Adaptation of SysML Blocks and Verification of Temporal Properties

Hamida Bouaziz

▶ **To cite this version:**

Hamida Bouaziz. Adaptation of SysML Blocks and Verification of Temporal Properties. Other [cs.OH]. Université de Franche-Comté, 2016. English. NNT : 2016BESA2015 . tel-01428887

HAL Id: tel-01428887

https://theses.hal.science/tel-01428887v1

Submitted on 6 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SPIM

Thèse de Doctorat

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

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Adaptation of SysML Blocks and Verification of Temporal Properties

Hamida Bouaziz

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X

THÈSE présentée par

Hamida Bouaziz

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : Informatique

Adaptation of SysML Blocks and Verification of Temporal Properties

Soutenue publiquement le 03 novembre 2016 devant le Jury composé de :

Yamine AIT AMEUR	Rapporteur	Professeur, IRIT, ENSEEIHT
Franck BARBIER	Rapporteur	Professeur, LIUPPA, Université de Pau
Khalil DRIRA	Examinateur	Directeur de Recherche CNRS, LAAS, Université de Toulouse
Hassan Mountassir	Directeur de thèse	Professeur, DISC, Université de Franche-Comté
Samir Chouali	Co-Directeur de thèse	MCF, DISC, Université de Franche-Comté
Ahmed Hammad	Co-Directeur de thèse	MCF, DISC, Université de Franche-Comté

Contents

List of Figures

1

Introduction

1.1/ Context and Challenges

At any time, the system can express new needs to new services. However, the fact of seeing and developing the system as one unit constitutes a barrier for its evolution, where it will be very difficult to specify the parts of the system which are altered by each evolution. Also, the verification of the system after modification will be more and more complex. In fact, the disadvantages of this approach have changed the manner of designing and developing these systems. That is what justifies the trend of the new approaches, such as CBD (Component-Based Development) approach which takes the system as a set of components. Developing systems by reusing and adapting a set of components constitutes the central topic of component-based development. It allows tackling the problems of the old approaches, but it also creates new challenges and criteria that must be taken into consideration during the development.

When assembling separately designed components, there is a high probability of encountering the problem of mismatches between them. These mismatches can concern for example the name of services, as well as the order in which the component asks (resp. offers) for environment services (resp. its services). That is what justifies the introduction of third entities or components which are used to solve these mismatches. This kind of components are called "adapters". A big part of the works done to adapt components start from a formal specification of these components, which makes difficult the communication between the various stakeholder in CBD projects. This implies the introduction of persons who are experts in the formal methods during the selection of the candidate components to buy and thus to reuse.

To tackle this problem and to make the communication between stakeholders easier. System engineering community proposes to use high level languages which adopts the principle of using the component as the development unit. This appears clearly through SysML [OMG12b], a language which is adopted by OMG, it is used to design systems that include software and hardware. The System Modelling Language (SysML), through its diagrams, fosters the view point that takes the system as a set of components. In SysML, we call them 'blocks'. A block is a modular unit of the system description. It may include both structural and behavioural features, such as properties and operations. To communicate with its environment, a block has a list of ports. These latter are characterised by interfaces that present the offered and the required services of the block. The use of these interfaces allows the preservation of the principle of black-box, where we can know what is the role of the component without having a need to see its implementation. SysML also

offers many diagrams to represent the behaviour of the blocks. It also puts at the disposal of developers the requirement diagram that allows capturing the different requirements and establishing the link between them and between the responsible blocks of their satisfaction.

This privilege given to SysML doesn't mean that it will take the place of formal methods. But it replaces them at a level of system representation, where we need a heigh level specification of the system, to allow a better communication between the CBD project stakeholders. We must also mention that SysML lacks of formal semantic, which makes very interesting the introduction of formal methods in component adaptation domain to compute the adapters and their behaviour semantics, and to verify the result of assembling components after the insertion of these adapters. In this context, the use of formal methods appears worthwhile because it allows to specify formally components interactions and thus to ensure component-based systems reliability by verifying components compatibility. Regarding the advantages and disadvantages of each of them, a combination of both in the same approach is the solution that will tackle the lack of each of them. That's what Model Driven Engineering (MDE) tries to do through the introduction of model transformation approaches.

In this context, we have identified these challenges:

- When assembling a set of components, it's very interesting to verify their compatibility. In this thesis, we are placed in the context of optimistic approach. According to the optimistic approach, two components are incompatible if it doesn't exist any environment to assemble them without leading their composition into a livelock situation . The verification of compatibility depends on the models used to represent the structure and the behaviour of these components. In the case where the components are modelled using SysML, there is a big question mark about the manner according to which the compatibility verification of blocks will be performed, could this verification be applicable directly on SysML models? or must we introduce SysML models into a transformation process to obtain their equivalents of formal models which are more suitable for a rigorous verification?

- Generally, the high level modelling languages as SysML, adopt some principles to manage the complexity of system representation and development. In SysML, the decomposition and the hierarchical organization constitute the major principles used to handle complexity. The utility of the decomposition and the hierarchy appears clearly through the structural and the behavioural specification of the system. Thus, an hierarchical representation of the blocks interactions, and a verification based on the abstraction introduced by this hierarchy can widely help in reducing the state space when we compose the interaction scenarios of blocks in order to verify the compatibility of these latter.

- The adaptation of components implies the introduction of a third entity called adapter. The major difference between the existing adaptation approaches concerns the detail given to generate the adapter. In [DBM14], the authors give only an adaptation contract that is resumed in a specification of the correspondences between blocks services. This will have an impact on the generation of the adapter, the adapter will contain all the possible interaction scenarios between the reused components (it can contains scenarios that are not necessary for the cooperation of the reused components). However, in [CPS06a, CPS08], the authors have increased

their adaptation contract by a specification of the adapter interactions by ordering the vectors of the adaptation contract using regular expressions. This requires that the developer, before making the specification of the adapter, must thoroughly know the interaction of each component with its environment, and he must have an idea about the synchronous execution of the reused components. In this context, we ask the question about the detail that will be enough to generate adapters to make a set of components cooperate with respect of the intention behind their assembling?

- In the context of an incremental development of a system by reusing and adapting components, the system, at each increment, will expose more blocks, and generally the verification of the satisfaction of a requirement by the assembled system implies the composition of scenarios of all components, which is considered as the source of state explosion problem. In this context, a proposition of a method which takes advantages from the mediator role played by the adapters, to reduce the state space during the verification of requirements satisfaction, appears very interesting.

1.2/ Contributions

In this section, we present a summary of the contributions proposed in this thesis described in Figure 1.1:

- In the first contribution, we focus on verifying the compatibility of components modelled with SysML diagrams. Thus, we model components interactions with SysML sequence diagrams (SDs) and components architecture with SysML blocks. The SysML SDs constitute a good start point for compatibility verification. However, this verification is still inapplicable directly on SDs, because they are expressed in informal language. Thus, to apply a verification method, it is necessary to translate the SDs into formal models, and then verify the wanted properties. In this thesis, we propose a high-level model-driven approach which consists of an ATL grammar that automatizes the transformation of SDs into interface automata. Also, to allow an easy use of Ptolemy tool to verify compatibility of blocks basing on interface automata, we have proposed some Acceleo templates that generate the Ptolemy entry specification.

- In SysML, the interactions between blocks are modelled with Interaction Block Diagram (IBD) and Sequence Diagram (SD). However, these interactions are modelled by the IBD only as architectural links. In other hand, a block can participate in multiple use cases, which makes its interaction protocol divided onto a set of sequence diagrams. For these reasons, there is a lack of a global view on the interaction protocol related to a given block. To allow a hierarchical representation of blocks interactions and to benefit from the abstraction introduced by this representation, we have proposed HPSM (Hierarchical Protocol State Machine) diagram. In order to permit the compatibility verification of blocks, we perform a translation of HPSMs into HIA-ILTs (Hierarchical Interface Automata with Inter-Level Transitions), a variant of interface automata (IA) which we propose for this purpose. Our major objective is to benefit from the hierarchy which is present in HIA-ILTs. Thus, we have adapted the existing approaches for compatibility verification based on IAs to be applicable on the HIA-ILTs. However, in order to avoid the flattening of the entire HIA-ILT, we pro-

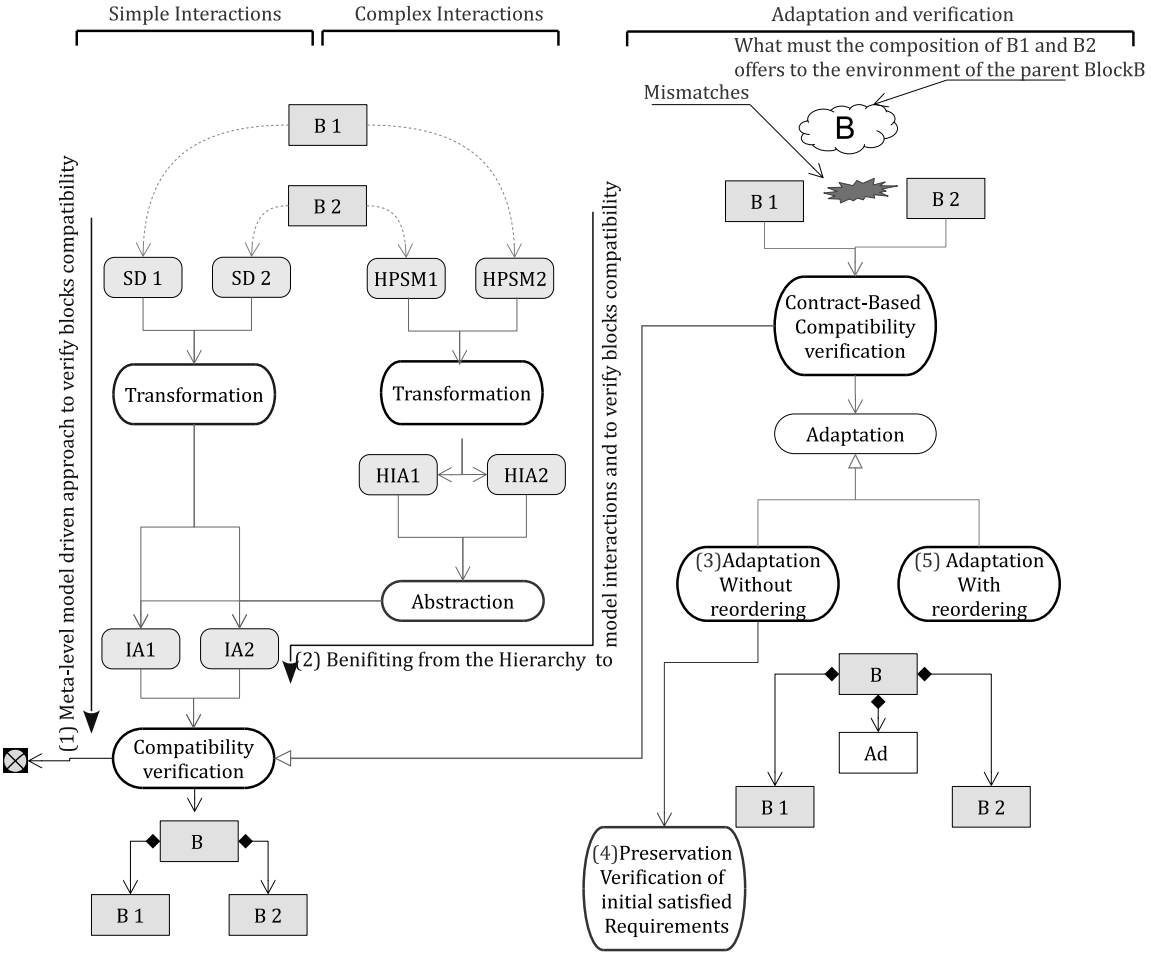


Figure 1.1: Thesis contributions.

pose a preliminary phase that allows selecting the composite states to flatten. The aim behind this is to alleviate the verification phase.

- In the third contribution, we propose a bottom-up approach to build systems, based on their partial specifications. The approach is based on reusing and formally adapting SysML blocks using converter-complement blocks. Our approach is completed by a verification phase which allows the verification of SysML requirements, formally expressed by temporal properties, on SySML blocks. In this phase, we exploit our manner of defining the adapter, to avoid the verification of the initial requirements, satisfied by the adapted blocks, on the whole system, and thus, we reduce the state space explosion problem.

- In the same context of the previous contribution, we have proposed a bottom-up approach to adapt SysML blocks but with different inputs and objectives. The major difference resides on that the adapter as we will define it in this case can solve more problems such as the reordering of services to eliminate livelock between blocks, it can also solve more types of mismatches ('one-to-many' rather than only 'one-to-one').

  To generate the adapters, in our approach, we are focusing on an incremental ap-

proach to construct the system. Where, at each increment the developer gives a specification of a part of the system that he want to build. This specification represents the interaction of the parent of the reused blocks with the rest of the system. The generation of the adapter is based on refinement relation between this specification and the specification of the reused blocks.

1.3/ Publications

In the following, we list the references for the published and submitted articles:

- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. SysML Blocks Adaptation. ICFEM'15, the 17th International Conference on Formal Engineering Methods, Springer, pages 417-433, Paris, France, 2015
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. A Model-Driven Approach to Adapt SysML Blocks. ICIST, the 22nd International Conference on Information and Software Technologies, Springer, pages *-*, Kaunas, Lithuania, 2016 (To appear)
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. Compatibility Verification of SysML Blocks Using Hierarchical Interface Automata. ISPS 12th International Symposium on Programming and Systems, IEEE, pages 313--322, Algiers, Algeria, 2015
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. Exploitation de la Hiérarchie pour la Vérification de la Compatibilité des Blocs SysML. CAL 9ème conférence francophone sur les architectures logicielles, Hammamat, Tunisie, 2015
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. Exploitation de la Hiérarchie pour la Vérification de la Compatibilité des Blocs SysML. RNTI. Revue des Nouvelles Technologies de l'Information, Volume RNTI-L-8, 2016, Pages : 99-118.
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. An Incremental Approach for Adapting and Verifying SysML Blocks. In SoSyM, Software and Systems Modelling (Re-submitted after revisions)
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. On the Use of Coloured Petri Nets to Adapt SysML Blocks. In JSS, Journal of Software and Systems (submitted)
- Hamida Bouaziz, Samir Chouali, Ahmed Hammad and Hassan Mountassir. SysML Model-Driven Approach to Verify Blocks Compatibility. In IJCAET. International Journal of Computer Aided Engineering and Technology (submitted)

1.4/ Document Outline

In this section we give a summary of the content of this thesis, which is structured in three parts as follows: In Part I, we introduce the scientific context and the related work of this

work, there, we first give, in Chapter 2, an overview of SysML language and its diagrams that allow to model the structure the behaviour and the requirements of systems. Then in Chapter 3, we present the transformation of models and the key concepts to describe them. We also introduce the concept of Interface Automata and Coloured Petri Nets, which we will use later in our approach to formally represent the behaviour protocols of SysML blocks and to verify component compatibility. Finally, in Chapter 4, we present the component-based domain and exactly the adaptation of components track.

In Part II, we present the contributions of this thesis regrouped in six chapters. In Chapter 5, we give a formal definition of SysML diagrams that we will used in our approaches of adaptation. In Chapter 6, we present our model-driven approach to verify blocks compatibility by transforming SysML sequence diagram into interface automata. Then, in Chapter 7, we introduce the hierarchy to model the interaction protocols of blocks, and we define our approach which takes advantage from the hierarchy to alleviate the compatibility verification of SysML blocks. Next, in Chapter 8, we explain our approach for adaptation based on the refinement relation between the blocks. After that, in Chapter 9, we focus on our adaptation manner to alleviate the verification of requirements initially satisfied by the adapted blocks. Finally, in Chapter 10, we extend our previous approach of adaptation to allow the reordering of requests for services, and to solve more type of mismatches by allowing more types of correspondences between blocks' services rather than only one-to-one correspondences.

In Part III, we conclude our work with Chapter 11, where we present the conclusions and perspectives of this thesis.

I

Scientific Context and State of the Art

2

SE and SysML Language

The systems engineering (SE) is an approach that proposes a range of processes and tools. This range allows controlling the development, the understanding and the reusing of complex systems. Particularly, these processes offer for developers the steps that they must follow to cover the different aspects related to the development of a given system. Each step proposes the use of some models for a better representation of a system aspect to which this step is dedicated.

The intention of creating a new community which focuses on the engineering of systems saw the light in the great institutions of American defence. In fact, the National Aeronautics and Space Administration (NASA) and the United States Air Force (USAF) have tried, in 1960s, to make a frame for the development of military programs and space exploration systems through more rational industrial approaches. This effort has led, in 1991, to the creation of the International Council on Systems Engineering (INCOSE), the first worldwide organism for system engineering [wik].

In the systems engineering domain, the system is seen not only as a set of software elements, but as a range of software and hardware elements which are in a constant interaction. In addition to the interactions inside the system, this last can interact with the environment. This interaction can be a request or an offer of a software service, or it can take the form of a signal or matter circulation. By intention to set up a language which

Contents

allows the modelling of all these aspects and sides of systems, OMG and INCOSE have unified their effort to create this language and to make it like what UML becomes for software engineering. This language is called System Modelling Language (SysML).

In the rest of this chapter, in section 2.1, we will define SysML, the need of system engineering to SysML, and the principles of this language. After that, in section 2.2, we demonstrate the emergence and wide spreading of SysML through a collection of works that have focused on SysML for modelling. Next, in section 2.3, we give a bref definition of SysML diagrams by mentioning the aspect covered by each diagram. In section 2.4, we give the example of some free platforms that allow to use SysML language for modelling. Finally, in section 2.5, we conclude.

2.1/ SysML

SysML (System Modeling Language) is a modelling language that allows the representation of the system as a set of diagrams. The appearance of SysML has been motivated by the intention of the systems engineering community to define a common modelling language. In fact, after ten years of its appearance, SysML has succeeded to take a place in the system engineering domain which is similar to that taken by the Unified Modelling Language (UML) in the software engineering domain.

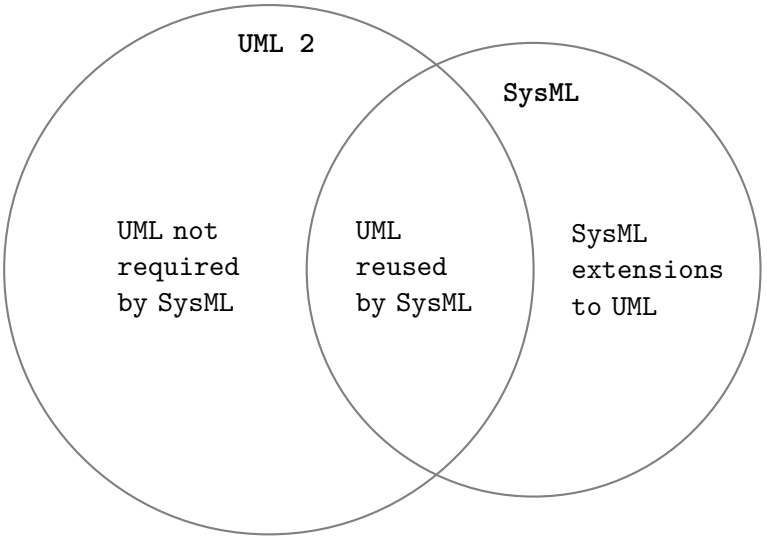


Figure 2.1: Relation between SysML and UML [OMG12a].

SysML can be defined as an extension of a sub-set of UML diagrams (Figure 2.1). This extension was made through the use of the profiling mechanism which is defined by UML. SysML allows the specification, analysis, design, verification and validation of a wide range of systems. It allows also to model the different aspects related to a given system; whether the requirement, the structural or the behavioural aspects. SysML is an open source specification, it includes an open source licence for its distribution and its use, its current version is **'OMG SysML v.1.3'**.

