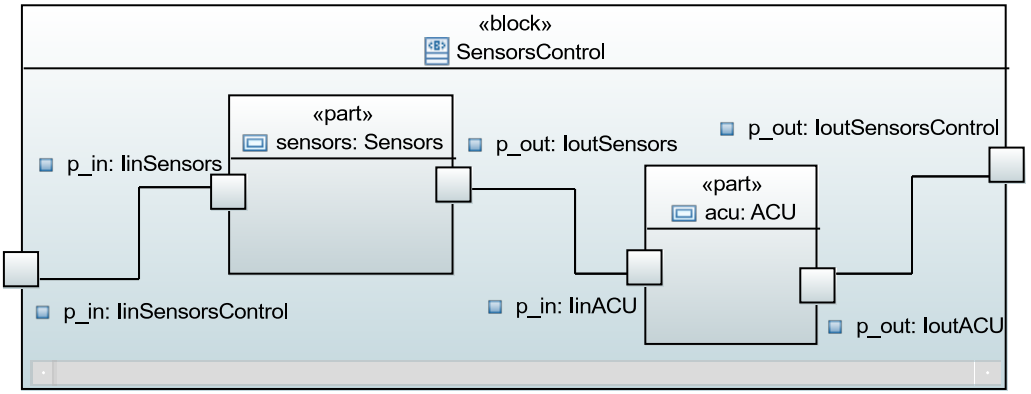

Figure 3.3 – An internal block diagram

block properties, that can be used later to compute energy use or response times.

3.1.4/ Package Diagram

The Package Diagram is used to describe how the system model is organized into packages, views, and perspectives. Organizing the elements by packages helps to establish unique naming of the model elements and avoid overloading a particular model element name. Other diagrams like the BDD, requirement diagram, and behavior diagrams can nest package elements.

3.2/ Behavioral Diagrams

Behavioral diagrams aim to describe all the actions performed by the system, showing the messages exchanged, the actors, the order of actions and the different states in which the system will be, once an action is executed. This set of SysML diagrams are meant to present the dynamic view of the system. In the following we present the diagrams grouped this category:

3.2.1/ The Activity Diagram

The Activity Diagram describes the actions done by the system and their sequence. For this end, it specifies the input and output data, which are used during the flows. This diagram is an extension of the activity model of UML 2.0. It includes additional properties to allow modeling of other type of flows (i.e. continuous or discrete object flows, controlled flows, probabilities of flows, etc).

3.2.2/ The Use Case Diagram

The Use Case Diagram describes the services a user can interact with. The services described are not limited to system functions, they can also be other users or physical objects. Hence, use case diagrams are very simple and their application fields are very large, including systems engineering, they were not modified from the UML 2.0 version.

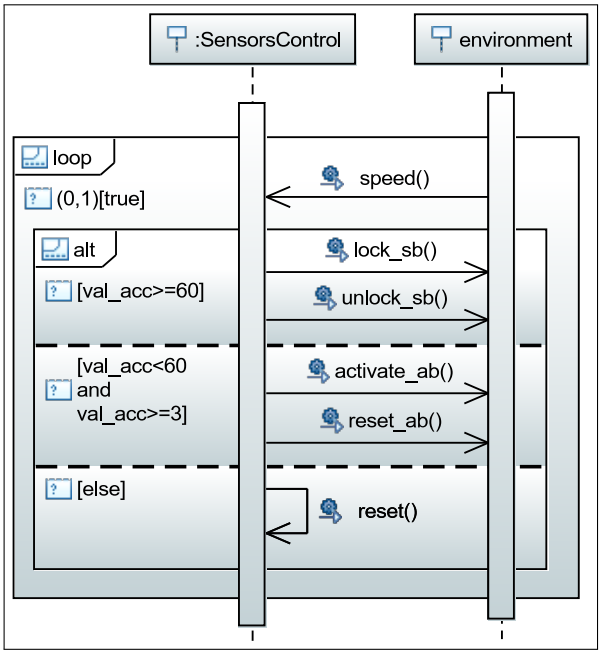

Figure 3.4 – A sequence diagram

3.2.3/ The Sequence Diagram

The Sequence Diagram describes the system behavior as interactions between system components. An interaction presents each part of the system represented as a lifeline that communicates through messages in the form of operation calls or signals. In addition to lifelines and messages, sequence diagrams uses combined fragments that condition the execution of messages, for example the `if` and `while` operands can be represented by the `alt` and `loop` combined fragments respectively.

In Figure 3.4 we present the interaction of the *ACU* block from our case study with its environment. We see that the ACU and its environment have a lifeline for each one and the exchanged messages are represented by arrows. We notice that there are two combined fragments, a `loop` fragment indicating that the message sequence is repeated and a nested `alt` fragment that decides which sequence of messages must be executed, in this case we decide whether to deploy the airbag by analyzing the deceleration value received in the `sensor_values` call from the environment.

3.2.4/ The State Machine Diagram

The State Machine Diagram describes through state machines the system behavior. A state machine presents a set of states and state transitions that represents the response of a system to events. Each state contains a set of values that are modified at each transition. The SysML state machine diagram is used unchanged from its UML 2.0 counterpart.

3.3/ Cross-Cutting Diagrams

In this category comes the diagrams that do not fit in structural neither behavioral diagrams since they describe the system needs in a wide view as for example the functional and non-functional requirements.

3.3.1/ The Requirements Diagram

The requirement diagram is a new diagram introduced by SysML to close the lack of requirement modeling in UML, besides the use case diagram that can only represent functional requirements though. This diagram aims to list the model functional and non-functional requirements, such as response times, size or functions of a system.

In the requirement diagram, each requirement is represented as a stereotype «*requirement*» of the UML element *class*, and the requirements can be associated by different relationships, i.e. the «*deriveReqt*» relationship to indicate that a requirement is derived from another requirement, or the containment relation, represented by the symbol $\oplus$ at the start of the association, to indicate that a requirement is contained in the other requirement.

The requirement diagram that specifies the system needs for our case study is shown on Figure 3.5. In this diagram, the initial requirement R_1 asks for ensuring passengers lives and it is decomposed by a containment relationship into two requirements $R_{1.1}$ and $R_{1.2}$ that ask for two safety devices: an airbag system which must be deployed whenever the car is in a collision, and the seat-belts that must be locked when the sensors detect strong movements. On the left side, requirement $R1.1$ is further decomposed into requirements $R_{1.1.1}$, $R_{1.1.2}$, and $R_{1.1.3}$. Requirement $R_{1.1.1}$ asks for the capture and sending of sensor values to an Airbag Control Unit (ACU). Requirement $R_{1.1.2}$ requests an ACU to decide whether to deploy the airbag and lock the seat-belts as soon as the sensors report new values. Finally, requirement $R_{1.1.3}$ demands to deploy an airbag device, once the signal from the ACU is received.

3.4/ SysML Tools

The OMG is responsible for releasing SysML and each version comes with a documentation describing the specification and a machine readable file in Extensible Markup Language (XML) format with the metadata for building SysML diagrams. From this metadata, developers can create tools to assist the designing of standards-compliant models. So, actually there are several tools available to assist the system designers in their projects, and is up to them to choose the one that fits better their needs. Here we list some of the most known tools classified in commercial and free software:

- Commercial Tools
  - *MagicDraw*: Since its release, this tool, produced by the NoMagic society [1], is intended for designing UML models, nevertheless as SysML is a UML stereotype then one can add a plugin named Paramagic (also a commercial plugin)

[1] http://www.nomagic.com

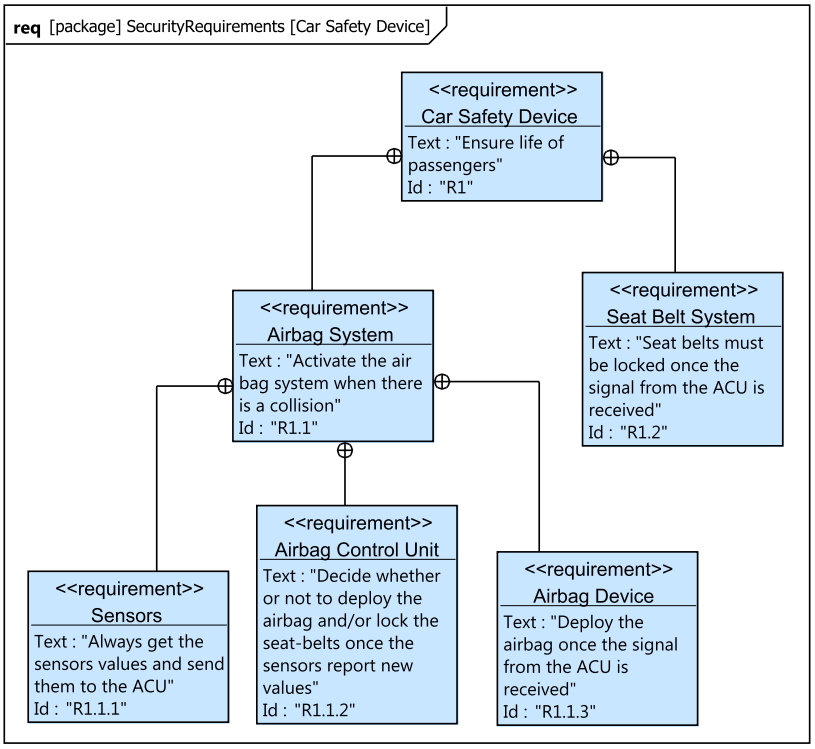

Figure 3.5 – A requirement diagram

that includes the SysML specification. This tool is the most used in the modeling community since it is the one used by the authors of the standards in their books and it integrates the new specification as soon as they are published.

- *Cameo Systems Modeler*: This tool, developed also by NoMagic and based in MagicDraw is exclusively intended for designing systems and we can say that is the most complete SysML tool as it integrates the new updates as soon as they are proposed by the OMG. It includes the SysML 1.4 version, which at this moment has not been released by the OMG though.
- *Artisan Studio*: Developed by the Atego society [2], this tool provides modeling capabilities to design UML and therefore SysML diagrams. One interesting option with this tool is the posibility to use a free version, which is intended for single users.
- *Enterprise Architect*: Developed by the Sparx Systems society [3], this tool can be extended to model SysML diagrams, by adding the MDG Technology plugin.

- Free Tools: Most of the free SysML tools are based on the EMF [4] and we can add them as plugins from their respective update sites.

  - *Papyrus*[5]: This plugin is intended to model any kind of EMF model, especially UML and related modeling languages as SysML.

[2] http://www.atego.com
[3] http://www.sparxsystems.com
[4] http://www.eclipse.org/emf/
[5] http://www.eclipse.org/papyrus/

- *TopCased*[6]: This plugin is intended to model critical embedded systems including hardware and/or software, and its main modeling language is SysML. We have to note that until now they were releasing a dedicated plugin to integrate in eclipse but actually they are migrating their tools to the PolarSys organization [7] which is an open source environment intended for embedded systems.

3.5/ Conclusion

As seen in this section, SysML is a language intended for the systems engineering community based in UML and allows designers to model not only structural and behavioral properties but also to organize the requirements. For the purpose of this dissertation we will particularly focus in four of the nine diagrams: Block Definition, Internal Block, Sequences, and Requirements. Requirement diagrams will be used to present the requirements to verify over the other ones. We also presented a vehicle safety system as a case study to show the use of these diagrams. Finally, we presented some of the main commercial and free tools used to model SysML diagrams.

[6]http://www.topcased.org/
[7]http://www.polarsys.org

4

Interface Automata

One of the several options to describe component behavior is the interface automata approach. In this chapter, we present an overview of the main concepts of this kind of automata proposed by Luca d'Alfaro and Thomas Henzinger en 2001. We will later used them in Part II to verify component compatibility in our proposed approaches. This chapter is organized as follows: in Section 4.1 we present a formal definition of interface automata and its properties, Section 4.2 presents the application of interface automata to verify component compatibility, Section 4.3 presents some tools that we can use to verify if two components are compatible and we conclude this chapter in Section 4.4 with a conclusion.

4.1/ Definition

Luca d'Alfaro and Thomas Henzinger introduced interface automata in [dAH01]. They used this automata to specify component interfaces and also to verify component assembly. For the matter of this dissertation, we exploit interface automata to model interfaces of SysML blocks (components).

The interface automata approach aims to describe the behavior of a component as an interface automaton, this automaton consists of a set of states and actions allowing the change of state. Actions are decomposed into three groups: input actions, output actions and internal actions. Input actions represent the methods than can be requested to the component, in which case they are the offered services. These actions are labeled by the character "?". Output actions model the method calls or messages sent to another component. Therefore, they represent the services required by the component. These actions are labeled by the character "!". Internal actions are operations that can be activated locally and are labeled by the character ";".

Formally we describe an interface automaton as follows:

Contents

Definition 1: Interface Automata

An *interface automaton* A is represented by the tuple $\langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ such that:

- S_A is a finite set of states;
- $I_A \subseteq S_A$ is a subset of initial states;
- Σ_A^I, Σ_A^O, and Σ_A^H, respectively denote the sets of input, output, and internal actions. The set of actions of A is denoted by Σ_A;
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ is the set of transitions between states.

We define by $\Sigma_A^I(s)$, $\Sigma_A^O(s)$, $\Sigma_A^H(s)$, respectively the set of input, output, and internal actions at the state s. $\Sigma_A(s)$ represents the set of actions at the state s.

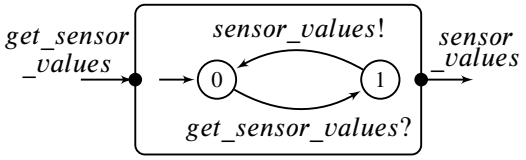

Figure 4.1 – Interface automaton A for the sensor block

As an example, we present in Figure 4.1 an interface automaton representing the behavior of the sensor block presented in the BDD of Figure 3.2. The process to obtain this automaton will be described later on Part II. Besides this graphical representation, we can describe it formally as follows:

- $S_A = \{0, 1\}$;
- $I_A = \{0\}$;
- $\Sigma_A^I = \{get\_sensor\_values\}$;
- $\Sigma_A^O = \{sensor\_values\}$;
- $\Sigma_A^H = \emptyset$;
- $\Sigma_A = \{get\_sensor\_values, sensor\_values\}$;
- $\delta_A = \{(0, get\_sensor\_values, 1), (1, sensor\_values, 0)\}$.

4.2/ Automata Compatibility Verification

The verification of the assembly of two components (blocks) is obtained by verifying the compatibility of their interface automata. Before doing this verification, it is necessary to ensure that the interface automata are *composable*.

Two interface automata A_1 and A_2 are *composable* if
$\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma_{A_2}^H = \emptyset$.
This means that they can not share the same set of input, output or internal actions. In Figure 4.2 we present an automaton representing the behavior of the ACU block in the BDD on Figure 3.2, which we compose with the automaton in Figure 4.1.

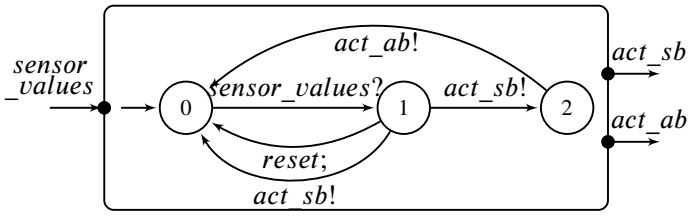

Figure 4.2 – Interface automaton B for the ACU block

We define by $Shared(A_1,A_2) = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^O_{A_1} \cap \Sigma^I_{A_2})$ the set of actions shared between A_1 and A_2. The verification of the compatibility of two interface automata is based on their synchronized product, $A_1 \otimes A_2$, obtained by synchronizing the interface automata on their shared actions (see Definition 3.4 in [dAH01]).

Definition 2: Synchronized Product

Let A_1 and A_2 be two composable interface automata. The *synchronized product* $A_1 \otimes A_2$ of A_1 and A_2 is defined by:

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ and $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ if
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

Two interface automata may be incompatible due to the existence of illegal states in their synchronized product. Illegal states are states from which a shared output action from an automaton can not be synchronized with the same enabled action as input on the other component.

Definition 3: Illegal States

Let A_1 and A_2 be two composable interface automata, the set of *illegal states* $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2).((a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2)) \vee (a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1)))\}$.

The interface automata approach is considered an optimistic approach. In this approach, the reachability of states in $Illegal\ (A_1, A_2)$ does not guarantee the incompatibility of A_1 and A_2. Indeed, in this approach one verifies the existence of an environment that provides appropriate actions to the product $A_1 \otimes A_2$ to avoid illegal states. The states in which the environment can avoid the reachability of illegal states are called compatible states,

and are defined by the set $Comp(A_1, A_2)$. This set is calculated in $A_1 \otimes A_2$ by eliminating illegal states, unreachable states, and states that lead to illegal states through internal actions or output actions, called also incompatible states. These states are eliminated by providing a *legal environment* which steers away from the illegal states by generating appropriate inputs. By eliminating these states in $A_1 \otimes A_2$, we obtain the composition $A_1 \parallel A_2$. As a consequence, the interface automata A_1 and A_2 are compatible *iff* $A_1 \parallel A_2 \neq \varnothing$ [dAH01].

Definition 4: Composition

The *composition* $A_1 \parallel A_2$ of two automata A_1 and A_2 is defined by:

i. $S_{A_1 \parallel A_2} = Comp(A_1, A_2)$,

ii. $I_{A_1 \parallel A_2} = I_{A_1 \otimes A_2} \cap Comp(A_1, A_2)$,

iii. $\Sigma_{A_1 \parallel A_2} = \Sigma_{A_1 \otimes A_2}$,

iv. $\delta_{A_1 \parallel A_2} = \delta_{A_1 \otimes A_2} \cap (Comp(A_1, A_2) \times \Sigma_{A_1 \parallel A_2} \times Comp(A_1, A_2))$

We call the automaton $A = A_1 \parallel A_2$, the composite automaton.

The verification of the compatibility between a component C_1 and a component C_2 is obtained by verifying the compatibility between their interface automata A_1 and A_2. The main steps of the verification algorithm of the compatibility between A_1 and A_2(the complete algorithm in [dAH01]) are listed as follows:

Compatibility verification algorithm:

1. verify that A_1 and A_2 are composable.
2. compute the product $A_1 \otimes A_2$.
3. compute the set of illegal states in $A_1 \otimes A_2$.
4. compute the set of incompatible states in $A_1 \otimes A_2$: the states from which the illegal states are reachable by enabling only internal and output actions (one supposes the existence of a helpful environment).
5. compute the composition $A_1 \parallel A_2$ by eliminating from the automaton $A_1 \otimes A_2$, the illegal states, the incompatible states, and the unreachable states from the initial states.
6. if $A_1 \parallel A_2$ is empty then A_1 and A_2 are not compatible, therefore C_1 and C_2 can not be assembled correctly in any environment. Otherwise, A_1 and A_2 are compatible and their corresponding component can be assembled properly.

The complexity of this approach is in time linear on $|A_1|$ and $|A_2|$ [dAH01].

4.3/ Compatibility Verification Utilities

The verification steps presented above have already been implemented in mainly two tools, The first one is *Ticc* [AdAD+06, dAFL06] which was developed by Henzinger and de

Alfaro, but is no longer available neither supported. The second one is *Ptolemy II* [BLL+05] which is a tool for modeling, simulation and design of concurrent, real-time and embedded systems. Ptolemy II integrates a module to work with interface automata [LX04], it represents the interaction between components that are represented as actors and evaluate their compatibility by various means, and one of those is the compatibility verification by interface automata.

4.4/ Conclusion

In this chapter we presented the interface automata approach introduced by d'Alfaro and Henzinger in which components are represented as automata. In this approach, component services are implemented as actions to change the component state. The goal of using interface automata is to verify the compatibility between components by synchronizing them in their shared actions and verify if there are states that are still accessible after the composition. In order to automatize the process of verification we presented also the tool Ptolemy which allows to verify if two interface automata are compatible.

5

Related Works

This chapter presents some ongoing research subjects in the scientific community related to our matter in this thesis. The chapter is organized as follows: in Section 5.1 we present some relevant approaches about the generation of software architectures for CBS, in Section 5.2 we list some interesting approaches exploring component compatibility verification, in Section 5.3 we list some proposals for describing component behavior. Section 5.4 introduce some approaches that propose how could we list and describe system requirements, particularly in SysML. We end the chapter with a conclusion in Section 5.5.

5.1/ Generation of CBS Architecture

In [LLH01], the authors propose to specify formally system requirements from UML use case diagram and to exploit this formal model to construct the class diagram of the system. The differences with our proposition are the use of SysML requirement diagram to specify requirements, and the situation of our work in the context of CBS, where we deal with the problem of component compatibility with the interface automata approach.

In [PEML10], the authors propose a system modeling approach that combines SysML safety requirements and block diagrams, and the model checking approach to prove that the local behavior of each component contributes to satisfy system requirements. In this work the problem of component compatibility and the preservation of the requirements are not treated.

To construct systems, other approaches take into account all the requirements at once. For example, [vL03], based on KAOS framework [BDD+97], and in [BDL05] the authors propose an incremental approach by adding structural and behavioral properties into a software architecture. Therefore, we were inspired by their vision concerning the require-

Contents

ments where a requirement should be further decomposed to meet the atomic requirements which can be then linked to elementary software components.

In [CLR+09], the authors propose a component-based design method called rCOS, its goal is to guide the process of system development from requirements elicitation through to coding, providing the formal definitions of the models in a UML-based RUP development process. Although, our approach is similar to theirs, in the way we use sequence diagrams to express component interfaces, it lacks the use of a component library to look for third party components that can be reused and verify their compatibility.

The works proposed in [Dro03, Dro07] are based on a behavior tree approach and translate atomic requirements into behavior trees. An interesting approach which inspired our approach in Chapter 8 was proposed in [LNRT10], the authors construct the system starting from raw requirements described in a natural language. Specifications of requirements are derived from intermediate requirement models, and these models approximate the raw requirements. These requirements are then directly mapped into system architecture, where a requirement is represented by an element in the system, thus, maximizing the match between the final system and the raw requirements. This approach is based on a component model that supports incremental composition. However, this model is restrictive and does not consider component protocols and compatibility verification, because their components do not have external dependencies, which ensures that newly added components do not alter the behavior of existing components. Our approach is different and proposes to map requirements, specified and organized with SysML diagrams, directly into system architecture, by exploiting the interface automata formalism and the composition of component interfaces. Using SysML to list the requirements has the main advantage of seeing them graphically modeled and their relationships are explicitly mapped, which allows seeing the decomposition of the system in early stages of the development [SV08b]. The construction of system architecture is guided by the requirements, and the preservation of these requirements in the final system is guaranteed by the compatibility of interface automata and the preservation of the component actions linked to the requirements.

5.2/ Compatibility Verification

In the field of the verification of component compatibility, Bauer et al. [BSBM05] suggested that *"two services are compatible if they can interact properly"*, which is derived into: firstly the components having opposite behaviors, this means that when one component emits a message, the other one must receive that message and vice-versa, secondly there must not be unspecified receptions, which means that all services requested from one component to the other one must be offered, and thirdly the system must be free of deadlocks, in other words, during the interaction between the two services, all reachable states must be deadlock free.

In the field of components, several approaches have been proposed to describe software architectures like those of [Szy02, MT00]. Most models consider the components with their behavior, connectors, and provided/required services. The assembly operation of components may occur at different levels of abstraction, from the design of Dynamic Software Architectures (DSA) to the implementation in platforms such as CORBA [McH07] or .NET [Pla02].

In our case, we are interested in SysML blocks specified by their interfaces and their behaviors modeled using Sequence Diagram (SD). We can cite as examples the model of Allen 1997 [AG97] where the protocols are associated with component connectors. Attie in 2006 [ALPC06] combines protocols to interfaces connecting two components. Others, like Becker in 2004 [BOR04] propose a framework for comparing models with three levels of interoperability using the signatures, the protocols associated to the components and quality of service. The protocols of Magee et al. 1999 [MKG99] are based on works on automata and competition using the formalism of transition systems, including the analysis of reachability. The composition operation is essential to define the assembly and verify the safety and liveliness properties.

The crucial question that arises to the developer is whether the proposed assembly is valid or not.

The approach of Moizan et al. 2003 [MRR03b] aims to provide UML components with the specification of their protocols. The behavior description language is based on hierarchical automata inspired by StateCharts. It can support mechanisms for composition and refinement of behaviors. Properties are specified in temporal logic.

Attiogbé et al. [AAA05, AAA06] define a component model based on services named Kmelia. In this approach they associate components through communication channels following a syntax inspired from Hoare's CSP [Hoa78]. The behaviors are described by automata and associated to services. To verify component compatibility, they encode the Kmelia components into LOTOS processes [Lot88] which are the input of the toolbox for protocol validation and verification CADP [FGK+96].

Teixeira in 2011 [TS11] proposed an approach to evaluate the compatibility of components specified in UML, they use the state machine diagram to describe component behaviors which are then translated to a Petri net to identify compatibility problems

Although a system architecture is composed by several components, and there are approaches like [CK13] which propose to verify compatibility in a multicomponent environment by using team automata [Ell97], we are constrain to verify components by pairs who must not share the same input (resp. output) actions, which allows us to use interface automata obtained from SD.

Other works deal with the inclusion of real-time constraints as in Etienne and Bouzefrane in 2006 [EB06]. It aims to determine the characteristics of components and to define some criteria to verify compatibility of their specifications during the assembly phase using the tool Kronos.

Our approach allows analyzing the consistency, composability and compatibility between blocks. It combines semi-formal models based on SysML and formal models based on interface automata for correct assembly between blocks.

5.3/ Component Behavior Description

In this thesis we explore the substitutability and compatibility of software components. Substitutability is the ability of the component to replace another one without been noticed by the clients. Compatibility is the ability of a component to interact properly with other one when connected.

[VVR06] proposes the use of session types, [HVK98] to describe the dynamic behavior of components additionally to the simple descriptions usually provided by software component interfaces, other like [JMO10] use timed automata.

The ability of SD to express the behavior of a system has been exploited before, like in [HL07, HK07, VT04, RF06, GMP11] where they are translated into state machines, Petri nets or even Java code skeleton, nevertheless they are not formally verified as it is proposed in [LTM+09]. This later proposes to translate SD into a Promela-based model to simulate its execution and therefore verify properties written in Linear Temporal Logic (LTL) [CGP99] that can be checked with the model-checker SPIM [Hol91].

5.4/ Requirements Specification in SysML

Soares [SV08b, SV08a] proposed a methodology for Model-based Requirements Engineering using the SysML Requirements and Use Case diagrams. They chose SysML because it allows to have a structured graphic model of the requirements and also a tabular format which may facilitate the traceability of requirements during the system life cycle. In this approach, first, all the atomic requirements are classified by type:

- Functional: describes what the users expect the system should do to be useful (functionalities), this type of requirements includes information about logical databases like frequency of use, data entities, and integrity constraints.
- Non-functional:are related to emergent system properties such as reliability, safety or response times. They do not depend from a single element in the system, they are evaluated on the emerging system by assembling the system components.
- External: describes all the inputs and outputs of the system, it can include other systems, users, hardware, software, or communication interfaces.

Then, they represent all the requirements in a SysML Requirement diagram which helps to define their relationships, which can be useful for discovering subsystems and limit system architectures. At this stage, requirements are also listed in a tabular form to allow their traceability during the system life cycle, this is important to trace when a requirement has been changed, satisfied or deleted. Finally, SysML Use Case diagrams are used to represent the actors involved in the system and the use cases which helps to delimit the system.

Matoussi et al. [Mat11, BMC+12, LSM+10] propose a methodology that allows generating a refinement architecture in EventB from SysML requirements specifications. They are based in the KAOS approach [vL03] and extend the SysML requirement diagram in the stereotype SysML/KAOS for functional goals. Once the goals diagram is described, they refine the requirements into atomic goals from which they generate Event-B functions that can be later verified.

In the field of requirements validation, Petin et al. [Pet07, PEML10] propose to model requirements through SysML diagrams which they extend in a prototype that allows to link formal properties to the requirements. The requirements can be then linked to the blocks in a BDD to indicate that that block satisfies that requirement, to verify the requirement, the linked property is verified with UPPAL model-checker. This model of linking blocks,

properties and requirements aims to facilitate the traceability of requirements validation. Similarly, Linhares et al. [LdOFV07] propose to formalize SysML requirements with LTL properties that they verify over Petri nets representing system behavior, for this end, they use the TINA toolbox [BRV04].

5.5/ Conclusion

In this chapter, we presented some related works of the proposals in this thesis, we cited notably the works of [CLR+09] who propose the design method rCOS, to develop a CBS architecture following a UML-Based development from requirements, other authors like Matoussi et al. [Mat11] and Petin et al. [Pet07] base their work in SysML requirements like we do in this thesis, the former using Event-B to verify requirements and the latter using UPPAL model-checker.

II

Contributions

6

Incremental Refinement of Abstract Components to Define CBS Architecture

In this chapter, we propose to exploit SysML language and the relation of refinement between components, to define the architecture of CBS from an abstract component. As seen above in Chapter 3, a SysML specification of a system is described by structural diagrams and behavior diagrams. Our approach is based on processing an incremental refinement from an abstract level toward more detailed levels. In our case it is a question of replacing an abstract block in a specification by a composition of blocks preserving its structural properties and its behavioral properties.

Structural diagrams of SysML describe the system in static mode and behavioral diagrams describe the dynamic operation of the system. We note that the term used in SysML for components is blocks, and they are modeled by two diagrams, the BDD, which defines the architecture of the blocks and their performed operations, and the Internal Block Diagram (IBD), which is used to define the ports of each block and transactions exchanged between them through their ports. During the refinement process, these two diagrams can be checked to decide whether the proposed architecture of components satisfies or not the requirements defined by an abstract component.

Contents

We focus on the decomposition of a SysML block into several blocks whose interactions are described by interface automata (see Chapter 4). These interface automata can be derived from SysML behavioral diagrams as proposed in [CH11]. The interface automata formalism [dAH01] allows us to model the temporal order for performing the required (output) and provided (input) services (operations) of a component. One problem that appears, may be the existence of anomalies in the interaction between blocks that can lead to illegal states. These states mean that one of the two blocks is requesting a service not offered from the other one. However, two interface automata A_1 and A_2 associated with two internal blocks B_1 and B_2 are compatible if there is an environment that prevent them from reaching illegal states during their interaction.

This chapter aims to formalize the decomposition process, by defining a refinement relation between blocks, and by focusing on the verification of architectural and behavioral aspects of SysML blocks. In Figure 6.1, we show the position of the contribution presented in this chapter, regarding the contributions of this thesis.

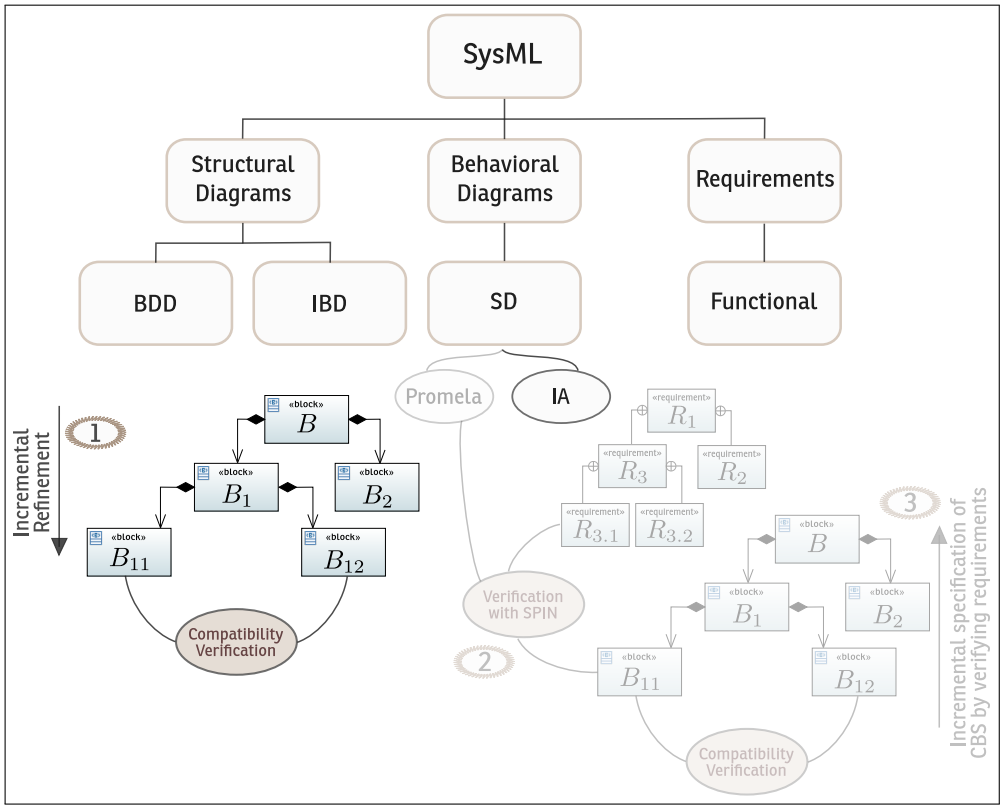

Figure 6.1 – Thesis contribution 1

It is organized as follows. In Section 6.1, we present a general explanation of our proposed refinement process, which allows us to verify refinement of an abstract SysML block into more concrete SysML blocks, then in Sections 6.2 and 6.3 we introduce the principles to model CBS with SysML and its formal representation, later in Sections 6.4 and 6.5 we present our proposal to validate structural and behavioral refinement of SysML blocks, and we end with a conclusion of this chapter in Section 6.6.

6.1/ Approach Overview

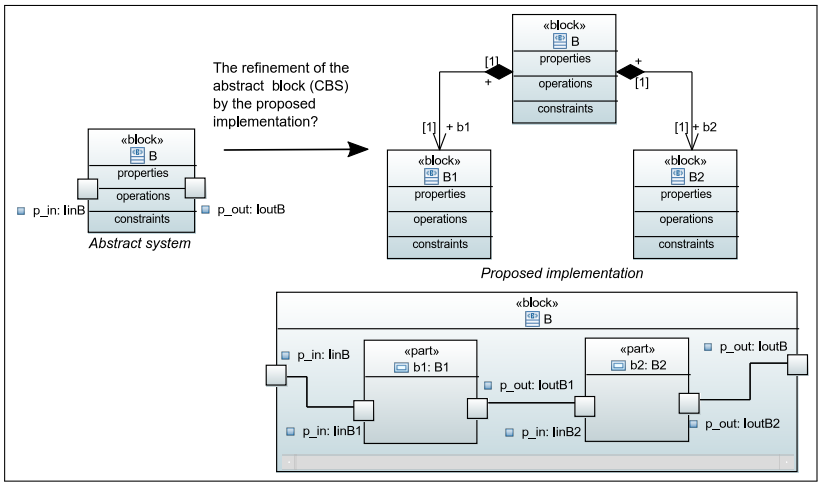

Figure 6.2 – Proposed approach for verifying structure and behavior refinement in SysML

The approach presented in this chapter aims to propose a method to formalize and verify SysML block decomposition in a refinement process. We show the general procedure in Figure 6.2.

Our approach aims to propose a formal method to construct SysML composite block from a set of elementary reusable ones. So, from an abstract composite block, such that its structure is modeled with SysML BDD and IBD diagrams, and its behavior with SD, we propose an approach which decides whether a composition of a set of elementary reusable blocks fulfill the structural and behavioral requirements, related to the composite block. Thus, we propose to verify the correct decomposition of the composite block into a selected set of sub-blocks (elementary). We achieve this decomposition by defining a *refinement by decomposition* relation between the abstract block and its sub-blocks. This relation is verified incrementally, between abstract blocks and their sub-blocks, at different levels, starting from the first abstract block until obtaining the elementary blocks. Our relation of refinement depends on the verification of two relations between the composite block and its sub-blocks:

- structural refinement relation: according to the illustration in the figure 6.2, this relation holds between the composite block B and, the sub-blocks $B1$ and $B2$ iff $B1$ and $B2$ are compatible and the offered and required services of $B1$ and $B2$ are consistent with those of B.

- behavioral refinement relation: this relation holds iff the behavior of the composition of $B1$ and $B2$ is a refinement of the behavior of B.

Definition 5: Refinement by decomposition of SysML blocks

Let B be an abstract block described by a BDD, and an IBD, these both diagrams specify the system architecture. Let $B_1, ..., B_n$ be the set of blocks composing B according to the BDD, so $B_1, ..., B_n$ refine by decomposition B iff:

- $B_1, ..., B_n$ refine structurally B,
- $B_1, ..., B_n$ refine behaviorally B,

Therefore, to verify the refinement between a block and its sub-blocks, we verify the conditions of consistency and compatibility for structural refinement and alternating simulation for behavioral refinement (see Figure 6.3). To achieve this verification, we need to specify by Sequence Diagrams (SD) the behavior description of the abstract block and its composing sub-blocks, and then, by exploiting the approach proposed in [CH11], we associate an interface automaton to each SD. These automata are exploited to verify the compatibility between sub-blocks by means of Ptolemy tool [LX04], which generates the composition automaton from two interface automata as input. Then we verify the behavioral refinement by means of the MIO Workbench [BMSH10], which verifies if a behavioral specification is refined by an implementation using Modal Input/Output (MIO) automata as data input.

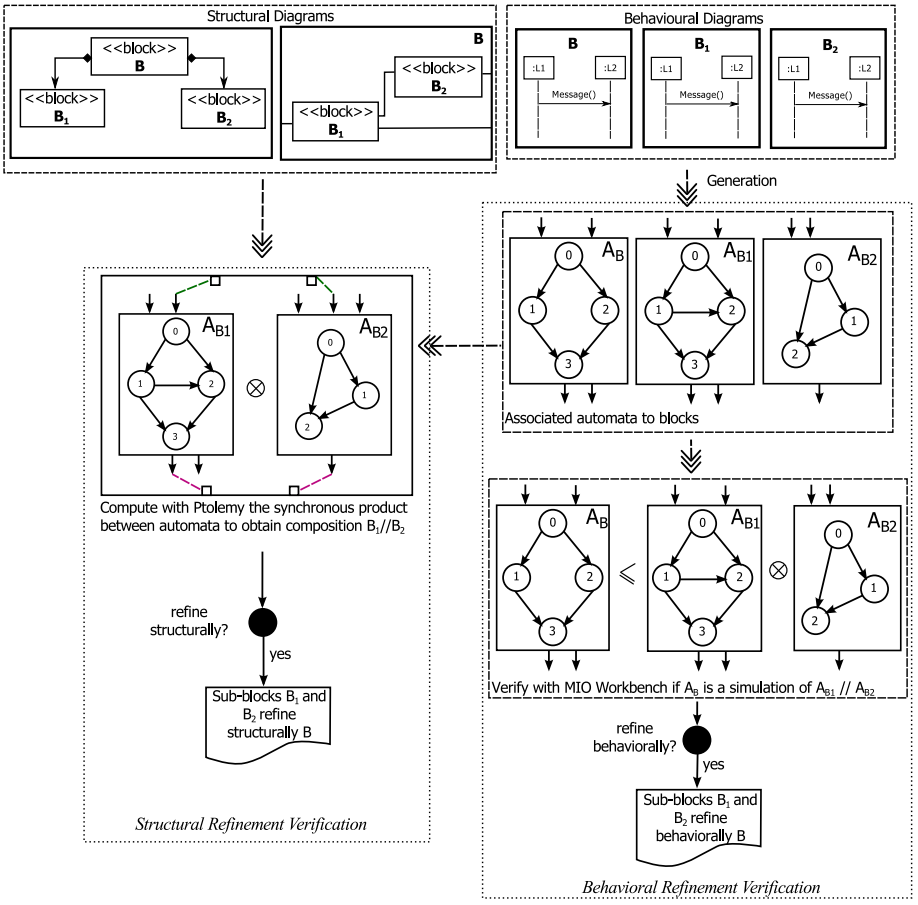

Figure 6.3 – Structural and behavioral refinement verification in a SysML Block Decomposition

Once we have verified the structural and behavioral refinement by two or more blocks that decompose an upper abstract block, we have also verified that this parent block can effectively be replaced by its composing entities. This verification process can be applied incrementally from the top abstract blocks until the lower level blocks to validate the final architecture of the CBS obtained from the initial abstract block.

6.2/ CBS Architecture Specification with SysML

In this section, we present the SysML models that we exploit to specify a CBS architecture. In Figure 6.4, we present the block definition diagram of the whole system. The system is represented by the main block: *ControlSystem*, this block is composed by two other blocks: *SensorsControl* and *DevicesControl*. The first one is responsible for collecting the data from the sensors all around the car to calculate the instant acceleration during movement and if it detects a sudden deceleration, it decides which safety devices must be activated: airbag or seat belts. The second one is charged of receiving calls to activate a safety device, in our case, it will manage the airbag unit and seat belts.

To illustrate our approach we only focus on analyzing a refinement for the block *SensorsControl*. Our goal is to propose a set of blocks, that when composed, can replace the block *SensorsControl*, and then verify if they can refine the structure and behavior of it. The approach can be applied in the same manner to obtain the architecture of the whole system corresponding to the block *SensorsControl*.

Thus, in Figure 6.5, we present the *SensorsControl* block using SysML in version 1.3. In this version of SysML, ports are used to describe the required and/or provided services of a system, these services are listed as operations in blocks outside the relationship. These blocks are used to type the interface of the port. For simplicity, we propose to assemble provided and required services in separate ports, *p_in* and *p_out* respectively, and interface blocks are named as *Iin<block_name>* and *Iout<block_name>* respectively.

In the diagram of Figure 6.5, we ask for a block named *SensorsBlock* that must provide a service named *speed()* as described by the interface block *IinSensorsControl*, and it demands the services *lock_sb()*, *unlock_sb()*, *activate_ab()*, and *reset_ab()*, listed by the interface block *IoutSensorsControl*. It also has an internal operation named *reset()* which is described as a private operation of the block.

In CBS development, systems are built by reusing already built components, rather than developing a new whole system. Thus, to build a concrete specification of the block *SensorsControl*, we propose to use a component library, with own and third party software

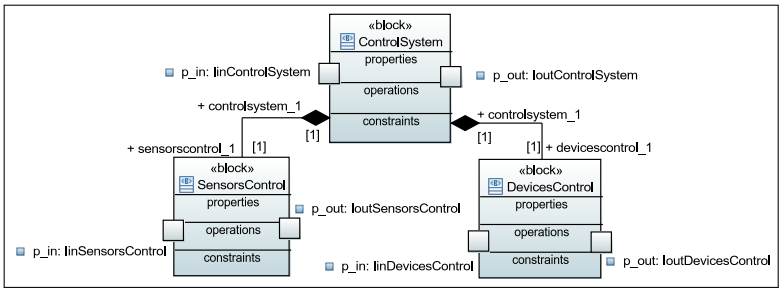

Figure 6.4 – The preliminary BDD of the safety vehicle system

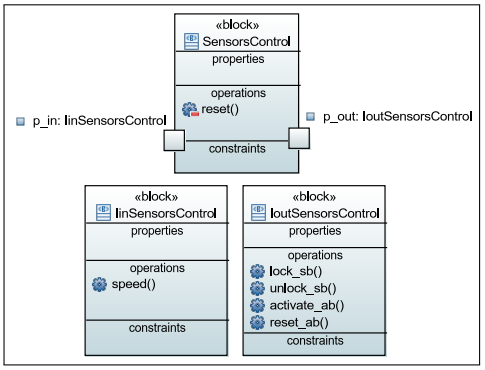

Figure 6.5 – Block definition diagram of *SensorsControl* block

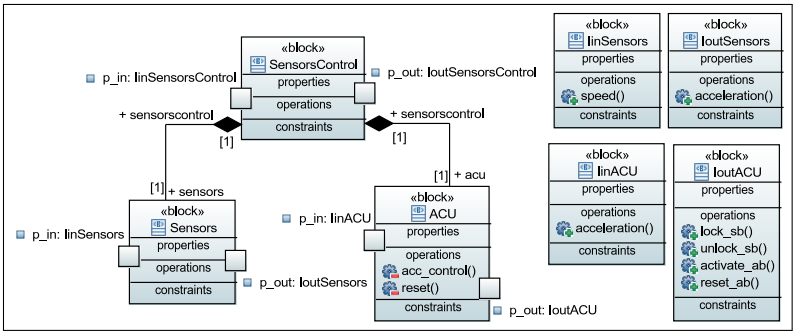

Figure 6.6 – Proposed block definition diagram for *SensorsControl* block

components, and assemble a set of these components, that once composed, will replace structurally and behaviorally the block *SensorsControl*, that we will refer to as an abstract block. Components in that library are described using SysML, so we will refer to them as blocks.

In this way, we propose to decompose our abstract block in two different sub-blocks: a *Sensors* unit and an Acceleration Control Unit (*ACU*). This proposed decomposition is shown in Figure 6.6 using a block definition diagram. On the left side we can see the representation of the two sub-blocks linked to the abstract block by a composition relation and on the right side the interface blocks used to type the input and output ports of sub-blocks. In this assembly, we have the block *Sensors* which receives the speed values through the *speed()* operation on its input port *p_in*, as listed in the interface block *IinSensors*, and it returns a computed acceleration through a call to the *acceleration()* operation described in the interface block *IoutSensors*. Also, there is the *ACU* block, which listens to acceleration values from the *acceleration()* operation in its input port *p_in* as described by the interface block *IinACU*, and decides whether is necessary to call the airbag or the seat belt system by calling the operations *lock_sb()* and *activate_ab()* respectively, these systems can be then unlocked by the operations *unlock_sb()* and *reset_ab()*, these operations are listed in the interface block *IoutACU*.

To complete the system architecture specification, we exploit the SysML internal block diagram to describe the internal composition of the abstract block and how its sub-blocks are connected. So, in Figure 6.7 we specify the internal composition of the *SensorsControl* block (our abstract block) in order to know how its composing blocks, $Sensors$ and ACU, are linked. In this diagram each composing block is instantiated and represented as a part of the system. Also, there are connectors between the ports to show how the parts are

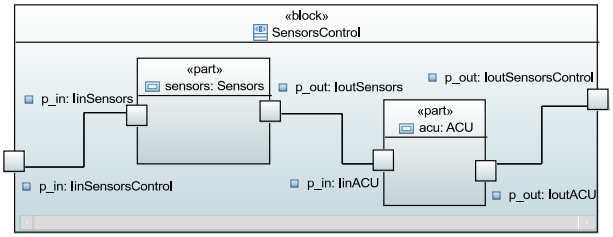

Figure 6.7 – Internal block diagram of the *SensorsControl* block

linked. In our example, the two sub-blocks are linked through a connector between the *p_out* port of block *Sensors* and the *p_in* port of block *ACU*. The ports of the composition that will communicate with the environment are linked with the external input and output ports of the abstract block. In our case study, the provided services of the *Sensors* block are externalized by the input port *p_in* of the *SensorsControl* block and it is represented by the connector between them. In the same way, the required services of the block *ACU* in port *p_out* are externalized by the *p_out* port of the *SensorsControl* block by a connector between them.

For the rest of this chapter, we will consider the interaction between the sub-blocks *Sensors* and *ACU* as a case study, and we will verify the correct assembly between these sub-blocks through our interface automata approach, and therefore if they refine structurally and behaviorally the block *SensorsControl*.

6.3/ Formal Specification of SysML models

To define formally the refinement relation, we have, first, to define formally SysML models exploited in our approach : BDD, Block interfaces, IBD, and the sequence diagram, defined in Chapter 3.

Definition 6: SysML Block

Let SB a set of blocks modeled with a BDD, a SysML block B in SB is a tuple $\langle \Phi_B, P_{in}, P_{out}, TypePort \rangle$, where:

- Φ_B is the set of the private operations in B,
- P_{in} the unique input port of B,
- P_{out} the unique output port of B.
- The function $TypePort : P_{in} \cup P_{out} \rightarrow SB$ determines the interface that types each port.

The input and output ports of blocks are typed by blocks called interfaces (see Definition 6). To determine the interface that types each port, we use the function $TypePort$.

For example, the formal specification of the block $SensorsControl$, described in Figure 6.5, is defined by $B_{sc} = \langle \Phi B, P_{in}, P_{out} \rangle$, where:

- $\Phi_B = \{reset()\}$,

- $P_{in} = p\_in$ where
  $TypePort(p\_in) = IinSensorsControl$, i.e. this port is typed by the block interface $IinSensorsControl$,
- $P_{out} = p\_out$ where
  $TypePort(p\_out) = IoutSensorsControl$, i.e. this port is typed by the block interface *IOutSensorsControl*.

According to the specifications of SysML 1.3 in [OMG12], each port associated to a block is typed by an interface block which exhibits the provided or the required services related to the port. These latter define the set of input and output actions of a block.

Definition 7: Block interfaces

Let P_{out} and P_{in} be respectively the output and input ports of a SysML Block B. The required and provided interfaces of B are defined by the blocks which type respectively P_{out} and P_{in}. The required interface of B, noted $I_{outB} = \langle \Phi_{outB}, P_{in}, P_{out} \rangle$, where Φ_{outB} defines the set of required services of B, i.e. the output actions, and P_{in}, P_{out} are both empty (the interfaces have no ports). And the provided interface of B, noted $I_{inB} = \langle \Phi_{inB}, P_{in}, P_{out} \rangle$ where Φ_{inB} define the set of provided services of B, i.e. the input actions, and P_{in}, P_{out} are both empty.

For example, the required interface of the block *SensorsControl* in Figure 6.5 is defined by *IoutSensorsControl* where $\phi_{IoutSensorsControl} = \{lock\_sb(), unlock\_sb(), activate\_ab(), reset\_ab()\}$.

Definition 8: SysML IBD

A SysML internal block diagram IBD, of a composite block, is a tuple $\langle \Phi Parts, iP_{in}, iP_{out}, eP_{in}, eP_{out}, Connector \rangle$, where:

- $\Phi Parts$ is the set of parts, where each part represents an instance of a block,
- iP_{in} and iP_{out} are respectively the sets of internal input and output ports. These ports are related to the parts in the IBD.
- eP_{in} and eP_{out} are respectively the external input and output ports. These ports are related to the composite block described by the IBD.
- the function $Connector : P_{in} \cup P_{out} \rightarrow P_{in} \cup P_{out}$ associates input and output ports to other input and output ports, it defines the links between blocks. P_{in} and P_{out} are respectively the sets of input and output ports.

For example, the internal block diagram for the block `SensorsControl` shown in Figure 6.7 will be formally specified as:

- $\Phi Parts = \{sensors, acu\}$,
- $iP_{in} = \{sensors\langle p\_in \rangle, acu\langle p\_in \rangle\}$,
  $iP_{out} = \{sensors\langle p\_out \rangle, acu\langle p\_out \rangle\}$.
- $eP_{in} = SensorsControl\langle p\_in \rangle$,
  $eP_{out} = SensorsControl\langle p\_ou \rangle$.

- $Connector(sensors\langle p\_in\rangle) = SensorsControl\langle p\_in\rangle$,
  $Connector(sensors\langle p\_out\rangle) = acu\langle p\_in\rangle$,
  $Connector(acu\langle p\_in\rangle) = sensors\langle p\_out\rangle$,
  $Connector(acu\langle p\_out\rangle) = SensorsControl\langle p\_out\rangle$.

6.4/ Structural Refinement of SysML Blocks

In this section we describe our structural refinement process by defining the refinement relation that we exploit to validate SysML architecture of systems. An initial version of this structural refinement approach was published in [CCM12a].

We define a refinement relation between composite blocks (abstract) and their sub-blocks in order to validate SysML system architecture, and to guarantee the decomposition between blocks and sub-blocks. Thus, we analyze the case where an abstract block described with SysML models (BDD and IBD) is refined by a set of sub-blocks, which are described by the internal block diagram of their abstract block. In this case, the sub-blocks refine structurally the abstract block *iff* it is possible to replace the abstract block by its sub-blocks without affecting the set of offered and required services of the system, and without causing system malfunction. Therefore, to define the refinement relation, we identify two conditions in which the refinement is based:

- Consistency: the set of sub-blocks are consistent with their abstract block if they offer at least the same services as the abstract block, and they require at most the same services. This relation ensures that the sub-blocks do not affect the services of their abstract block.
- Compatibility: the set of sub-blocks are compatible if the interoperability holds between them, which means that they interact correctly without causing deadlocks or system malfunction.

Definition 9: Structural refinement of SysML blocks

Let B be an abstract block described with the BDD, and IBD_B the internal block diagram of B. Let $B_1, ..., B_n$ be the set of blocks composing B according to the BDD, so $B_1, ..., B_n$ refine structurally B iff:

- $B_1, ..., B_n$ are consistent with B,
- the interacting blocks $B_1, ..., B_n$ according to IBD_B are compatible.