2.1.1/ The Need of SE to SysML

The systems engineering is interested by the different sides of complex systems, whether the software or the hardware sides. However, the development of a system, which is characterized by an order of complexity, is still not obvious if there is no suitable tools which assist and guide its development. It was due to this need that the SysML language has seen the light as a communication language between the different members of the development teams. It allowed to unify the visual modelling principles using a small set of diagrams, which makes it easy to learn and to use. The introduction of SysML in this domain was not only for simplifying the modelling and the communication but also to offer the development community a good pillar to analyse the requirements of the system since the first steps of the development through a model driven process.

2.1.2/ Who Created SysML?

Many parts have contributed to the creation of SysML, they are all united in one association called 'The SysML Partners'. The goal behind this union was the creation of an UML profile that will be more adapted for the system engineering domain. This association that is created in 2003 under the leadership of Cris Kobryn, have defined SysML as an open source specification. On november 2005, the association 'The SysML Partner' has finished the draft copy of SysML specification v1.0. This specification has been revised and adapted by Object Management Group (OMG) on july 2006[sys]. Among the contributors to the creation and the persistence of SysML, there is for example: Gentleware [Gen], Motorola [Mot], INCOSE [INC], etc.

2.1.3/ Principles of SYML

The development of SysML has been mainly guided by these principles:

- Parsimony: SysML bases on a part of UML. This part is considered as the minimal sub-set of UML diagrams that allows the satisfaction of the requirements of the systems engineering community. The other needed elements have been added in function of the new needs expressed by the systems engineering domain.
- Reuse: SysML reuses the concepts of UML. However, the additional requirements have been satisfied by adding new concepts. This extension of SysML concepts has always been guided by the principle of parsimony.
- Layering: This principle is used to organize the SysML profile in two ways. The first way bases on the fact that SysML is defined as strict UML profile. Hence, all SysML packages are considered as an extension layer of UML meta-model. However, the second way concerns only the SysML constructs, where they are organized into two levels of compliance, Basic and Advanced, which constitutes an additional layering.
- Extensibility: SysML supports the same extension mechanisms provided by UML (metaclasses, stereotypes, model libraries), therefore the language can be further extended for specific systems engineering domains, such as automotive, aerospace, manufacturing and communications.

- Interoperability: SysML is aligned with the semantics of the ISO data interchange standard to support interoperability among engineering tools. It inherits the XMI interchange from UML which makes possible the use of generated models files of a tool by other several tools.

2.2/ Emergence of SysML

The introduction of SysML into the systems engineering domain has opened the door to many studies, which intend to evaluate its capacity to model the different aspects of systems, with all necessary details, through industrial case studies [LdSd006, PSTV13]. In [LdSd006], a proposition is made to model an experimental unit of a factory plant system in Santa Catarina university. Also, the study, in [PSTV13], concerns the use of SysML to model an industrial system, which is a part of a system that controls the power of a boiling water reactor of a nuclear plant at Finland.

SysML, through its structural diagrams, tries to foster the view point that takes the system as a set of components, where the component represents the basic unit of the development. In [MTO+11], the authors try to benefit from the advantages of component-based development to design a module-based software for a robot. This software must allow for the robot to capture a target object using a camera, to move toward it, and to move it. Regarding the compatibility between the component-based development approach and the SysML language, the authors made their choice on SysML as language to model their system. They have used the Internal Block Diagram (IBD) of SysML to represent the multilayer architecture of their system, where the blocks communicate using tasks, the tasks take their entry data from the sensors, and they activate the moving material parts.

The decomposition of the system on a set of blocks reinforces the reusing of its parts, and facilitates its adaptation over time, which increases the life time of the system. These advantages are due mainly to the interfaces which are used by the blocks to communicate with the rest of the system (encapsulation principle). These decomposition and encapsulation principles of SysML offer a better control of the models size that are used to model the system, they allow targeting the details of the system through a succession of steps (from a high level of abstraction to a low level), which makes the diagrams that capture the different aspects of the system more clear. In [LWMY11], the authors show the structure and the dynamic aspects of a maintenance assistance system of the military planes. The system is exposed at high level of abstraction through a range of SysML diagrams, they have used the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD) to represent the architectural part, and for modelling the dynamic part, they have based on the activity and sequence diagrams.

In [GCRJ08], an automotive driver information system of 4*4 vehicle was modelled using SysML. The objective was to present, through a case study, the capacity of SysML diagrams to model the different aspects of electronic systems in automotive vehicles. This study proved the distinction of SysML in term of the capacity to model the different structural and behavioural properties. Yue et Peter, in [GJ09] try to foster their view point on SysML, by establishing an evaluation report of SysML, where they compare it with Simulink/MATHLAB. Contrary to Simulink/MATHLAB, SysML offers a mean to cover the structural aspect. Concerning the functional aspect, SysML also offers more constructs and diagrams to model this aspect (sequence diagram, activity diagram, state machine

diagram) than Simulink/MATHLAB.

SysML could also take a place in the domain of physic, and exactly in the field of the development of particle accelerators. In [GGA+08], the authors have presented a set of SysML diagrams that allows representing a part of LLRF (Low Level Radio Frequency) system. The authors, through this orientation, try to prove the utility of using a model-driven language as SysML instead of basing on documents of thousands lines that make the communication between team members difficult and slow.

There is also an attempt to integrate the reliability analysis of mechatronic systems techniques into the approach of systems engineering, by focusing on the SysML models. In [MCR+12], the authors present the utility of this integration. The idea can be resumed on generating the failure possibilities that can arrive in a given system using structural and behavioural models of SysML. The authors have used an electromechanical actuator for aircraft ailerons to illustrate their approach. In [DIK09], the authors, through their experience of combining modelling languages and systems reliability analysis techniques, have found that SysML models are more suitable then those of UML to support the reliability study of systems. They have demonstrated this distinction of SysML through a study of a system that controls the level of a tank.

SysML was also introduced in the field of Radio frequency and microwave engineering. In [LCKB08], the authors show how SysML diagrams (block definition diagram, internal block diagram, requirement diagram) can be used to model a UMTS (Universal Mobile Telecommunication Standard) transceiver system. SysML has also proved its capacity to model the control software of complex systems, where a change in a requirement may alter all the system. In [JT13], an approach was made to regulate the development of control systems. This approach can alleviate the effect of requirements evolution. The authors have proposed the use of SysML models to represent the controls systems IEC 61131-3. These models can be used, later, as the start point of an MDE (Model Driven Engineering) process to generate the code for an implementation in the standard IEC 61131-3 languages. The authors have presented their approach through a control system of a motor and a pump.

2.3/ SysML Diagrams

SysML has nine diagrams, where four diagrams (package, use case, sequence and state machine diagrams) are directly copied from UML 2.0, three diagrams (activity, block definition and internal block diagrams) are copied with some modifications, these modifications deal with the differences between the software engineering and the system engineering. The last two diagrams are considered as new (parametric and requirement diagrams) (see Figure 2.2).

Another taxonomy decomposes the SysML diagrams on three sub-sets (see Figure 2.2). It is based on the aspect to which each diagram is associated. It differentiates between the structural, behavioural and requirement diagrams. In the following, we give a description of each sub-set, with a definition of each diagram. We focus more on the diagrams that we will use in our work.

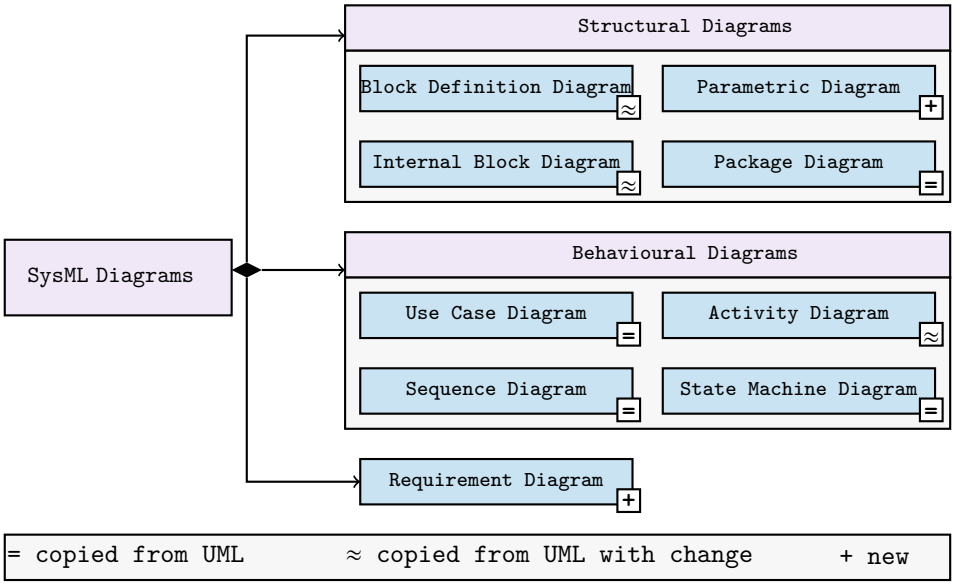


Figure 2.2: SysML Diagrams

2.3.1/ Structural Diagrams

In SysML language, the block represents the basic unit used to build the system architecture. It can refer to a material part as well as a software part, it can also represent a person which uses or interacts with this system. The system structure can be modelled using a range of SysML diagrams, which includes the block definition diagrams, the internal block diagrams, the package diagrams and the parametric diagrams. This range of diagrams allows the decomposition of the system into a set of blocks and organizing them inside packages, also it allows establishing the architectural links between blocs and packages, and define the relations between their quantitative features. In the following, we give a basic description of each architectural diagram by listing the basic constructs of each diagram that we have used. Because SysML is a graphical language, we base on graphical models to show the basic anatomy of these diagrams.

- `The block Definition Diagram`

  The Block Definition Diagram (BDD), as it is captured in Figure 2.2, is a copied diagram from UML with some modifications. It bases on the UML class diagram, with exclusion of some capabilities, such as some specialized forms of associations. On the other hand, it has modified some concepts such as the notion of class was replaced by the notion of block. It added also new concepts such as the blocks ports. The BDDs allow the modelling of the system parts using blocks and they offer the possibility to visualize the dependences between these blocks and the existed hierarchy. This hierarchy helps to identify two groups of blocks: Atomic and composite blocks. It is possible to obtain a flatten BDD of the system, by replacing each composite block by its sub-blocks.

  Each block (as it is shown in Figure 2.3) has a name, a set of values, properties, referenced blocks, parts, operations and constraints expressed on its properties. To interact with the rest of the system, the block uses the ports which are placed on its sides.

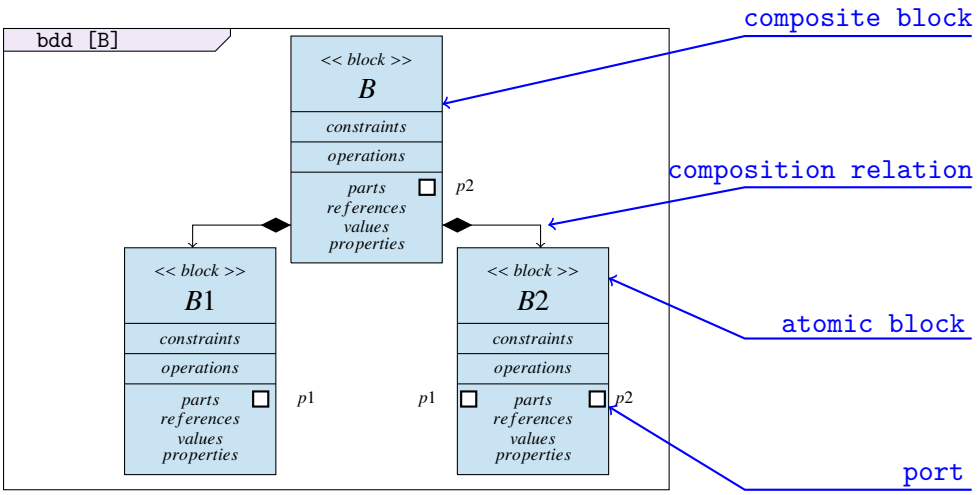


Figure 2.3: A Block Definition Diagram (BDD)

- **The Internal Block Diagram**

  The Internal Block Diagram (IBD) is an adaptation of UML composite structure diagram. It is used to model the internal structure of each composite block. This internal structure is resumed on the set of parts and connectors. The parts represent the instances of blocks, each part has the same ports as the block that instantiates. The connectors specify the topology of connecting parts, they link the ports of parts. If a connector links a port of the father block and one port of its parts, it will be considered as a delegation connector (see Figure 2.4).

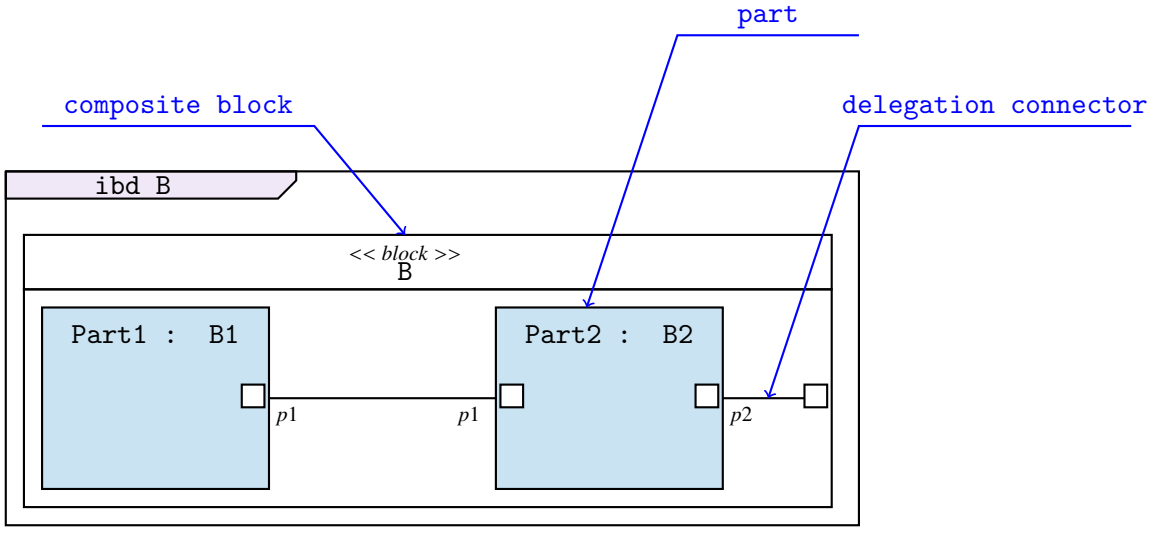


Figure 2.4: An Internal Block Diagram (IBD)

- **The Parametric Diagram**

  The Parametric Diagram is a new diagram introduced in SysML. Its use is for expressing the constraints (equations) on the blocks properties. Thus, it constitutes a good mean to evaluate the system performance.

- **The package Diagram**

  The Package diagram is used to organize the global model of the system, where the other SysML diagrams will be elements of the system packages. This organization can be done in different ways by considering some different aspects. It can be done by considering the system hierarchy (enterprise, system, design, .., verification). It can also be guided by the domain (requirements, use cases, structure, behaviour,...) or by the view points.

2.3.2/ Behavioural Diagrams

SysML offers four behavioural diagrams. They allow modelling the behaviour of blocks using a set of steps (actions, states, ...) provided with a set of evolution rules. They take generally the form of oriented graphs. In the following, we give a simple definition of each behavioural diagram, we will focus more on the sequence diagram.

- **The Use Case Diagram**

  The Use Case Diagram models the functionalities of the system that require an interaction between the system and its users. Anything that users would to do with the system has to be made available as a use case or a part of a use case [GBB05].

- **The Sequence Diagram**

  The Sequence diagram (SD) is a copied diagram from UML2.0. It represents the interactions by focusing on the observable exchange of messages between blocks. A sequence diagram has two dimensions, where the vertical dimension represents time and the horizontal one represents the blocks which participate in the interaction ([RJB04]).

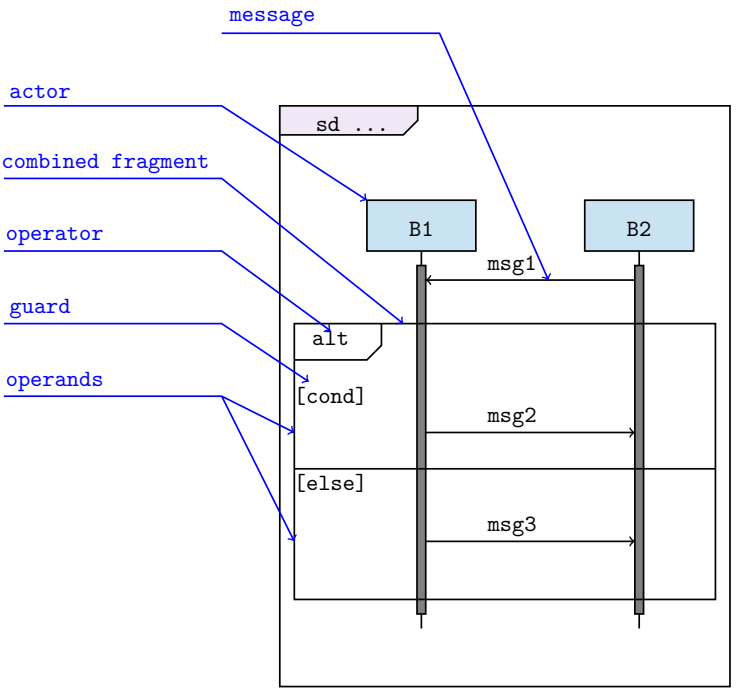


Figure 2.5: Basic elements of a Sequence Diagram (SD)

It consists of a set of lifelines which represent the interacted blocks. The temporal execution of interactions is shown as a succession of messages. A message takes the form of an arrow originates at the sender and ends at the receiver. A SD can also contain a set of combined fragments (CFs). CFs are used to express different types of control flows, such as concurrency, choice and loop ([RJB04]). They are defined by interaction operators (Alt, Loop, Break, etc) and corresponding interaction operands (see Figure 2.5).

- `The Activity Diagram`

  An activity diagram illustrates one activity. It models its fundamental elements which are the actions, the control elements (decision, division, merge,..) and the inputs/outputs/control flows.

- `The State Machine Diagram`

  The State Machine Diagram captures the different states of the block to which this diagram is associated. The transitions between states are labelled by the actions executed by the block to change its state. The execution of a transition can imply some modifications of the values that describe the last state. State machine diagram didn't know any modification from UML 2.0.

2.3.3/ Requirement Diagram

The Requirement Diagram (RD) is a new diagram in SysML. It specifies the system requirements which are expected by the users. This diagram offers a way to model the functional and no-functional requirements and the different links between them.