In the following section, we define formally the consistency conditions that must be respected between abstract block and its sub-blocks to verify partly the relation of structural refinement.

6.4.1/ Consistency and Composability Verification between Blocks

We propose to verify that the sub-blocks are consistent with their parent block by looking if the provided and required services in the abstract block are in accord to those in the composing sub-blocks. Indeed, the consistency condition proposed here allows us

to determine if the offered services of the abstract block are provided by the sub-blocks that compose it. Similarly, it is verified whether the required services by the composed sub-blocks are required by the abstract block. In addition, composability ensures that the blocks in question do not share the same inputs and/or the same outputs. These conditions are described in the following:

We consider an abstract block B, and B_i and B_j two linked sub-blocks, by a connector, in the set $B_1, ..., B_n$ of composing sub-blocks described in an internal block diagram IBD_B, Φ_{inB}, Φ_{inBi}, Φ_{inBj} are respectively the offered services (input actions) of the blocks B, B_i, B_j, Φ_{outB}, Φ_{outBi}, Φ_{outBj} are respectively the required services (output actions) of the blocks B, B_i, B_j, and Φ_B, Φ_{Bi}, Φ_{Bj} are respectively the internal actions of the blocks B, B_i, B_j. We define the set of shared actions between B_i and B_j by the set $Shared(B_i, B_j) = (\Phi_{inBi} \cap \Phi_{outBj}) \cup (\Phi_{outBi} \cap \Phi_{inBj})$.

The composition of the blocks $B_1, ..., B_n$ is consistent with B iff :

- **Condition 1 (Composability)**:
  For every pair of connected sub-blocks $\{B_i, B_j\}$, it holds that: $\Phi_{inBi} \cap \Phi_{inBj} = \Phi_{outBi} \cap \Phi_{outBj} = \Phi_{Bi} \cap (\Phi_{Bj} \cup \Phi_{inBj} \cup \Phi_{outBj}) = \Phi_{Bj} \cap (\Phi_{Bi} \cup \Phi_{inBi} \cup \Phi_{outBi}) = \varnothing$
  This condition ensures to compose B_i and B_j and to apply later the interface automata theory to verify their compatibility.
- **Condition 2 (At least the same inputs)**:
  For a sub-block B_i connected to the external input port eP_{in} it holds that: $\Phi_{inB} \subseteq \Phi_{inBi}$
  This condition ensures that the sub-block B_i offers at least the same services (inputs) as the abstract block B.
- **Condition 3 (At most the same outputs)**:
  For a sub-block B_i connected to the external port eP_{out} it holds that: $\Phi_{outBi} \subseteq \Phi_{outB}$.
  This condition ensures that the sub-block B_i requires at most the same services (outputs) as the abstract block B.

Remark: Note that according to formal definitions of conditions 1,2, and 3, their verification is possible on to the formal specifications of the used SysML model: blocks, BDD, IBD, and block interfaces (see the Algorithm 1 on page 55).

Case Study Consistency Verification To verify the consistency between the composite block $SensorsControl$ and its composing sub-blocks (see Figures 6.5 and 6.7), we first identify the connected sub-blocks in its BDI. From the BDI in Figure 6.7 we obtain:

The list of parts in the block $SensorsControl$ will be instances of its composing sub-blocks:

$$\Phi Parts_{SensorsControl} = \{sensors, acu\},$$

we then identify the ports of its composing sub-blocks from the list of internal ports in $SensorsControl$ block:

$$iP_{in} = \{sensors\langle p\_in\rangle, acu\langle p\_in\rangle\},$$
$$iP_{out} = \{sensors\langle p\_out\rangle, acu\langle p\_out\rangle\},$$

for each of the identified ports we use the $Connector$ function to determine its associated counterpart:

$$Connector(sensors\langle p\_in\rangle) = SensorsControl\langle p\_in\rangle,$$
$$Connector(sensors\langle p\_out\rangle) = acu\langle p\_in\rangle,$$
$$Connector(acu\langle p\_in\rangle) = sensors\langle p\_out\rangle,$$
$$Connector(acu\langle p\_out\rangle) = SensorsControl\langle p\_out\rangle.$$

We identify the counterparts that are other sub-blocks to determine the connected sub-blocks, that in this case are *sensors* and *acu*.

Then, as exposed in Condition 1 we check the sub-blocks composability by considering the input, output, and internal actions of each sub-block. These actions are described in the interface blocks of the proposed BDD of Figure 6.6, we formally list them as follows:

$$\Phi_{inSensors} = \{speed\},$$
$$\Phi_{outSensors} = \{acceleration\},$$
$$\Phi_{Sensors} = \varnothing$$

$$\Phi_{inACU} = \{acceleration\},$$
$$\Phi_{outACU} = \{lock\_sb, activate\_ab, unlock\_sb, reset\_ab\},$$
$$\Phi_{ACU} = \{acc\_control, reset\}$$

$$Shared(Sensors, ACU) = \{acceleration\}$$

Notice that we do not find inputs or outputs that are present simultaneously in both sub-blocks, ie.

$$\Phi_{inSensors} \cap \Phi_{inACU} = \Phi_{outSensors} \cap \Phi_{outACU} =$$
$$\Phi_{Sensors} \cap (\Phi_{ACU} \cup \Phi_{inACU} \cup \Phi_{outACU}) =$$
$$\Phi_{ACU} \cap (\Phi_{Sensors} \cup \Phi_{inSensors} \cup \Phi_{outSensors}) = \varnothing;$$

Then we check the condition 2, and we find that the input $\Phi_{inSensorsControl} = \{speed\}$ of the abstract block *SensorsControl* is present in the connected sub-block *Sensors*, ie.

$$\Phi_{inSensorsControl} \subseteq \Phi_{inSensors}.$$

Finally for Condition 3, we check that the outputs

$$\Phi_{outSensorsControl} = \{lock\_sb, activate\_ab, unlock\_sb, reset\_ab\}$$

of the abstract block are the same required outputs of the connected sub-block *ACU*, ie.

$$\Phi_{outSensors} \subseteq \Phi_{outSensorsControl}.$$

We can therefore conclude that the blocks $Sensors$ and ACU are consistent with $SensorsContol$.

6.4.2/ Interface Automata Generation

The aim of this section is to show the process of generating the interface automata that describe the behavior of the analyzed sub-blocks. We can generate an interface automaton from a sequence diagram that represents the behavior of each block. In the following we present an overview of the approach described in [CH11], which we exploit to generate interface automata. This approach is based on an algorithm that accepts as entry the formal definition of a sequence diagram (see Definition 11), which is also based on the formal definition of messages (see Definition 10) between blocks and environment, and generates as output the components of the corresponding interface automata.

Definition 10: Message

A message is a tuple:
$\langle B_s, action, B_f, \rangle$ where :

- B_s is the source block of the message,
- B_f is the target block of the message,
- $action$ is the called method

We consider that in a set of messages Mes and an $Environment$ block, the following condition is valid :
$\forall mes_i = \langle B_{s_i}, action_i, B_{f_i}, \rangle \in Mes$ we have $B_{s_i} = Environment$ or $B_{f_i} = Environment$. This condition allows us to translate sequence diagrams into interface automata. In the following definition we consider only the combined fragments $loop$, alt, and seq which are sufficient to the translation to interface automata.

Definition 11: Sequence diagram formal model

A sequence diagram representing the protocol of a block B is defined by :
$SD_B = \langle IM, Mes, Loop, Alt, Seq \rangle$, where

- IM : the initial message,
- Mes: the set of messages,
- $Loop = \langle loop_1, ..., loop_i, ..., loop_n \rangle$, is the list of the loop combined fragments,
  $loop_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$, obj_i is a message or a fragment, and $card(loop_i) >= 1$
- $Alt = \langle alt_1, ..., alt_i, ..., alt_n \rangle$, is the list of alternative combined fragments,
  $alt_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$, obj_i is a message or a fragment, and $card(alt_i) >= 2$
- $Seq = \langle seq_1, ..., seq_i, ..., seq_n \rangle$, is the list of sequence combined fragments,
  $seq_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$, obj_i is a message or a fragment, and $card(seq_i) >= 2$

The formal model corresponding to the sequence diagram specifying the protocol of the block sensors (see Figure 6.8) is $SD = \langle IM, Mes, Loop, Alt, Seq \rangle$, where:

- IM : the message designed by the action
  $\langle Environment, speed(), Sensors \rangle$;
- Mes: $\{\langle Environment, speed(), Sensors \rangle$,
  $\langle Sensors, acceleration(), Environment \rangle\}$;
- $Loop = \langle loop \rangle$,

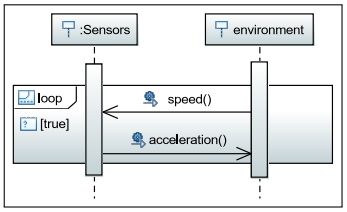

Figure 6.8 – Sensors sequence diagram

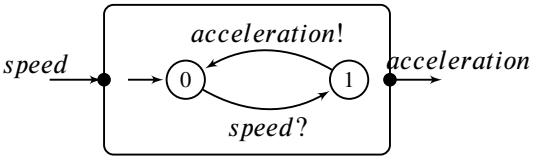

Figure 6.9 – Interface automaton A_1 associated to the *Sensors* block

$loop = \{\langle Environment, speed(), Sensors\rangle,$
$\langle Sensors, acceleration(), Environment\rangle\};$

- $Alt = \emptyset$;
- $Seq = \emptyset$

The derivation algorithm requires as main input a sequence diagram SD_B, and also a list l of objects, which are messages and fragments composing SD_B. The algorithm provides as output an interface automaton A.

For our case study, we applied the algorithm to generate the automata that describe the behavior of the $Sensors$ and ACU blocks. Their corresponding sequence diagrams shown in Figures 6.8 and 6.10 detail the actions done by these blocks.

The generated interface automata A_1 and A_2 are shown in Figures 6.9 and 6.11.

6.4.3/ Compatibility Verification

The compatibility verification between two blocks B_1 and B_2 is obtained by verifying the compatibility between their interface automata A_1 and A_2. To verify the compatibility between two blocks B_1 and B_2, this approach verifies if there is an environment where it is possible to correctly assemble B_1 and B_2. Thus, we assume the existence of an environment that accepts all output actions of the synchronized product automaton of A_1 and A_2, and does not trigger any input action of $A_1 \otimes A_2$.

Condition 4 (Compatibility): Two interface automata A_1 and A_2 are compatible *iff* their composition $A_1 \parallel A_2$ has at least one reachable state.

The composition $A_1 \parallel A_2$ is calculated by computing the product $A_1 \otimes A_2$ in which we eliminate the states that are illegal and those that lead to illegal ones through internal or output actions (see Definition 4). The algorithm to calculate the composition is described as a part of Algorithm 1. We can also obtain the composition by means of the tool *Ptolemy* [LX04].

Case Study Compatibility Verification To verify the compatibility of the interface automata associated to the two sub-blocks, $Sensors$ and ACU, we compute the synchronous product $A_1 \otimes A_2$ and we eliminate unreachable states to obtain the composition $A_1 \parallel A_2$ as indicated by condition 4 of our approach, this synchronous product is given in Figure 6.12. The calculated automaton contains the illegal states $11'$, $12'$, $13'$, and $14'$, which we must eliminate to obtain the composition $A_1 \parallel A_2$ shown in Figure 6.13. This composition has

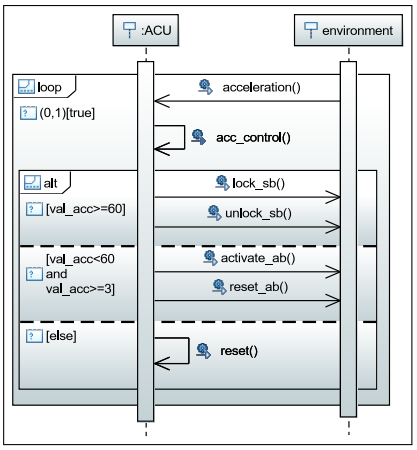

Figure 6.10 – ACU sequence diagram

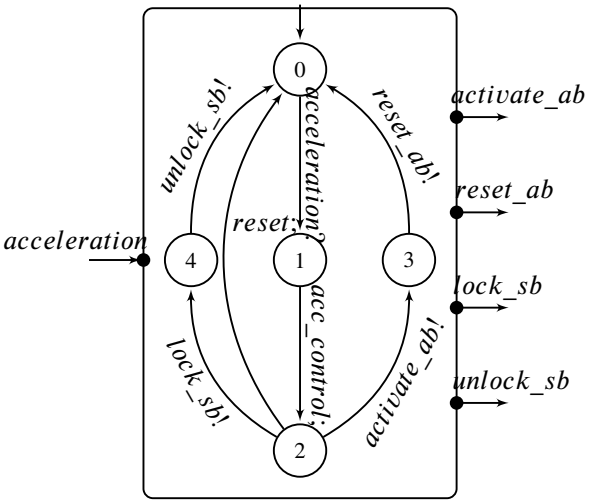

Figure 6.11 – Interface automaton A_2 associated to the *ACU*

no illegal states and more than one reachable state, and therefore we conclude that they are compatible. We deduce that the two blocks *Sensors* and *ACU* are consistent and compatible, and that the block *SensorsControl* can be refined (structurally) into the sub-blocks *Sensors* and *ACU*.

In the following section, we present the algorithm to verify the refinement between an abstract block and its sub-blocks at the structural level.

6.4.4/ Verification Algorithm of the Structural Refinement Relation between an Abstract Block and its Sub-blocks

The Algorithm 1 shows the pseudo-code to implement our approach to verify the structural refinement relation. For every pair of connected sub-blocks in an internal block diagram, we verify, first, the consistency between the abstract block and the sub-blocks, and if they are consistent, we continue to verify the compatibility between the sub-blocks.

The complexity of this algorithm is in time linear on the size of the interface automata and is given by $O(|A_1 \times A_2|)$. In fact, the complexity of the compatibility verification is $O(|A_1 \times A_2|)$ [dAH01], and we can easily verify that the complexity of the verification of conditions 1,2, and 3 do not increase this complexity.

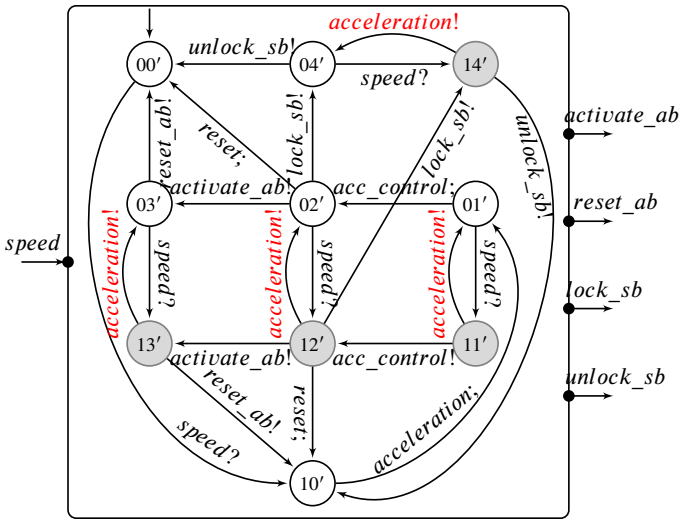

Figure 6.12 – Product $A_1 \otimes A_2$ between the automata *Sensors* and *ACU*

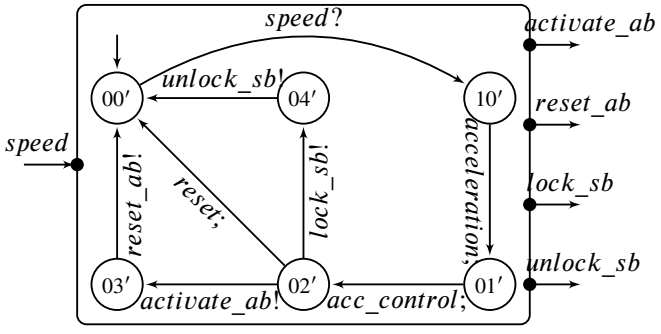

Figure 6.13 – Composition $A_1 \parallel A_2$ between the automata *Sensors* and *ACU*

Algorithme 1 : Verification of the consistency and the compatibility between blocks

Input : An abstract block B described with a $BDD = \langle \Phi_B, P_{in}, P_{out}, TypePort \rangle$ and a $IBD = \langle \Phi Parts, iP_{in}, iP_{out}, eP_{in}, eP_{out}, Connector \rangle$, and a sequence diagram $SD_{i=1..n} = \langle IM, Mes, Loop, Alt, Seq \rangle$ for each sub-block $B_{i=1..n}$ described in the IBD.

Output : The results on the verification of the structural refinement between the abstract block and its sub-blocks

Consistency verification:

1. Analyze the BDD by exploring the set Φ_B and the block interfaces from the sets (P_{in}, P_{out}) (obtained using the $TypePort$ function) to identify the offered, required and internal operations of $B, B_{i=1..n}$.
2. Analyze the IBD by exploring the set $\Phi Parts$ (which are instances of blocks in ΦB) and identify the connected sub-blocks using the $Connector$ function on each of the internal ports in iP_{in} and iP_{out}.
3. For each pair of connected sub-blocks B_i and B_j verify the condition 1 (composability), by considering their sets of private operations (Φ_{Bi}, Φ_{Bj}) and the sets of offered and required services (Φ_{inBi}, Φ_{inBj}, Φ_{outBi}, and Φ_{outBj}).
4. Analyze the IBD and identify which internal input port in iP_{in} is connected to the external input port eP_in and select the sub-block B_i that owns it.
5. Verify the condition 2 (at least the same inputs)
6. Analyze the IBD and identify which internal output port in iP_{out} is connected to the external input port eP_{out} and select the sub-block B_i that owns it.
7. Verify the condition 3 (at most the same outputs)

if *one of the conditions is not verified* **then**
  $B_1,...,B_n$ are inconsistent with B;
else
  Compatibility verification (Condition 4):

  1. For each pair of connected sub-blocks $\{B_i, B_j\}$, obtain the interface automata A_i and A_j from the sequence diagrams SD_i and SD_j
  2. Compute the product $A_i \otimes A_j$
  3. Compute the composition $A_i \parallel A_j = A_i \otimes A_j - Illegal(A_i, A_j)$ and remove the unreachable states
  4. Verify the condition 4

  if *condition 4 is verified for every pair* $\{B_i, B_j\}$ **then**
    the composition of $B_1,...,B_n$ refine structurally B
  else
    the structural refinement does not hold between the composition of $B_1,...,B_n$ and B
  end
end

6.5/ Behavioral Refinement Verification of SysML Blocks

In this section, we describe our behavioral refinement process by defining a relation between SysML Sequence Diagrams (SD). In our approach, behaviors are described with SysML SD and when an abstract block is replaced by two or more concrete blocks, we refine its behavior with the actions of the composing blocks. To verify if the abstract block behavior is well refined by the actions of the composing blocks, we verify if the alternating simulation [AHKV98] holds between the IA of the composition and the IA obtained from the SD of the abstract block.

6.5.1/ Alternating Simulation

In order to verify if the composition of a set of blocks refines the behavior of an abstract block, we use the concept of *Alternating Simulation* for interface automata [AHKV98].

To define alternating simulation formally, we use the notation $s \xrightarrow{\tau}{}^{*} s'$ for interface automata to mean that there exists a sequence of internal transitions leading from s to s'. Then, we define *alternating simulation* for interface automata as commonly used in software specification [dAH05].

Definition 12: Alternating Simulation

For a pair of interface automata
$P = \langle S_P, I_P, \Sigma_P^I, \Sigma_P^O, \Sigma_P^H, \delta_P \rangle$ and $Q = \langle S_Q, I_Q, \Sigma_Q^I, \Sigma_Q^O, \Sigma_Q^H, \delta_Q \rangle$
with the same signature, a binary relation $\leq_a \subseteq S_P \times S_Q$ is an alternating simulation if whenever $p \leq_a q$ and $a \in \Sigma_P$ it holds that:

if $q \xrightarrow{a?} q'$ and $a \in \Sigma_Q^I$ then $\exists p'.p \xrightarrow{a?} p'$ and $(p', q') \in \leq_a$
if $p \xrightarrow{a!} p'$ and $a \in \Sigma_P^O$ then $\exists q'.q \xrightarrow{\tau}{}^{*} q'.\exists q''.q' \xrightarrow{a!}{}^{*} q''$ and $(p', q'') \in \leq_a$
if $p \xrightarrow{a;} p'$ and $a \in \Sigma_P^H$ then $\exists q'.q \xrightarrow{\tau}{}^{*} q'$ and $(p', q') \in \leq_a$

Definition 13: Interface Automata Refinement

An interface automaton P refines an interface automaton Q, writen $P \leq_a Q$, if

1. $\Sigma_Q^I \subseteq \Sigma_P^I$ and $\Sigma_Q^O \supseteq \Sigma_P^O$
2. there is an alternating simulation $\leq_a$ by Q of P such that $I_P \leq_a I_Q$

Actually, we can not verify behavioral refinement through Ptolemy tool but we can do that thanks to the MIO Workbench [BMSH10], an Eclipse-based editor and verification tool for modal I/O automata.

6.5.2/ Modal I/O Automata

Modal automata are an extension of interface automata with modality and control information proposed by Larsen et al. [LNW07].

Definition 14: Modal automaton

A modal automaton S is a six tuple: $S = (S_S, I_S, \Sigma_S^{ext}, \Sigma_S^H, \longrightarrow_{\Diamond}^S, \longrightarrow_{\Box}^S)$ where

S_S: is a finite set of states,

$I_S \in S_S$: is the initial state,

Σ_S^{ext} and Σ_S^H: are disjoint sets of external and internal actions,

$\Sigma_S = \Sigma_S^{ext} \cup \Sigma_S^H$,

$\longrightarrow_{\Diamond}^S \subseteq S_S \times \Sigma_S \times S_S$: is the may transition relation describing allowed behavior,

$\longrightarrow_{\Box}^S \subseteq S_S \times \Sigma_S \times S_S$: is the must transition relation describing required behavior.

Usually interface automata refinement is verified by alternating simulation and instead of building a new tool we propose to use the MIO Workbench module to verify modal refinement. Larsen et al. proposed Theorem 1 (a proof can be found in [LNW07]) to show that we can use observational modal refinement as it coincides with alternating simulation.

Theorem 1: Alternating simulation and observational modal refinement

Alternating simulation and observational modal refinement coincide for interface automata in the following sense:

for any two interface automata P, Q: $P \leq_a Q$ iff $\mathcal{T}(P) \leq_m^* \mathcal{T}(Q)$

Indeed, interface automata can be translated into modal automata to use observational modal refinement. This can be achieved by applying the $\mathcal{T}$ translation function.

Let s_{mayall} be a fresh state that allows all behavior but does not require any behavior. If U denotes the universe of all inputs, such that for all interface automata P , $\Sigma_P^I \in U$, then we define the translation function $\mathcal{T}$ as follows:

Definition 15: Translation function $\mathcal{T}$

$\mathcal{T}(S_P, I_P, \Sigma_P^I, \Sigma_P^O, \Sigma_P^H, \delta_P) = (S_S, I_S, \Sigma_S^{ext}, \Sigma_S^H, \longrightarrow_{\Diamond}, \longrightarrow_{\Box})$ where

$S_S = S_P \cup \{s_{mayall}\}$,
$I_S = I_P$,
$\Sigma_S^{ext} = U \cup \Sigma_P^O$,
$\Sigma_S^H = \Sigma_P^H$

and $s_1 \xrightarrow{a}_{\Diamond}^S s_2$ if $(s_1, a, s_2) \in \delta_P$ and $a \in \Sigma_P^O \cup \Sigma_P^H$

and $s_3 \xrightarrow{a}_{\Box}^S s_4$ and $s_3 \xrightarrow{a}_{\Diamond}^S s_4$ if $(s_3, a, s_4) \in \delta_P$ and $a \in \Sigma_P^I$

and $s_3 \xrightarrow{a}_{\Diamond}^S s_{mayall}$ if $\forall s' \in S_P.(s_3, a, s') \notin \delta_P$ and $a \in U$

and s_{mayall} is a fresh state such that $\forall a \in \Sigma_S.s_{mayall} \xrightarrow{a}_{\Diamond}^S s_{mayall}$.

As an example, we apply the function $\mathcal{T}$ over the interface automaton A_2 associated to the

block *ACU* (see Figure 6.11).

Let P be the automaton representing the behavior of the block *ACU*, such that:

- $S_P = \{0, 1, 2, 3, 4\}$;
- $I_P = \{0\}$;
- $\Sigma_P^I = \{acceleration\}$;
- $\Sigma_P^O = \{activate\_ab, reset\_ab, lock\_sb, unlock\_sb\}$;
- $\Sigma_P^H = \{acc\_control, reset\}$;
- $\delta_P = \{(0, acceleration, 1),\ (1, acc\_control, 2),\ (2, activate\_ab, 3),\ (3, reset\_ab, 0),\ (2, reset, 0), (2, lock\_sb, 4), (4, unlock\_sb, 0)\}$.

The result automaton S from the translation $\mathcal{T}(P)$ is a modal automaton such that:

- $S_S = \{0, 1, 2, 3, 4, s_{mayall}\}$;
- $I_S = \{0\}$;
- $\Sigma_S^{ext} = \{activate\_ab, reset\_ab, lock\_sb, unlock\_sb, U\}$;
- $\Sigma_S^H = \{acc\_control, reset\}$;
- $\longrightarrow_{\Diamond}^{S} = \{(0, acceleration, 1),\ (1, acc\_control, 2),\ (2, activate\_ab, 3),\ (3, reset\_ab, 0),\ (2, reset, 0),\ (2, lock\_sb, 4),\ (4, unlock\_sb, 0),\ (0, a, s_{mayall}),\ (1, b, s_{mayall}),\ (2, b, s_{mayall}), (3, b, s_{mayall}), (4, b, s_{mayall})\}.\ a, b \in U \wedge a \neq acceleration$;
- $\longrightarrow_{\Box}^{S} = \{(0, acceleration, 1)\}$.

Definition 16: Observational Modal Refinement

For a pair of modal automata P and Q with the same signature, a binary relation $R \subseteq S_P \times S_Q$ is an observational modal refinement if whenever pRq and $a \in \Sigma_P$ it holds that:

if $q \xrightarrow{a}_{\Box} q'$ and $a \in \Sigma_Q^O$ then $\exists p'.p \xrightarrow{a}_{\Box} p' \wedge (p', q') \in R$

if $p \xrightarrow{a} p'$ and $a \in \Sigma_P^O$ then $\exists q'.q \xrightarrow{\tau}_{\Diamond}^{*} q'.\exists q''.q' \xrightarrow{a}_{\Diamond} q'' \wedge (p', q'') \in R$

if $p \xrightarrow{a} p'$ and $a \in \Sigma_P^H$ then $\exists q'.q \xrightarrow{\tau}_{\Diamond}^{*} q' \wedge (p', q') \in R$

We say that a modal automaton P observationally refines a modal automaton Q, writen $P \leq_m^* Q$, if there exists an observational modal refinement containing (I_P, I_Q).

6.5.3/ Case Study Application

To verify the behavior refinement for the *SensorsControl* block we need the sequence diagram shown in Figure 6.14, it shows the messages and replies that must be implemented

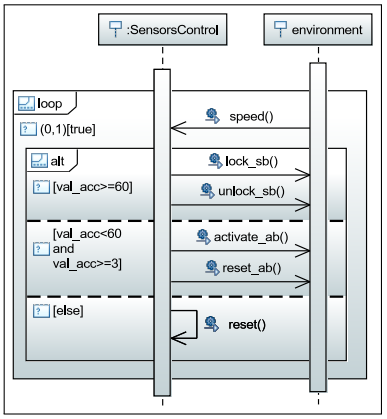

Figure 6.14 – SD for *SensorsControl* abstract block

by the composition of sub-blocks. In order to apply our approach we generate the interface automaton shown in Figure 6.15, this automaton is then verified against the composition automaton of the *sensors* and *ACU* blocks (see Figure 6.13). The two automata are loaded into the MIO Workbench and the refinement is verified with the module *Strict-Observational Modal Refinement*.

In Figure 6.16, we show the verification of the two automata in MIO Workbench. The composition automaton $(sensors \parallel acu)$ is presented at the left and the $SensorsControl$ automaton at the right. To perform the verification we select in the refinement verification menu the option *Strict-Observational Modal Refinement*, being the other refinement verification options (*Strong Modal Refinement, May-Weak Modal Refinement,* and *Weak Modal Refinement*) intended for modal automata. Once the verification is executed, MIO Workbench will determine the binary relations between the states of the two automata (shown by the gray arrows from one automaton to the other one) and will verify if the conditions for an observational modal refinement hold. For our example, we can see the tool bar at the top, this bar turns green to indicate that the verification was successful and that the automaton $(sensors \parallel acu)$ is a refinement of the automaton $SensorsControl$.

In this way we continue also treating the block *DevicesControl*, and we obtain the final system whose BDD is shown in Figure 6.17. In this BDD, we note the decomposition of the abstract block $ControlSystem$ into the elementary blocks $sensors$, ACU, $airbag$, and $seatbelt$.

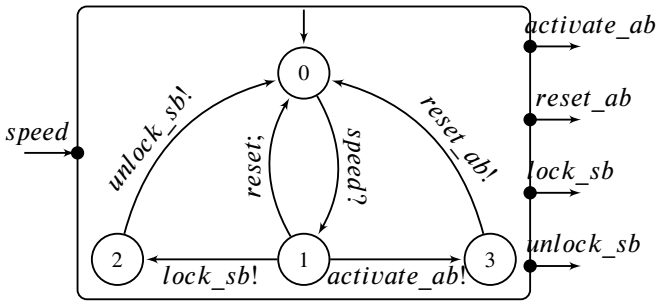

Figure 6.15 – IA associated to the *SensorsControl* abstract block

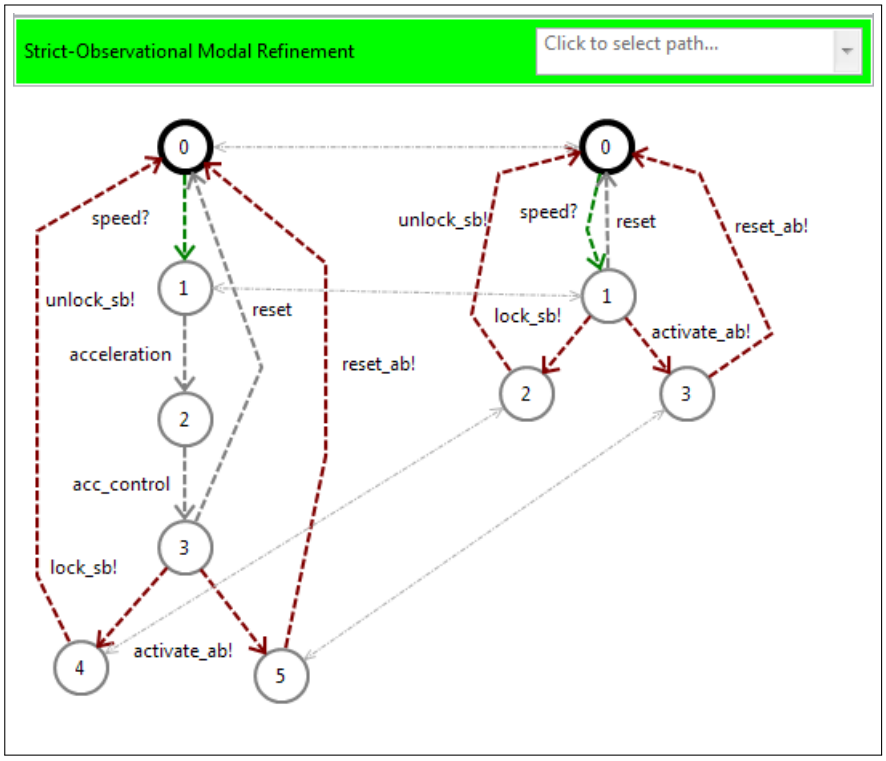

Figure 6.16 – Refinement in MIO Workbench

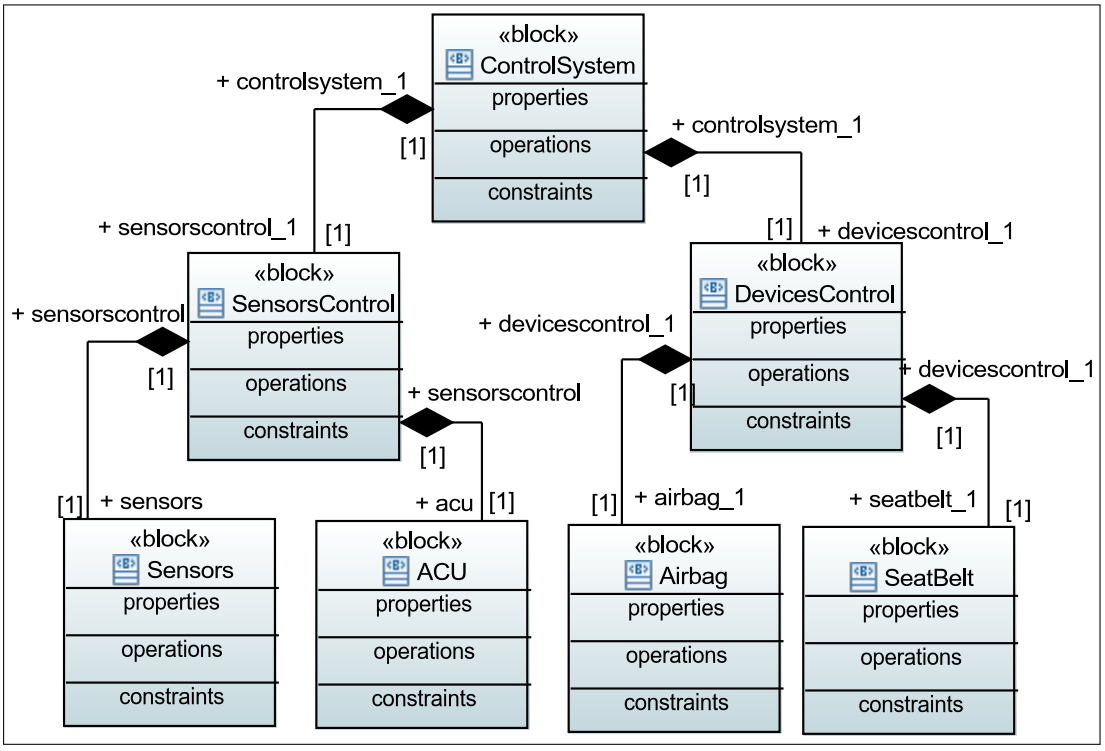

Figure 6.17 – The final BDD of the safety vehicle system

6.6/ Conclusion

We have shown in this chapter, how to specify formally systems described by SysML models with BDD, IBD and SD diagrams. Then, we defined a refinement relation between SysML system blocks, described by structural and behavioral diagrams. The refinement in SysML is an essential concept and it is based on the development of a process from an abstract level towards more detailed levels, which can end in its implementation. Our refinement ensures an incremental substitutability of an abstract block in a specification by a composition of blocks preserving its structural and behavioral properties.

To verify structural refinement specified in BDD and IBD diagrams, we verified first if the sub-blocks were consistent with the abstract block specification and then we verified if they were compatible by applying the model of interface automata. Interface automata were obtained from the SysML SD of each composing sub-block.

To verify behavioral refinement, we applied the approach of alternating simulation for interface automata to verify if the composition of the set of sub-blocks simulated the expected behavior in the abstract block. To ease the task of verifying alternating simulation we proposed to use the module *Observational Modal Refinement* of the tool MIO Workbench. To use this tool, we translated interface automata into modal automata.

7

Formal Verification of SysML Requirements

This chapter presents a formal verification technique based on the approach proposed by V. Lima *et al.* in [LTM+09]. This technique proposes to create a Promela-based model from UML interactions expressed in Sequence Diagrams (SD), and uses SPIN model checker [Hol91] to simulate the execution and to verify properties written in Linear Temporal Logic (LTL) [CGP99]. In Figure 7.1, we show the position of the contribution presented in this chapter, regarding the contributions of this thesis.

7.1/ Approach Overview

To verify if a component satisfies a given requirement, we propose to use the tuple Promela/SPIN. We choose them because it provides important concepts for implementing SD: sending and receiving primitives, parallel and asynchronous composition of concurrent processes, and communication channels. Our adaptation of the approach proposed by V. Lima *et al.* concerns a particular type of sequence diagrams that we exploit to specify the block behaviors.

We propose to use a particular type of SD with only two lifelines, one for the block and one for the environment. Thus, SD can be further translated into interface automata as exposed in [CH11]. In this diagram the exchanged messages will be the offered services as calls from the environment and the required services as calls to the environment. The main advantage of using SD for verification is that we can verify temporal properties over

Contents

it. Messages follow a sequence order that we can trace to detect deadlocks or execution of paths. Figures 7.2 and 7.4 show the interfaces, by SD, for two blocks: sensors and ACU. These are blocks from our block library. In these diagrams we notice that there are only two lifelines and messages are sent to/received from the environment.

Before presenting the verification approach, we show in the following section a short presentation of LTL.

7.2/ Linear Temporal Logic (LTL)

Linear temporal logic is a modal temporal logic with operators referring to time and used for reasoning about infinite behaviors of reactive systems. LTL is mostly used as the logic to specify the properties to verify in model checking environments, such as SPIN.

7.2.1/ Syntax

The set of well-formed LTL formulas Φ is constructed from a set of atomic propositions $\mathscr{P} = \{p_0, p_1, ...\}$, the standard boolean operators: $\neg$ (not),$\vee$ (or), $\wedge$ (and), and the temporal operators: $\bigcirc$, to be read as "next", and $\cup$, to be real as until. The formula s $\bigcirc p$, means that p is true at the next step. And the formula $p \cup q$ means that q is true at some point, and p is true until that time. The until operator allows deriving the temporal operators

- Eventually $\diamond p := true \cup p$, means p will become true at some point in the future.
- Always $\Box p := \neg \diamond \neg p$, means p is always true.

The set of LTL formulas over $\mathscr{P}$ is inductively defined as follows:

- every atomic proposition in $\mathscr{P}$ belong to Φ.
- if p and q are formulas in Φ, then $\neg p$, $p \vee q$, $\bigcirc p$, and $p \cup q$ are formulas in Φ.

7.2.2/ Semantics

Let φ be an LTL formula, φ can be satisfied by an infinite word w that can be viewed as ω-word on an execution path of a Kripke structure. Let $w = a_0, a_1, a_2, ...$ over $2^{\mathscr{P}}$ (the set of propositions), be such an ω-word, where at some time point $i \in \mathbb{N}$, a proposition p is true iff $p \in a_i$. We note by w_i the suffix of w starting at i. The satisfaction relation $\vDash$ between a word and an LTL formula is defined as follows:

- $w \vDash \varphi$ iff $\varphi \in w_0$.
- $w \vDash \neg\varphi$ iff not $w \vDash \varphi$.
- $w \vDash \varphi \vee \psi$ iff $w \vDash \varphi$ or $w \vDash \psi$.
- $w \vDash \varphi \cup \psi$ iff $\exists i \in \mathbb{N}$ such that $w_i \vDash \psi \wedge \forall j \in \mathbb{N}, 0 \leq j < i, w_j \vDash \varphi$.

Table 7.1 – Mapping of basic concepts from sequence diagrams to Promela

SD element	Promela Element	Promela Statement
Lifeline	Process	`proctype{...}`
Message	Message	`mtype{m1,...,mn}`
Connector	Communication channel for each message arrow	`chan chanName = [1] of {mtype}`
Send and receive events	Send and receive operations	Send ⇒ `ab!m`, Receive ⇒ `ab?m`

Alt combined fragment	if condition	`if`
		`::(guard)->ab_p?p;`
		`:: else -> ab_q?q;`
		`fi;`

Loop combined fragment	do operator	`do`
		`::ab_p?p;`
		`od`

In this work we exploit LTL to express properties, specifying SysML requirements, to be verified on components by means of the model-checker SPIN. These properties are expressed as LTL formulas, and SPIN requires, as input, their corresponding negative formulas, which are converted into Büchi automata to be exploited in the model-checking algorithm.

7.3/ Verification with SPIN Model Checker

As exposed above in the overview, we exploit and adapt the approach proposed in [LTM+09] to translate SD to Promela-based models to verify properties with the model-checker SPIN. Table 7.1 shows the Promela representation of the main elements in SD. Alternative and loop combined fragments are represented as `if` condition and do operator in Promela respectively, guard condition is declared globally and the non-deterministic behavior is implemented at `init` time by assigning different values to the guards.

Figures 7.3 and 7.5 show partially the Promela representation for the sensors SD and ACU SD respectively (the complete code is presented at the end of this chapter in Listings 7.1 and 7.2). In both diagrams, we notice that their two lifelines are translated as processes in the Promela code, one process for the block and one other for the environment. Both processes are started at the same time thanks to an atomic call at the main process `init`. We also notice that *loop* combined fragments are translated as do statements. The alternative combined fragment, *alt*, in ACU SD is translated as `if` statement. There, the three possible range values for deceleration are assigned at `init` time by using an `if` clause, this way, SPIN will choose non-deterministically, which of the three values will be used to simulate the system.

Once the sequence diagram is translated, the component can be simulated as a SPIN system. However, in order to verify whether the component satisfies an LTL property, the authors propose to use a series of flags to keep track of *who is sending/receiving what message to/from whom* at any time of the execution. In our approach we verify properties over independent components with only two lifelines in their SD, one line for the selected component and the other for the environment. So, we do not use a flag related to *to/from whom* is sent a message as it will always be *the other* lifeline. These flags are updated together at each *send/receive* event using a `d_step` statement. The flags for our example in Figure 7.2 will be `send` and `receive` to indicate the performed action, `msg_get_sensor_values` and

`msg_sensor_values` to indicate the message exchanged, and `sensors` and `environment` to indicate who performed the action.

7.4/ Requirement specification with LTL

After defining the flags to track the execution state of the system, LTL properties can be written as boolean expressions over the flags. In our approach, we propose to translate SysML requirements to LTL properties by respecting this formalism with flags. Hence, for example requirement $R1.1.1$ in Figure 8.3 can be expressed as: *always after receiving a call to get_sensor_values, the sensor block will send a message with the sensor_values*. The boolean expression, using the flags described before, will be:

```
□((sensors && receive && msg_get_sensor_values) → ◊ (sensors && send &&
                          msg_sensor_values))
```

Similarly, requirement $R1.1.2$ can be expressed as: *always after receiving a message with the sensor_values, the ACU will send a message deciding to lock the seat-belt, activate the airbag or wait for another call*, and the boolean expression with flags will be:

```
□((acu && receive && msg_sensor_values) → ◊ (acu && send && (msg_reset
                    || msg_act_sb || msg_act_ab)))
```

These properties are further verified over their corresponding Promela model by using SPIN model-checker, which indicates if blocks satisfy the properties. Once a corresponding block is found for a requirement, we continue with another requirement to start building the system architecture.

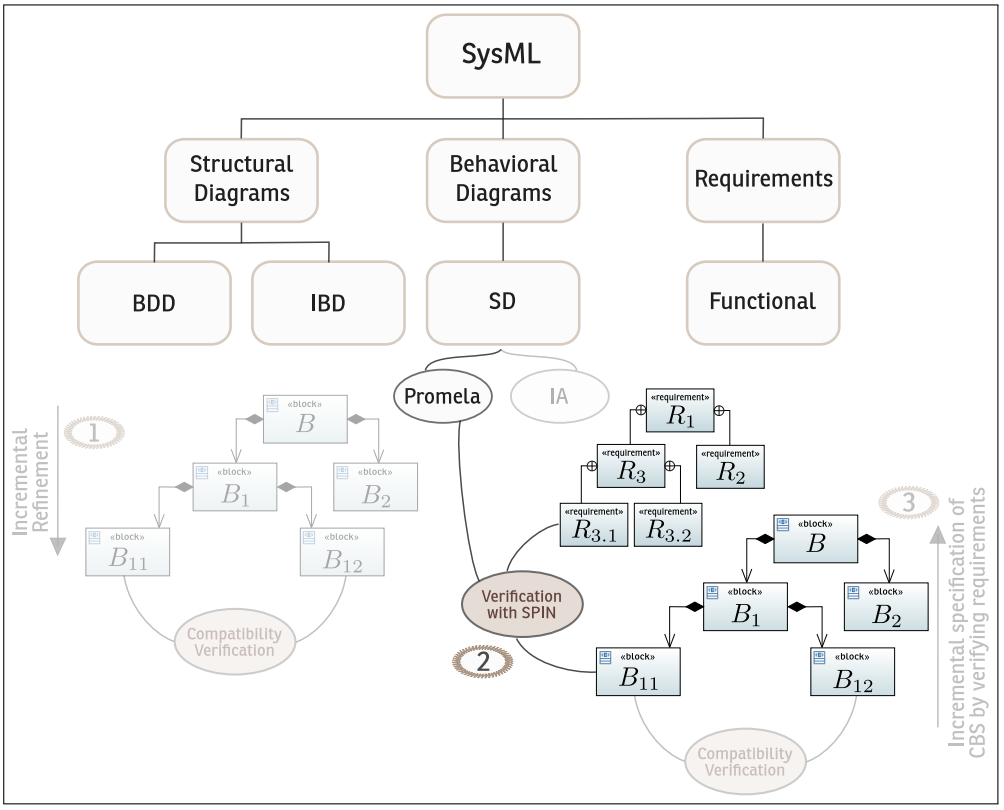

Figure 7.1 – Thesis contribution 2

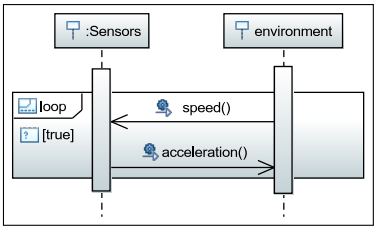

Figure 7.2 – SD for *Sensors* block

```
...
proctype proc_sensors(){
do
  sensors_environment_get_sensor_values?get_sensor_values;
  d_step{send=0; receive=1; msg_get_sensor_values=1;
  msg_sensor_values=0; sensors=1; environment=0;};
  sensors_environment_sensor_values!sensor_values; ...
od
}
proctype proc_environment(){
do
  sensors_environment_get_sensor_values!get_sensor_values;
  ...
  sensors_environment_sensor_values?sensor_values; ...
od
}
init{atomic{run proc_sensors();run proc_environment();}}
```

Figure 7.3 – Promela code for *Sensors* block

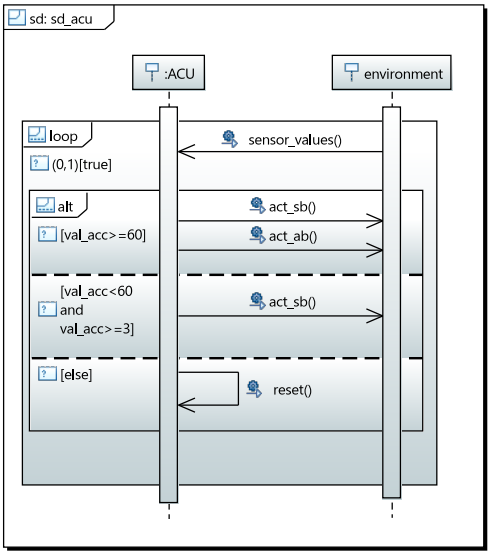

Figure 7.4 – SD for the *ACU* block

```
...
proctype proc_acu(){
do
 ::acu_environment_sensor_values?sensor_values;
 if
  ::(val_dec>=60)->{acu_environment_act_sb!act_sb; ...
    acu_environment_act_ab!act_ab;
    d_step{send=0; receive=1; ...};}
  ::((val_dec<60) && (val_dec>=3))->
    acu_environment_act_sb!act_sb; ...
  ::else{acu_reset!reset; ...
    acu_reset?reset; ...}
 fi;
od}

proctype proc_environment(){
do
 ::acu_environment_sensor_values!sensor_values; ...
 if
  ::((val_dec<60) && (val_dec>=3))->
    acu_environment_act_sb?act_sb; ...
  ::(val_dec>=60)->{acu_environment_act_sb?act_sb; ...
    acu_environment_act_ab?act_ab; ...}
 fi;
od}
init{
 if
  ::(true)->val_dec=0;
  ::(true)->val_dec=10;
  ::(true)->val_dec=60;
 fi;
 atomic{run proc_acu();run proc_environment();}
}
```

Figure 7.5 – Promela code for *ACU* block

7.5/ Case Study Promela descriptions

In the following, we present the complete Promela descriptions for the blocks sensors and ACU. To obtain these listings, we used a tool developed by us under the Eclipse framework, by means of an ADL and Acceleo transformation.

Listing 7.1 – "Promela Description for Sensors Block"

```
/*Messages declaration*/
mtype={get_sensor_values,sensor_values};

/*Channels declaration*/
chan sensors_environment_get_sensor_values=[1] of {mtype};
chan sensors_environment_sensor_values=[1] of {mtype};

/***FLAGS***/
/*Last performed action*/
bit send=0;
bit receive=0;
/*Message used in the last action*/
bit msg_get_sensor_values=0;
bit msg_sensor_values=0;
/*Lifeline that performed last action*/
bit sensors=0;
bit environment=0;

ltl p0 {[]((sensors && receive && msg_get_sensor_values) ->
          <>(sensors && send && msg_sensor_values))};
/*Lifelines specification */
proctype proc_sensors(){
do
    ::atomic{sensors_environment_get_sensor_values?get_sensor_values;
            d_step{send=0; receive=1;
                msg_get_sensor_values=1; msg_sensor_values=0;
                sensors=1; environment=0;};};
    atomic{sensors_environment_sensor_values!sensor_values;
            d_step{send=1; receive=0;
                msg_get_sensor_values=0; msg_sensor_values=1;
                sensors=1; environment=0;};};
od}

proctype proc_environment(){
do
    ::atomic{sensors_environment_get_sensor_values!get_sensor_values;
            d_step{send=1; receive=0;
                msg_get_sensor_values=1; msg_sensor_values=0;
                sensors=0; environment=1;};};
    atomic{sensors_environment_sensor_values?sensor_values;
            d_step{send=0; receive=1;
                msg_get_sensor_values=0; msg_sensor_values=1;
                sensors=0; environment=1;};};
od}

/*System Instantiation*/
init{
    atomic{run proc_sensors();run proc_environment();}
    }
```

Listing 7.2 – "Promela Description for ACU Block"

```
/*Messages declaration*/
mtype={sensor_values,act_sb,reset,act_ab};

/*Channels declaration*/
chan acu_environment_sensor_values=[1] of {mtype};
chan acu_environment_act_sb=[1] of {mtype};
chan acu_environment_act_ab=[1] of {mtype};
chan acu_reset=[1] of {mtype};
```

```
9
10 /*Variable global*/
11 byte val_acc=0;
12
13 /***FLAGS***/
14 /*Last performed action*/
15 bit send=0;
16 bit receive=0;
17 /*Message used in the last action*/
18 bit msg_sensor_values=0;
19 bit msg_act_sb=0;
20 bit msg_act_ab=0;
21 bit msg_reset=0;
22 /*Lifeline that performed last action*/
23 bit acu=0;
24 bit environment=0;
25
26 ltl p0 {[]((acu && receive && msg_sensor_values) ->
27         <>(acu && send && (msg_reset || msg_act_sb || msg_act_ab)))}
28 /*Lifelines specification */
29 proctype proc_acu(){
30 do
31     ::atomic{acu_environment_sensor_values?sensor_values;
32              d_step{send=0; receive=1;
33                 msg_sensor_values=1; msg_act_sb=0;
34                 msg_act_ab=0;  msg_reset=0;
35                 acu=1; environment=0;};};
36     if
37         ::(val_acc>=60)->{
38                         atomic{acu_environment_act_sb!act_sb;
39                                d_step{send=1; receive=0;
40                                     msg_sensor_values=0; msg_act_sb=1;
41                                     msg_act_ab=0;  msg_reset=0;
42                                     acu=1; environment=0;};};
43                         atomic{acu_environment_act_ab!act_ab;
44                                d_step{send=1; receive=0;
45                                     msg_sensor_values=0; msg_act_sb=0;
46                                     msg_act_ab=1;  msg_reset=0;
47                                     acu=1; environment=0;};};
48                         }
49         ::((val_acc<60) && (val_acc>=3))->
50             atomic{acu_environment_act_sb!act_sb;
51                    d_step{send=1; receive=0;
52                         msg_sensor_values=0; msg_act_sb=1;
53                         msg_act_ab=0;  msg_reset=0;
54                                      acu=1; environment=0;};};
55         ::else{
56             atomic{acu_reset!reset;
57                    d_step{send=1; receive=0;
58                         msg_sensor_values=0; msg_act_sb=0;
59                         msg_act_ab=0;  msg_reset=1;
60                         acu=1; environment=0;};};
61             atomic{acu_reset?reset;
62                    d_step{send=0; receive=1;
63                         msg_sensor_values=0; msg_act_sb=0;
64                         msg_act_ab=0;  msg_reset=1;
65                         acu=1; environment=0;};};}
66     fi;
67 od}
68
69 proctype proc_environment(){
70 do
71     ::atomic{acu_environment_sensor_values!sensor_values;
72             d_step{send=1; receive=0;
73                 msg_sensor_values=1; msg_act_sb=0;
74                 msg_act_ab=0;  msg_reset=0;
75                 acu=0; environment=1;};};
76     if
77         ::((val_acc<60) && (val_acc>=3))->
78             atomic{acu_environment_act_sb?act_sb;
```

```
79                      d_step{send=0; receive=1;
80                           msg_sensor_values=0; msg_act_sb=1;
81                           msg_act_ab=0;  msg_reset=0;
82                                     acu=0; environment=1;};};
83          ::(val_acc>=60)->{
84                           atomic{acu_environment_act_sb?act_sb;
85                                  d_step{send=0; receive=1;
86                                       msg_sensor_values=0; msg_act_sb=1;
87                                       msg_act_ab=0;  msg_reset=0;
88                                       acu=0; environment=1;};};
89                           atomic{acu_environment_act_ab?act_ab;
90                                  d_step{send=0; receive=1;
91                                       msg_sensor_values=0; msg_act_sb=0;
92                                       msg_act_ab=1;  msg_reset=0;
93                                       acu=0; environment=1;};};
94                           }
95      fi;
96  od}
97
98  /*System Instantiation*/
99  init{
100     if
101         ::(true) ->val_acc=0;
102         ::(true) ->val_acc=10;
103         ::(true) ->val_acc=60;
104     fi;
105     atomic{run proc_acu();run proc_environment();}
106 }
```

7.6/ Conclusion

We have presented in this chapter our proposal to verify SysML functional requirements over SysML blocks. To solve this issue, we have translated requirements into formal properties using linear temporal logic. LTL properties were then verified over a formal model of a SysML block. Formal models of SysML blocks were obtained by applying the approach of V. Lima et al. to translate SysML sequence diagrams into Promela descriptions. Finally we used the model-checker SPIN to verify LTL properties over the Promela descriptions.