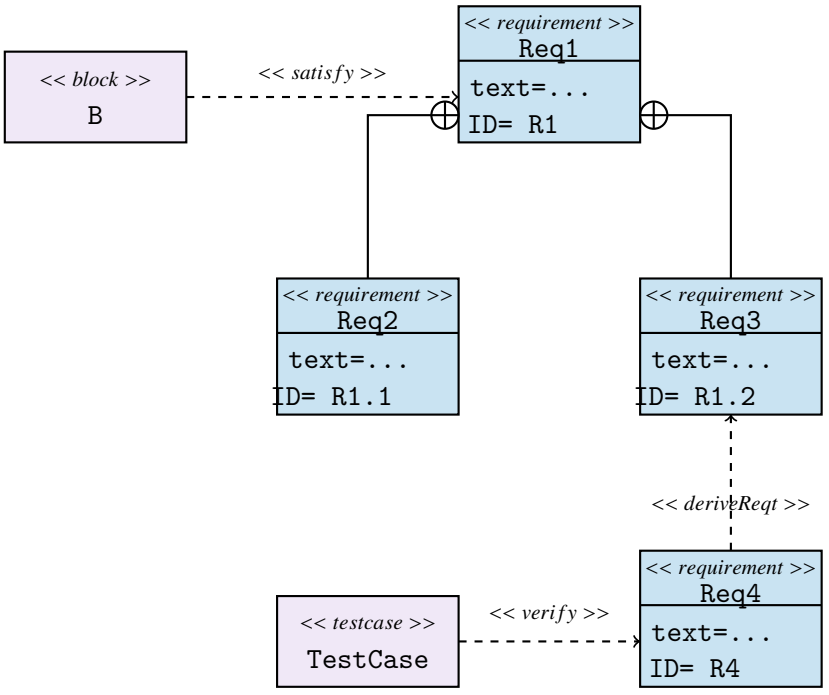


Figure 2.6: Basic elements of a requirement diagram

It is possible to represent the existed hierarchy between requirements using the composition and the derivation (<<deriveReqt>>) relations. It is also possible to link a block to

a requirement using the satisfaction relation (<<satisfy>>), or to associate a requirement with a test case using a verification relation (<<verify>>) (see Figure 2.6). Each requirement is defined by its name, its description and its own and unique identifier.

2.4/ Free Platforms for SysML Modelling

2.4.1/ TOPCASED

TOPCASED [top] is the acronym of **T**oolkit in **O**pen **S**ource for **C**ritical **A**pplications and **S**ystems **D**evelopment. It is a free computer-aided engineering software. It bases on the Eclipse development platform [wik].

The objective of TOPCASED is to cover the set of requirements for developing software and systems (the descending branch of the V cycle), as well as the transversal needs such as configuration management, change management and requirements engineering. Its development is based on a MDE (Model Driven Engineering) approach. It is based on a global ecore meta-model which includes all the classes of SysML diagrams elements. It provides also the elements with their graphical notations which simplify the modelling. Based essentially on standardized language for modelling software (UML, SysML, AADL, etc.), TOPCASED works with XMI files.

2.4.2/ Papyrus

Papyrus [pap] is a set of eclipse plugin that belongs to the Eclipse Modelling Project. It aims to offer an integrated environment which is dedicated for users. Thus, Papyrus is a new environment for editing all sorts of EMF models (Eclipse Modelling Framework), and to particularly support UML and its profiles such as SysML, MARTE, etc.

Thus, Papyrus offers editors for diagrams of EMF based modelling languages. It offers also the glue for linking its editors and other tools. Its generation of XMI files, allows the use of its models with other applications and tools. Papyrus is graphical but also textual tool. Thus, it is possible to edit the models using textual editors which offer the assistance to edit the model content.

2.5/ Conclusion

In this chapter, we have presented SysML and its diagrams, by focusing more on those that we will base our work on. SysML can be used to simplify the modelling of systems. However, its graphical and high level aspect constitutes a barrier in front of its use for verification, which makes the introduction of its diagrams into a transformation chain very essential. In the next chapter, we will present some transformation works which have targeted SysML diagrams to generate formal specifications.

3

Model-Driven Development and Models Transformation

The model-driven development is centred on the use of models. It is a form of generative engineering that has its own processes, where all or a part of an application is generated from models. It covers many approaches such as: Model-Driven Architecture (MDA) [MDA, Dav03], Model-Integrated Computing (MIC) [SK97] and Software Factories [GS03]. The MDA is defined and supported by Object Management Group (OMG). MDA changed the software development style, it describes each artefact as a model, which shifts the developers from code-oriented to model-oriented approach. It bases on three major types of models: Computation Independent Model (CIM) involves specification of system functionalities, Platform Independent Model (PIM), and Platform Specific Model (PSM). MIC focuses on the formal representation, composition, analysis, and manipulation of models during the design process. However, the software factories focus on the product-line systems, where it tries to automatize the product development.

In the context of model-driven development, the notion of model transformations [CH06] plays a fundamental role. This transformation can concern different kinds of models (e.g. formal or informal, graphical or textual, etc.), and it can be done for different purposes (e.g. formalizing for verification objective, for generating application code, etc.)

In the remainder of this chapter, we will give an overview of the transformation of models and its application on SysML models, where in section 3.1, we present the basic concepts of model-driven development. In section 3.2, we summarize the different types of transformation. After that, in section 3.3, we present some works which have been done on SysML

Contents

diagrams. However, in section 3.4, we focus on the transformation works that targeted sequence diagram of SysML. In our contribution, we need to transform sequence diagrams into interface automata and CPNs, that is why we reserve the section 3.5 and the section 3.6 to present the formal models: Interface automata and coloured Petri nets. Finally, in section 3.7, we conclude.

3.1/ Basic Concepts

The model-driven development defines a toolbox that contains the necessary tools for expressing and structuring the different works in this field. These tools allow for architects and developers to share the same vocabulary. In the following, we give an overview of the basic notions and concepts used in this domain.

- **Model and Modelling:**

  A model is an abstract description of the real system, this description can be considered as a simplification and a restriction of the reality according to a given viewpoint. This viewpoint is controlled by the needs and objectives behind the construction of the system. Generally, in addition to the textual annotations, the model includes many more of the graphical components.

  The modelling is the art of projecting the studied system on conventional diagrams. Data modelling is an abstract representation, where the individual values of data are ignored [wik].

- **Model-driven**

  The model-driven approaches are based on the notion of model, they provide a set of models that help in understanding and steering the system design, construction, deployment and maintenance. In these approaches, the models are seen as the base of each activity.

- **Meta-Model**

  A meta-model is the model that defines the expression language of other models [OMG06] in a high level of abstraction. A meta-model has two principal features: firstly, it must capture the essential features of the modelling language, and secondly, it must be able to depict the concrete syntax and semantic of this language.

- **Meta-meta-model**

  The meta-model is the model of a modelling language. Consequently, meta-model, in turn, is expressed in a meta-modelling language specified by the meta-meta-model.

All these concepts are represented in Figure 3.1:

- **In the first level:** There is the system to model with all its entities and all factors around them.
- **In the second level:** There is the model which is created by projecting the system on a given schema, and separating the elements of interest according to given objectives. It can be an UML class diagram, a conceptual model MERISE, or all other schema that represents an abstract view of the modelled objects.

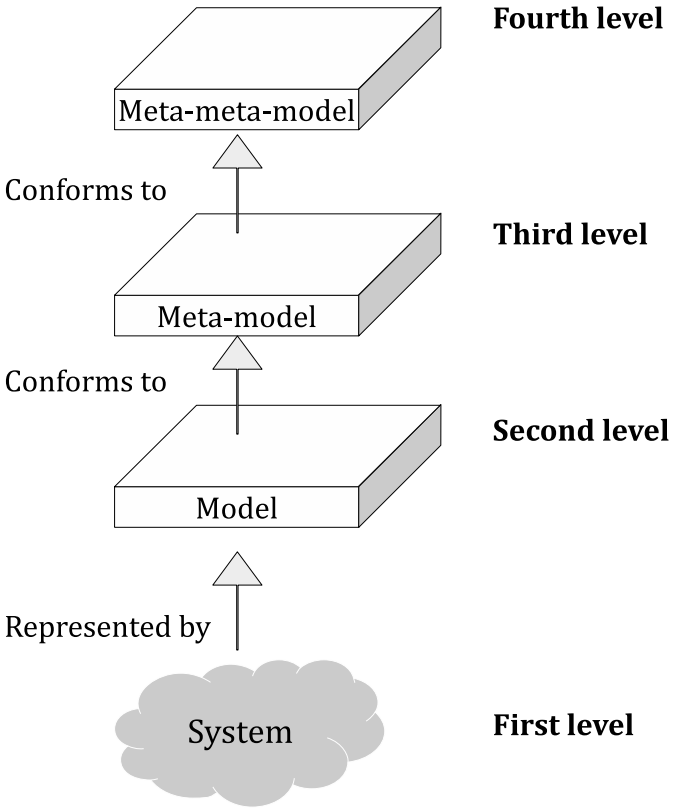


Figure 3.1: The abstraction levels of modelling.

- **In the third level:** There is the modelling language or meta-model that specify the classes of concepts used in the second level.
- **In the fourth level:** There is the meta-meta-model. It must be generic enough to define many other languages. Also, it must be precise to express the rules that each language must respect.

3.2/ Model Transformations

A model transformation, regardless of each type it is, can be seen as a function that takes as parameters (inputs) a set of models and provides as results (outputs) another set of models. The input and the output models are structured according to their meta-models. The implementation of a transformation between a set of models is possible if there exists semantic correspondences between their meta-models. These correspondences are materialized using transformation rules which can be implemented in a given language (e.g. ATL, ATOM^3, XSLT, MTL,TGG, etc.).

Generally, a transformation may take multiple source and target models. Furthermore, in some cases, this transformation may have the same source and target meta-model. In this case, we talk about endogenous transformation. Otherwise, when the source and the target meta-models are different, we talk about exogenous transformation. In literature, it exists another classification, where a transformation belongs to one of these types:

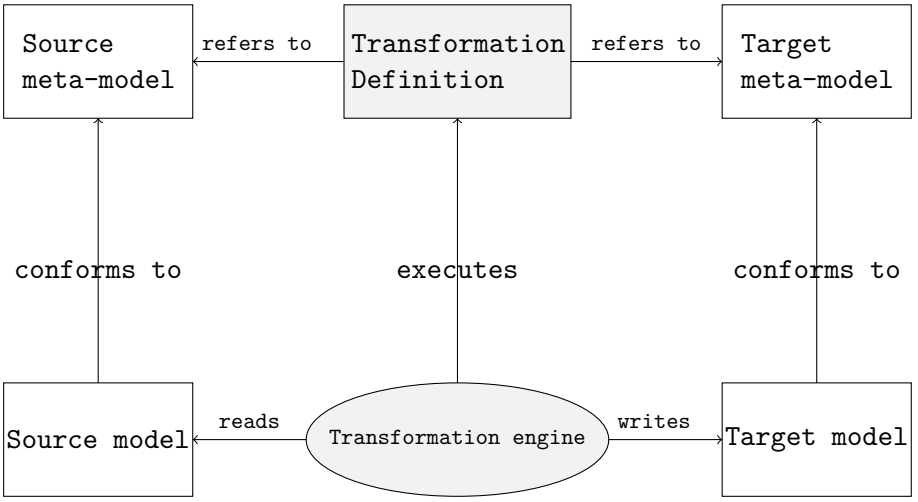


Figure 3.2: The basic concepts of models transformation [CH06].

- **Vertical transformation:** It is the type where the source and target models are defined in different level of abstraction. There is two directions of transformation, which decreases the level of abstraction (i.e. refinement) and the other which increases it.

- **Horizontal transformation:** This trend aims to modify the presentation of the source model with preserving the same level of abstraction. This modification can concern for example an add or a deletion of some model elements, or merging two source models.

- **Oblique transformation:** In this kind of transformation, we find a couple of the vertical and horizontal transformations. It is used generally by compilers to generate the executable code after optimizing the source code.

In addition to the horizontal and vertical transformations, there is another classification, which differentiates between the type of models. When the model takes the form of a textual specification, it is called code rather than model. Thus, we can distinguish two categories of transformation:

- Transformations Model → Code:

  It consists in generating textual specifications from models. There are many languages and tools that allow us to implement this kind of transformation (i.e. AToM^3, Acceleo, etc.). AToM^3 allows graphical transformation where the source model and the transformation rules are specified graphically. Acceleo, the tool that we use in our work, is the result of several man-years of R&D started in the French company Obeo [obe]. Acceleo is a source code generator of the eclipse foundation that implements the MDA (Model driven architecture) approach to realize application starting from EMF (Eclipse Modelling Framework) models. Thus, Acceleo is an implementation of the norm of the Object Management Group (OMG) for transforming models to text (M2T), where the transformations take the form of templates.

- Transformations Model → Model:

In this kind, the target model is not a text. Also, to perform this category of transformation, we find many languages and tools (i.e. AToM^3, ATL, etc.). In this work, we are interested to ATL (Atlas Transformation Language). ATL [atl] is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. These rules are based on a mixture of declarative and imperative constructs. The set of the rules constitutes the ATL grammar. Each ATL rule is characterized by two mandatory elements:

- `from` : A pattern on the source model with possible constraints.
- `to` : One or more elements of the target model, it indicates how target elements must be initialized from the corresponding source elements.

In literature, the transformation is usually used to generate the source code of applications. Another use is for formalizing models of high level languages. Various research works have been done to transform informal models to formal ones [RC15] [AEC14] [GCA13] [DHJ+10]. These transformations are generally implemented and offered as tools which can assist architects during the verification of their systems. Many of these works are dedicated to generate formal models from UML and SysML diagrams [RC15] [AEC14] [RKBIH15], which are considered as informal models.

3.3/ Transformation of SysML Diagrams

Many works (e.g. [Vas09], [GBHP15], [JKPB12]) have been done to integrate the descriptive power of SysML models with the analytic and computational power of Modelica models. This integration provides a capability that is significantly greater than provided by SysML or Modelica individually.

A mapping between SysML and Modelica, considering a small subset of the Modelica language, has been proposed by Vasaiely [Vas09]. This work presents a mapping between SysML parametric diagrams and Modelica equations. A representation of Modelica models in SysML have also been processed by Johnson et al. in [JKPB12]. In this work, the authors explore the definition of continuous dynamic models in SysML and the use of graph grammar to maintain a bidirectional mapping between SysML and Modelica constructs. In [GBHP15], Gautier et al. proposed a tooled MDE (Model Driven Engineering) approach to validate requirements of complex systems at the earliest stages of design process. This approach consists on generating Modelica simulation code from SysML models.

In [GBHP13], Gauthier et al. presented their approach to verify the SysML models consistency with the VHDL-AMS (e.g. the naming of a component with reserved words of the VHDL-AMS language (syntactic error) or the connection of two ports with different types (semantic error)). They have used an ATL transformation to generate problems from SysML models and after generating VHDL-AMS code. A test approach to validate model-to-model transformation with EUnit [GDKR+11] has been presented there.

The authors, in [PBG14], showed a translation of SysML state machines models into a class of non-autonomous Petri net models using ATL. The target formalism of the transformation is the class of Input-Output Place Transition Nets (IOPT), which extends the known

low-level Petri net class of Place/Transition Petri nets with input and output signals and events dependencies. In [RKBIH15], the authors present a transformation process from SysML diagrams into another variant of Petri nets. They proposed an approach which describes a verification methodology of SysML activity diagrams based on their transformation into RECATNet model.

In [CLY+14], the authors used SysML and requirement elicitation templates to collect and model user requirements, and then transform requirement diagrams into other SysML diagrams for design and analysis (use-case and activity diagrams) using the transformation rules which are defined using ATLAS Transformation Language (ATL).

There is another trend to transform SysML specification into UML models. The work introduced in [LBLP11] is in line with this trend that tries to make possible to re-use the test generation techniques initially developed for UML4MBT (restriction of UML for Model Based Testing process). The introduction of SysML4MBT is justified by its capacity to model more constructs and thus more reach semantic. The approach is a model-based testing approach that takes as input a SysML specification of a system under test and automatically translates it into an equivalent behavioural UML model. This generated UML model is finally used to derive test cases and executable test scripts.

3.4/ Transformation of Sequence Diagram

The sequence diagram, which is a shared model between UML and SysML, was be the matter of many transformation works. The most of the proposed approaches are based on using transformation rules, and they essentially differ in the target model of transformation.

In [KBSB10], [RF06], we find a description of an automated transformation method, which allows transforming sequences diagrams into their equivalents of coloured Petri nets. In [ES09], the authors proposed some correspondences to transform sequence and use case diagrams to Petri Net. These correspondences formalize the interactions composed of messages and combined fragments (alternative, optional and loop). Authors in [Mer14], basing on Meta-modelling and ATL grammars, they defined a set of ATL rules to transform SDs to Petri Nets. They proposed rules for the basic constructs of SDs and for a sub-set of combined fragments kinds (Alt, Par). In [CESK09], a grammar, which based on graph transformation, was proposed to transform the sequence diagram into ECATNets, a variant of Petri Net. The authors used the $AToM^3$ tool to implement the meta-models of SDs and ECATNets, to generate the modelling tools and to implement the graph grammar that performs the transformation. They are also some works that have as target models a textual specification. In [AYAM11], a graph grammar was used to generate Promela code starting from SDs. The authors used also the tool $AToM^3$ for meta-modelling and for implementing the graph transformation grammar. In [MMSC13], The authors proposed a grammar to transform the communication diagram, which has a near semantic to that of SD, into Buchi automata.

In [CH11a], some correspondences between sequence diagram and interface automata are given. This work was be the reference in [CCM12a] to prepare the sequence diagram of SysML blocks for the compatibility verification phase. But, in [CCM12a], this transformation have done manually, which can be considered as a source of user errors. That is why, we propose, later in this thesis, the correspondences for more constructs, and we propose,

also, a set of ATL rules to automatize this transformation. Contrary to the works mentioned before, which they don't take into consideration the case of nested combined fragments, in our work, we explain the different cases, and how we deal with them. Also, at a stage of this thesis, we need to transform the sequence diagrams into coloured Petri nets. This transformation will be guided by our adaptation objective, and the generation of the coloured Petri nets will be steered by the adaptation contract in a meta-model-driven approach. Thus, in the next two sections, we present the different possible operations on Interface Automata (IAs) and Coloured Petri Nets (CPNs).

3.5/ Interface Automata

Interface automata [dAH01] were introduced by Alfaro and Henzinger to specify component interfaces and also to verify component assembly based on their actions. The set of actions is decomposed into three groups: input actions, output actions and internal actions. Input actions allow to model the methods that the component exposes to its environment. These actions are labelled by the character '?'. The output actions model the methods that the component needs to invoke from other components. These actions are labelled by the character '!'. Internal actions are methods that can be activated locally and are labelled by the character ';'.

Definition 1: Interface Automaton

An *interface automaton* A is represented by the tuple

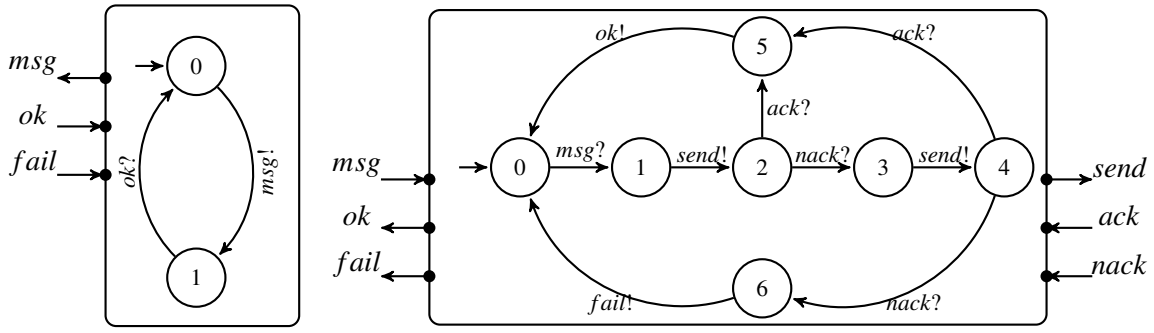
$$\langle \; \mathtt{S}_A, \; \mathtt{I}_A, \; \Sigma_A^I, \; \Sigma_A^O, \; \Sigma_A^H, \; \delta_A \; \rangle$$

Where:

- S_A is a finite set of states.
- $I_A \subseteq S_A$ is a set 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.

Example:

In Figure 3.3, we give the example of interface automata used in [dAH01] to model a software component that implements a message-transmission service. The component has a method msg, used to send messages. Whenever this method is called, the component returns either ok or fail. To perform this service, the component relies on an interface to a communication channel that indicates a successful transmission, and nack, which indicates a failure. When the method msg is invoked, the component tries to send the message, and resends it if the first transmission fails. If both transmissions fail, the component reports failure; otherwise, it report success.



(a) Interface automaton `User` (b) Interface automaton `Comp`

Figure 3.3: example of interface automata.

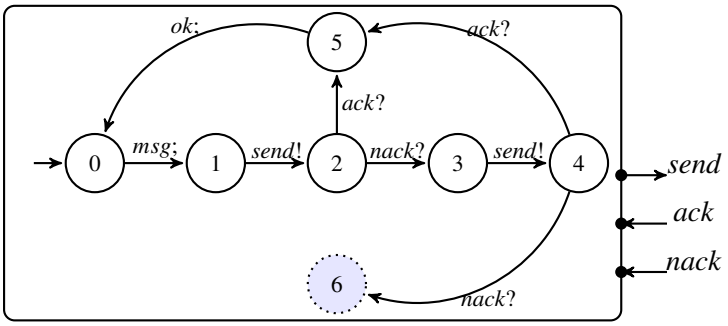


Figure 3.4: User ⊗ Comp. The illegal state of the product is depicted with dotted border.

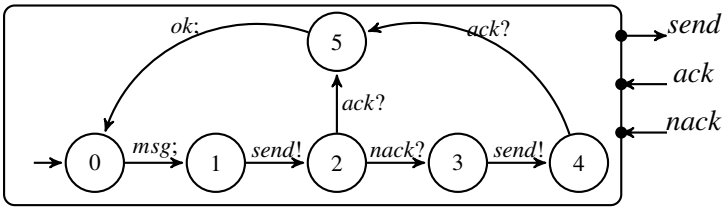


Figure 3.5: User || Comp.

3.5.1/ Operations on interface automata

The synchronous product is used to capture the parallel execution of two components represented by their interface automata. Before computing the global behaviour of the two components, it is mandatory to verify if they can be assembled by testing their composability. Two interface automata A_1 and A_2 are composable if:

$$\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^H_{A_2} = \emptyset.$$

Definition 2: Synchronous product

The synchronous product between *two composable interface automata* A_1 and A_2 is defined as:

$$A_1 \otimes A_2 = \langle \; S_{A_1 \otimes A_2}, \; I_{A_1 \otimes A_2}, \; \Sigma^I_{A_1 \otimes A_2}, \; \Sigma^O_{A_1 \otimes A_2}, \; \Sigma^H_{A_1 \otimes A_2}, \; \delta_{A_1 \otimes A_2} \; \rangle$$

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ 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
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

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 shared actions between A_1 and A_2.

The synchronous product of two interface automata may contain illegal states. This can happen when, at a given state of the parallel execution a component is ready to send a shared action, and the other component doesn't anticipate this, and it doesn't perform a reception of this shared action.

Definition 3: Parallel composition

The composition of two interface automata A_1 and A_2 is denoted by $A_1 \parallel A_2$, it is computed by eliminating from the product $A_1 \otimes A_2$ the illegal states and all states reached from these illegal states by enabling output and internal actions.
The set of *illegal states* of two interface automata A_1, A_2 is defined as :

$$Illegal(A_1, A_2) = \left\{ (s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2). \begin{pmatrix} a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2) \\ \vee \\ a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1) \end{pmatrix} \right\}$$

We define by $\Sigma^I_A(s_1)$ and $\Sigma^O_A(s_1)$, respectively, the set of input and output actions at the state s_1.

Definition 4: Compatibility

Two interface automata A_1 and A_2 are compatible iff $A_1 \| A_2 \neq \emptyset$.

From the definition of compatibility, we can deduce that two interface automata are compatible if the result of their parallel composition contains at least one state.

3.5.2/ Refinement of interface automata

The refinement relation can be defined as alternating simulation [dAH01]. An interface automaton P refines an interface automaton Q, if all input steps of Q can be simulated by P and all the output steps of P can be simulated by Q. We need some preliminary notions:

Given an interface automaton P and a state v $\in S_P$, the set ε-closure$_P$(v) is the smallest set U $\in S_P$ such that:

- v $\in$ U and
- if u $\in$ U and (u, a, u') $\in \delta_P$ and a $\in \Sigma_P^H$ then u' $\in$ U.

The ε-closure of a state v consists of the set of states that can be reached from v by taking only internal steps.

Consider an interface automaton P, and a state v $\in S_P$. We let:

- ExtEn$_P^O$(v)= { a | $\exists u \in \varepsilon - closure_P(v).a \in \Sigma_Q^O(u)$ }, and
- ExtEn$_P^I$(v)= { a | $\exists u \in \varepsilon - closure_P(v).a \in \Sigma_P^I(u)$ }

be the sets of externally enabled output and input actions, respectively, at v.

Consider an interface automaton P and a state v $\in S_P$. For all externally enabled input and output actions a $\in$ ExtEn$_P^O$(v) $\cup$ ExtEn$_P^O$(v), we let:

- ExtDest$_P$(v, a)= {(u,a,u')$\in \delta_P$. u $\in \varepsilon - closure_P(v)$}.

Using this notation, the alternating simulation on interface automata is defined as follows:

Definition 5: Alternating simulation

Consider two interface automata P and Q. A binary relation $\succeq \subseteq S_P \times S_Q$ is an alternating simulation from Q to P if for all states $v \in S_P$ and $u \in S_Q$ such that $v \succeq u$, the following conditions hold:

- ExtEn$_P^I$(v) $\subseteq$ ExtEn$_Q^I$(u) and ExtEn$_P^O$(u) $\subseteq$ ExtEn$_P^O$(v).
- For all actions $a \in ExtEn_P^I(v) \cup ExtEn_Q^O(u)$ and all states $u' \in ExtDest_Q(u, a)$, there is a state $v' \in ExtDest_P(v, a)$ such that $v' \succeq u'$.

Thus, we can define the refinement relation between interface automata as follows:

Definition 6: Refinement

There is a refinement relation between two interface automata P and Q, if there is an alternating simulation between their initial states.

3.6/ Coloured Petri Nets

Coloured Petri Nets (CPNs) preserve useful properties of Petri nets and at the same time extend initial formalism to allow the distinction between tokens [Jen96]. In CPNs, a token has a data value attached to it. This attached data value is called token colour.

Definition 7: Coloured Petri Net

Formally, a CPN is defined as a tuple:

$$\langle\ P,\ T,\ A,\ \Sigma,\ C,\ N,\ E,\ G,\ I\ \rangle$$

Where:

- P: is a set of places.
- T: is a set of transitions.
- A: is a set of arcs.
- Σ: is a set of colour sets defined within CPN model. This set contains all possible colours, operations and functions used within CPN.
- C: is a colour function. It maps places in P into colours in Σ.
- N: is a node function. It maps A into $(P \times T) \cup (T \times P)$.
- E: is an arc expression function. It maps each arc $a \in A$ into the expression e.
- G: is a guard function. It maps each transition $t \in T$ into guard expression g. The output of the guard expression should evaluate to Boolean value true or false.
- I: is an initialization function. It maps each place p into an initialization expression i.

3.7/ Conclusion

In this chapter, we have presented the transformation of models, its types and its objectives, where we have seen that the major objective is to prepare system models for a verification phase. In our work, the transformation concerns SysML models and exactly SysML sequence diagram. In a stage of our work, we need to transform sequence diagrams into interface automata, that is why we have reserved a section in this chapter to present the different operations that can be done on interface automata. In another stage, we need to transform our sequence diagrams into coloured Petri nets. Because our resulted models of transformation will be the input of an adaptation process, we will give in the next chapter an overview of component-based software engineering and the works have already been done to adapt software components.

4

CBSE and Component Adaptation

The software engineering (SE) is interested especially by the working methods and the systematic procedures that allow the development of large software, which respond to customers expectations [MR07]. It searches for reliability and good performances of software [Jal08], while respecting time limitation and cost constraints.

The development of large software requires a lot of effort and asks for cooperation of many entities, which makes crucial the decomposition of the development on several phases. The organisation of these phases is guided by the chosen development method. The common phases between these methods are essentially the analysis, the design, the implementation and the test phases. The analysis allows capturing the different requirements, without making reference to how these requirements will be satisfied. The design phase defines how the requirement, specified in the last phase, will be achieved. A language must be used (such as UML, SysML) to represent the different solutions. The implementation phase allows coding the different proposed solutions, and the test phase allows verifying that the specified requirements and the searched quality are respected by the final product (see Figure 4.1).

Another way to organize the development is by decomposing the software on a set of components, developing from scratch some components, reusing others and adapting them. In the remainder of this chapter, we will talk more on the notion of components and their

Contents

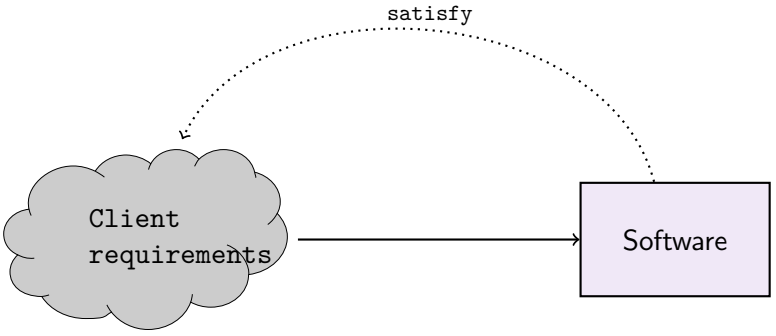


Figure 4.1: The goal of software engineering.

adaptation.

This chapter is organized as follows: We will expose in sections 4.1, 4.2, 4.3, 4.4 and 4.5 definitions of Component-Based Software Engineering (CBSE), software component, abstraction of a component, the notion of interfaces, and we give examples of component models. In section 4.6 and 4.7, we present some works which concern the verification of assembled systems. In section 4.8, we present some works that have already been done in the context of component adaptation, we present also a taxonomy of adaptation techniques. Finally, in section 4.9, we conclude.

4.1/ Component-Based Software Engineering

The Component-Based Software Engineering (CBSE) is a branch of the software engineering which is based on the separation of preoccupations. It has emerged in 1990 as an approach which bases on reusing entities called 'software components'.

At any time, the system can express new needs to new services. However, the fact of seeing and developing the system as one unit constitutes a barrier in front of its evolution, where it will be very difficult to specify the system parts which are altered by each evolution. Also, the verification of the system after modification will be more and more complex, because the totality of the system will be concerned. In fact, the disadvantages of this approach have changed the manner of designing and developing these systems. That is what justifies the trend of the new approaches which take a system as a set of basic units such as CBSE approach.

Thus, the CBSE is seen as the process of defining, implementing, assembling and integrating independent components or weakly coupled components. It searches essentially to make the construction of software more easier by assembling pre-existed components. Thus, it aims to the reuse, where a component can be used by several systems, which reduces the cost and the time of development. This approach also contributes to control the evolution of the system and its maintenance, where the altered parts by maintenance can be defined in term of components (see Figure 4.2).

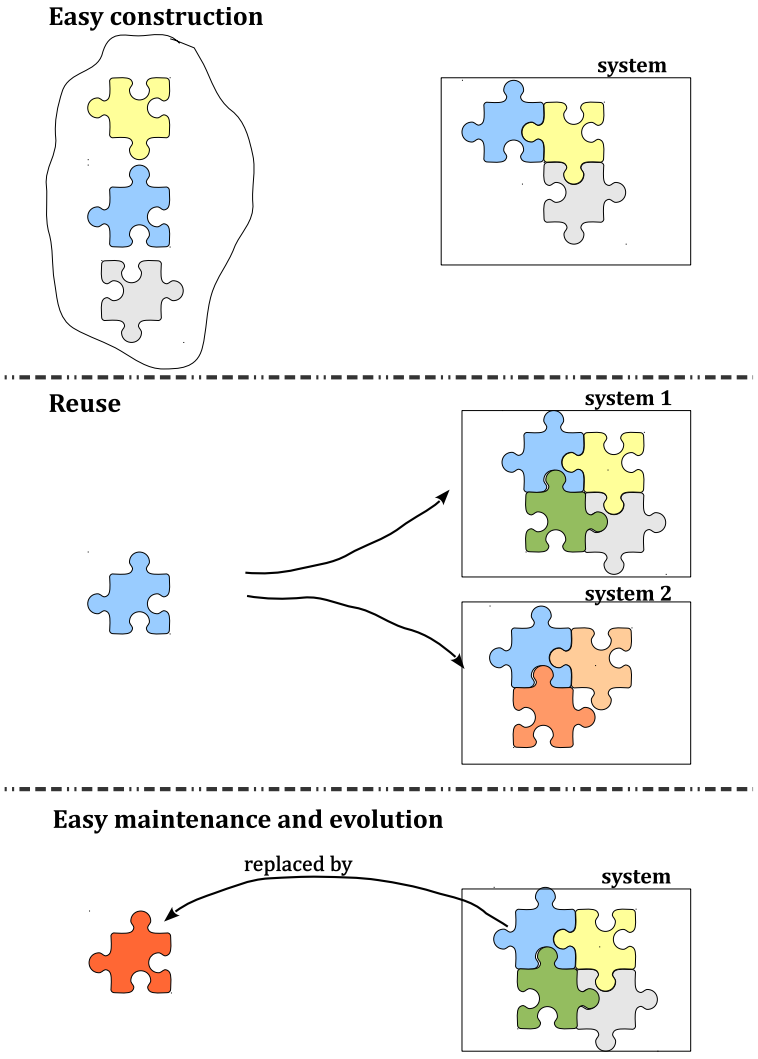


Figure 4.2: The goals of CBSE.

4.2/ Definition of Software Component

In literature, there is several definitions of the software component term. Among these definitions, we choose the one cited in [Mic95], in which the component is defined as a binary and reused piece of software which can be used and connected with other components. In [Szy98a], Clemens Szyperski defined the software component as a unit of composition with contractually specified interfaces and explicit context dependences. According to Szyperski, a software component can be deployed independently and can be the subject to composition by third-parties.

From the definition of Szyperski, we can extract the essential characteristics that must be fulfilled by a software component:

- Standardized: this means that the component must conform to a model. This model can define the interfaces of the component, its meta-data, its documentation, its

composition ad its deployment.

- Independent: The component can be composed and deployed without having need to a specific set of components. When a component, in the new system to which it was composed, requires a set of exterior services, this need can be specified using the required interface of the component.

- Composable: For a component to be composable, all its external interactions must take place through publicly defined interfaces

- deployable: To be deployable, the component must be able to operate as stand-alone entity on some components platforms that implement the component model.

- Documented: A component must be fully documented to allow to potential users to decide whether a component can satisfy their needs or not.

A component is considered as a black box. Thus, it must be equipped by a set of interfaces which allow to the component to interact with its environment, it must provide essentially a description of its interfaces set. More precisely, it has to specify what it can offer and what it waits from its environment. Each interface must be provided with a contract that specifies the mode of use and the constraints according to which the functionalities of the component will be executed [Szy98a].

4.3/ Abstraction of Components

The abstraction allows hiding the detail of implementation through the use of interfaces. The most ideal abstraction is that where the environment doesn't need any detail about the implementation of the component interfaces, which justifies the notion of 'black box' component. The components interact using their input and output actions. Input actions represents procedures or methods that can be called, and the receiving end of communication channels, as well as the return locations from such calls. Output actions are procedures or method calls, message emissions, the act of returning after a call or method terminates, and exceptions that arise during method execution [dAH01]. The verification of these components consists on evaluating their outputs in function of their inputs.

Figure 4.3: Black-box component.

Often, knowing some detail about the component activities is necessary to have more interactive component, so called grey box component. In this case, the component can give more detail concerning, for example, the conditions under which the exterior services are called.

4.4/ Component Interfaces

In the approaches where the components are considered as black-boxes, the definition of the components interfaces is very important to make them able to communicate and collaborate with others entities. The interfaces must be defined very clearly to give a good description of the roles of the components. The description of interfaces must be separated from the components implementations [Crn02, Szy98b]. This separation helps to (i) replace the implementation without changing the interface, (ii) improve the system performance rather than rebuild it, and (iii) add new interfaces without changing the existed ones.

A component must be, at least, equipped by one type of interfaces, required interface or provided interface. The provided interfaces of a component represent the services that the component can offer to its environment. However, the required ones are the imported interfaces of the other components. They summarize the services that the component requires from its environment. Each service is represented concretely as an operation (see Figure 4.4).

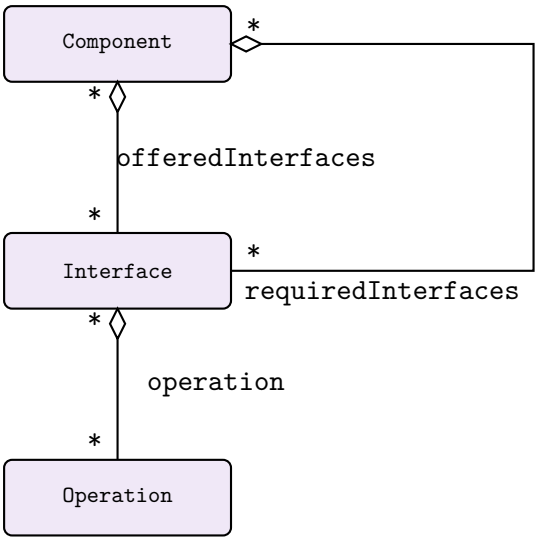


Figure 4.4: UML sub-meta-model of syntactic specification of a software component.

4.5/ Component Models

The most fundamental problem of component-based development is how a system can be constructed by assembling and making interoperate a set of components conceived by different providers. Thus, the standardization of component models, that the components must respect, appears indispensable. That's why, many component models have seen the light,such as CCM, EJB, etc. According to Rainer Weinreich and Johannes Sametinger [WS01], a component model must define the standards set that will reign the design and the implementation of components. These standards concern the naming, interoperability, personalisation, composition, evolution and the deployment. A component model defines also the standards which specify the implementation of the component model on a given platform and the set of executable software entities which are necessary to support

the execution of a component which conforms to the model.

A component model must define at least three elements, the syntax of components (i.e. how they are constructed and represented), their semantics (i.e. what the role of the component in the assembled system), and their composition. There exists many component models for different application domains. The industrial models (e.g. EJB (Entreprice Java Beans)[Sun06], CCM (CORBA Component Model) [OMG06], COM (Component Object Model) et .NET [M. 07]) provide the extra-functional services in containers, such as the security and the persistence services, which discharge the developers from defining these extra-functionnal services and allows them to concentrate on the business logic of their applications. These component models have different views on the component notion. Some models are used only for the architectural design without any execution support (e.g. UML 2.0). Some models define the component as a run-time entity (i.e. COM/NET [M. 07], Fractal[BCL+04, BCL+06]), however, there are other models that consider it as a design entity (i.e. KobrA [ABM00], SOFA [PV02, BHP06], PCM [BKR07]).

The most well-known of such component models are CORBA and EJB [Sun06]. The traditional CORBA Component Model, as it is defined in CORBA 2.4 [OMG00] and its anterior versions, have many limitations. Among its major limitations, the lack of a standard to deploy objects on the server side, it also lacks of mechanisms of managing objects life cycles. In these versions, there is no separation between the functional and no-functional requirements. To overcome these limitations, the CCM (CORBA Component Model) have seen the light. It defines the functions and the services that allow the implementation, the management, the configuration and the deployment of components. The most famous component model is, without doubt, CORBA, which is the most appropriate for distributed applications. However, the cumbersome and the complexity of this model constitute its major disadvantages. The model EJB is more restrictive and more efficient. It bases essentially on JAVA, and it has earned an important part of the market [Mou11].