8

Incremental Specification of CBS Architecture Guided by the Verification of SysML Requirements

The CBS are widely used in the industrial field, and they are built by assembling various reusable components (third party components), allowing reducing their development cost. The success of the CBS development is related to the process of building complex systems by assembling smaller and simpler components. Generally these systems are made larger because they are developed with software frameworks. However, this development is a hard task due to two reasons. The first is the difficulty to decide what to build and how to build it, by considering only system requirements and reusable components. Therefore, the question that arises is: how to specify a CBS architecture satisfying all system requirements? The second reason concerns the compatibility between the set of reusable components that compose the system, which must be guaranteed. Indeed, generally, one exploits reusable components from a component library to construct CBS, thus we need to guarantee component compatibility.

In this chapter, we discuss the relationship between system requirements and CBS architecture specification. Our goal is to propose a methodology, to the CBS specifier, to build a consistent system architecture that formally fulfills all the system requirements. To achieve this goal, we exploit the SysML requirement diagram to specify and organize system requirements, SD to describe components behavior, and BDD and IBD to specify

Contents

system architecture. In Figure 8.1, we show the position of the contribution presented in this chapter, regarding the contributions of this thesis.

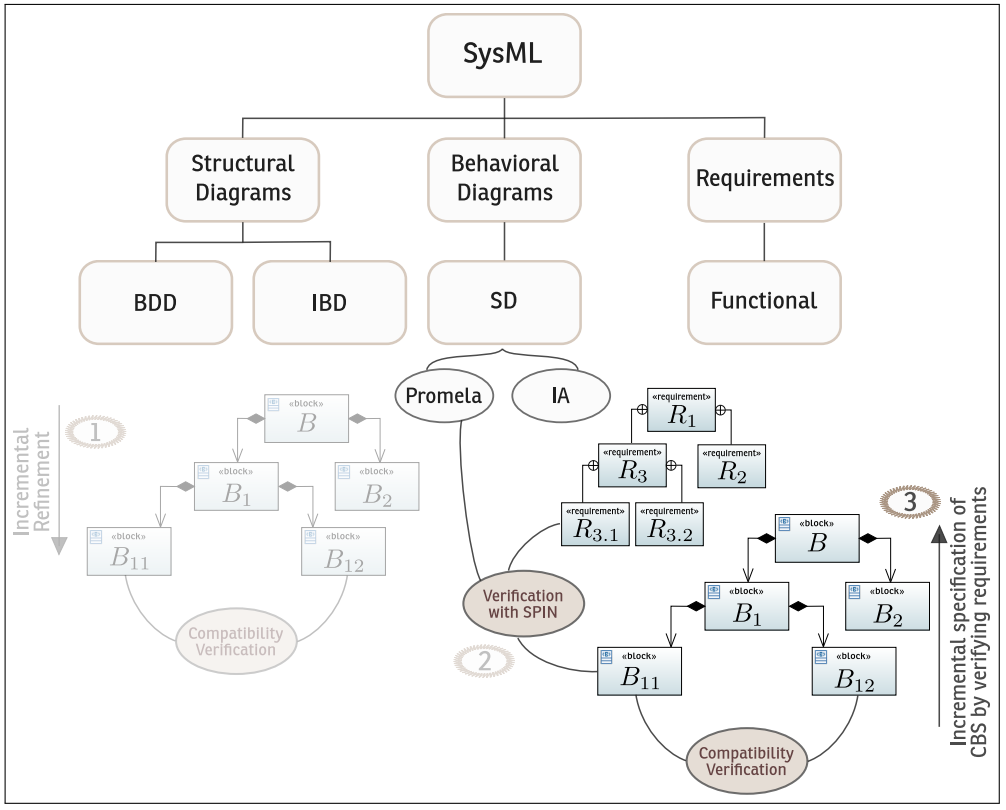

Figure 8.1 – Thesis contribution 3

We propose to extract the atomic requirements from a requirement diagram and treat them one by one, to construct a partial architecture of the system, composed of atomic components and composite components. At each step, we propose to select an atomic requirement from a SysML requirement diagram, and choose a component from a library that should satisfy the selected requirement. Then we verify whether the component satisfies the requirement thanks to the LTL formula which specifies the requirement and the Promela program which specifies the component SD (see Chapter 7). After that, we verify the compatibility between the selected component, and the selected one in the precedent step, and we verify also the preservation of the requirements treated in the precedent steps. This process ends when all atomic requirements are treated, or when we detect incompatibility between components, or the non preservation of the requirements by component composition. When the process ends correctly, we guarantee the architecture consistency of the final CBS which then fulfills all the requirements.

This chapter is organized as follows: An overview of the approach is presented in Section 8.1, then we present in Section 8.2 the case study used to validate the approach, and later in Sections 8.4, and 8.5 we describe the main steps of the approach.The illustration on the case study is shown in Section 8.6, and we end with the conclusion in Section 8.7.

8.1/ OVERVIEW

We propose an approach to construct a CBS system and to specify its architecture directly from SysML requirements. Our goal is to obtain a consistent architecture respecting all the specified requirements. To specify this architecture, the software architect exploits a library of reusable components (or blocks). These components are considered as black boxes and described only by their interfaces, specified with SD. So, we propose to specify CBS requirements with SysML requirement diagram, then analyze this diagram in order to associate one by one its atomic requirements (can not be decomposed) to software components that satisfy them. The satisfiability is evaluated by performing a formal verification step with a model-checker. Each verified component is tested for compatibility with the other components in the composition and then added to the partial architecture that must preserve the atomic requirements.

In our approach, a CBS is specified with a SysML requirement diagram that shows the functional requirements, and component interfaces describe component protocols by sequence diagrams.

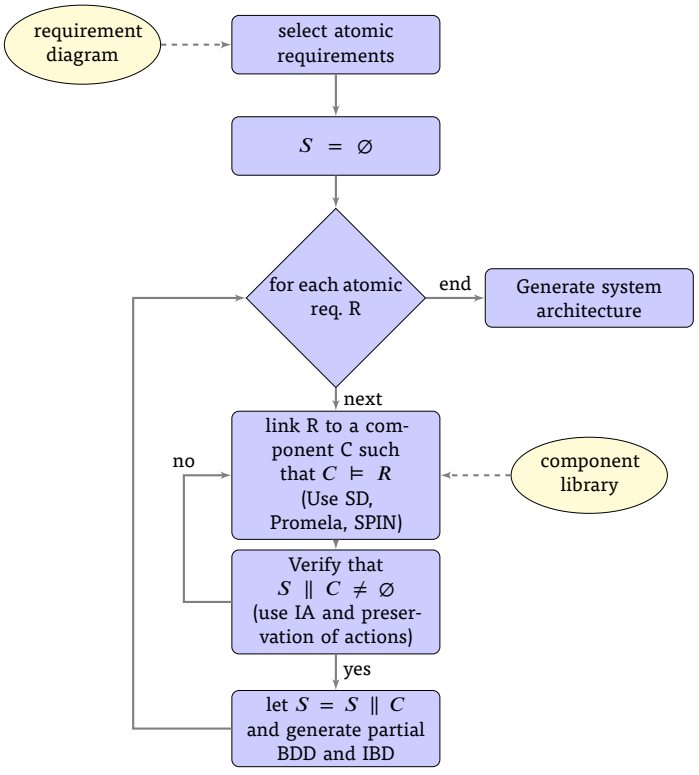

Figure 8.2 – Proposed approach to generate CBS architecture from SysML requirements

The main steps of our approach, presented in Figure 8.2, can be described as follows:

1. Start by analyzing the SysML requirement diagram to obtain the atomic requirements because they are more precise, and it is easier to find components that satisfy them (see Section 8.3).

2. Let R_i be the first atomic requirement, let C_i be a component from the component library, described by the sequence diagram SD_i. Specify R_i with the LTL formula

F_i and translate SD_i to the Promela code PRO_i, then verify that C_i satisfies R_i by verifying that PRO_i satisfies F_i with the model checker SPIN (see Section 7.3). The selection of the component C_i in the library is done by the software architect. However, it is possible to guide this selection (or to automate it) because R_i is a functional requirement, and describes constraints on offered and required services (Input/output actions). These services are also described in component interfaces. So it is easy to extract these services from R_i and to match them with those described in the interfaces. If this step returns *false*, then C_i does not satisfy R_i, therefore one has to obtain the appropriate component in other libraries, or to develop it from scratch.

3. Let A_i be the interface automaton describing the component protocol and obtained from the sequence diagram SD_i (see Section 8.4).
4. Identify the input and output actions in A_i related to R_i (Section 8.4).
5. Repeat until all the requirements are treated .
   (a) Let R_{i+1} be the next atomic requirement, *connected* to R_i (see Definition 19), let C_{i+1} be a component satisfying R_{i+1}, thanks to the LTL formula F_{i+1} and the Promela code PRO_{i+1}. Let A_{i+1} be the interface automaton describing the component protocol.
   (b) Identify the set of input and output actions in A_{i+1} related to R_{i+1}.
   (c) Verify that C_i and C_{i+1} are compatible thanks to their interface automata, so verify that $A_i \parallel A_{i+1} \neq \emptyset$ (see Section 4.2).
   (d) Verify that the requirements R_i and R_{i+1} are preserved by the composition, so they are satisfied by the composite $C = C_i \parallel C_{i+1}$ (see Section 8.4).
   (e) Define the consistent partial architecture of the system by the composite $C = C_i \parallel C_{i+1}$, according to Definition 20.
   (f) Let $C_i = C_i \parallel C_{i+1}$, $A_i = A_i \parallel A_{i+1}$, and $R_i = \{R_i, R_{i+1}\}$.
6. End repeat

According to the main steps of our approach, we validate the final architecture of our CBS when all the atomic requirements are analyzed without problems of component compatibility and/or requirement preservation.

8.2/ Case Study

To illustrate our approach, we use the case study of the vehicle safety system presented in Section 1.4. The associated requirement diagram that specifies the system needs is shown on Figure 8.3.

In this diagram, the initial requirement $R1$ asks for ensuring passengers lives and it is decomposed into two requirements $R1.1$ and $R1.2$ that ask for two safety devices: an airbag system, which must be deployed whenever the car is in a collision, and the seatbelts that must be locked when the sensors detect strong movements, therefore, this last is an atomic requirement as it is not decomposed. On the left side, requirement $R1.1$ is further decomposed into requirements $R1.1.1$, $R1.1.2$, and $R1.1.3$ which are atomic ones. Requirement $R1.1.1$ asks for the capture and sending of sensor values to an Airbag Control

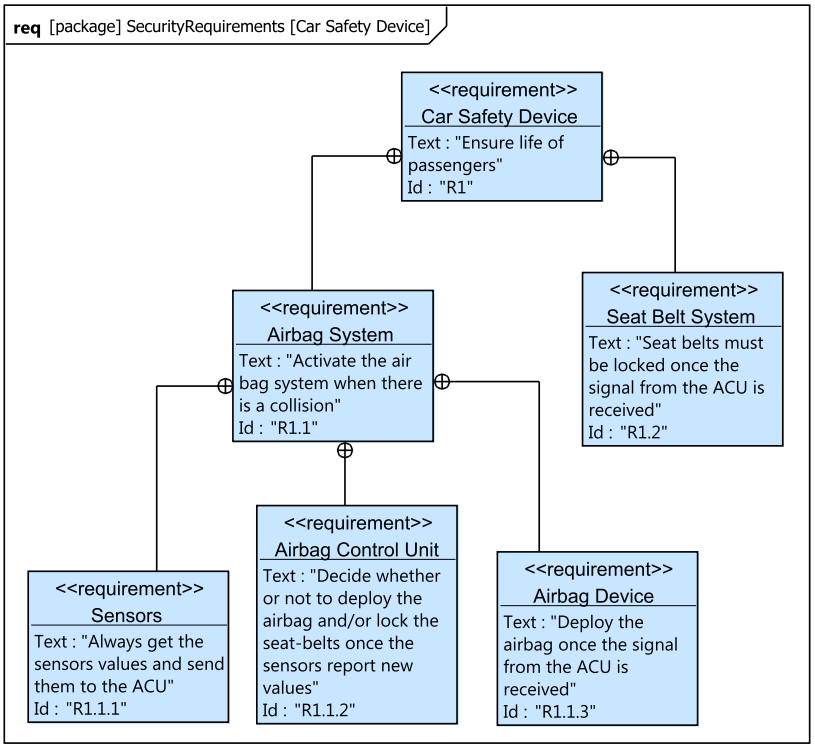

Figure 8.3 – SysML requirements diagram for a safety car system

Unit (ACU). Requirement $R1.1.2$ requests an ACU to decide whether to deploy the airbag and lock the seat-belts as soon as the sensors report new values. Finally, requirement $R1.1.3$ demands to deploy an airbag device, once the signal from the ACU is received.

8.3/ SysML Requirement Diagram Analysis

In this section, we specify formally the SysML requirement diagram in order to analyze it and to extract formally the atomic requirements. Then, we show that it is sufficient to a CBS to satisfy only the atomic requirements in order to satisfy all the requirements specified in the requirement diagram. In the following definition we consider two relations of SysML requirement diagram.

- **Containment**: exploited to decompose a requirement into other ones more precise.
- **Derivation**: exploited to connect a requirement with other ones that derive from it.

Definition 17: Requirement diagram specification

We specify a SysML requirement diagram by $RD = \langle IR, SR, RelC, RelD \rangle$ such that:

- IR: define the set of initial requirements, the first requirements that the specifier defines in the requirement diagram. Generally, they are not precise, and it is necessary to connect them, with the containment relation, to more refined requirements.
- SR: the set of all requirements.
- $RelC \subseteq SR \times P(SR)$ the relation of containment, where $P(SR)$ is the set of the subsets of SR.
- $RelD \subseteq SR \times P(SR)$ the relation of derivation.

For example in our case study, the specification of the requirement diagram described in Figure 8.3 is $RD = \langle IR, SR, RelC, RelD \rangle$, where:

- $IR = \{R1\}$,
- $SR = \{R1, R1.1, R2, R1.1.1, R1.1.2, R1.1.3\}$,
- $RelC = \{(R1, \{R1.1, R1.2\})$,
- $(R1.1, \{R1.1.1, R1.1.2, R1.1.3\})\}$, and
- $RelD = \emptyset$.

Definition 18: Atomic requirements

The set of atomic requirements in the requirement diagram specified by $RD = \langle IR, SR, RelC, RelD \rangle$ is the set $AR = \{R | R \in SR, \nexists (R, \{R_i, ...R_n\}) \in RelC\}$

An atomic requirement is a requirement that can not be decomposed. It expresses a constraint on input and output actions which are related to one component (see Section 8.4).

The atomic requirements in our case study are (see Figure 8.3): $\{R1.1.1, R1.1.2, R1.1.3, R1.2\}$.

Remark: To compute the set of atomic requirements, it is necessary to analyze the set SR of all requirements and to identify the requirements that are not related by the relation $RelC$ (containment).

Theorem 2: System satisfying all atomic requirements

Let S be a CBS, let $RD = \langle IR, SR, RelC, RelD \rangle$ be the specification of a requirement diagram, and let AR be the set of atomic requirements of RD. S satisfies all the requirements in SR iff it satisfies the atomic requirements AR.

Theorem 2 states that it is sufficient for a system to satisfy the atomic requirements, in order to satisfy all requirements represented in a SysML requirement diagram.

To illustrate the proof of this theorem, we propose a simple requirement diagram presented in Figure 8.4. The requirements are connected with a containment relation, with

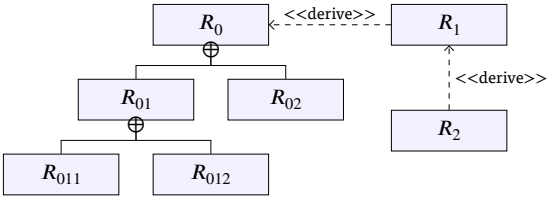

Figure 8.4 – A SysML requirement diagram

continuous arrows, and a derivation relation, with dashed arrows. So the requirement $R0$ is decomposed in the requirements $R01$ and $R02$. The requirement $R1$ derives from $R0$, and $R2$ derives from $R1$.

Proof. Due to the semantic of the relation derive in SysML requirement diagram (and also the semantic of requirement diagram), it's obvious to state that a system satisfies all requirements that are specified by a requirement diagram iff it satisfies the initial requirements and all the ones that are derived (linked by the relation derive) directly or indirectly from them. Indeed, the satisfaction of the derived requirements does not guarantee the satisfaction of the initial ones. Since the atomic requirements are either derived (directly or indirectly) from initial requirements, or related by the relation of containment (directly or not) to initial requirements. And due to the semantic of the containment and the derive relations, the satisfaction of atomic requirements leads to the satisfaction of the requirement which are linked to them. Therefore it is sufficient to satisfy atomic requirements to satisfy all requirements. For example in Figure 8.4, the requirements to satisfy (initial and derived) are $\{R0, R1, R2\}$. The derived requirements are $R1$ and $R2$, and the initial requirement is $R0$. However, to satisfy a requirement composed of other ones, it is sufficient to satisfy the requirements that compose it. This process is repeated until all the atomic requirements are satisfied. So, to satisfy $R0$, it is sufficient to satisfy $R01$ and $R02$. And to satisfy $R01$ it is enough to satisfy $R011$ and $R012$. Therefore, to fulfill all requirements, it is necessary to satisfy $\{R1, R2, R02, R011, R012\}$, which defines the set of atomic requirements in our requirement diagram. □

8.4/ Component Assembly Preserving SysML Requirements

In this section, we specify interface automata from sequence diagrams thanks to the approach proposed in [CH11], then we propose to compose components and to verify compatibility between them, using their interface automata, by applying the algorithm presented in Chapter 4. We propose also to verify, at the same time as the compatibility verification is done, whether the atomic SysML requirements are preserved over the composition. Indeed, this verification allows avoiding the requirement verification over the obtained composite component, thus, we avoid the state explosion problem for the model checker. Before presenting the algorithm to verify the preservation, we show in the following sections that SysML requirements are related to Input/Output actions of interface automata, and their preservation is related to the preservation of their related actions by the composition.

8.4.1/ Functional Requirements and Input/Output Actions

The atomic requirements considered in this work concern the functional properties of a CBS. They are related directly to input and output actions of components. Therefore, for each atomic requirement we associate the sets of input and output actions provided by a component.

Let AR be the set of atomic requirements in the specification RD of a requirement diagram. Let R_i be an atomic requirement satisfied by the component C_i. Let A_i be the interface automaton describing the protocol of C_i. So, R_i is associated to input actions $I_{R_i} = \{i_{ri_1}, ..., i_{ri_n}\}$, and output ones $O_{R_i} = \{o_{ri_1}, ..., o_{ri_n}\}$.

For example the first atomic requirement in our case study is $R1.1.1$: *always get the sensor values and send them to the ACU*. It is satisfied by the component Sensor. The interface automaton of this component is described in Figure 8.6. The set of input actions related to $R1.1.1$ is $\{get\_sensor\_values\}$, and the set of output actions is $\{sensor\_values\}$.

The actions related to atomic requirements are formalized by transitions in the interface automata, labeled with these input/output actions.

Definition 19: Connected requirements

Let R and R' be two atomic requirements specified in a SysML requirement diagram. R and R' are related respectively to the set of input actions I_R, and I'_R, and output ones O_R, and O'_R. R and R' are connected iff $I_R \cap O'_R \neq \emptyset$ or $I'_R \cap O_R \neq \emptyset$.

According to Definition 19 and to the condition of composability of interface automata (see Chapter 4), it is obvious to state that two components satisfying two connected atomic requirements are composable. We exploit this definition in our approach: at each iteration i of our approach, we choose an atomic requirement which is connected with the requirement in the iteration $i-1$, in order to compose their components, otherwise the composition is not allowed.

8.4.2/ Preservation of Input/Output Actions in Automata Composition

In this section, we show that the composition of two interface automata does not guarantee the preservation of their non shared input/output actions in the obtained composite automaton, despite their compatibility.

In fact, in the item (iii) of Definition 4, the authors in [dAH01] indicate that the set of actions in the composite automaton $A = A_1 \parallel A_2$ is the same as the set of actions in the synchronized product $A_1 \otimes A_2$, however, the set of transitions in A is not the same as the one in $A_1 \otimes A_2$ (according to Definition 4). Indeed, the set of transitions in A is included in the one of $A_1 \otimes A_2$. Thus, there may be input/output actions in Σ_A which are not associated to transitions in A. In fact according to the optimistic approach of interface automata, despite that A_1 and A_2 are compatible, and $A \neq \emptyset$, there may be shared input/output actions between A_1 and A_2 which do not synchronize, but certainly, there are also shared actions which synchronize (because $A \neq \emptyset$). Therefore, the transitions labeled with the shared input/output actions, which do not synchronize, will be eliminated from $A = A_1 \parallel A_2$ because they lead to illegal states. But the related input/output actions (which label the eliminated transitions) remain in the set of actions in A, because the composite component described by the composite automaton A could provide these

actions, and with the optimistic approach, one decides that it is compatible, because one supposes the existence of the helpful environment which never enables these actions (for more illustration see the example in [dAH01]).

8.4.3/ Verification of the Preservation of the Atomic Requirements by the Composition.

In this section, we show the conditions that the composition of components should respect to preserve the requirements of the composed components. And we show also how to verify these conditions by adapting the compatibility verification algorithm of interface automata (see Chapter 4).

The preservation of the atomic requirements by the composition of components is necessarily related to the preservation of the input/output actions, associated to these requirements, by the composition of their interface automata. Furthermore, in Section 8.4.2, we indicate that some input/output actions may belong to the set of actions of a composite automaton, but they do not label transitions in this automaton. Hence, in this case we state that these actions are not preserved.

Condition of Input/Output action preservation: An action act (input or output) is preserved by the composition of two interface automata, A_1 and A_2, iff there is at least one transition in the composite automaton, $A = A_1 \parallel A_2$, which is labeled with act. Which means that the action act belongs to the set of Input/Output actions in A, when act is not shared between A_1 and A_2, and belongs to the internal actions in A otherwise.

Verification algorithm overview: To verify the preservation of atomic requirements by the composition, we propose to adapt the compatibility verification algorithm [dAH01] (Chapter 4). We verify whether the transitions labeled with input/output actions, related to atomic requirements, are preserved in the transition set of the obtained composite automaton $A_i \parallel A_{i+1}$. This adaptation consists on: to calculate in the step (2) of the compatibility verification algorithm, the set of transitions in $A_i \otimes A_{i+1}$, noted T, related to the requirements. When we eliminate transitions in the step (5) of the compatibility verification algorithm, we eliminate also these transitions in T. Finally, we verify that all the actions related to the requirements, are associated to at least one transition in T, after step (6).

We notice that this adaptation does not increase the complexity of the compatibility verification algorithm (this can be easily verifiable). So the complexity of the presented algorithm is $\mathcal{O}|A_i \otimes A_{i+1}|$. However, in order to calculate the time complexity of one step in our approach, we have to consider a component C_c associated to the current requirement to analyze, R_c . This component is selected from the components library specified by the set $C = \{C_1, C_2, ..., C_n\}$. We consider also the sequence diagrams SD_c that specifies the protocol of C_c. In each step we have to verify that the current component satisfies the current requirement thanks to the Promela code of SD_c and to the model checker SPIN. And we verify also the compatibility between the current component and the composite component, C_p, obtained in a precedent step. So the time complexity of one step in our approach is analyzed as follows:

- to select a component from the set C that should satisfy an atomic requirement, the complexity is: $\mathcal{O}(|C|)$

- to verify that C_c satisfies R_c, the complexity is : $\mathcal{O}(|TS_c| \times 2^{|Pc|})$, where TS_c is the automaton calculated by SPIN from the Promela code associated to SD_c (this is the complexity of the LTL model checking), and Pc the LTL formula that specifies the requirement R_c.

- After verifying the atomic current requirement on the component, we verify the compatibility between C_c and C_p and the preservation of the requirements by the composition. The complexity of this step is : $\mathcal{O}(|A_c \otimes A_p|)$, where A_c and A_p are the interface automata associated respectively to C_c and C_p.

Therefore, to calculate the complexity of the whole approach, we have to consider the complexity of one step and the number of the atomic requirements which defines the number of steps.