4.6/ Verification of Component Compatibility

Building a system by assembling a set of components implies the adoption of some precautions during the assembling time. First of all, the components must be compatible, after that, it is necessary to check that these components fulfil the requirements to which they are dedicated. The compatibility check of components is performed by computing the compatible states of their parallel execution. According to the optimistic approach of

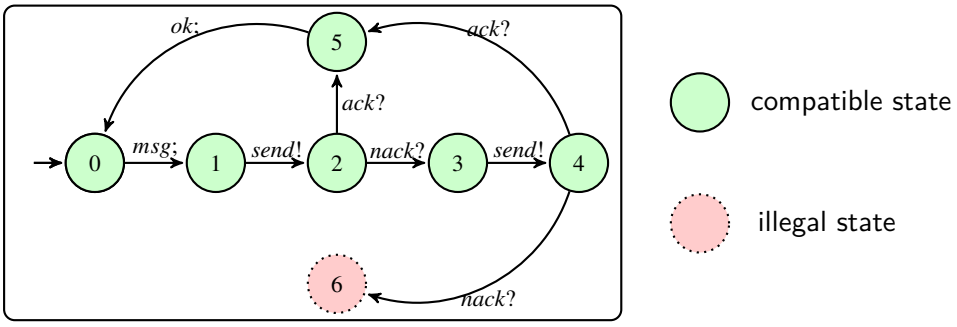


Figure 4.5: User ⊗ Comp. The set of compatible states is not empty

de Alfaro and Henzinger [dAH01], which is defined on interface automata, two compo-

nents are compatible if the parallel composition of their interface automata is not empty. This means that there is at least one compatible state in their synchronous execution. To illustrate that (see Figure 4.6), we reuse the example presented in Figure 3.5.

A decision about the compatibility or not of components must be taken on several different levels. The first level concerns the signature of component services. In [CPS06a], the authors insist on the fact that the signature of a component is no longer sufficient to ensure the good interaction with other components. The definition of signature, which is resumed in a presentation of the offered and required services of components, must be duly completed by a specification of the interaction protocol of the components with their environment [dAH05], [CCM12a]. Also, Bordeaux et al. in [BSBM04], have insisted on the fact that the compatibility of web services depends not only on static properties like the correct typing of their message parameters, but also on their dynamic behaviour, where providing a simple description of the service behaviour, based on process-algebraic or automata-based formalisms, can help detecting many subtle incompatibilities in their interactions.

Teixeira in [TeS11] proposed an approach to evaluate the compatibility of components specifid in UML, they use the state machine diagram to describe component behaviours which are then translated to a Petri net to identify compatibility problems.

Carrillo et al., in [CCM12a], have proposed an approach that allows combining the semi-formal models of SysML and the formal models (interface automata) to verify the consistency and the compatibility of SysML blocks.

In the component domain, there is many approaches that have targeted the component compatibility issue, and the manner to make them cooperate. The main difference lies in the formalism which is used to model the interactions of components with their environment. The choice of a model among others, is guided and justified by the expressiveness of the model semantic and the properties to verify. In [EPK02] and [KEP07a], the authors have used PRES+ [CEP00], which is a variant of timed Petri net. They justify this choice by the capacity of PRES+ to capture intuitively the concurrency and real-time aspects, and can expose both the data and the control flow of the system. The process algebra is used into many works (e.g. [BBC05a] use a subset of Pi-Calculus [MPW92], [AG97] use CSP [Hoa85]). This choice of process algebra is justified by its capacity to feature more expressive descriptions of protocols, enable more sophisticated analysis of concurrent systems, and support formal derivation of safety and liveness properties.

In our approach for verifying SysML blocks compatibility, we have tried to benefit from the hierarchy present in the system models. We have based on Hierarchical Interface automata (HIAs) as formal hierarchical formalism that allows modelling the interactions of each block with its environment using composite and nested states. After, basing on some relations between the HIAs of blocks that we want to verify their compatibility, we select some states to flatten and others to consider as abstract states. We have introduced this step into our approach to alleviate the compatibility verification.

4.7/ Formal Analysis of Assembled Systems

It is very interesting to make call to formal methods to verify the result of assembling a set of components to construct a system [KEP07a]. The use of formal methods appears worthwhile because it allows spotting the interaction gaps and the parallel execution defaults of

these components. A formal analysis can concern several errors, such as the deadlock, the unfairness, etc. It can also target the temporal properties [CGP99, Sch04, CCM14], such as the order of components services invocation, or verifying if when a component requests a service, the other components can always answer this request by offering the corresponding service. This verification can also concern the defaults in one component but after its assembling with others. In this case, it targets for example the local blocking in this component [BCS+08, CK96]. In [CK96], Cheung and Kramer present their approach for verifying if a component contains the anomalies by using some environment hypothesis. These hypothesis or constraints allow to enhance the verification by reducing the problem of state explosion. Because verifying a component, without having information about its execution context, makes the verification more difficult. This difficulty can be resumed by analysing no interesting scenarios, or by verifying the scenarios that can never happen.

There are several manners that allow the verification of the assembled systems. On one hand, there is those that take the system in its totality as one unit. They base on the traditional verification methods such as model checking [CGP01] and theorem proving [Sam76]. On the other hand, we find the methods which take advantage of the fact that the system is considered as a set of components [CIP04, XB03, CCM14, DOP13, DOP14]. In [CCM14], Carrillo et al. prove that if a temporal property is verified on a SysML block (a SysML block is the equivalent of a component in the component-based paradigm), it will be verified on the parent block which contains this block with others. There is also some methods that put some hypothesis on some properties of some components or of environment, to guarantee that other properties on other components will be verified. This method is called Assume-Guarantee [DOP13, DOP14]. It bases on iterative processes to prove that the initial hypothesis is always verified or it is verified in the context where the system will take place.

In our case, we assemble the system parts after adapting them using special blocks called adapters. To verify that a temporal requirement which is initially satisfied by a block, is still satisfied after assembling it with other blocks, we base only on a sub-set of the generated adapters. Later, in chapter 9, we give more details about our approach.

4.8/ Components Adaptation

When assembling separately developed components, there is a high probability of encountering the problem of mismatches. These latter may concern for example the name of services, as well as the order in which the component asks for (resp. offers) environment services (resp. its services). That is what justifies the introduction of third entities or components that are used to solve these mismatches. These components are called 'adapters'. Thus, the adapter plays the role of a mediator (see Figure 4.6) between the reused components. In fact, all the interactions pass through this adapter which acts as an orchestrator and allows the involved components to work correctly.

4.8.1/ Adaptation Taxonomy

In [CMP06], there is a classification of the different types of the adaptation that can be applied on software parts.This classification is based on a set of criteria, where each criterion produces two categories. The first criterion concerns the phase in the life cycle in which

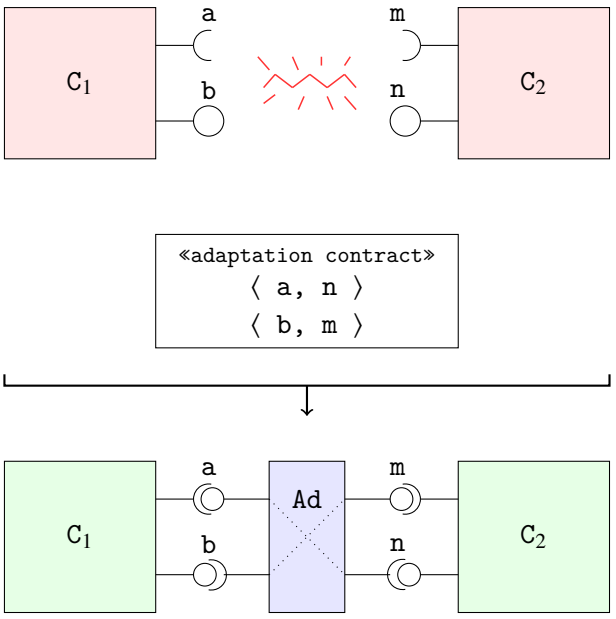


Figure 4.6: The role of the adapters.

adaptation takes place:

- Static adaptation (design time adaptation):

  This category concerns the adaptation done before the execution of the system. It can refer to both requirement (or model) adaptation, or to the adaptation of already developed pieces of code. The first case may happen when we need to extend the system specification by new requirements. Or, when we want to modify the requirements. However, the adaptation of already existed pieces concerns for example the modification of the manner whose the services are requested or offered. In the static adaptation, all steps are known and have been planned before proceeding with adaptation.

- Dynamic adaptation (runtime adaptation):

  It takes place in the execution time of the software. In this case, the components to adapt and the steps to follow are unknown before the the adaptation time.

Another classification is based on the the way in which the adaptation is managed. According to this criterion, the adaptation could be:

- Manual adaptation:

  The adaptation steps as well as the adaptors are specified and developed by the architects and developers involved in the development process. It can be assisted by some tools.

- Automatic adaptation:

  All adaptation steps and adaptors are automatically generated by software tools. These tools are able to deduce if its necessary to apply an adaptation. It is also to these tools to decide about the steps to follow for generating the adapter.

Yahiaoui et al., in [YTL04], have introduced an alternative criterion. This criterion concerns the kinds of the properties concerned by the adaptation:

- Functional adaptation:

  It concerns the organisation of required and offered services. It can involve both adding new services, and modifying the existed ones.

- Technical adaptation:

  In this case, the adaptation intends to modify the way and the constraints under which the services are provided. It is usually done by adding or removing constraints to the behaviour of these services.

Another criterion concerns the restriction of the component interaction protocol. According to this criterion, it is possible to distinguish two kinds of adaptation:

- Restrictive adaptation:

  It bases on reducing the interaction protocol of the components to remove the behaviours that lead to errors (as in [IS01, IT03a]). The idea consists on synthesizing a controller between components. Deadlocks detected on the controller are avoided by removing some branches. By this way, the controller enables the maximum set of interactions between the components which do not lead to deadlock states.

- Generative adaptation:

  It is more general, it takes into consideration the effect that when two components are developed separately, there is a great chance that they do not agree on the same name for their services. It may also have a problem in the combination of their protocols (parallel execution). Thus, it seems very essential to establish a mapping between their services. This mapping can take the form one-to-one [YS97, BCP04, BBC05a] or more complex form one-to-many [CMM10b]

4.8.2/ General Adaptation Process

The component adaptation process contains essentially three phases:

1. The specification of the components interaction protocols using suitable formalism (IDL: Interface Description Language): This formalism must be able to represent the concerned concepts by the adaptation phase.

2. The specification of the adaptation contract: IDLs of components must be extended by some adaptation rules which form the adaptation contract. When assembling two components developed separately, there is a high probability to confront the problem of mismatches between their services. Essentially, the adaptation contract in CBSE is used to solve this problem of mismatches between components [CS14]. An adaptation contract is specified by a set of rules, where a rule takes the form of a synchronous vector v_i [CPS08] (see Figure 4.7). The number of elements of each vector is the number of components. A synchronous vector v_i for a set of components ($\{C_i\}_{i\in\{1..n\}}$), is a tuple $\langle e_1, ..., e_n\rangle$ with e_i can belong to the set of actions of the component C_i, or it can be equal to ε. ε means that the component C_i doesn't participate in

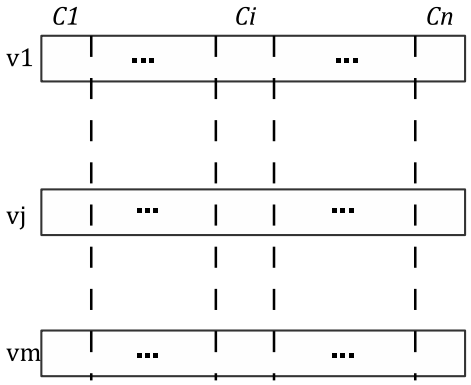


Figure 4.7: The adaptation contract.

this synchronization. For example, the vector v= $\langle \varepsilon, \ldots, e_j, \ldots, e_k, \ldots \varepsilon \rangle$ means that the service e_j of the component C_j corresponds to the service e_k of the component C_k.

3. The generation of the adapter: It can be done automatically. Basing on the adaptation contract and the IDLs of components, the adapter will be generated.

4.8.3/ Principal Adaptation Approaches

In literature, there are many approaches for adaptation. The most known differ essentially in the formalisms used to describe the interaction of components with their environment. In this section, we will present briefly two approaches, the first one uses the π-calculus process algebra and the second bases on the labelled transition systems and Petri nets.

Adaptation of π-calculus protocols

In [BBC05a], the authors have proposed an approach based on process algebra to generate automatically the adaptors. In this work, the specification of the component interfaces is extended by a description of the interaction protocols using π-calculus. The used variant of *pi*-calculus is the following:

```
E::= 0 | a.E | (s) E | [x=y]E | E||E' | E+E'
a::= τ | x?(d) | x!(d)
```

The input and the output actions, which represent the requests of services and the reception of these requests, are represented using the notation x!(d) and x?(d), where x is the name of the action, and d represents a set of parameters or data emitted or received through x. The internal actions (no observable actions) are represented using τ.

The actions are included in processes E where 0 represents the inaction, (x)E represents the creation of new name x in the process E. the operator [x=y]E represent the conditional behaviour, [x=y]E can be translated by E if x=y, and by inaction in the other case. The choice operator + between E and E' is translated with the execution of E or E'. However the parallel operator || indicates the parallel execution of E and E'.

We will explain this approach through this example:

```
role Client={
     signature request! (Data url); reply? (Data page);
     behaviour request! (url) . reply? (page).0
}
role Server={
     signature query? (Data url); return! (Data file);
     behaviour query? (url) . return! (file).0
}
```

The Client is a web browser, where the user can enter an URL and send a request to visualize the web page or to open a file. The Server is a web sites host which contains an application that allows to receive the requests of clients and send a web page or a file. The interaction behaviours of the Client and the Server are represented using π-calculus.

Basing on the correspondences between request! and query? and between reply? and return!, the adapter A that satisfies this specification can be defined as follows:

```
A=request?(url) . query!(url) . return?(file) . reply!(page).0
```

Adaptation based on LTSs and Petri nets

The work presented in [CPS06a] is based on using regular expressions and labelled transition systems to specify the component protocols. The correspondences between component services are expressed using synchronous vectors. However, the adapter specification is represented using regular expressions which can be represented as a LTS that has the synchronous vectors as labels of its transitions.

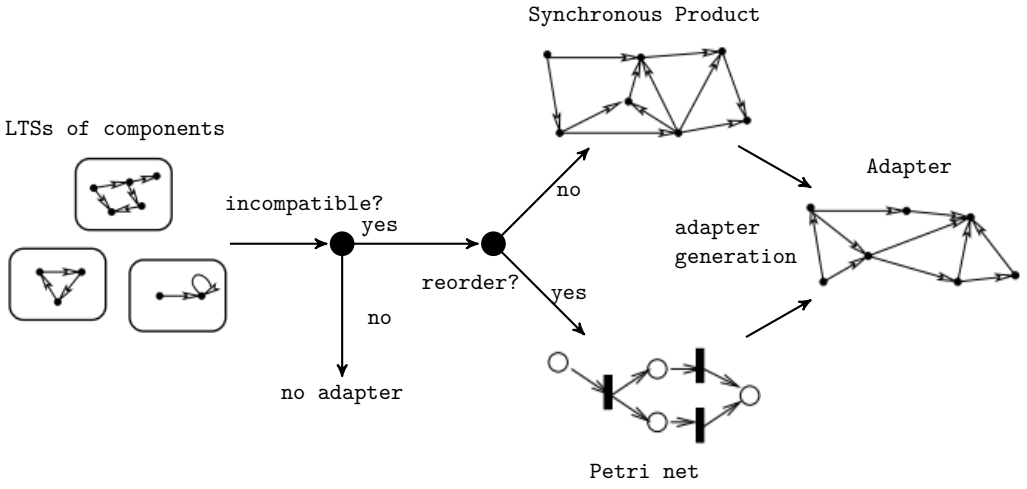


Figure 4.8: Adaptation approach [CPS06a].

The automatic generation of the adapter can be done with two manners:

- Adaptation without reordering: In this approach, the adapter must respect the order of actions, and it must deliver the received action to the concerned component before executing any other action. In this case, the adapter is simply represented by

the mirror of the synchronous product of the LTSs of components. The mirror consists on inverting the type of actions, where the input (resp. output) actions become output (resp. input) actions.

- Adaptation with reordering: It consists in reordering the calls for services, the idea is to encode in a Petri net the set of the adaptations constraints. The places assigned to the states of the components (Control states) and the places called resource-places (the states having a label which finishes by '??' or '!!') must be created. After that, the transitions of the Petri net, their incoming and their outgoing arcs will be created by following the information manifested by the transitions of the LTSs of the components. This set of incoming arcs includes also the arcs that link resource-places with the transitions of components (they represent receptions of calls). However, the set of ongoing arcs contains also the arcs that link the transitions of the components with the resource-places (they represent the generation of a service call). After that, the LTS of the adapter and a set of 'tau' transitions will be used to links the resource-states where the components put their calls for services with the resource-places from where the components consume these calls. The Algorithm 1 details the creation of this Petri net.

Algorithm 1 Adaptation with reordering using Petri net

INPUT: $\{\langle A_i, S_i, I_i, F_i, T_i \rangle\}_{i=1..n}$ // the LTSs of the components
$\langle A_R, S_R, I_R, F_R, T_R \rangle$ //the LTS of the adapter specification

1: **for all** component i **do**
2: **for all** $s_j \in S_i$ **do**
3: - create a place Control-i-s_j
4: **end for**
5: -add a token in the place Control-i-I_i, where I_i is the initial state of i
6: **for all** a! $\in A_i$ **do**
7: - add a place ??a
8: **end for**
9: **for all** a? $\in A_i$ **do**
10: - add a place !!a
11: **end for**
12: **for all** (s,l,s') $\in T_i$ **do**
13: - add a transition t labelled with $\bar{l}$
14: - add an arc from the place Control-i-s to the transition t
15: - add an arc from the transition t to the state Control-i-s'
16: - if l has the form a! then add an arc from the transition t to the place ??a
17: - if l has the form a? then add an arc from the place !!a to the transition t
18: **end for**
19: **end for**
20: **for all** $s_R \in S_R$ **do**
21: - create a place Control-R-s_R
22: **end for**

```
23: -add a token in the place Control-R-I_R
24: for all (s_R,(l_1,...,l_n),s'_R) ∈ T_R do
25:    - add a transition t labelled with tau
26:    - add an arc from the place Control-R-s_R to the transition t
27:    - add an arc from the transition t to the place Control-R-s'_R
28:    for all l_i of the form a! do
29:       - add an arc from the place ??a to the transition t
30:    end for
31:    for all l_i of the form a? do
32:       - add an arc from the transition t to the place !!a
33:    end for
34: end for
```

The LTS of the adapter will be obtained by computing the marking graph (non recursive adapters) or the recoverability graph (recursive adaptors) of the resulted Petri net.