To demonstrate the correctness of our approach, we should prove that the composition of two components preserves the atomic requirements iff the composed components are compatible and the input and output actions related to these requirements are preserved according to the condition of preservation of input/output actions. Indeed, each step in the incremental approach is based on the *compatibility* and the *preservation* of the atomic requirements by the composition. Thus, it is sufficient to show the correctness of a step i in our approach.

Theorem 3: Preservation of requirements

Let C_i be a component satisfying the atomic requirement R_i and A_i the interface automaton of C_i, let I_i be the set of input actions related to R_i and O_i the output ones. Let C_{i+1} be a component satisfying R_{i+1} and A_{i+1} the interface automaton of C_{i+1}, let I_{i+1} be the set of input actions related to R_{i+1} and O_{i+1} the output ones. The composite component $S = C_i \parallel C_{i+1}$ preserves the requirements $\{R_i, R_{i+1}\}$ iff the interface automata A_i, and A_{i+1}, are compatible, and the input and output actions, I_i, I_{i+1}, O_i, and O_{i+1} are preserved in S.

Proof. The component C_i satisfies R_i means that the program Promela describing the component behaviors satisfies the LTL property specifying the requirement R_i. In our approach, component behaviors are also described with an interface automaton A_i, and these behaviors are execution paths in the interface automaton. The functional requirement R_i is related to the sets of input/output actions, I_i, O_i, and they express constraints and the order of executing these actions. For example R_i could express: always when C_i enables an input action $i \in I_i$ then it will inevitably enable the output actions $o \in O_i$. So C_i satisfies this requirement iff in all the execution paths in A_i where a transition labeled by i belongs, it will be followed by a transition labeled with o. Since our composition approach preserves at least one of these paths, when the compatibility and the preservation of Input/Output actions hold, then the requirements are preserved.

Indeed, the composite $S = C_i \parallel C_{i+1}$ preserves the input/output actions related to the requirements means that for each input/output actions related to R_i and R_{i+1}, the transitions labeled with these actions are preserved, therefore at least one execution path, in $A_i \parallel A_{i+1}$ containing these transitions is preserved in S. Indeed, we have the following possibilities when A_i and A_{i+1} are compatible and the actions related to R_i and R_{i+1} are preserved (illustration concerning only one action a related to R_i or R_{i+1}):

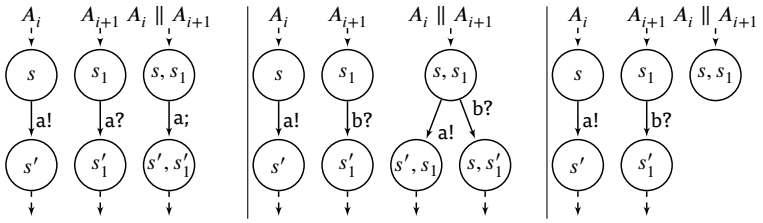

Figure 8.5 – Interface automata composition alternatives

- if there are no illegal states the preservation is guaranteed, because all the paths are preserved. This case (we have two possibilities) is illustrated in the Figure 8.5(a) and (b). In the case (a), we suppose that there is a synchronization between the two automata on the shared action a (related to a requirement), so in the composition we obtain a transition labeled with the internal action a. Therefore, the action a is preserved and becomes internal. And in the second case (b), we suppose that a is not shared and there is interleaving in the composite automaton, and a is preserved.

- if there are illegal states (and the automata are compatible due the optimistic approach): in this case (see Figure 8.5(c)), we suppose that the first automaton provides a shared output action a, in the state s, and the second automaton does not provide the input action a, in $s1$. So we obtain an illegal state and the action a is not enabled in the illegal state $(s, s1)$. In this case we have to verify that a is preserved in other paths of the composite automaton $A_i \parallel A_{i+1}$, with our approach. So, if we find a transition labeled with a, in $A_i \parallel A_{i+1}$, so it is preserved (according to the condition of preservation), and the associated requirement also, otherwise the related requirement is not preserved. So when the preservation of the Input/Output action is verified, then the related requirements are preserved.

□

8.5/ Specification of System Architecture

The construction of a CBS with our approach is based on constructing, at each incremental step, one SysML composite component, which defines a partial architecture of a CBS. This architecture is based on the interface automata of the assembled components and particularly on their shared actions. So, in the following definition, we describe the SysML composite by specifying the relation between SysML BDD and IBD diagrams, and the interface automata describing the behaviors of the composed components.

Definition 20: SysML Composite component

Let C_1 and C_2 be two components, let A_1 and A_2 be their respective interface automata. When A_1 and A_2 are compatible, $A_1||A_2 \neq \emptyset$, the composite component C composed of C_1 and C_2, is well formed and it is written $C = C_1 \parallel C_2$. This composite is described with the SysML BDD diagram, BDD_C, composed of the composite block C, and the blocks C_1 and C_2. The interactions between the components C_1 and C_2 are described with the SysML IBD diagram, IBD_C such that, IBD_C is composed of the parts C_1 and C_2 which communicate through internal ports, labeled with the names of the synchronized input and output actions, which are shared between A_1 and A_2. The external ports of IBD_C are labeled with the names of actions which are not shared.

This definition is illustrated in Section 8.6 in Figures 8.9 and 8.10 (BDD and IBD).

8.6/ Illustration on the Case Study

In this section we apply our approach on the case study shown in Section 8.2. As exposed in the approach, we start by analyzing the SysML requirement diagram to obtain the atomic requirements. These requirements are $R1.1.1$, $R1.1.2$, $R1.1.3$, and $R1.2$. Then, we link LTL properties for each of these atomic requirements. These properties are used to verify whether a block in a component library satisfies the requirement in order to match them.

For the first requirement $R1.1.1$ we take a sensor block with its associated SD shown in Figure 7.2 respectively. This sensor block gets information from several sensors (accelerometers, impact sensors,...) all around the car at each call of the service `get_sensor_values`, and sends them through a service `sensor_values`. These services are respectively the input {`get_sensor_values`} and output actions {`sensor_values`} related to requirement $R1.1.1$. To validate if the block *sensors* satisfies requirement $R1.1.1$, we first describe the requirement as a LTL property like "*always, after the sensors block receives a call for* `get_sensor_values`*, it sends a message* `sensor_values` *to the environment*". Then we translate the associated SD to a Promela description as exposed in Chapter 7, the generated code is not shown here for lack of space. Following the approach of flags from [LTM+09], the LTL property in Promela language is:

```
□((sensors && receive && msg_get_sensor_values)
        → ◊(sensors && send && msg_sensor_values))
```

The next requirement to be analyzed is $R1.1.2$ which is connected to $R1.1.1$. For this requirement, we find the ACU block and its associated SD in Figure 7.4, this block offers an input action {`sensor_values`} and requires the output actions {`act_sb,act_ab`} to lock the seat-belts and deploy an airbag respectively, this block analyzes each arrival of sensor values and decides whether the seat-belts must be locked, an airbag must be deployed or wait for another sensor values arrival (`reset` action). To verify if this block satisfies requirement $R1.1.2$, we express it as a Promela description (the generated code was shown in Listing 7.2) and the requirement is expressed as a LTL property: "*always after receiving a message with the sensor_values, the ACU will send a message deciding to lock the seat-belt (*`act_sb`*), activate the airbag (*`act_ab`*) or wait for another call (*`reset`*)*", which expressed in Promela code using flags will be:

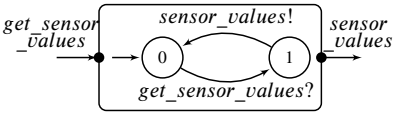

Figure 8.6 – IA for the *Sensors* block

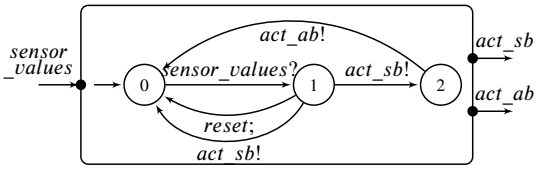

Figure 8.7 – IA for the *ACU* block

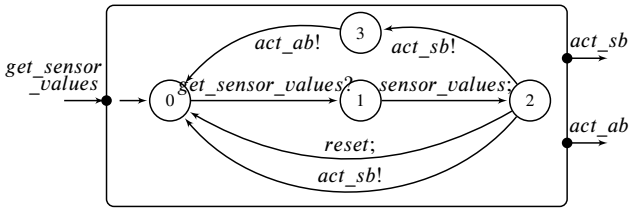

Figure 8.8 – IA for the composition of *Sensors* and *ACU* blocks

$\square$((acu && receive && msg_sensor_values)
$\rightarrow \Diamond$ (acu && send && (msg_reset || msg_act_sb || msg_act_ab)))

These properties are verified using SPIN model-checker which outputs no errors for both models, therefore, the models satisfy the properties.

Then, to link the blocks that satisfy requirements $R1.1.1$ and $R1.1.2$, we verify that they are compatible thanks to their interface automata. These interface automata are generated from SD following the approach in [CH11], and they are shown in Figures 8.6 and 8.7. To verify compatibility we compute the composition; we use Ptolemy Interface Automata tool [LX04] which computes the composition of two given interface automata as input. The output composite automaton is shown in Figure 8.8, this automaton is not empty, so the blocks Sensors and ACU are compatible. This composition had illegal states that were eliminated automatically by Ptolemy tool, so we have to validate that the actions related to the requirements are still present on the transitions of the composite automaton to guarantee preservation of the requirements over the composition.

Looking at the transitions in the composite automaton, we find that the set of input/output actions, related to the requirements, are still present, so the requirements are still preserved over the composition and we can proceed to define a partial architecture of the system, by presenting a BDD with the refinement of an abstract block into the blocks Sensors and ACU, this diagram is presented in Figure 8.9.

The interactions between the composed blocks are then described by an IBD (see Figure 8.10) where the ports representing the synchronized input and output actions are linked with connectors and the unshared actions are exposed as offered and demanded services of the composition.

Subsequently, we continue adding requirements $R1.1.3$, with related input action {act_ab}, and $R1.2$, with related input action {act_sb}, to our architecture in the same manner, until all atomic requirements are treated.

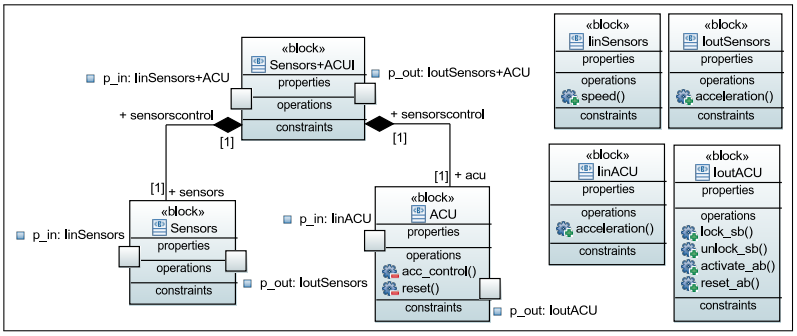

Figure 8.9 – BDD for the second iteration

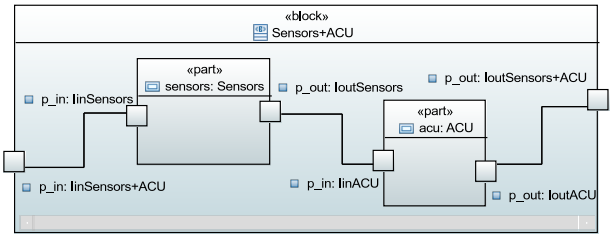

Figure 8.10 – IBD for the second iteration

8.7/ Conclusion

In reliable applications, it is important to specify a system architecture in accord with the requirement specifications. To achieve this goal, in this chapter we proposed an approach to specify system architecture directly from SysML functional requirements. SysML requirement diagram was analyzed to extract its atomic requirements. Then, we associate these requirements, one by one, to reusable components, and LTL properties representing the requirements were verified on the components. To verify an LTL property, component behavior represented in SD is translated into Promela statements and then verified with the SPIN model-checker. These components were then added to a partial architecture by the composition of their component interfaces described through interface automata. We guarateed preservation of requirements over the composition by the conservation of related Input/Output actions on the transitions of the composite automaton. Finally, we illustrated this approach by the case study of a safety vehicle system.

III

Conclusion

9

Conclusion and Perspectives

The conclusion of this thesis summarizes the obtained results and it presents some perspectives for our work.

We present in Figure 9.1 the global approach and the perspectives.

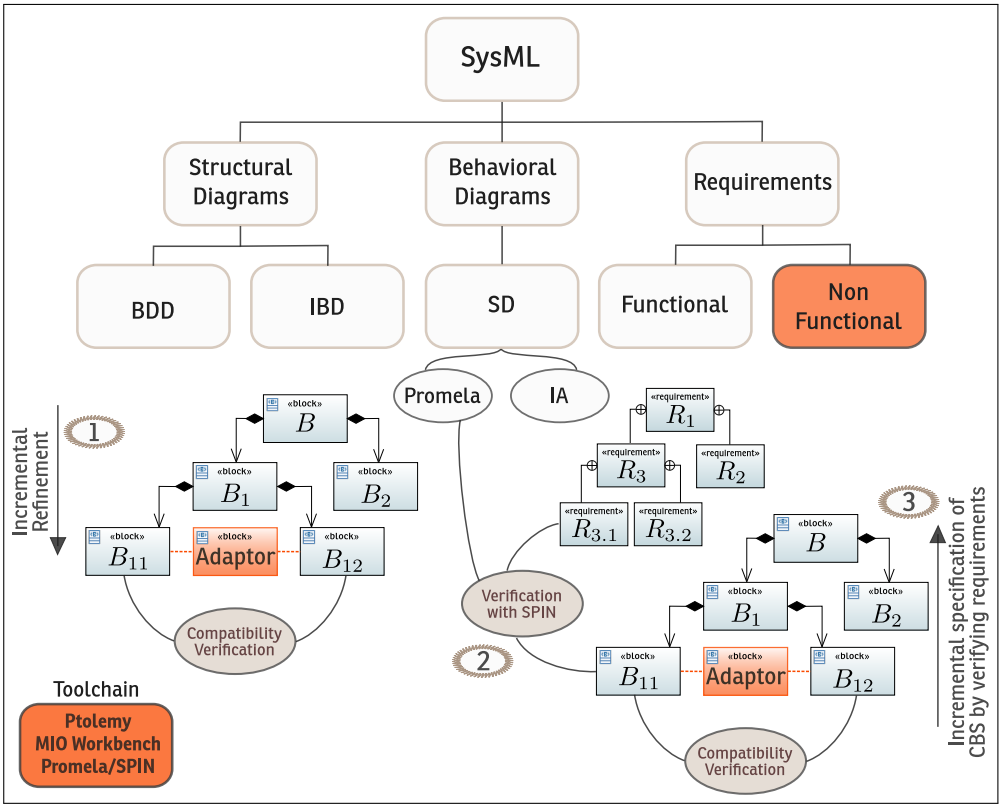

Figure 9.1 – Thesis perspectives

CBS are a promising solution widely exploited in industrial applications. The approach to develop CBS is based on the idea that software systems can be developed by selecting appropriate components and then assemble them in a valid architecture. The success of the CBS development is related to the process of building complex systems by assembling various reusable components, allowing reducing their development cost. Generally, these systems are made larger because they are developed with software frameworks. However, this development can become a difficult task when the designer must decide what to build and how to build it, by considering only system requirements and reusable components.

Therefore, there are three challenges that a designer must confront:

1. The first one is how to specify an architecture that satisfies all system requirements?
2. The second one concerns the compatibility between the set of reusable components that compose the system, which must be guaranteed. Indeed, generally, one exploits reusable components from a component library to construct CBS, thus, we need to guarantee component compatibility.
3. And finally, we ask for how to integrate formal verification in the assembly process to build reliable systems.

The philosophy of this thesis is inspired from the concept of CBS, to build, step by step, SysML specifications of a system, by exploiting the notion of SysML refinement. Structural diagrams of SysML describe the system in static mode, and behavioural diagrams describe the dynamic operation of the system. In SysML, blocks are modeled by two diagrams. The BDD, which defines the architecture of the blocks and their performed operations, and the IBD, which is used to define the ports of each block and transactions exchanged between them through their ports. During the refinement process, these two diagrams can be checked to decide whether the proposed architecture satisfies or it is inappropriate to the requirements diagram.

9.1/ Main Contributions

In this thesis we presented three contributions based on an incremental approach.

1. The first contribution aimed to formalize the decomposition process, by defining a refinement relation between an abstract block and its sub-blocks. It consisted on exploiting the architecture description in SysML language when a system is described by structural diagrams and behavioral diagrams. The refinement in SysML is an essential concept and it is based on the development of a process from an abstract level towards more detailed levels, which can end in its implementation. Our refinement ensured an incremental substitutability of an abstract block in a specification by a composition of blocks preserving its structural and behavioral properties.

   To verify structural refinement specified in BDD and IBD diagrams, we verified first if the sub-blocks were consistent with the abstract block specification and then we verified if they were compatible by applying the model of interface automata of d'Alfaro et al. Interface automata were obtained from the SysML SD of each composing sub-block and then verified for compatibility by means of the *Ptolemy* tool .

   To verify behavioral refinement, we applied the approach of alternating simulation for interface automata to verify if the composition of the set of sub-blocks simulated the expected behavior in the abstract block. To ease the task of verifying alternating simulation we proposed to use the module *Observational Modal Refinement* of the *MIO Workbench* tool. To use this tool, we translated interface automata into modal automata.

2. The second contribution focused on properties verification in our system. We were inspired from works proposed by V. Lima et al. The technique proposes to generate Promela-based models from UML interactions expressed in Sequence Diagrams

(SD), and uses conjointly the SPIN model checker in order to simulate and verify properties written in Linear Temporal Logic(LTL). Our minor adaptation of this approach concerned a particular type of sequence diagrams that we exploited to specify the block behaviors. We chose Promela to describe SD specifications and LTL properties to describe functional requirements. We then used SPIN tool to verify these properties. We chose this environment implementation because it is a popular tool in verification activity and it is easy to specify and implement SD concepts like sending and receiving primitives, parallel and asynchronous composition.

3. The third contribution discussed an interesting approach to describe the relationship between system requirements and a CBS architecture specification in SysML. The goal was to propose a methodology to build incrementally a consistent system architecture that formally fulfills all the system requirements. We used SD to describe component behavior and BDD and IBD to specify system architecture. In details, the proposed construction extracted the atomic requirements from a requirement diagram and treated them one by one in order to construct the final system. We obtained then a partial architecture of the system, composed with elementary blocks (components) and composite blocks (components). At each step, we selected an atomic requirement from a SysML requirement diagram, and we chose a block from a library that should satisfy the selected requirement. Then we verified whether the block satisfied the requirement thanks to the LTL properties which specified the requirement and the Promela specification which described the component behavior from SD. We then verified the compatibility between the selected block, and the selected ones in the precedent steps, and we verified also the preservation of requirements treated in precedent steps. The process ended when all atomic requirements were treated, or when we detected incompatibility between components, or the non preservation of the requirements by the component composition. In this way we guaranteed the architecture consistency of the final system which therefore fulfilled all the system requirements.

9.2/ Perspectives of the Work

The works of this thesis targeted the main question: how to introduce formal verification on informal SyML specifications in the process of the development of CBS?

Our contributions addressed some solutions in order to build consistent CBS, thus we established relations between refinement, concepts of SysML blocks, and CBS characteristics. In the following, we present some perspectives and future works related to our contributions.

Toolchain for verification Develop and provide for the designer a toolchain to support the automatic verification of the refinement relation between abstract blocks and sub-blocks, and also the verification of SysML requirements on blocks to decide or not the validation of CBS SysML architecture. This toolchain will be composed of tools that allow (1) verifying conditions of consistency between blocks and exploit Ptolemy tool to verify compatibility (2) translating directly diagrams into Promela language (3) and verifying LTL properties with the model-checker Spin. Some significant examples can be experimented to evaluate the proposed approach.

Combining verification and simulation for non functional properties In reliable applications, it is important to specify a system architecture in accord with functional and non functional requirement specifications. To achieve this goal, we have proposed an approach to specify system architecture directly from SysML functional requirements. So, as a future work, it is interesting to address the validation problem with non functional requirements and use simulation techniques.

Adapting and Generating adapters for incompatible blocks The problem of adapting blocks is crucial in the development of CBS by reusing blocks. The adaptation consists to generate automatically, when it is possible, an adaptor block between incompatibles blocks in order to ensure a reliable interaction. The idea of a future work, is to generate an entity capable of ensuring the interaction between two incompatible blocks when the conditions of consistency and compatibility are failed, allowing thus achieving our approach of refinement of abstract blocks, and the generation of CBS architecture based on SysML requirements.

IV

Résumé étendu

10

Vérification Formelle et Incrémentale de Spécifications SysML pour la Conception de Systèmes à Base de Composants

Le travail présenté dans cette thèse est une contribution à la spécification et à la vérification des Systèmes à Base de Composants (SBC) modélisé avec le langage SysML. Les SBC sont largement utilisés dans le domaine industriel et ils sont construits en assemblant différents composants réutilisables, permettant ainsi le développement de systèmes complexes en réduisant leur coût de développement. Malgré le succès de l'utilisation des SBC, leur conception est une étape de plus en plus complexe qui nécessite la mise en œuvre d'approches plus rigoureuses.

Pour faciliter la communication entre les différentes parties impliquées dans le développement d'un SBC, un des langages largement utilisé est SysML, qui permet de modéliser, en plus de la structure et le comportement du système, aussi ses exigences. Il offre un standard de modélisation, spécification et documentation de systèmes, dans lequel il est possible de développer un système, partant d'un niveau abstrait, vers des niveaux plus détaillés pouvant aboutir à une implémentation. Généralement ces systèmes sont faits plus grands parce qu'ils sont développés avec des cadres logiciels.

Dans ce contexte nous avons traité principalement deux problématiques :

La première est liée au développement par raffinement d'un SBC modélisé uniquement

Contents

par ses interfaces SysML. Notre contribution permet au concepteur des SBC de garantir formellement qu'une composition d'un ensemble de composants élémentaires et réutilisables raffine une spécification abstraite d'un SBC. Dans cette contribution, nous exploitons les outils: Ptolemy pour la vérification de la compatibilité des composants assemblés, et l'outil MIO Workbench pour la vérification du raffinement

La deuxième problématique traitée concerne la difficulté de déterminer quoi construire et comment le construire, en considérant seulement les exigences du système et des composants réutilisables, donc la question qui en découle est la suivante: comment spécifier une architecture SBC qui satisfait toutes les exigences du système? Nous proposons une approche de vérification formelle incrémentale basée sur des modèles SysML et des automates d'interface pour guider, par les exigences, le concepteur SBC afin de définir une architecture de système cohérente, qui satisfait toutes les exigences SysML proposées. Dans cette approche nous exploitons le model-checker SPIN et la LTL pour spécifier et vérifier les exigences.

Dans ce chapitre nous présentons un résumé de ces contributions structuré ainsi : d'abord nous présentons un contexte scientifique qui liste les concepts sur lesquels nous nous sommes basés (SBC, SysML, Automates d'Interface), ensuite nous présentons nos contributions et enfin nous listons nos conclusions et perspectives de ces travaux.

10.1/ Contexte Scientifique

Dans cette section nous présentons les concepts de base utilisés dans cette thèse. D'abord, nous introduisons les SBC, puis SysML qui est le langage choisi pour modéliser les SBC et enfin les automates d'interface que nous utilisons pour la vérification de la consistance et compatibilité des blocs.

10.1.1/ Systèmes à Base de Composants

Au fils des années, le domaine du développement logiciel a evolué à travers de différents paradigmes. La programmation structurée a changé dans le temps vers le paradigme des classes et puis vers la révolution de la programmation orientée objets. Les objets des nos jours ont grandit et ils sont identifiés comme des *composants logiciels*. Dans cette section, nous allons définir et décrire les propriétés de ces derniers pour mieux comprendre les differences entre objets et composants.

Plusieurs définitions ont été proposés pour définir les composants logiciels et une des plus complètes a été proposé dans [SP97]. Cette définition est :

"Un composant logiciel est une unité de composition avec des interfaces spécifiés contractuellement et seulement dépendances de contexte explicites. Un composant logiciel peut être déployé indépendamment et est sujet à la composition par un tiers".

À partir de cette définition, un composant logiciel est une unité de composition avec d'autres paires, un composant doit encapsuler son implémentation et interagir avec son environnement en tenant compte de seulement ses interfaces bien définis. Ces interfaces doivent donner l'information sur les exigences du composant de la part d'autres composants et les services qu'il peut offrir.

Cependant, pour utiliser un composant correctement, il est nécessaire de satisfaire un

contrat. Ce contrat liste une série de contraintes sur la manière d'exécuter le composant pour que celui-là exécute ses fonctionnalités [Szy02]. Il est aussi requis de définir ce que l'environnement de composition et déploiement doit fournir pour faire interagir les composants proprement.Cet environnement est composé d'un modèle à composants avec des règles de composition et un cadre qui établi déploiement, installation et activation des règles des composants. Ainsi, les systèmes logiciels conçus pour être un assemblage de composants avec une architecture prédéfinit sont appelés Systèmes à Base de Composants (SBC).

Les composants peuvent être raffinés et améliorés par des versions ultérieurs. Une entreprise qui vend des composants tiers peut proposer différentes versions améliorées du même composant. Une gestion traditionnelle des versions supposerait que la version d'un composant évolue d'une seule source. Cependant, dans le marché ouvert, l'évolution des versions est plus complexe et la gestion des versions peut devenir un problème en soi, surtout parce que les versions peuvent aussi changer au niveau de son interface.

10.1.2/ Le Langage SysML

SysML (Systems Modeling Language) [OMG12] est un langage de modélisation dédié aux applications d'ingénierie système. Il a été conçu comme réponse à l'appel de propositions (RFP) fait en mars 2003 par l'Object Management Group (OMG) [OMG15] pour l'utilisation d'UML en Ingénierie Système [OMG03], il a été proposé par l'OMG et l'International Council on Systems Engineering (INCOSE) et il a été adopté en tant que standard en Mai 2006. SysML est un profil d'UML 2.0 [OMG05] qui réutilise un sous-ensemble de ses diagrammes et ajoute de nouvelles fonctionnalités pour mieux s'adapter aux besoins de l'ingénierie système de sorte qu'il permet la spécification,l'analyse, la conception, la vérification et la validation d'un large éventail de systèmes complexes. Ces systèmes peuvent inclure le logiciel, le matériel, les données, les processus, les personnes et les installations.