4.8.4/ Other Approaches

Several surveys have been done on existing works which have proposed solutions in the software adaptation area [CMP06, SEG09]. In the literature, many approaches [KEP07b, CPS06b, CMM12] have been proposed to adapt components designed separately. These approaches can differ, for example, in the formalism used to represent the interfaces of the components and to model their interaction protocols and the context of adaptation. There exist many frameworks that allow modelling components and establishing links between them such as Papyrus [pap], BIP [BBB+11], etc. Papyrus put at the disposal of designers a set of SysML diagrams that allows modelling the architecture, the behaviour and the requirements of components. BIP (Behaviour, Interaction, Priority) is a general framework that supports rigorous design of components. BIP language allows building complex systems by coordinating the behaviour of a set of components. The behaviour is described with Labelled Transition Systems (LTS) extended with data and functions written in C. In [BBJ+10], the authors have introduced the notion of dispatcher component. Their dispatcher links the required and the provided ports of BIP components, but it doesn't impose a scheduling on component execution, where the global interaction of these components is guided only by their execution scenarios, and the dispatcher plays its role roughly as a connector. In fact, most of these frameworks use the concept of connector to express the coordination between components. Nonetheless, connectors are stateless [BBB+11]. In our work, we use the connectors to express coordination between components of systems but after a stage of behaviour adaptation. In our work, the coordination between components cannot be assured without inserting special components called adapters that allow solving the mismatches between the name of component services, and they allow also restricting some coordination scenarios which are not essential for assembling these components. This restriction is allowed by respecting the specification of the interactions of the parent (of the reused blocks) block toward its environment.

In [CMP06], we find a census of the different methods and works that have proposed solutions on the software adaptation track. The authors have addressed the different antecedents and motivations that justify the interest given to this track, where the major objective is the creation of a software component market (the so called COTS : Commercial

Off The Shelf), where the developers can find some components for their applications. The authors opened a debate on the characteristics that must be present in a component, this debate can be summarized into the question: How can we offer components with specifications that will serve and help in the totality of CBSE process?

In [CPS06a], the authors insist on the fact that the component signature is no longer sufficient to ensure its good adaptation with other components. The definition of signature, which is summarized into a presentation of the offered and required services of the component, must be duly completed by a specification of the interaction protocol of the component with its environment. This protocol helps to order the interaction of this component, allows defining its functional semantic and gives an idea of its behaviour towards its environment.

Most adaptation proposals focused on solving the behavioural mismatches between abstract descriptions of software components [MPS12, CPS08, TI08, BBC05b]. In [TI08], Tivoli et al. present an approach that allows generating adapters by enforcing behavioural properties. The entry of the adaptation process is the set of MSCs (Message Sequence Charts) that represent the components to assemble, and the set of LTL properties (liveness and safety) that the resulted system must satisfy. The authors have adopted on a restrictive adaptation approach to generate the adapter. This approach consists in reducing the interaction protocol of the components by removing the behaviours that lead to errors. The idea consists in synthesizing an adapter that plays the role of a controller between components. Deadlocks detected on the controller are avoided by removing some branches. In this way, the controller enables the maximum set of interactions between the components which do not lead to deadlock states. Bracciali et al. in [BBC05b], have proposed their methodology for generative behavioural adaptation where components behaviours are specified using a subset of the π-calculus and composition specifications are specified with name correspondences. In [MPS12], Mateescu et al. have presented their on-the-fly approach to adapt a set of services initially modelled by STSs (Symbolic Transition Systems), where these STSs are extracted from the BPEL description of the services. A contract which contains a set of synchronization vectors is used to specify the correspondences between the interfaces of services. They perform a verification of this contract using CADP [GLMS13], a rich verification toolbox. They explain there, how it is possible to represent these STSs and LTS using LOTOS process algebra. They explain also their forward approach to eliminate, from the adapter, the states and the branches that can cause anomalies (e.g. livelock states) in the interactions of the adapter with web services. Canal et al. in [CPS08], show how to automatically generate a concrete adaptor from a specification of the component interfaces and the adaptation contract (which takes the form of a set of correspondences between the actions of components). Also in [CS14], Canal et al. have focused on asynchronous communication between components, where components exchanged messages via buffers. Our approach takes place in the context of synchronous communication, where we adopt a generative-restrictive approach to construct the adapter block that can solve name mismatches between blocks. We have used interface automata to formally specify the interactions of the reused blocks, where their parallel composition, which consists in eliminating the illegal states, allows us to generate adapters that avoid the blocking of components. Also, our adapter is generated according to the specification of the composite block that will contain the reused blocks (which is not the case for the works presented above), this specification can be seen as a reflection of the environment where the components will take place.

The approaches which intend to assemble the components can also differ in the direction

of the design: bottom-up vs to-down. We find in [CCM12b], Carrillo and al. adopt a top down approach, where they verify if a specification of a SysML composite block can be divided on a set of sub-blocks specifications, the authors didn't refer to the adaptation issue. In [IT03b, PST07], the authors have adopted a bottom-up approach, they construct the wanted system by assembling existing components. Thus, they start from existing components which represent the leaves of the system. They take components designed separately, hence to allow the correct interaction between them, they synthesize a third entity called adapter. Our approach, which we present in this paper, concerns SysML blocks as in [CCM12b], it is a bottom up approach like in [IT03b, PST07]. However in our process, we don't give only the mapping rules between actions like in [IT03b], and we don't give the specification of the adapter as in the works already done in [PST07]. But, we give the interaction protocol of the composite block which will include the reused blocks. This block represents the part of the system that the architect wants to develop and integrate to the system. Thus, the interactions of this block must be specified with respect of the role which it will play in the system where it will be integrated.

The main difference between the existing adaptation approaches concerns the detail given to generate the adapter. In [IT03b, CH11b, DBM14, BBC05b, TI08, CMM10a], the authors give only an adaptation contract which is resumed in a specification of the correspondences between the services of the blocks (see Figure 4.9 (i)). In fact, this will have an impact on the generation of the adapter that will contain all the possible interaction scenarios between the reused components, it can contain scenarios that they are not necessary for the cooperation of the reused components. However, in [PST07, CPS06a, CPS08,

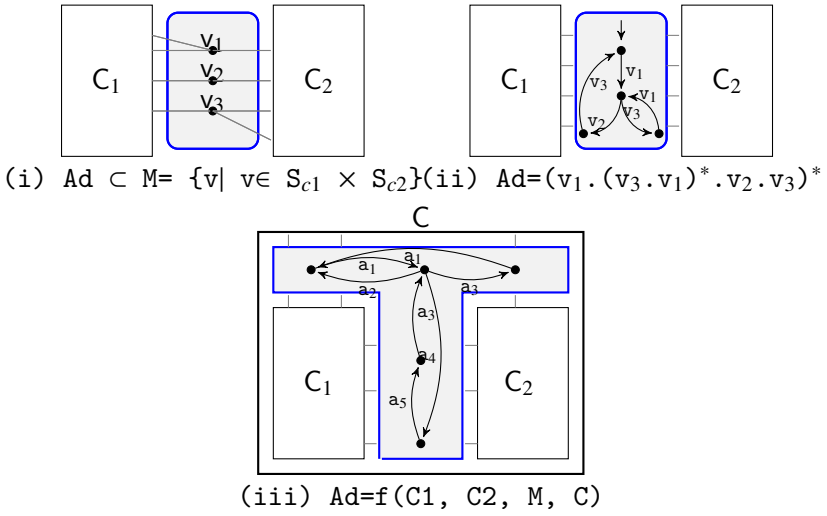


Figure 4.9: The difference between our approach *(iii)* and the existing approaches *(i, ii)* of adaptation.

MPS12], the authors increment their adaptation contract by a specification of the adapter interactions by ordering the vectors of the adaptation contract using regular expressions or a LTS of vectors (see Figure 4.9 (ii)). This requires that the developer, before making the specification of the adapter, must know very well the interaction of each component with its environment, and he must have an idea about the synchronous execution of the reused components. In this context, we ask the question about the detail that will be enough to generate adapters to make a set of components cooperate, where this cooperation must respect the intention behind their assembling? Therefore, in our approach (see Figure 4.9

(iii)), we propose to moderate the specification that must be given to generate the adapter. In our approach, the developer gives the interactions of the part of the system to be built and integrated to the system with its environment. The generation of the adapter is based on the specification of this part, by taking into consideration that this last will be the box that will include and represent the reused blocks during their interactions with the rest of the system.

In our approach, we don't use the conditions of consistency used by Carrillo et al. in [CCM12b] concerning the inclusion relation between the set of services offered and required by the composite block vs the set of services provided and required by its sub-blocks, because in the present work we take into consideration the possibility of making an adapter as a complement to achieve the specification of the parent block. In fact, our notion of adapter differs from the notion used in the existing works. In previous works, the adapter is defined as a protocol converter (it solves the problem of name mismatches between components). However, in our approach the adapter has two roles. It plays the first role as a converter between sub-blocks on the one hand, and between the reused blocks and their environment on the other hand. It plays the second role as a complement, i.e., when the environment of the reused blocks asks for a service, and this service is not offered by any of the reused sub-blocks, the adapter gives way thus to the developer to implement this service. Also, when a reused block needs a service and this last can't be offered by neither its sibling blocks nor by the environment of its parent, in this case the adapter gives way to the developer to define this service.

We can consider that our approach introduces a new branch to the taxonomy of component adaptation. In [BBC05b] , the adapter is defined binary, and in [IT03b] is defined system-wide. However, in our approach, the adapter is defined as a composite-block-wide adapter.

4.9/ Conclusion

In this last chapter of the state of the art part, we have presented some preliminaries concerning the notion of component. We have also presented some works which have given interest to the verification of systems developed as a set of components. We have focused more on the adaptation and the works that have targeted this issue.

In the next part of this thesis, we will present our contributions. They concern, as we have mentioned slightly in this chapter, the compatibility verification, the adaptation, and the verification of the assembled systems after adaptation.

II

Contributions

5

Formalizing SysML Diagrams

The goal of proposing and introducing SysML into the system engineering domain is to put at the disposal of designers a language that allows resuming the thousands of document lines (textual documents used to describe the requirements, the architecture and the behaviour of the system) using a set of convivial and graphical models. Thus, SysML was created to be a high level language that offers to designers sufficient design flexibility. However, this flexibility and no-formal aspect of SysML have contributed to the emergence of some ambiguities. These ambiguities can be summarized in the different interpretations of the semantic of its concepts and diagrams (e.g. there is no indication at the level of SysML diagrams about whether the communication is synchronous or asynchronous). In order to make the semantic of SysML diagrams that we will use more precise and clear, we propose, in this chapter, to formalize these diagrams.

In the rest of this chapter, we will give a formal definition of four SysML diagrams: In Section 5.1, we present the elements of the Requirement Diagram (RD). After, in Section 5.2, we give a formal definition of the Block Definition Diagram (BDD), where we focus on its different elements. Next, in Section 5.3, we formalize the SysML Internal Block Diagram (IBD). However, we reserve Section 5.4 for a behavioural diagram of SysML which is the Sequence Diagram (SD).

Contents

5.1/ Requirement Diagram (RD)

The requirement diagram, as its name indicates, allows us to represent the system requirements and the relations between them using graphical notation. In this section, we formally define the SysML requirement diagram.

Definition 8: Requirement Diagram (RD)

We formally define the requirement diagram as follows:

$$\texttt{RD= } \langle \texttt{ Req, AtReq, ExtElm, CRel, DRel, SRel, VRel } \rangle$$

Where:

- **Req:** is the set of all requirements.
- **AtReq:** is the set of atomic requirements, we have $\texttt{AtReq} \subset \texttt{Req}$. Each requirement belongs to this set can't be decomposed into other requirements. We have:

$$\texttt{AtReq=\{r}\in\texttt{Req} \mid \nexists(\texttt{r}_1,\ldots,\texttt{r}_n) \in \texttt{Req}^n, \ (\texttt{r}, \ (\texttt{r}_1,\ldots,\texttt{r}_n))\in\texttt{CRel}\}.$$

- **ExtElm:** is the set of external elements. The elements of others diagrams to which the requirements are linked. An external element can be a block, a test case, etc.
- **CRel:** is the set of all containment relations between requirements. This relationship enables a complex requirement ($\forall \ \texttt{r} \in \texttt{Req} \wedge \texttt{r} \notin \texttt{AtReq}$) to be decomposed into its containing child requirements. We have:

$$\texttt{CRel} \subset \texttt{Req} \times \mathcal{P}(\texttt{Req}).$$

- **DRel:** is the set of derivation relations («deriveReqt»). We have:

$$\texttt{DRel} \subset \texttt{Req} \times \texttt{Req}.$$

- **SRel:** is the set of satisfaction relations («satisfy»). A satisfaction relation specifies which block is responsible of satisfying the given requirement. We have:

$$\texttt{SRel} \subset (\texttt{ExtElm} \times \texttt{Req})$$

- **VRel:** is the set of verification relations («verify»), we have:

$$\texttt{VRel} \subset (\texttt{ExtElm} \times \texttt{Req})$$

  In this type of relation the external element is a set of scenarios represented by a state machine, sequence or activity diagrams, we call it a test case.

This formalisation helps in defining our approach for adapting SysML blocks and verify-

ing the set of requirements. It allows us also to exploit the relations between the requirements and the blocks during the verification phase.

5.2/ Block Definition Diagram (BDD)

The Block Definition Diagram (BDD) in SysML defines features of blocks and relationships between them such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties and operations [OMG12b].

5.2.1/ BDD Formal Definition

Definition 9: Block Definition Diagram (BDD)

Formally, we define a block definition diagram by the tuple:

$$\mathtt{BDD} = \langle\ \mathtt{B},\ \mathtt{R}\ \rangle$$

Where:

- $\mathtt{B}$: is the set of blocks that compose the system, where each block represents a part of the system that has its own properties and behaviour. We have:

  $$\mathtt{B} = \{\ \mathtt{B}_i \mid \mathtt{i} \in \{1..\mathtt{n}\},\ n \in \mathbb{N}\ \},$$

  where n is the number of blocks in the BDD.

- $\mathtt{R}$: is the set of relations between blocks. We have:

  $$\mathtt{R} = \{\ \mathtt{R}_i \mid \mathtt{i} \in \{1..\mathtt{m}\}\ ,\ m \in \mathbb{N}\ \} = \mathtt{Herit} \cup \mathtt{Ass} \cup \mathtt{Comp} \cup \mathtt{Aggreg}.$$

We will define the sets Herit, Ass, Comp and Aggreg later in this chapter.

- We have $\mathtt{B} \neq \emptyset$. This means that our system is composed at least of one block.
- if $\mathtt{Card(B)} > 1 \Rightarrow \mathtt{R} \neq \emptyset$, where Card: $\mathtt{B} \rightarrow \mathbb{N}$ is the function that returns the number of blocks which compose our system. This relation means that our blocks are not isolated, and there are some interactions between them.

5.2.2/ Block

The blocks are modular units of the system description. The Block represents the basic structural element of the BDD. It may include both structural and behavioural features, such as properties and operations. To communicate with its environment, a block has a list of ports.

Definition 10: Block

Formally, we define a block as:

$$\texttt{Block} = \langle\ \texttt{name, Values, Operations, Constraints, Parts, References, Ports}\ \rangle$$

Where:

- `name : is the name of the block.`
- `Values:` is the set of attributes of the block.
- `Operations:` is the set of the operations of the block. It describes its internal behaviour.
- `Constraints:` this set gives some conditions about the values. A constraint may be a comparison or an equation for computing a value of the block.
- `Parts:` this set includes the list of the blocks connected with the current block through a composition relation.
- `References:` this set includes the list of the blocks connected with the current block through a navigable association. We mean by `navigable association`, an association which has one direction. In our case, its direction is from the current block to the referenced blocks.
- `Ports:` is the list of the ports positioned on the block used to interact with the blocks which belong to its environment. We define by

  $$\texttt{BlockPorts:B} \rightarrow \texttt{Ports},$$

  the function that returns the set of ports associated to a given block.

5.2.3/ Ports

The ports allow to block to interact with the other blocks. In SysML 1.2, we distinguish flow ports and standard ports. SysML 1.3 has deprecated the flow specification. However, it has focused more on standard ports. Formally, we define a standard port by its name, its type and its direction. The type of the port is represented by a block containing a list of operations, we call it `"interface_Block"`. The direction specifies if the port is a `required` or a `provided` port, it takes two different values: `req` or `prov`.

we formalize a standard port as:

$$\texttt{port=}\ \langle\ \texttt{name, type, Direction}\ \rangle$$

A block which types a port contains a set of operations, we call it `Interface_Block`, it specifies one of the interfaces associated to the block.

Definition 11: Interface_Block

We define an Interface_Block as follows:

$$\texttt{Interface\_Block= } \langle \texttt{ name, Op } \rangle$$

Where:

- `name:` is the name of the interface_block.
- `Op:` is the set of provided operations by the block, which are provided through the provided port whose type is this `interface_block`. It can also be the set of required operations which are required through the required port whose type is this `interface_block`.

5.2.4/ Parts

The list of parts of a block contains the list of blocks which are connected to the current one by means of a composition or an aggregation relations.

We have:

- The set of parts of a block is included in the set of the BDD blocks: $\texttt{Parts} \subseteq \texttt{B}$.
- The function $\texttt{BlockParts} : \texttt{B} \rightarrow \texttt{B}^n$ returns the set of parts of the block passed in parameter. It returns thus the set of blocks which are targeted by composition relations that start at the current block.
- $\forall\ \texttt{B}_i, \texttt{B}_j \in \texttt{B}\ (\texttt{B}_j \in \texttt{BlockParts}(\texttt{B}_i)$
  $\Rightarrow \exists\ \texttt{r} \in (\texttt{Aggreg} \cup \texttt{Comp})\ (\texttt{Sides}\ (\texttt{r}) = (\texttt{B}_i\ ,\ \texttt{B}_j)))$

  Where, `Sides` is a function which affects to each relation $\texttt{r} \in \texttt{R}$ the two blocks which delimit this relation.

  $$\texttt{Sides} : \texttt{R} \rightarrow \texttt{B}^2$$
- The function `BlockParts` allows us to identify if a block is an atomic or a composite block.
- $\forall\ \texttt{B}_i \in \texttt{B}$, we have:
  - $\texttt{BlockParts}(\texttt{B}_i) = \emptyset \Leftrightarrow \texttt{B}_i$ is an atomic block.
  - $\texttt{BlockParts}(\texttt{B}_i) \neq \emptyset \Leftrightarrow \texttt{B}_i$ is a composite block.

5.2.5/ References

The compartment References represents the list of blocks that the current block interacts with, and in the same time, it indicates the Min and Max cardinalities which specify the minimum and respectively the maximum number of the instances of target block referenced by the current block. It uses also a boolean variable to indicate if these referenced instances blocks must be ordered or not.

We have:

- The function **References** : $B \rightarrow (B \times \mathbb{N} \times \mathbb{N} \times \mathbb{B})^n$ returns a set of tuple, where each tuple specifies: **(1)** a `block` which is referenced by the block passed in parameter, **(2)** the `min` and the `max` of instances of the referenced block, and **(3)** a `boolean` to specify whether this set of instances is ordered or not.
- We use the function ReferencedBlocks: $B \rightarrow \mathcal{P}(B)$ to represent the set of blocks which are referenced by the block passed in parameter:

$$\text{ReferencedBlocks}(B_i)=\{B_j \mid \exists\ e \in \text{References}(B_i) \wedge e(1)=B_j\}$$

- The existence of a block B_j in the set of references of a block B_j means that it exists a reference relation between them:

$$\forall\ B_i, B_j \in B\ (B_j \in \text{ReferencedBlocks}(B_i) \Rightarrow \exists\ r \in \text{Ass}\ (\text{Sides}\ (r) = (B_i\ ,\ B_j)))$$

- If there is a reference between two blocks B_i and B_j, this implies that it exists in each of them a port which are linked, and these two ports have a different direction.

$$\forall\ B_i, B_j \in B\ (B_j \in \text{ReferencedBlocks}(B_i) \Rightarrow \exists\ p_1 \in \text{BlockPorts}(B_i) \wedge p_2 \in \text{BlockPorts}(B_j) \wedge ((p_1.\text{Direction=prov}) \wedge (p_2.\text{Direction=req})) \vee ((p_1.\text{Direction=req}) \wedge (p_2.\text{Direction=prov})))$$

5.2.6/ BDD Relations

The set of relations in a BDD serves to construct bridges between blocks. Thus, from the definition of the different compartments of a block, we can deduce that a relation can express composition, aggregation, heritage and associations between system blocks.

1. **Heritage:**

   The heritage relations allow to simplify the development of system, and to create clear models by factorizing the shared elements between blocks.

   We have:

   - The function **Inherit** : $B \times B \rightarrow \mathbb{B}$, is a boolean function that allows us to know if the first parameter inherits the second, where the two parameters are blocks.
   - $\forall\ B_i, B_j \in B\ (\text{Inherit}(B_j, B_i) \Rightarrow \exists\ r \in \text{Herit}\ (\text{Sides}(r) = (B_i\ ,\ B_j)))$

2. **Composition and aggregation:**

   The composition and aggregation relations allow to preserve the principle of black box vs white box, which guides the development of system through multiple phases by transiting across abstraction levels. As we know, the difference between the composition and aggregation relations concerns the importance of the parts for their parents, where in the composition relation, the parent block can't exists without its parts. Hence, we can consider that the composition is a strong aggregation.

- The function **IsComponentOf** : $B \times B \rightarrow \mathbb{B}$, is a boolean function that allows us to know if the block passed in the first parameter is a component of the second block passed in the second parameter.

- The set of all blocks linked with composition or aggregation relations which starts from $B_i \in B$ is equal to the set of blocks which belong to the parts of B_i (BlockParts(B_i)).

$$\forall\ B_i \in B\ (\{B_j \in B |\ \text{IsComponentOf}(B_j, B_i)\} = \text{BlockParts}(B_i))$$

- if a port p_i is owned by a block B_i, and this port has no corresponding port in the blocks in the same hierarchy with B_i. Thus, we deduce that p_i is a port of the parent block (PB) which contains B_i (i.e. it exists a delegation between the composite block PB and the component blocks $\{B_i\}$).

  $\forall\ PB \in B, \forall\ B_i \in B, \forall\ p_i \in \text{BlockPorts}(B_i)\ ($

  $\text{isComponentOf}(B_i\ ,\ PB) \wedge \nexists\ p_j \in \bigcup\ \text{BlockPorts}\ (B_j)($

  $\text{Correspond}(p_i\ ,\ p_j)) \Rightarrow \Rightarrow \exists\ p \in \text{BlockPorts}(PB)\ (\text{Correspond}(p_i\ ,\ p)))$

  where **Correspond**: Ports x Ports $\rightarrow \mathbb{B}$ is the function that returns true when the ports in parameters are corresponding ports (two ports are corresponding ports if they are conceived to be linked together).

3. Association:

The Block Definition Diagram uses the associations with restraint navigability. This kind of associations expresses a directional connexion between two blocks.

- The function **Associated** : $B \times B \rightarrow \mathbb{B}$ is a boolean function that allows us to know if the block passed in the first parameter is the target of a navigation relation which starts from the block passed in the second parameter.

- We have, the set of all blocks linked to $B_i \in B$ with an association that starts from B_i is equal to the set of references of the block B_i (BlockReferences(B_i)):

  $\forall\ B_i \in B\ (\{B_j \mid \text{Associated}(B_j\ ,\ B_i)\} = \text{BlockReferences}(B_i))$

5.3/ Internal Block Diagram (IBD)

The Internal Block Diagram (IBD) in SysML captures the internal structure of a composite block B ($\text{BlockParts}(B) \neq \emptyset$) [OMG12b]. It represents the relations between the required and the provided ports of blocks instances. These relations are represented using the connectors.

Definition 12: Internal Block Diagram (IBD)

Formally, an IBD can be defined as a graph, where instances of blocks (parts) are vertexes and the connectors are edges.

$$\textbf{\textit{IBD}} = \langle\ \textbf{\textit{Parts, Ports, Connectors}}\ \rangle$$

Where:

- **`Parts`** is a set of instances of the blocks linked to the composite block B with composition relations.
- **`Ports`** is a set of ports. Each port is assigned to a part.
- **`Connectors`** is a set of connectors linking provided ports with required ports of blocks instances.

We can divide the set of connectors into two sub-sets, a set of delegation connectors (DC) and a set of standard connectors (SC). Thus, we have:

- `Connectors = DC ∪ SC`

- `DC`: is the set of connectors that link the composite block with its parts.

$$\texttt{DC= \{ } (p_i \texttt{ , } p_j) \texttt{ | } p_i \in \texttt{Ports} \wedge p_j \in \texttt{Ports} \wedge$$

$$((p_i\texttt{.Direction= req} \wedge p_j\texttt{.Direction=req}) \vee$$

$$(p_i\texttt{.Direction= prov} \wedge p_j\texttt{.Direction=prov}))\texttt{\}}$$

- `SC`: is the set of connectors that link the parts.

$$\texttt{SC= \{ } (p_i \texttt{ , } p_j) \texttt{ | } p_i \in \texttt{Ports} \wedge p_j \in \texttt{Ports} \wedge$$

$$((p_i\texttt{.Direction= req} \wedge p_j\texttt{.Direction=prov}) \vee$$

$$(p_i\texttt{.Direction= prov} \wedge p_j\texttt{.Direction=req}))\texttt{\}}$$

5.4/ Sequence Diagram (SD)

Definition 13: Sequence Diagram (SD)

We specify a SysML sequence diagram by:

$$SD = \langle\ Blocks,\ Msgs,\ CF,\ Ref\ \rangle$$

where:

- **Blocks:** is the set of blocks, they represents the actors involved in the interaction described by this SD.
- **Msgs:** is the set of the messages exchanged between the blocks.
- **CF:** is the set of combined fragment,

  $CF = \{\ (operator,\ operands)\ |\ operator \in \{alt,\ loop,\ par,\ opt,\ seq,\ strict,\ break\},\ operand = \{msg|\ msg \in Msgs\}\}$
- **Ref:** is like anchors which indicate that this sequence diagram includes other sequence diagrams. $Ref = \{SD_i\}$.

5.5/ Conclusion

In this chapter, we have formalized four SysML diagrams which are the Requirement Diagram (RD), the Block Definition Diagram (BDD), the Internal Block Diagram (IBD) and the Sequence Diagram (SD). We will use RD to represent the requirements of systems. The BDD and IBD will be used to model the architecture of systems. However, we will focus on the SD to model the behaviour (interaction) of the parts (blocks) of systems. The formalisation of these diagrams will help us to define our approaches of adaptation and verification without ambiguities. Thus, this chapter will be the reference in the next chapters when we deal with high level modelling.

6

A SysML Model Driven Approach to Verify Blocks Compatibility

In the component paradigm, the system is seen as an assembly of heterogeneous components, where the system reliability depends on these components compatibility. As we have mentioned in chapter 2, SysML uses the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD) to model the structure of the blocks (the components) and to establish links between them. To model the behaviour, SysML uses the State Machine (SM), the Sequence Diagram (SD) and the Activity Diagram (AD).

In SysML, the interactions between blocks are modelled using IBDs and SDs. These interactions take the form of architectural links in the IBDs. However, SDs, that interest us in this chapter, allow us to model the scheduling of these interactions using the life lines of blocks. Thus, the SDs constitute a good start point to verify the interactions inside the system. Since formal verification is still inapplicable directly on SysML models [BCHM15], therefore to apply a verification method, it's necessary to translate the SysML models into formal ones, and then verify the wanted properties.

In this chapter, the interactions of blocks are represented using a set of SDs. Each SD is associated with a block, and it describes the interaction scenarios of a block with its environment. To formalize the semantic of SDs, we transform them into interface automata (IAs) [dAH01]. As we have mentioned in chapter 3, IAs constitute a good formal model to represent the scenarios of requesting (output actions) and performing (input actions) services of a block. The composition of interface automata allows verifying some rela-

Contents

tions and properties on blocks such as the consistency and the compatibility (i.e. verify the existence of an environment where it's possible to connect these blocks).

Our approach of transforming SDs into IAs is mainly based on meta-modelling [dLVA04] and meta-model transformations [CH03]. Such approach consists on defining the meta-models of the source and the target models, and then specifying the correspondences between them in the meta-level. To avoid user errors during the transformation from the SDs to IAs, we have proposed an automated ATL [atl] grammar, which performs this transformation automatically. After, to verify some properties on the resulted interface automata, we have opted for Ptolemy [Pto] tool that requires as entry a textual specification. For this purpose, we have used Acceleo [Acc] to define a templates set on our meta-model of interface automata, that allows generating automatically the Ptolemy entry specification. Thus, this tool chain, that we have developed, can assist the architect during the compatibility verification phase by discharging him from doing many tasks.

The remainder of this chapter is organized as follows: In Section 6.1, we introduce our proposed approach. After, Section 6.2 gives details of transforming sequence diagrams into interface automata. Next, Section 6.3 presents the Acceleo templates that we have proposed to generate Ptolemy entry specification. In Section 6.4, we present how we verify the compatibility of the blocks. In Section 6.5, we illustrate our approach by a case study. Finally, in Section 7.5, we conclude.

6.1/ Our Methodology

Our approach aims to prepare the SysML blocks for the compatibility verification phase. We show an overview of our methodology in Figure 6.1. Thus, we start from sequence diagrams of the blocks that we want to verify their compatibility. After, by applying the ATL grammar, that we will expose later in this chapter, we obtain their equivalents of interface automata. For verifying the compatibility of the blocks, we use the Ptolemy tool. Ptolemy contains a module which allows the verification and the composition of interface automata. To discharge the user from redrawing the interface automata using the Ptolemy user interface, we propose a set of Acceleo templates to automatically generate the Ptolemy entry specification.

6.2/ Transforming SDs of Blocks into Interface Automata

In our work, the sequence diagrams are used to visualize the scheduling of the different interactions of each block with its environment. In the sequence diagram of a block B, the environment life line will represent the set of all blocks with which the block B can interact.

To transform the sequence diagrams into interface automata, we give the correspondences in this chapter. To implement these correspondences and to automatize the transformation, we propose a set of ATL rules. Our ATL grammar doesn't deal with combined fragments as an isolated units as in the works have already been done on Petri nets and the other kinds of automata. It deals with the different cases of nested combined fragments.

This ATL grammar is defined on the meta-model of sequence diagram as source and the meta-model of interface automata as target.

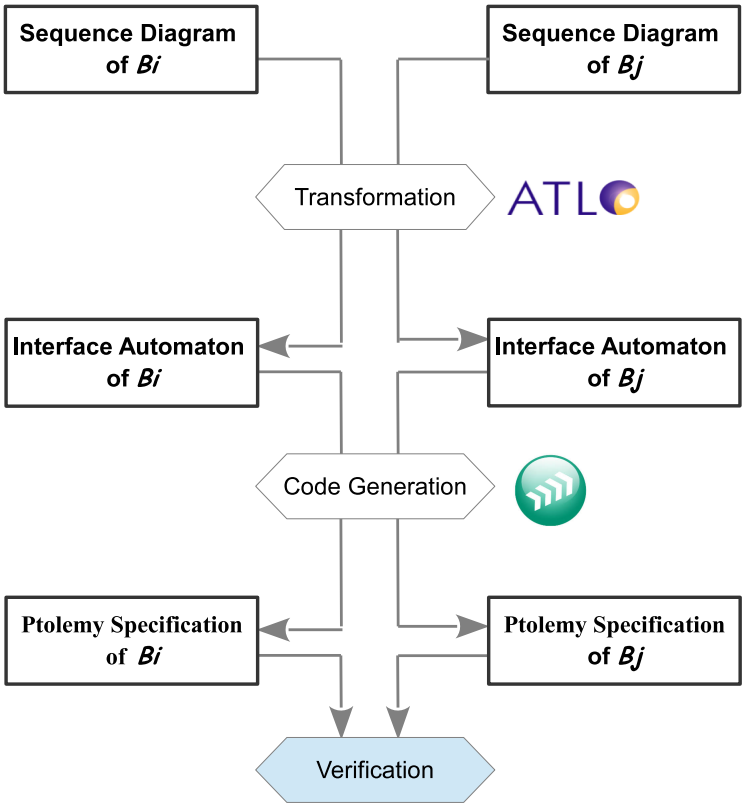


Figure 6.1: Our Methodology.

6.2.1/ Sequence Diagram Meta-Model

By intention to reuse existed modelling tools, we have used the sub-set of Papyrus [pap] SysML meta-model and its graphical editor to draw the sequence diagrams. In Figure 6.2, we represent the set of the classes of Papyrus meta-model that allows modelling sequence diagrams, and in Figure 6.3, we give an example of a sequence diagram which is modelled using Papyrus editor.

In Figure 6.2, the root class is the class Interaction. Sequence diagrams, that we will model, will be the instances of this class. Each interaction can include a set of life lines, a set of messages and finally a set of interaction fragments. The classes:

- **LifeLine:** each instance of this class represents an object which participates in the interaction. It will be the support of sending (resp. receiving) events executed (resp. intercepted) by the object.
- **Message:** defines the messages set interchanged between objects. Each message has two ends; a send end and a receive end.
- **InteractionFragment:** is the super class of the classes: Interaction, CombinedFragment, InteractionOperand and OccurenceSpecification.
- **CombinedFragment:** each combined fragment includes a set of interaction operands, and it has its own interaction operator. The interaction operator takes a value of this list [alt, opt, break, loop, par, ...].

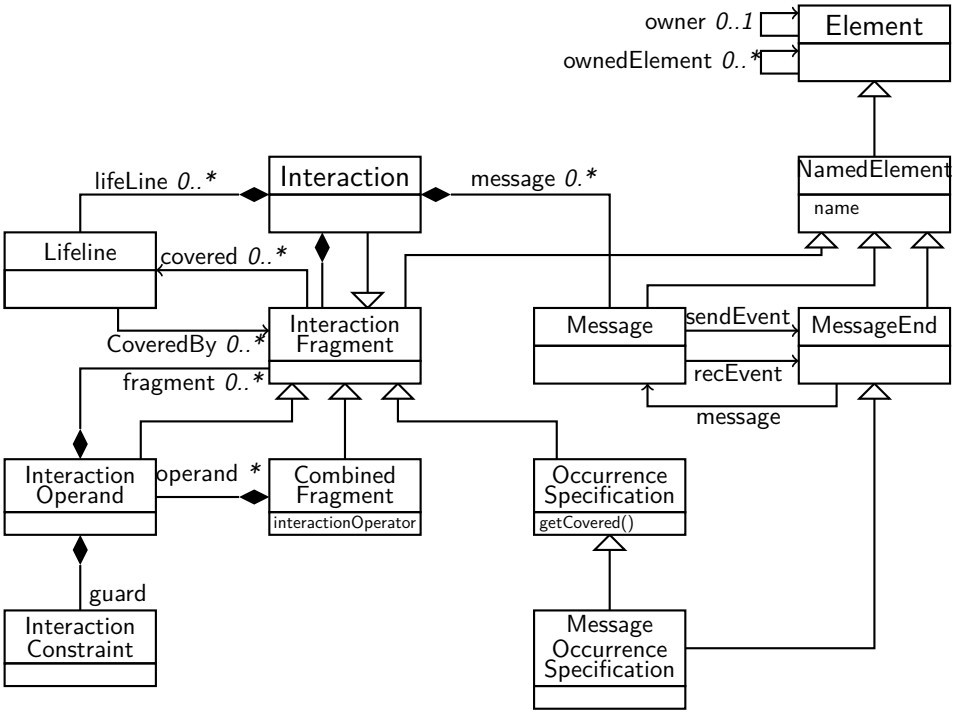


Figure 6.2: Papyrus Meta-Model of SysML Sequence Diagram.

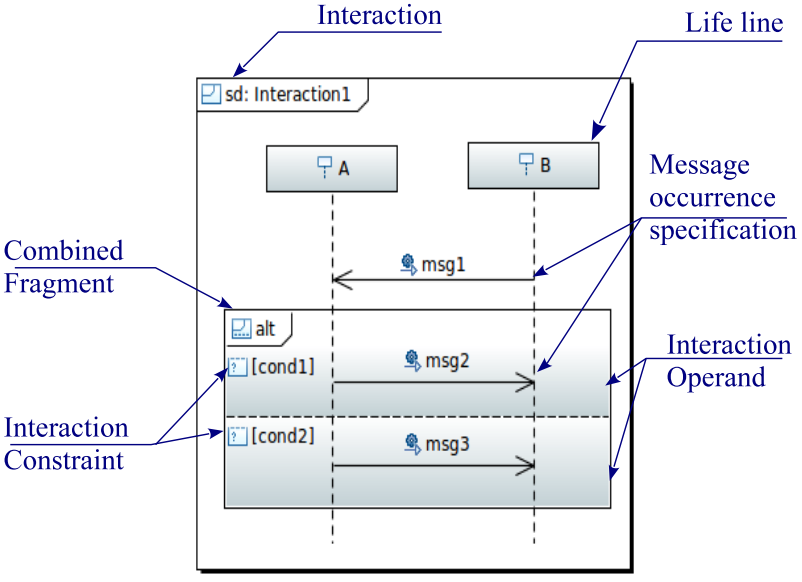


Figure 6.3: Sequence diagram elements.

- **InteractionOperand:** each operand is associated to a combined fragment, and it can have a guard.
- **MessageOccurenceSpecification:** Each event associated to the life line is represented as a message occurrence specification. it represents an extremity of a message. We can know the life line, to which the specification is associated, by executing the method getCovered() of the super class OccurenceSpecification. We can also obtain the message started or finished at this specification, by navigating through the

association message of the super class MessageEnd.

The classes MessageEnd, Message and InteractionFragment inherit the class NamedElement which its self inherits the class Element. The association 'owner' allows obtaining the father element of the current element, however the association 'ownedElement' allows us to obtain the children elements of the current element.

6.2.2/ Interface Automata Meta-Model

Basing on the formal definition of interface automata formalism given in section 2, we have proposed their meta-model in Figure 6.4. The classes:

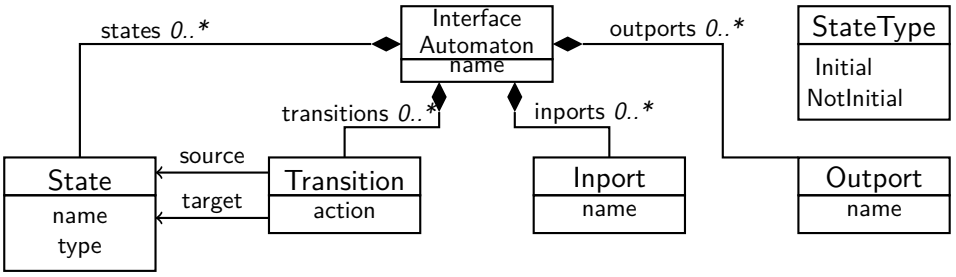


Figure 6.4: Interface Automata Meta-Model.

- **InterfaceAutomaton:** is the root. Each interface automaton has a name which represents the name of the block to which this automaton is associated. Each instance of this class can include a set of states, a set of transitions and a set of ports(in-ports, out-ports).
- **State:** Each instance of this class has a name and a type. The type allows specifying if this instance is an initial state or not.
- **Transition:** each instance of this class has three values to specify. The action which is the label of this transition, the source state and the target state.
- **Inport:** represent the ports associated with the input actions.
- **Outport:** represent the ports associated with the output actions.

In Figure 6.5, we present an interface automaton which is modelled using our editor. We have used the Graphical Modelling Framework (GMF) to specify and generate our graphical editor of interface automata.

6.2.3/ Basic Interaction Transformation Rules

An interface automaton specifies the interactions of a block 'B' with its environment, where the environment represents all the blocks with which the block 'B' can interact. Thus, the interface automaton will be associated to the life line of the block 'B'.