SysML comprend donc 9 diagrammes et selon [OMG12] on peut les définir comme suit:

- *Activités*: décrit le comportement du système comme flux de contrôle et de données.
- *Définition de blocs (DDB)*: décrit la structure architecturale du système comme composants avec leur propriétés, opérations et relations.
- *Bloc interne (DBI)*: décrit les structures internes des composants, en ajoutant leurs parties et connecteurs.
- *Paquets*: décrit comment un modèle est organisé en paquets, vues et points de vue.
- *Paramétrique*: décrit les contraintes paramétriques entre les éléments structurels.
- *Exigences*: décrit les exigences du système et leurs relations avec d'autres éléments.
- *Séquences (DS)*: décrit le comportement du système comme interactions entre les composants du système.
- *Machines d'état*: décrit le comportement du système comme états qu'un composant a en réponse à des événements.
- *Cas d'utilisation*: décrit les fonctions du système et leurs acteurs en train de les utiliser.

10.1.3/ Les Automates d'Interface

Les automates d'interface ont été introduits par Alfaro et Henzinger [dAH01] pour modéliser les interfaces dans une approche à composants. Ces automates sont issus des automates Input/Output où il n'est pas nécessaire d'avoir des actions d'entrée activables dans tous les états. Chaque composant est décrit par un seul automate d'interface. L'ensemble des actions est décomposé en trois ensembles : les actions d'entrée, les actions de sortie, et les actions internes. Les actions d'entrée permettent la modélisation des méthodes qui vont être appelées dans un composant, dans ce cas elles représentent les services offerts pour un composant. Elles peuvent aussi modéliser une réception de messages dans un canal de communication. Ces actions sont étiquetées par le caractère "?". Les actions de sortie modélisent les appels des méthodes d'un autre composant. Donc, elles représentent les services requis par un composant. Elles peuvent aussi modéliser la transmission de messages dans un canal de communication. Ces actions sont étiquetées par le caractère "!". Les actions internes sont des opérations activables localement et elles sont étiquetées par le caractère ";".

Definition 21: Automate d'Interface

Un *automate d'interface* A est représenté par le tuple $\langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ tels que :

- S_A *est ensemble fini d'états*;
- $I_A \subseteq S_A$ *est un sous ensemble des états initiaux*;
- Σ_A^I, Σ_A^O *et* Σ_A^H, *représentent, respectivement, les ensembles des actions d'entrée, de sortie et internes. L'ensemble des actions de A est noté par* Σ_A;
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions entre les états*.

Les actions d'entrée et de sortie d'un automate d'interface A sont notées actions externes ($\Sigma_A = \Sigma_A^I \cup \Sigma_A^O$). L'ensemble des actions internes Σ_A^H peut contenir l'action, *epsilon* ϵ, qui symbolise un événement non opérationnel. Nous définissons par $\Sigma_A^I(s)$, $\Sigma_A^O(s)$, $\Sigma_A^H(s)$, respectivement les ensembles des actions d'entrée, de sortie et internes activables à l'état s. $\Sigma_A(s)$ représente l'ensemble des actions activables de l'état s.

La vérification de l'assemblage de deux composants (blocs) s'obtient en vérifiant la compatibilité de leurs automates d'interface. Pour vérifier l'assemblage de deux composants B_1 et B_2, on vérifie s'il existe un environnement pour lequel il est possible d'assembler correctement B_1 et B_2. Cela se traduit par la *composition* de leurs automates d'interface et la vérification si cette dernière n'est pas vide.

Deux automates d'interface A_1 et A_2 sont *composables* si $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_2}^H \cap \Sigma_{A_1} = \emptyset$. $Shared(A_1, A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ est l'ensemble des actions partagées entre A_1 et A_2.

Definition 22: Produit Synchronisé

Soient A_1 et A_2 deux automates d'interface composables. Le *produit synchronisé* $A_1 \otimes A_2$ de A_1 et A_2 est défini par :

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ et $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ si
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

Deux automates d'interface pourraient être incompatibles à cause de l'existence des états illégaux dans leur produit synchronisé. Les états illégaux sont des états à partir lesquels une action de sortie partagée d'un automate ne peut pas être synchronisée avec la même action activée en entrée dans l'autre composant.

Definition 23: Etats Illégaux

Soient deux automates d'interface composables A_1 et A_2, l'ensemble des *états illégaux* $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ est défini par $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2).$ telle que la condition C est satisfaite$\}$

$$C = \begin{pmatrix} a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2) \\ \vee \\ a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1) \end{pmatrix}$$

L'approche des automates d'interface est considérée comme une approche optimiste, car l'atteignabilité des états dans $Illégal(A_1, A_2)$ ne garantit pas l'incompatibilité des A_1 et A_2. En effet, dans cette approche on vérifie l'existence d'un environnement qui fournit les services appropriés au produit $A_1 \otimes A_2$ afin d'éviter les états illégaux. Les états dans lesquels l'environnement peut éviter l'atteignabilité des états illégaux sont appelés états compatibles, et sont définis par l'ensemble $Comp(A_1,A_2)$. Cet ensemble est calculé dans $A_1 \otimes A_2$ en éliminant les états illégaux, les états inatteignables et les états qui conduisant vers des états illégaux en passant par des actions internes ou des actions de sortie.

Definition 24: Composition

La *composition* $A_1 \parallel A_2$ de deux automates A_1 et A_2 est définie par
(i) $S_{A_1 \parallel A_2} = \text{Comp}(A_1, A_2)$,
(ii) $I_{A_1 \parallel A_2} = I_{A_1 \otimes A_2} \cap \text{Comp}(A_1, A_2)$ et
(iii) $\delta_{A_1 \parallel A_2} = \delta_{A_1 \otimes A_2} \cap \text{Comp}(A_1, A_2) \times \Sigma_{A_1 \parallel A_2} \times \text{Comp}(A_1, A_2)$

10.2/ CONTRIBUTIONS

Dans cette section nous présentons un résumé des contributions proposées dans cette thèse. Ces contributions sont orientées à donner une réponse à la problématique présentée en début de chapitre. Nous présentons dans la Figure 10.1 un diagramme représentant nos contributions (numérotées 1, 2 et 3) à la spécification des CBS en SysML.

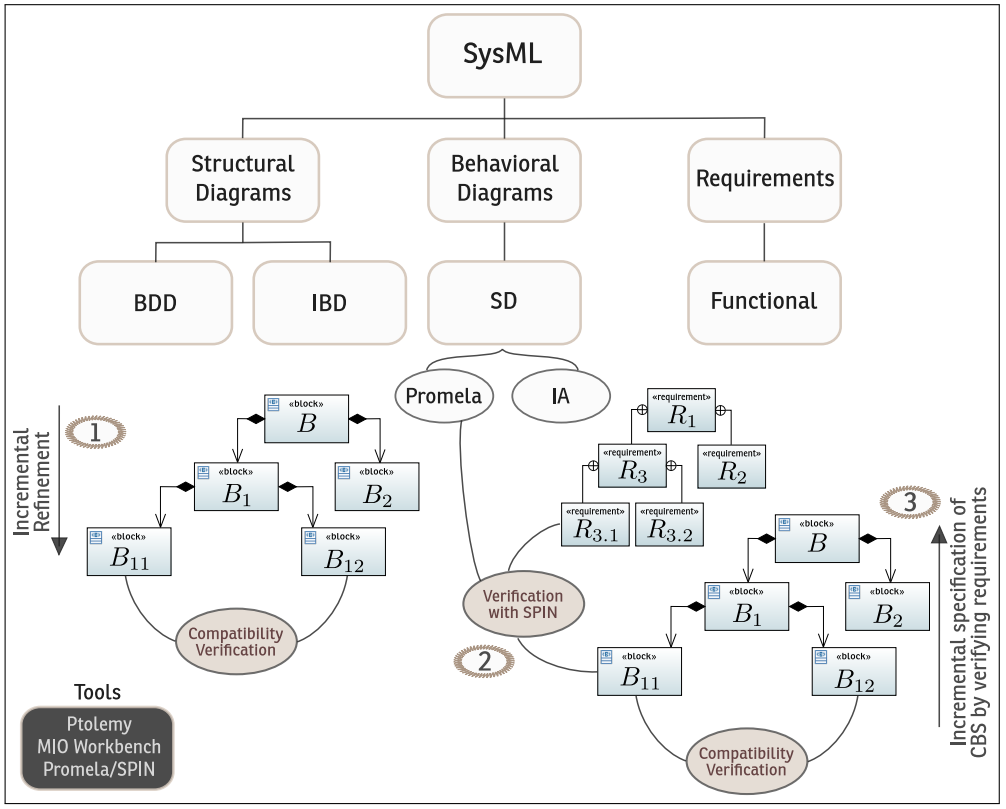

Figure 10.1 – Contributions de la thèse

En premier lieu, nous nous focalisons à la réalisation d'une spécification SysML abstraite en assemblant plusieurs blocs concrets selon un processus de raffinement (voir Contribution 1 à la Figure 10.1). Les comportements des blocs sont décrits par des automates d'interface qui peuvent être obtenus à partir des diagrammes de comportement comme proposé dans [CH11]. Cette approche cherche à formaliser le processus de décomposition, en définissant les relations de raffinement entre blocs, et en se focalisant sur la vérification des aspect structurels et comportementaux des blocs SysML. Dans cette contribution nous exploitons les outils: Ptolemy II pour vérifier la compatibilité entre les composants assemblés, et MIO Workbench pour vérifier le raffinement.

En deuxième lieu, en nous basant sur des diagrammes d'exigences et interfaces des composants spécifiés par SD, nous proposons une approche formelle et méthodologique pour spécifier incrémentalement l'architecture de système qui préserve tous les exigences du système (voir Contribution 3 à la Figure 10.1). De cette manière, nous proposons de traiter, une à une, les exigences atomiques, extraites du diagramme d'exigences (fourni par le concepteur), pour construire une architecture partielle du système, composée de composants atomiques et composites. À chaque étape, nous proposons de sélectionner une exigence atomique du diagramme d'exigences SysML, et choisir un composant d'une bibliothèque de composants qui devrait satisfaire l'exigence choisie. Après nous vérifions si le composant satisfait l'exigence en vérifiant le programme Promela qui spécifie le DS du composant (voir Contribution 2 à la Figure 10.1). Puis, nous vérifions la compatibilité entre les composants choisis dans les étapes précédentes et le nouveau composant et nous vérifions aussi la préservation des exigences traitées aux étapes précédentes. Ce processus finit quand tous les exigences atomiques ont été traitées, ou si les exigences ne sont pas préservées par la composition des composants. Quand le processus finit correctement, nous garantissons la consistance de l'architecture du CBS final qui satisfait tous les exigences du système.

Dans ce context, cette thèse propose de nouvelles contributions :

- L'exploitation du diagramme d'exigences SysML pour spécifier des exigences de CBS [CCM13, CCM14a].
- La spécification d'exigences SysML avec des formules en Logique Temporelle Linéaire (LTL) pour les vérifier sur les composants [CCM14a], grâce à leur DS qui est traduit vers Promela en adoptant l'approche proposée dans [LTM+09].
- La vérification de la compatibilité des composants en exploitant le formalisme des automates d'interface [dAH01], obtenus à partir des DS des composants, grâce à l'approche proposée dans [CH11]. Dans ce travail nous avons adapté l'algorithme de vérification de la compatibilité pour gérer des exigences SysML et pour vérifier aussi sa préservation dans la composition [CCM12a, CCM12b, CCM15].
- La vérification du raffinement du comportement en appliquant la simulation alternée dans les automates d'interface [CCM15].
- La proposition d'une approche incrémentale pour construire des SBC et pour vérifier leurs exigences pour éviter le problème de l'explosion combinatoire du nombre d'états des composants vérifiés. En fait, la vérification des exigences est réalisée sur des composants élémentaires que généralement son petits, de tel sorte que nous évitons la vérification sur des composants composites grâce à la préservation des exigences dans la composition. Cette contribution permet d'obtenir l'architecture SBC qui satisfait tous les exigences. En effet, cette architecture est construite incrémentalement et aussi validée incrémentalement par rapport aux exigences SysML à chaque étape [CCM13, CCM14a].

10.2.1/ Raffinement Incrémental d'une Architecture SBC

L'approche présenté dans cette section cherche proposer une méthode pour formaliser et vérifier la décomposition d'un bloc SysML dans un processus de raffinement. Nous avons présenté la procédure générale à la Figure 6.2.

Notre approche s'oriente à proposer une méthode formelle pour construire un bloc composite SysML à partir d'un ensemble des blocs élémentaires réutilisables. Ainsi, à partir d'un bloc composite abstrait, tel que son structure est modélisé par DDB et DBI, et son comportement par DS, nous proposons une approche qui décide si une composition d'un ensemble de blocs élémentaires réutilisables satisfont les exigences structurelles et comportementales, par rapport au bloc composite. Ainsi, nous proposons de vérifier la décomposition correcte du bloc composite en un ensemble de composants élémentaires sélectionnés. Nous accomplissons cette décomposition en définissant une relation de *raffinement par décomposition* entre le bloc abstrait et ses sous-blocs. À différents niveaux, en commençant à partir du premier bloc abstrait jusqu'à obtenir les blocs élémentaires. Notre relation de raffinement dépende de la vérification de deux relations entre le bloc composite et ses sous-blocs :

- relation de raffinement structurel : selon l'illustration de la Figure 6.2, cette relation se maintient entre le bloc composite B et les sous-blocs $B1$ et $B2$ si $B1$ et $B2$ sont compatibles et les service requis et demandés de $B1$ et $B2$ sont consistants avec ceux de B.

- relation de raffinement comportemental : cette relation se maintient si le comportement de la composition de $B1$ et $B2$ est un raffinement du comportement de B.

Definition 25: Raffinement par une décomposition de blocs SysML

Soit B un bloc abstrait décrit par un DDB et un DBI, ces deux diagrammes spécifient l'architecture du système. Soit $B_1, ..., B_n$ l'ensemble de blocs composant B selon son DDB, alors $B_1, ..., B_n$ raffinent par décomposition B si:

- $B_1, ..., B_n$ raffine structurellement B,
- $B_1, ..., B_n$ raffine comportementalement B,

Donc, pour vérifier le raffinement entre un bloc et ses sous-blocs, nous vérifions les conditions de consistance et compatibilité pour le raffinement structurel, et de simulation alternée pour le raffinement comportemental (voir Figure 6.3). Pour réussir cette vérification, nous avons besoin de spécifier par diagramme de séquences la description du comportement du bloc abstrait et ses sous-blocs composants, et puis, en exploitant l'approche proposé dans [CH11], nous associons un automate d'interface à chaque DS. Ces automates sont exploités pour vérifier la compatibilité entre les sous-blocs en utilisant l'outil Ptolemy [LX04], qui génère l'automate de la composition de deux automate d'interface en entrée. Ensuite nous vérifions le raffinement comportementale à travers l'outil MIO Workbench [BMSH10], qui vérifie si une spécification de comportement est raffinée par une implémentation en utilisant un automate Modal Input/Output (MIO) comme donnée d'entrée.

Une fois que nous avons vérifié le raffinement structurel et comportemental par deux ou plus blocs qui décompose un bloc abstrait de niveau supérieur, nous avons aussi vérifié que ce bloc parent peut effectivement être substitué par ses entités composants.

Ce processus de vérification peut être appliqué incrémentalement depuis les blocs abstraits de plus haut niveau jusqu'aux blocs de plus bas niveau pour valider l'architecture finale du SBC obtenu à partir du bloc abstrait initial.

Table 10.1 – Règles de correspondance des concepts basiques entre DS et Promela

Élément DS	**Élément Promela**	**Déclaration Promela**
Ligne de vie	Processus	`proctype{...}`
Message	Message	`mtype{m1,...,mn}`
Connecteur	Chaîne de communication pour chaque flèche de message	`chan chanName = [1] of {mtype}`
Envoi et réception d'événements	Envoi et réception d'opérations	Send ⇒ `ab!m`, Receive ⇒ `ab?m`

Fragment combiné Alt	condition if	`if`
		`::(guard)->ab_p?p;`
		`:: else -> ab_q?q;`
		`fi;`

Fragment combiné Loop	opérateur do	`do`
		`::ab_p?p;`
		`od`

10.2.2/ Vérification Formelle d'Exigences SysML

Pour vérifier si un composant satisfait une exigence donnée, nous proposons d'utiliser l'ensemble Promela/SPIN. Nous les choisissons car ils fournissent d'importants concepts pour implémenter des DS : primitives d'envoi et réception des messages, composition parallèle et asynchrone de processus concurrents, et chaînes de communication. Notre adaptation de cette approche proposé par V. Lima *et al.* [LTM+09] concerne un type particulier de diagramme de séquence que nous exploitons pour spécifier le comportement de blocs.

Nous proposons d'utiliser un type particulier de DS avec seulement deux lignes de vie, une pour le bloc et une pour son environnement Ainsi, un DS peut par la suite être traduit en automate d'interface comme montré dans [CH11]. Dans ce diagramme de séquence les messages échangés seront les services offerts comme appels de l'environnement et les services requis comme appels à l'environnement La principale avantage d'utiliser les DS pour la vérification est que l'on peut vérifier des propriétés temporelles sur eux. Les messages suivent un ordre séquentiel que nous pouvons tracer pour détecter blocages ou exécution de chemins. Les Figures 7.2 et 7.4 montrent les interfaces, à travers des DS, pour deux blocs : *sensors* et *ACU*. Ils sont des blocs pris d'une à partir d'une bibliothèque de blocs. Dans ces diagrammes nous voyons qu'il y a seulement deux lignes de vie et que des messages sont envoyés par/reçus de l'environnement.

Ensuite, dans les Figures 7.3 et 7.5 nous montrons partiellement la représentation Promela pour ces deux blocs respectivement (le code complet est présenté dans les Listings 7.1 et 7.2). Ce code est obtenu en appliquant les règles de transformation listées dans la Table 10.1.

Dans les deux diagrammes nous pouvons voir qu'il y a deux lignes de vie traduites en tant que processus dans le code Promela, un processus pour le bloc et un autre pour son environnement. Les deux processus commence au même moment grâce à un appel atomique dans le processus main : `init`. Nous pouvons aussi voir que les fragments combinés de boucle `loop` sont traduits en tant que déclarations `do`, et le fragment combiné d'alternative `alt` est traduit comme une déclaration `if`. Pour pouvoir parcourir tous les chemins d'exécution, il est nécessaire de définir les valeurs possibles pour les variables affectées, comme c'est le cas des valeurs `deceleration` qui est assignée au moment `init` dedans une déclaration `if`, de cette manière SPIN va choisir de façon non-déterministe quelle valeur sera utilisée pour simuler le système.

Une fois que le DS est traduit, le composant peut être simulé comme un système SPIN. Pour pouvoir vérifier si le composant satisfait une propriété LTL, V. Lima propose d'utiliser une série de drapeaux pour garder une trace de *qui est en train d'envoyer/recevoir quel message à/de qui* à tout moment de l'exécution. Néanmoins, dans notre approche nous vérifions des propriétés sur des composants indépendants avec seulement deux lignes de vie dans son DS, une ligne pour le composant sélectionné et l'autre pour son environnement. Donc, nous n'utilisons pas le drapeau lié à *à/de qui* un message est envoyé car il sera toujours l'autre ligne de vie. Ces drapeaux sont mis à jour au même temps à chaque événement *envoi/réception* en utilisant une déclaration `d_step`.

Après la définition des drapeaux pour garder la trace de l'exécution du système, des propriétés LTL peuvent être écrites comme expressions booléennes sur les drapeaux. Dans notre approche, nous proposons de traduire des exigences SysML à propriétés LTL en respectant le formalisme des drapeaux (pour plus des détailles, voir Chapitre 7). Ces propriétés sont par la suite vérifiées sur leur modèle Promela correspondant en utilisant le model-checker SPIN, qui indiquera si les blocs satisfont les propriétés. Une fois qu'un bloc correspondant est trouvé pour une exigence, nous continuons avec l'exigence suivante pour commencer à construire l'architecture du système.

10.2.3/ Spécification Incrémentale d'une Architecture SBC

Nous proposons une approche pour construire un SBC et spécifier son architecture directement à partir de ses exigences SysML. Notre objectif est d'obtenir une architecture consistante en respectant toutes les exigences spécifiées. Pour spécifier cette architecture, l'architecte logiciel exploite une bibliothèque de composants réutilisables (ou blocs). Ces composants sont considérés comme boîtes noires et ils sont décrits seulement par ses interfaces, spécifiés par DS. Ainsi, nous proposons de spécifier des exigences de SBC avec un diagramme d'exigences SysML, puis analyser ce diagramme pour associer une à une ses exigences atomiques (celles qui ne peuvent pas être décomposées) à des composants logiciels qui les satisfont. La satisfiabilité est évalué en effectuant une étape de vérification formelle avec un model-checker. Chaque composant vérifié est évalué pour compatibilité par rapport aux autres composants dans la composition et par la suite ajouté à l'architecture partielle qui doit préserver les exigences atomiques.

Dans notre approche, un SBC est spécifié avec un diagramme d'exigences SysML qui présente les exigences fonctionnelles, et des interfaces des composants qui décrivent leurs comportements à travers de DS.

Les étapes principales de notre approche sont présentées dans la Figure 10.2, elles peuvent être décrites comme suit :

1. Commencer par analyser le diagramme d'exigences SysML pour obtenir les exigences atomiques car elles sont plus précises et il sera plus facile de trouver des composants qui vont les satisfaire (voir Section 8.3).

2. Soit R_i la première exigence atomique, soit C_i un composant dans une bibliothèque de composants, décrit par le DS SD_i. Spécifier R_i avec la formule LTL F_i et traduire SD_i au code Promela PRO_i, puis vérifier que C_i satisfait R_i en vérifiant que PRO_i satisfait F_i avec le model-checker SPIN (voir Section 7.3). La sélection du composant C_i dans la bibliothèque est faite par l'architecte logiciel. Néanmoins, il est possible de guider cette sélection (ou l'automatiser) car R_i est une exigence fonctionnelle et

décrit des contraintes dans les services offert et requis (actions d'entrée/sortie). Ces services sont aussi décrits dans les interfaces des composants. Alors, il est facile d'extraire ces services de R_i et de faire une correspondance avec ceux décrit dans les interfaces. Si cette étape retourne *false*, alors C_i ne satisfait pas R_i et donc l'on doit obtenir le composant approprié dans une autre bibliothèque, ou le développer dès zéro.

3. Soit A_i l'automate d'interface qui décrit le protocol du composant obtenu à partir du DS SD_i (voir Section 8.4).
4. Identifier les actions d'entrée et sortie en A_i liées à R_i (voir Section 8.4).
5. Répéter jusqu'à avoir traité tous les exigences :
   (a) Soit R_{i+1} l'exigence atomique suivante, *connectée* à R_i (voir Définition 19), soit C_{i+1} un composant satisfaisant R_{i+1}, grâce à la formule LTL F_{i+1} et le code Promela PRO_{i+1}. Soit A_{i+1} l'automate d'interface décrivant le protocole du composant.
   (b) Identifier l'ensemble des actions d'entrée et sortie en A_{i+1} liées à R_{i+1}.
   (c) Vérifier que C_i et C_{i+1} sont compatibles grâce à leurs automates d'interface, en vérifiant que $A_i \parallel A_{i+1} \neq \emptyset$ (voir Section 4.2).
   (d) Vérifier que les exigences R_i et R_{i+1} sont préservées par la composition, c'est-à-dire qu'elle sont satisfaites par le composant composite $C = C_i \parallel C_{i+1}$ (voir Section 8.4).
   (e) Définir l'architecture consistante partielle du système pour le composant composite $C = C_i \parallel C_{i+1}$, en accord avec la Définition 20.
   (f) Soit $C_i = C_i \parallel C_{i+1}$, $A_i = A_i \parallel A_{i+1}$, et $R_i = \{R_i, R_{i+1}\}$.
6. Fin Répéter

Selon les étapes principales de notre approche, nous validons l'architecture finale pour notre SBC, une fois que tous les exigences atomiques ont été analysées sans problèmes de compatibilité des composants et/ou préservation des exigences.

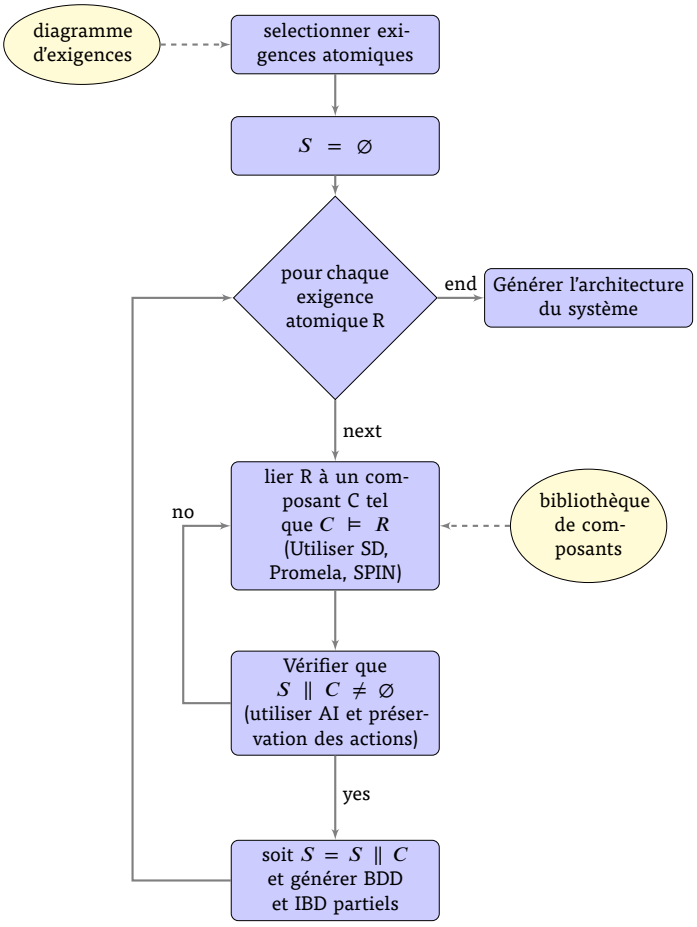

Figure 10.2 – Approche proposé pour la génération d'une architecture SBC à partir d'exigences SysML

10.3/ CONCLUSIONS

La Figure 10.3 montre l'approche globale des contributions de cette thèse et ses perspectives. Dans cette thèse nous avons présentées trois contributions basées sur une approche

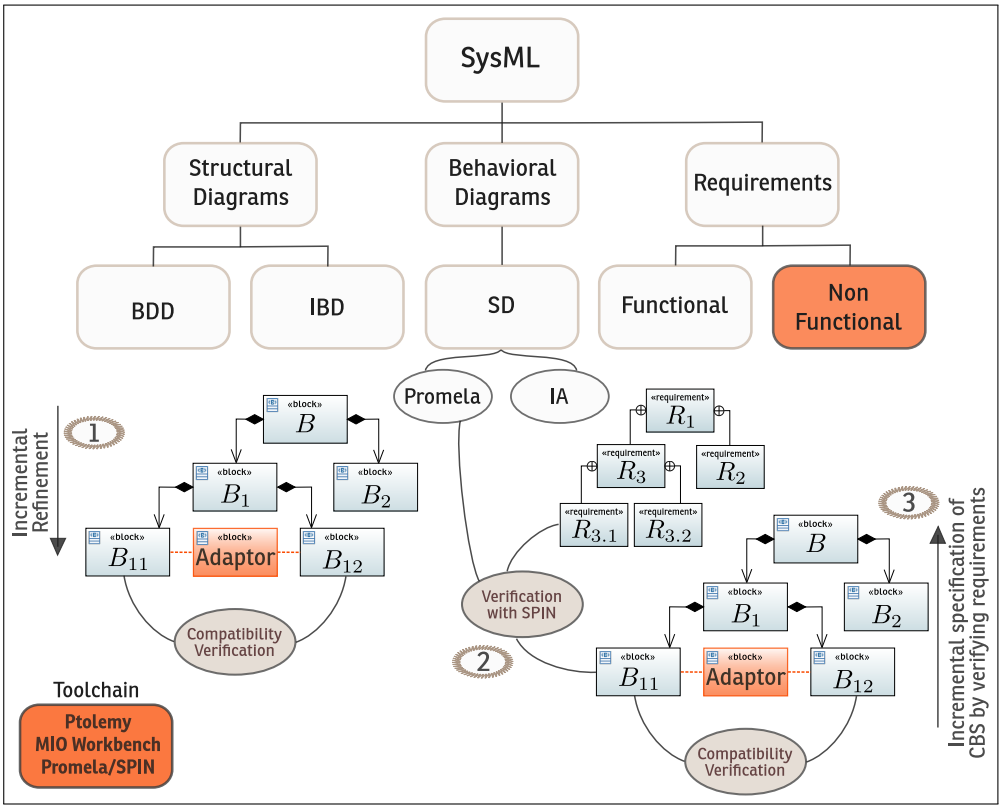

Figure 10.3 – Perspectives de la thèse

incrémentale :

1. La première contribution orientée à formaliser le processus de décomposition, en définissant une relation de raffinement entre un bloc abstrait et ses sous-blocs. Il consistait à exploiter la description d'architectures en langage SysML quand un système est décrit par des diagrammes structurels et comportementaux. Le raffinement en SysML est un concept essentiel et il est basé sur le développement d'une démarche à partir d'un niveau abstrait vers de niveaux plus détaillés, qui peuvent aboutir en son implémentation. Notre raffinement assure une substitution incrémentale d'un bloc abstrait d'une spécification par une composition des blocs en préservant ses propriétés structurelles et comportementales.

   Pour vérifier le raffinement structurelle spécifié en diagrammes DDB et DBI, nous avons vérifié d'abord si le sous-blocs étaient consistants par rapport à la spécification du bloc abstrait et par la suite nous avons vérifié s'ils étaient compatibles entre eux en nous appuyant sur le modèle des automates d'interface de d'Alfaro et al. Les automates d'interface ont été obtenus à partir du DS SysML de chaque sous-bloc et puis vérifiés en compatibilité avec l'outil *Ptolemy*.

   Pour vérifier le raffinement structurelle, nous avons appliquée l'approche de la simulation alternée pour les automates d'interface pour vérifier si la composition de

l'ensemble de sous-blocs simulait le comportement attendu dans le bloc abstrait. Pour faciliter la tâche de vérification de la simulation alternée nous avons proposé d'utiliser le module *Observational Modal Refinement* de l'outil *MIO Workbench*.

2. La deuxième contribution a été orientée sur la vérification des propriétés de notre système. Nous nous avons inspiré des travaux proposés par V. Lima et al. Cette technique propose de générer des modèles basés en Promela à partir des interactions UML décrites en Diagrammes des Séquences (DS) et utilise l'outil de model-checking SPIN pour simuler et vérifier des propriétés écrites en Logique Temporelle Linéaire (LTL).

   Notre adaptation de cette approche s'est orientée à un type particulière de DS que nous exploitons pour spécifier le comportement d'un bloc. Nous avons choisi Promela pour décrire des spécifications de DS et des propriétés LTL pour décrire des exigences fonctionnelles. Pour la suite, nous avons utilisé le model-checker SPIN pour vérifier ces propriétés. Nous avons choisi cet environnement d'implémentation car c'est un outil de vérification très répandu et permet d'une manière relativement facile de spécifier et implémenter des concepts des DS comme l'envoi et réception de primitives et la composition parallèle et asynchrone.

3. La troisième contribution propose une approche pour décrire la relation entre les exigences du système et la spécification d'une architecture CBS en SysML. L'objectif était de proposer une méthodologie pour construire incrémentalement une architecture de système consistante que formellement satisfait toutes les exigences du système. Nous avons utilisés des SD pour décrire les comportements des blocs et DDB et DBI pour spécifier l'architecture du système. La construction proposé extrait les exigences atomiques d'un diagramme d'exigences et les traite un par un pour construire le système final.

   Nous obtenons donc une architecture partielle du système, composé de blocs élémentaires et blocs composites. À chaque étape, nous sélectionnons une exigence atomique à partir d'un diagramme d'exigences SysML, et nous choisissons un bloc d'une bibliothèque des blocs qui devrait satisfaire l'exigence choisie. Puis, nous vérifions si le bloc satisfait l'exigences grâce aux propriétés LTL spécifiés par l'exigence et la spécification Promela qui décrit le comportement du bloc à partir du DS.

   Ensuite, nous vérifions la compatibilité entre le bloc sélectionné et ceux sélectionnés aux étapes précédentes. Le processus finit quand toutes les exigences atomiques ont été traitées ou quand on détecte des incompatibilités entre les blocs, ou si les exigences ne sont pas conservées par le bloc composite formé. De cette manière nous garantissons la consistance de l'architecture du système final qui satisfait toutes les exigences du système.

10.4/ PERSPECTIVES

Les travaux de cette thèse répond à une question principale : comment introduire la vérification formelle dans des spécifications informelles SysML pendant le processus de développement des SBC ?

Dans ce but, nos contributions ont introduit quelques solutions pour construire de SBC consistants, ainsi nous avons établi des relations entre le raffinement, concepts de blocs

SysML et caractéristiques des SBC. Ce travail peut continuer à être enrichi et dans ce contexte nous envisageons d'autres perspectives de travail comme :

Chaîne d'outils pour la vérification Développer une chaîne d'outils pour aider le concepteur dans la vérification automatique de la relation de raffinement entre les blocs abstraits et ses sous-blocs et aussi la vérification des exigences SysML dans les blocs pour décider la validité de l'architecture CBS en SysML. Cette chaîne d'outils serait composé d'outils qui permettrait (1) vérifier les conditions de consistance entre les blocs et utiliser l'outil Ptolemy pour la vérification de la compatibilité, (2) traduire directement des DS en programmes Promela et (3) vérifier des propriétés LTL avec le model-checker SPIN. Quelques exemples significatifs pourrait être expérimentés pour évaluer l'approche proposée.

Combiner vérification et simulation pour la validation de propriétés non fonctionnelles Dans les applications fiables, il es important de spécifier une architecture de système en accord avec des spécifications d'exigences fonctionnelles et non-fonctionnelles. Dans ce but, nous proposons une approche pour spécifier l'architecture du système directement en SysML à partir des exigences fonctionnelles SysML. Ainsi, pour des travaux futurs, il serait intéressant d'adresser le problème de validation pour les exigences non-fonctionnelles et utiliser des techniques de simulation.

Adapter et générer des adaptateurs pour les blocs incompatibles Le problème d'adapter des blocs est crucial dans le développement de SBC en réutilisant des blocs. L'adaptation consiste à générer automatiquement, quand c'est possible, un bloc adaptateur entre blocs incompatibles pou assurer une interaction fiable. L'idée d'un travail futur est de générer une entité capable d'assurer l'interaction entre deux blocs incompatibles quand les conditions de consistance et compatibilité ne sont pas validées.

Bibliography

[AAA05] Pascal André, Gilles Ardourel, and Christian Attiogbé. Behavioural Verification of Service Composition. In *First International Workshop on Engineering Service Compositions (WESC'05)*, pages 77–84, Amsterdam, The Netherlands, dec 2005. IBM Research Report RC23821.

[AAA06] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. *Proceedings of the 5th International Symposium on Software Composition*, 4089:18–33, 2006.

[AAA07] Pascal André, Gilles Ardourel, and Christian Attiogbé. Protocoles d'Utilisation de Composants: Spécification et Analyse en Kmelia. *Proceedings of the 13th Conférence Francophone de Languages et Modèles à Objets*, pages 19–34, 2007.

[AaDB+02] Yamine Ait-ameur, Bruno D'Ausbourg, Frédéric Boniol, Rémi Delmas, and Virgine Wiels. A component based methodology for description of complex systems, an application to avionics systems. In *3rd European Systems Engineering Conference (EuSEC'2002)*, Toulouse, France, 2002.

[AdAD+06] B.Thomas Adler, Luca de Alfaro, Leandro Dias Da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A Tool for Interface Compatibility and Composition. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62. Springer Berlin Heidelberg, 2006.

[Adm98] National Highway Traffic Safety Administration. Federal Motor Vehicle Safety Standards. http://www.nhtsa.gov/cars/rules/rulings/AAirBagSNPRM/, 1998.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, jul 1997.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin Heidelberg, 1998.

[ALPC06] PC Attie, DH Lorenz, Aleksandra Portnova, and Hana Chockler. Behavioral Compatibility Without State Explosion: Design and Verification of a Component-Based Elevator Control System. In I. Gorton et Al, editor, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, pages 33–49. Springer Berlin Heidelberg, 2006.

[BCK03] Len. Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, volume 2nd. Addison-Wesley, Boston, MA, USA, second edition, 2003.

[BDD+97] P Bertrand, Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde. GRAIL/KAOS : An Environment for Goal-Driven Requirements Engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 612—-613. ACM, 1997.

[BDL05] Olivier Barais, Laurence Duchien, and Anne-Françoise Le Meur. A Framework to Specify Incremental Software Architecture Transformations. *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 62–69, 2005.

[Ben87] John K. Bennett. The design and implementation of distributed Smalltalk. *ACM SIGPLAN Notices*, 22(12):318, 1987.

[BLL+05] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California, Berkeley, 2005.

[BMC+12] Erwan Bousse, David Mentré, Benoît Combemal, Benoît Baudry, and Takaya Katsuragi. Aligning SysML with the B method to Provide V & V for Systems Engineering. In *Model-Driven Engineering, Verification, and Validation (MoDeVVa 2012)*, Innsbruck, Austria, sep 2012.

[BMFN01] Jo Barnes, Andrew Morris, Brian Fildes, and S.V. Newstead. Airbag effectiveness in real world crashes. *Road Safety Research, Policing, Education Conference*, 2001.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On Weak Modal Compatibility, Refinement, and the MIO Workbench. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2010.

[BOR04] Stefen Becker, Sven Overhage, and Ralf Reussner. Classifying Software Component Interoperability Errors to Support Component Adaption. In Crnkovic Ivica, Stafford Judith, Schmidt Heinz, and Wallnau Kurt, editors, *Component-Based Software Engineering*, pages 68–83. Springer Berlin Heidelberg, 2004.

[BRV04] Bernard. Berthomieu, Pierre-Oliver Ribet, and François Vernadat. The tool TINA – Construction of abstract state spaces for Petri nets and Time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[BSBM05] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible? *Technologies for EServices*, 3324:15–28, 2005.

[BW97] Martin Büchi and Wolfgang Weck. A Plea for Grey-Box Components. Technical report, Turku Centre for Computer Science, Turku, 1997.

[CCM12a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Formalizing and verifying compatibility and consistency of SysML blocks. In *ACM SIGSOFT Software Engineering Notes (UML-FM 2012)*, volume 37, pages 1–8, Paris, France, 2012. ACM.

[CCM12b] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Vérification de la consistence et de la compatibilité entre blocs SysML. In *Conférence en Architectures Logicielles (CAL 2012)*, Montpellier, France, 2012.

[CCM13] Samir Chouali, Oscar Carrillo, and Hassan Mountassir. Specifying System Architecture from SysML Requirements and Component Interfaces. In *Software Architecture (ECSA 2013)*, pages 348–352, Montpellier, France, 2013.

[CCM14a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Incremental Modeling of System Architecture Satisfying SysML Functional Requirements. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software (FACS 2013)*, Lecture Notes in Computer Science, pages 79–99. Springer International Publishing, Nanchang, China, 2014.

[CCM14b] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Modélisation Incrémentale d'une Architecture de Système Satisfaisant des Exigences Fonctionnelles SysML. In *Conférence en Architectures Logicielles (CAL 2014)*, Paris, France, 2014.

[CCM15] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Verification of a SysML Block Decomposition in a Refinement Process. In *Under consideration for publication in Software & Systems Modeling (SOSYM)*. Springer Berlin Heidelberg, 2015.

[CFTV00] Carlos Canal, Lidia Fuentes, José M. Troya, and Antonio Vallecillo. Extending CORBA Interfaces with Pi-Calculus for Protocol Compatibility. In *Proceedings of the 33th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, pages 208–225, St. Malo, France, 2000. IEEE.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CH11] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on SysML and interface automata. *Innovations in Systems and Software Engineering*, 7(4):265–274, oct 2011.

[CK13] Josep Carmona and Jetty Kleijn. Compatibility in a multi-component environment. *Theoretical Computer Science*, 484:1–15, 2013.

[CL02] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc, Norwood, MA, 2002.

[CLR+09] Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and Verification in Component-Based Model-Driven Design. *Science of Computer Programming*, 74(4):168–196, feb 2009.

[dAFL06] Luca de Alfaro, Marco Faella, and Axel Legay. An introduction to the tool Ticc. In *Proc. of Workshop on Trustworthy Software*, pages 1–32, 2006.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-Based Design. In *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 83–104. Springer Netherlands, 2005.

[Dav10] Brett Davis. Mercedes-Benz celebrates 30 years of using airbag technology. http://www.caradvice.com.au/84716/mercedes-benz-celebrates-30-yearsof-using-airbag-technology/, 2010.

[Dro03] R. Geoff Dromey. From requirements to design: formalizing the key steps. *First International Conference on Software Engineering and Formal Methods*, pages 2–11, 2003.

[Dro07] R Geoff Dromey. Engineering Large-Scale Software-Intensive Systems. In *Software Engineering Conference, ASWEC 2007.*, pages 4–6. IEEE, 2007.

[DW98] Desmond F. D'souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[EB06] Jean-Paul Etienne and Samia Bouzefrane. Vers une approche par composants pour la modélisation d'applications temps réel. In *MOSIM'06 Conférence Francophone de Modélisation et Simulation*, volume 6, pages 1–10, Rabat, jan 2006. Lavoisier.

[Ell97] Clarence Ellis. Team Automata for groupware systems. *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 415–424, 1997.

[FGK+96] Jean Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP a protocol validation and verification toolbox. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Berlin Heidelberg, 1996.

[GDTS10] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: A UML2 tool for domain-specific language modeling. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6100:361–368, 2010.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[GMP11] Roy Grønmo and Birger Møller-Pedersen. From UML 2 Sequence Diagrams to State Machines by Graph Transformation. *The Journal of Object Technology*, 10:8:1, 2011.

[HK07] Rolf Hennicker and Alexander Knapp. Activity-Driven Synthesis of State Machines. *Fundamental Approaches to Software Engineering*, pages 87–101, 2007.

[HL07] Zbigniew Huzarr and Grzegorz Loniewski. Deriving prototypes from UML sequence diagrams. In *Proceedings of the 10th International Conference on Information System Implementation and Modeling*, Hradec nad Moravicí, Czech Republic, apr 2007.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, New Jersey, 1991.

[Hol97] Gerard J Holzmann. The Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98 Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, 1998. Springer-Verlag.

[JMO10] Jacques Julliand, Hassan Mountassir, and Emilie Oudot. Incremental Verification of Component-Based Timed Systems. *International Journal of Identification Modelling and Control special issue on Formal Modeling and Verification of Critical Systems*, 1(3/4):19, 2010.

[LdOFV07] Marcos V. Linhares, Rômulo S. de Oliveira, Jean-Marie Farines, and François Vernadat. Introducing the modeling and verification process in SysML. In *IEEE International Conference on Emerging Technologies and Factory Automation. ETFA'2007*, pages 344–351. IEEE, 2007.

[LLH01] Xiaoshan Li, Zhiming Liu, and Jifeng He. Formal and Use-Case Driven Requirement Analysis in UML. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 215–224, Washington, DC, USA, 2001. IEEE Computer Society.

[LNRT10] Kung-Kiu Lau, Azlin Nordin, Tauseef Rana, and Faris Taweel. Constructing Component-based Systems Directly from Requirements using Incremental Composition. In *Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 85–93. IEEE, 2010.

[LNW07] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In Rocco Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin Heidelberg, 2007.

[Lot88] ISO Lotos. A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807. *International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva*, page 142, 1988.

[LSM+10] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouet. A first attempt to combine SysML requirements diagrams and B. *Innovations in Systems and Software Engineering*, 6(1):47–54, dec 2010.

[LST+11] Vincent Lorenzo, Rémi Schnekenburger, Yann Tanguy, Patrick Tessier, and Sébastien Gerard. L'outil de modélisation graphique MDT Papyrus: État actuel et perspectives. *Génie logiciel*, (97), 2011.

[LTE+09] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML: an open source toolset for MDA. *5th ECMDA-FA: Proceedings of the Tools and Consultancy Track*, pages 1–4, 2009.

[LTM+09] Vitor Lima, Chamseddine Talhi Talhi, DDjedjiga Mouheb, Mourad Debbabi, Lingyu Wang, and Makan Pourzandi. Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. *Electronic Notes in Theoretical Computer Science*, 254:143–160, oct 2009.

[LX04] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, 2004.

[Mat11] Abderrahman Matoussi. *Construction de spécifications formelles abstraites dirigée par les buts*. PhD thesis, Université Paris-Est, 2011.

[McH07] Ciaran McHale. *Corba Explained Simply*. Ciaran McHale, 2007.

[MHH+09] J.D. Meier, David Hill, Alex Homer, Jason Taylor, Prashant Bansode, Lonnie Wall, Rob Boucher Jr., and Akshay Bogawat. *Microsoft Application Architecture Guide: Patterns & Practices*. Microsoft Press, 2nd edition, 2009.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour Analysis of Software Architectures. In *Software Architecture*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Springer.

[Mou11] Sebti Mouelhi. *Contributions à la Vérification de la Sûreté de l'Assemblage et à l'Adaptation de Composants Réutilisables*. PhD thesis, Université de Franche-Comté, Besançon, 2011.

[MRR03a] Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. A Behavioral Model of Component Frameworks. Technical report, INRIA, 2003.

[MRR03b] Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. Behavioral substitutability in component frameworks: A formal approach. In *Proceedings of ESEC*, volume 3, pages 22–28, 2003.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[OMG03] OMG. UML for Systems Engineering. Request for Proposal. http://syseng.omg.org/UML_for_SE_RFP.htm, 2003.

[OMG05] OMG. Unified Modeling Language Specification, UML 2.0. http://www.omg.org/spec/UML/2.0/, jul 2005.

[OMG12] OMG. Systems Modeling Language (SysML) Version 1.3. http://www.omg.org/spec/SysML/1.3/, 2012.

[OMG15] OMG. OMG Organization. http://www.omg.org, 2015.

[PEML10] Jean-François Pétin, Dominique Evrot, Gérard Morel, and Pascal Lamy. Combining SysML and formal methods for safety requirements verification. In *22nd International Conference on Software & Systems Engineering and their Applications*, Paris, France, 2010.

[Pet07] Jean-François Petin. *Méthodes et modèles pour un processus sûr d'automatisation*. PhD thesis, Université Henri Poincaré - Nancy I, Nancy, 2007.

[Pla02] David S. Platt. *Introducing Microsoft .Net*. Microsoft Press, Washington, second edition, 2002.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57. IEEE, sep 1977.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RF06] Óscar R. Ribeiro and João M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 237–256, 2006.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SP97] Clemens Szyperski and Cuno Pfister. Workshop on Component-Oriented Programming, summary. In *Special Issues in Object-Oriented Programming - ECOOP '96 Workshop Reader*, Heidelberg, 1997. dpunkt Verlat.

[SV08a] Michel Dos Santos Soares and Jos Vrancken. A proposed extension to the SysML requirements diagram. *Proceedings of the IASTED International Conference on Software Engineering*, pages 220–225, 2008.

[SV08b] Michel dos Santos Soares and Jos Vrancken. Model-driven user requirements specification using SysML. *Journal of Software*, 3(6):57–68, 2008.

[Szy02] Clemens Szyperski. *Component Software: beyond Object-Oriented Programming*. Addison-Wesley Professional, New York, 2 edition, 2002.

[TB85] P. P. Tanner and W. A. S. Buxton. Some Issues in Future User Interface Management System (UIMS) Development. In *User Interface Management Systems*, pages 67–79. Springer-Verlag, 1985.

[TS11] Nara Sueina Teixeira and Ricardo Pereira E Silva. Compatibility Evaluation of Components Specified in UML. *30th International Conference of the Chilean Computer Science Society*, pages 90–99, nov 2011.

[vL03] Axel van Lamsweerde. From system goals to software architecture. *Formal Methods for Software Architectures*, pages 25–43, 2003.

[VT04] Simona Vasilache and Jiro Tanaka. Synthesis of State Machines from Multiple Interrelated Scenarios Using Dependency Diagrams. In *Proceedings of the 8th World Multiconference on Systemics, Cybernetics and Informatics*, pages 49—-54, Orlando, Florida, USA, 2004.

[VVR06] Antonio Vallecillo, Vasco T Vasconcelos, and António Ravara. Typing the Behavior of Software Components using Session Types. *Fundamenta Informaticae*, 72(4):583–598, 2006.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[ZVB+08] Barbora Zimmerova, Pavlína Vařeková, Nikola Beneš, Ivana Černá, Luboš Brim, and Jiří Sochor. Component-Interaction Automata Approach (CoIn). *The Common Component Modeling Example*, 5153(102):146–176, 2008.

Document generated with XeLaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

Abstract:

The work presented in this thesis is a contribution to the specification and verification of Component-Based Systems (CBS) modeled in SysML. CBS are widely used on the industrial field, and they are built by assembling various reusable components, allowing developing complex systems at lower cost.
To ease the communication between the various stakeholders in a CBS development project, one of the widely used modeling languages is SysML, which besides allowing modeling of structure and behavior, it has capabilities to model requirements. It offers a standard for modeling, specifying and documenting systems, wherein it is possible to develop a system, starting from an abstract level, to more detailed levels that may lead to an implementation.
In this context, we have dealt mainly two issues. The first one concerns the development by refinement of a CBS, which is described only by its SysML interfaces and behavior protocols. Our contribution allows the designer of CBS to formally ensure that a composition of a set of elementary and reusable components refines an abstract specification of a CBS. In this contribution, we use the tools: Ptolemy for the verification of compatibility of the assembled components and MIO Workbench for refinement verification.
The second one concerns the difficulty to decide what to build and how to build it, considering only system requirements and reusable components. Therefore, the question that arises is: how to specify a CBS architecture, which satisfies all system requirements? We propose a formal and incremental verification approach based on SysML models and interface automata to guide, by the requirements, the CBS designer to define a coherent system architecture that satisfies all proposed SysML requirements. In this approach we use the SPIN model-checker and LTL properties to specify and verify requirements.

Keywords: Modeling, SysML specifications, CBS architecture, Refinement, Compatibility, Requirements, LTL properties, Promela/SPIN, Ptolemy, MIO Workbench

Résumé :

Le travail présenté dans cette thèse est une contribution à la spécification et la vérification des Systèmes à Base de Composants (SBC) modélisé avec le langage SysML. Les SBC sont largement utilisés dans le domaine industriel et ils sont construits en assemblant différents composants réutilisables, permettant ainsi le développement de systèmes complexes en réduisant leur coût de développement. Malgré le succès de l'utilisation des SBC, leur conception est une étape de plus en plus complexe qui nécessite la mise en œuvre d'approches plus rigoureuses.
Pour faciliter la communication entre les différentes parties impliquées dans le développement d'un SBC, un des langages largement utilisé est SysML, qui permet de modéliser, en plus de la structure et le comportement du système, aussi ses exigences. Il offre un standard de modélisation, spécification et documentation de systèmes, dans lequel il est possible de développer un système, partant d'un niveau abstrait, vers des niveaux plus détaillés pouvant aboutir à une implémentation.
Dans ce contexte nous avons traité principalement deux problématiques. La première est liée au développement par raffinement d'un SBC modélisé uniquement par ses interfaces SysML. Notre contribution permet au concepteur des SBC de garantir formellement qu'une composition d'un ensemble de composants élémentaires et réutilisables raffine une spécification abstraite d'un SBC. Dans cette contribution, nous exploitons les outils: Ptolemy pour la vérification de la compatibilité des composants assemblés, et l'outil MIO Workbench pour la vérification du raffinement
La deuxième problématique traitée concerne la difficulté de déterminer quoi construire et comment le construire, en considérant seulement les exigences du système et des composants réutilisables, donc la question qui en découle est la suivante: comment spécifier une architecture SBC qui satisfait toutes les exigences du système? Nous proposons une approche de vérification formelle incrémentale basée sur des modèles SysML et des automates d'interface pour guider, par les exigences, le concepteur SBC afin de définir une architecture de système cohérente, qui satisfait toutes les exigences SysML proposées. Dans cette approche nous exploitons le model-checker SPIN et la LTL pour spécifier et vérifier les exigences.

Mots-clés : Modélisation, Spécifications SysML, Architecture SBC, Raffinement, Compatibilité, Exigences, Propriétés LTL, Promela/SPIN, Ptolemy, MIO Workbench

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