To perform the basic transformations (interactions without combined fragments), we have three rules:

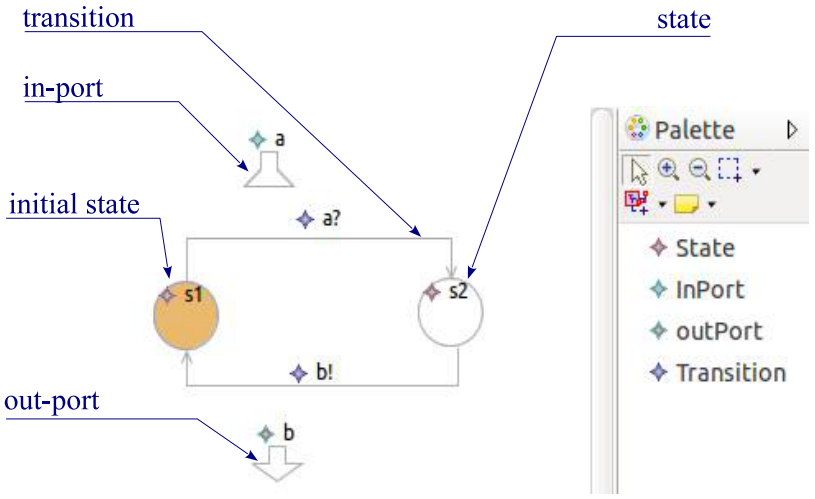


Figure 6.5: Generated Interface Automata Editor.

- **Rule 1:** LifeLine2InterfaceAutomaton

  This rule allows us to initialize the interface automaton which is associated to the block 'B'. The name of the interface automaton will be the name of the block 'B'. The in-ports are created using the helper '**createInports**', because we need to create one in-port for all messages having the same name, which are received by the block 'B'. To create the out-ports, we have used the helper '**createOutports**', which creates one out-port for all messages having the same name, which are emitted by 'B'.

ATL Rule 1:

```
rule LifeLine2InterfaceAutomaton {
from lifeline : SD!Lifeline (lifeline.name<>'ENV')
to   ia : IA!InterfaceAutomaton (
name <-lifeline.name,
states <-IA!State.allInstances(),
transitions <-IA!Transition.allInstances(),
inports <- thisModule.createInports(lifeline),
outports <- thisModule.createOutports(lifeline)
) }
```

- **Rule 2:** MessageOccurrenceSpecification2State

  We are only interested with events (sending and receiving of messages) associated to the life line of our block 'B'. These events (mos) are the instances of the class 'MessageOccurrenceSpecification', where their life line is not the environment but the current block 'B'(mos.getCovered ≠ ENV). Thus, must create a state for each message occurrence specification.

ATL Rule 2:

```
rule MessageOccurenceSpecification2State {
from mos :SD!MessageOccurrenceSpecification(mos.getCovered().name <> 'ENV')
to s : IA!State (name<-mos.message.name.concat('start'))
}
```

- **Rule 3:** Message2Transition

  A message, which has an extremity that starts or ends at the life line of the current block 'B', must be transformed into a transition in the interface automaton of 'B'. For a message $mos_1 \xrightarrow{mes} mos_2$ (where mos1 and mos2 are message occurrence specifications), we create a new transition. This transition will be labelled with the action 'mes', but to specify the type of this action and to fix the beginning and the end states of this transition, we must analyse three cases:

  – Only mos_1 is associated to the life line of 'B' (see Figure 6.6 (1)): In this case, the label will be an output action 'mes!'. The transition starts at the state associated to mos_1 and ends at the state associated to mos_i (the next message occurrence specification of mos_1 on the life line of 'B').
  – Only mos_2 is associated to the life line of 'B' (see Figure 6.6 (2)): In this case, the label will be an input action 'mes?'. The transition starts at the state associated to mos_2 and ends at the state associated to mos_i (the next message occurrence specification of mos_2 on the life line of 'B').
  – mos_1 and mos_2 are associated to the life line of 'B' (see Figure 6.6 (3)): In this case, the label will be an internal action 'mes;'. The transition starts at the state associated to mos_1 and ends at the state associated to mos_2.

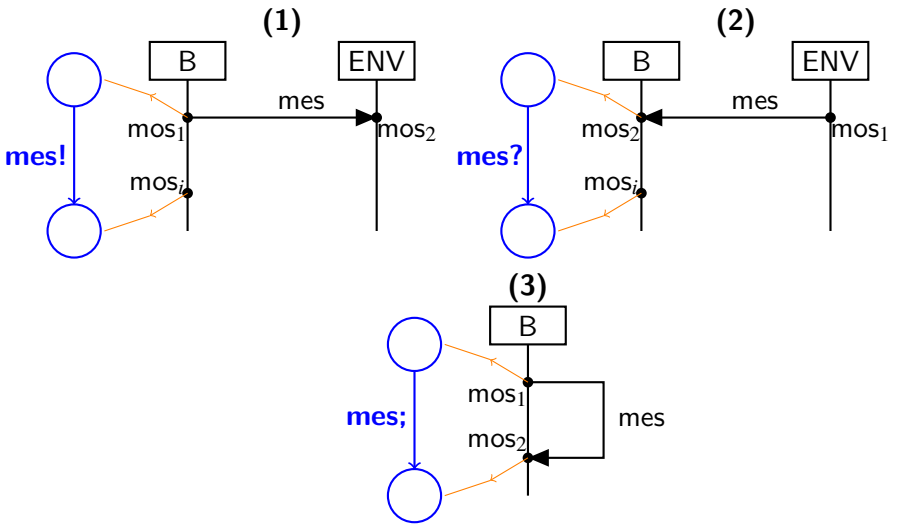


Figure 6.6: Message transformation.

ATL Rule 3:

```
rule message2Transition {
from mes : SD!Message, mos : SD!MessageOccurrenceSpecification
.            (mos.getCovered().name <> 'ENV' )
to t : IA!Transition (
action <- mes.name.concat(
      if(mes.sendEvent.getCovered()=mes.receiveEvent.getCovered())then ';'
       else    if (mes.sendEvent=mos)then '!'  else '?'endif  endif),
source <- thisModule.resolveTemp(mos, 's'),
target <- thisModule.resolveTemp(
            thisModule.NextMsgOcSpec(mos.getCovered(), mos), 's')
) }
```

– `NextMsgOcSpec (lfn, mos)` is an ATL helper that returns the next occurrence specification of 'mos' on the life line 'lfn'.

6.2.4/ ALT Combined Fragment Transformation Rules

The **alt** fragment allows us to express alternative behaviours according to guards. It is the most used of fragments. Also, the fragments **loop** and **par**, that express respectively iterative scenarios and the parallel execution of execution scenarios, are very used in SDs. In Figure 6.7, we give an overview about the equivalence between the fragments **alt**, **loop** and the iterface automata.

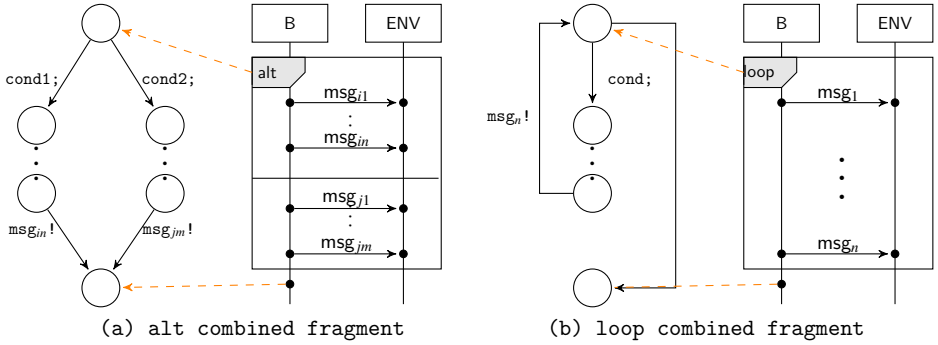


Figure 6.7: The transformation of loop and alt into interface automata

In the following we focus on the fragment **alt**. To transform the alt combined fragment, we have proposed three rules. Two rules for the beginning of alt, and the third one is for processing the end of alt.

For the beginning of alt, we distinguish between two cases:

- **Rule 1:** TransformAltWichFollowsAMsg

  In the case when the combined fragment 'alt' follows a message 'mes', to transform alt, we create a state which represents the beginning of 'alt', and three transitions (see Figure 6.8(1)).

  – **'t1'** allows us to connect the beginning of 'alt' with the previous behaviour using the message just before 'alt'.

  – **'t2'** allows us to connect the beginning of 'alt' with the behaviour of the first operand using guard as internal action.

  – **'t3'** allows us to connect the beginning of 'alt' with the behaviour of the second operand using guard as internal action.

ATL Rule 1:

```
rule TransformAltWichFollowsAMsg {
from alt : SD!CombinedFragment (thisModule.FollowedAMessage(alt))
to
s : IA!State (name <- 'BeginAlt'),
t1 : IA!Transition (
action <- thisModule.previousMessage(alt).name.concat(...
          - specify the type of action as in rule 2),
source<-thisModule.resolveTemp(thisModule.PreviousMessageOccurence(alt),'s'),
target <- s),
t2 : IA!Transition (
action <- alt.operand->at(1).guard...concat(';'),
source <- s,
target<-thisModule.resolveTemp(thisModule.
            getTheFirstElement(alt.operand->at(1)), 's')),
t3 : IA!Transition (
action <- alt.operand->at(1).guard...concat(';'),
source <- s,
target<-thisModule.resolveTemp(thisModule.
            getTheFirstElement(alt.operand->at(2)), 's')),
}
```

- `FollowedAMessage (alt)` is an ATL helper that returns true if alt follows a message.
- `previousMessage(alt)` is an ATL helper that returns the message which is before alt.
- `PreviousMessageOccurence(alt)` is an ATL helper that returns the message occurrence specification which is before alt.
- `getTheFirstElement(op)` is an ATL helper that returns the first element in the operand op.

- **Rule 2:** TransformFirstAltInInteractionOrOperand

  This rule processes 'alt' in case when it is the first element of the global interaction, the first element in an operand, or when it follows a combined fragment. The difference between this rule and the last one, reside in the transition t1. In this rule, we don't create the transition t1 because when 'alt' is :

  - the first element in the interaction (see Figure 6.8(2)): we don't need this transition.
  - directly after a combined fragment 'cf' (see Figure 6.8(4)): this transition will be created by the rule which processes the end of 'cf'.
  - the first element of an operand 'op'(see Figure 6.8(3)): this transition will be created by the rule which processes the beginning of the combined fragment of 'op'.

ATL Rule 2:

```
rule TransformFirstAltInInteractionOrOperand {
from alt : SD!CombinedFragment (thisModule.FirstElementOrFollowsCF(alt))
to
s : IA!State (- the same as rule 1),
t2 : IA!Transition (- the same as rule 1),
t3 : IA!Transition (- the same as rule 1),
}
```

– `FirstElementOrFollowsCF (alt)` is an ATL helper that returns true if alt is the first element in the interaction, the first element inside an operand, or it follows directly a combined fragment.

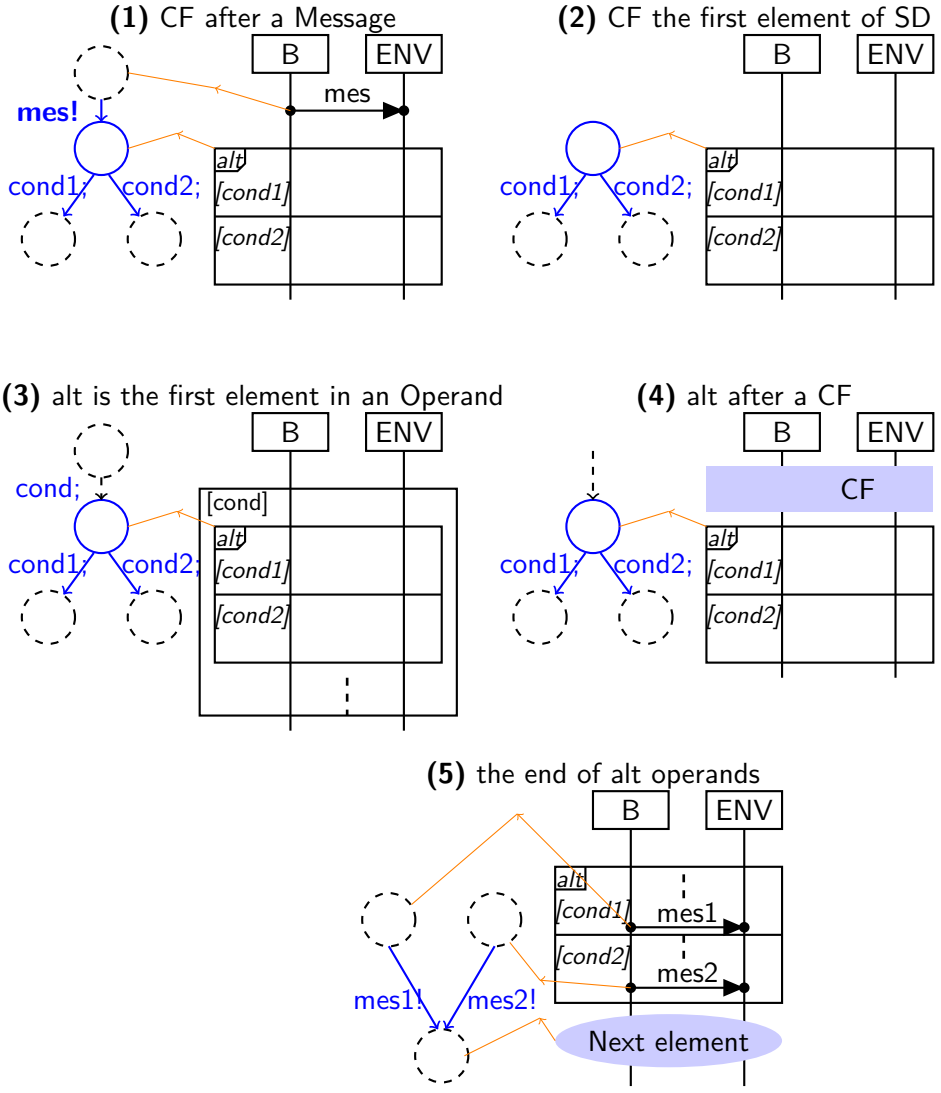


Figure 6.8: Alt transformations.

- **Rule 3:** TransformEndAlt

This rule (see Figure 6.8(5)) allows processing the end of an 'alt' operand. It takes as parameters this operand and the last message occurrence specification (mos) inside it.

ATL Rule 3:

```
rule TransformEndAlt {
from op:SD!InteractionOperand, mos:SD!MessageOccurrenceSpecification
   (thisModule.isTheLastMessageInOperand(mos.message, op)
                and mos.getCovered().name<>'ENV')
to
t : IA!Transition (
action <- mos.message.name.concat(-specify the type of action as in
rule2),
source <- thisModule.resolveTemp(mos,'s'),
target <- thisModule.resolveTemp(thisModule.getNextElement(op.owner),
's'),
}
```

It creates a transition between the state associated to mos and the state associated to the next element of the combined fragment to which this operand belongs (op.owner). The next element may be a message or a combined fragment. The transition takes as label the name of message whose 'mos' is one of its ends.

- `isTheLastMessageInOperand(mes, op)` returns true if the message mes is the last message in the operand op.
- `getNextElement(cf)` returns the next message or combined fragment of the combined fragment cf.

Using the same manner of thinking, we can define rules for other combined fragments.

6.3/ GENERATION OF PTOLEMY SPECIFICATION

At this step, and to discharge the user from redrawing the interface automata using the Ptolemy user interface, we propose a set of Acceleo templates to generate automatically the Ptolemy entry specification.

By analysing an entry file of ptolemy interface automaton, and by eliminating informations related to the position of nodes on the ptolemy canvas, we have obtained its skeleton and we have defined six Acceleo templates. We have eliminated the informations related to the position of nodes on the canvas, because the ptolemy, when it doesn't find informations about the position of a node, it uses its default values.

The first Acceleo template 'generateIA' is the main template, it creates the file of the Ptolemy specification and its header, and calls the other templates. The templates, after the principal one, each one has a name that corresponds to its role.

- `generateInport(inport : Inport)`: it allows us to generate the Ptolemy specification of each in-port of the concerned interface automaton.

- `generateOutport(outport : Outport)` : it will be called iteratively (as the previous template) by the main template to generate the Ptolemy specification for each outport of the concerned interface automaton.
- `generateState(state : State)` : it allows us to generate the Ptolemy specification for automaton states.
- `generateRelation(transition : Transition, i:Integer)` and `generateLinks ( transition : Transition, i:Integer)` : these two templates allow to generate the Ptolemy specification for transitions of the automaton.

```
[comment encoding = UTF-8 /]
[module generate('http://www.interfaceAutomata.ecore')]
[template public generateIA(IA : InterfaceAutomaton)]
[comment @main/]
[file (IA.name.concat('.xml'), false, 'UTF-8')]
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
      "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="[IA.name/]" class="ptolemy.domains.modal.kernel.ia.InterfaceAutomaton">
      [for (outport :Outport| IA.outports )]
      [generateOutport(outport)/]
      [/for]
      [for ( inport :Inport| IA.inports )]
      [generateInport(inport) /]
      [/for]
      [for (state :State| IA.states )]
      [generateState(state) /]
      [/for]
      [for (transition :Transition| IA.transitions)]
      [generateRelation(transition, i)/]
      [/for]
      [for (transition :Transition | IA.transitions)]
      [generateLinks(transition,i)/]
      [/for]
</entity>
[/file]
[/template]
```

```
[template private generateInport(inport : Inport)]
<port name="[inport.name/]" class="ptolemy.actor.TypedIOPort">
<property name="input"/> </port>
[/template]
```

```
[template private generateOutport(outport : Outport)]
<port name="[outport.name/]" class="ptolemy.actor.TypedIOPort">
<property name="output"/>
</port>
[/template]
```

```
[template private generateState(state : State)]
<entity name="[state.name/]" class="ptolemy.domains.modal.kernel.State">
[if (state.type=StateType::Initial)]
   <property name="isInitialState" class="ptolemy.data.expr.Parameter" value="true">
   </property>
[/if]
</entity>
[/template]
```

```
[template private generateRelation(transition : Transition, i:Integer)]
<relation name="relation[if (i>1)][i/][/if]"
 class="ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="[transition.action/]"></property>
</relation>
[/template]
```

```
[template private generateLinks ( transition : Transition, i:Integer)]
<link port="[transition.source.name/].outgoingPort" relation="relation[if (i>1)][i/][/if]"/>
<link port="[transition.target.name/].incomingPort" relation="relation[if (i>1)][i/][/if]"/>
[/template]
```

6.4/ THE BLOCKS VERIFICATION

We want to verify the consistency and the compatibility of the blocks. To do that, we base on the interface automata that describe the interaction protocols of these blocks.

Definition 14: Consistency of SysML blocks

Two blocks B_1 and B_2 are considered as consistent if their interface automata A_1 and A_2 are composable:
$\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^H_{A_2} = \emptyset$.

Definition 15: Compatibility of SysML blocks

Two blocks are compatible, if they are consistent and their interface automata are compatible. According to the optimistic approach of *Henzinger* [dAH01], two interface automata are compatible if their composition is not empty:
$A_1 || A_2 \neq \emptyset$

To verify the consistency and the compatibility of the blocks, we use the Ptolemy tool. We give it, as entry, the generated files(the files that we have generated using our Acceleo templates). Ptolemy computes the composition of interface automata and delivers the result. If the result of composition is not empty, this means that the blocks are consistent and compatible.

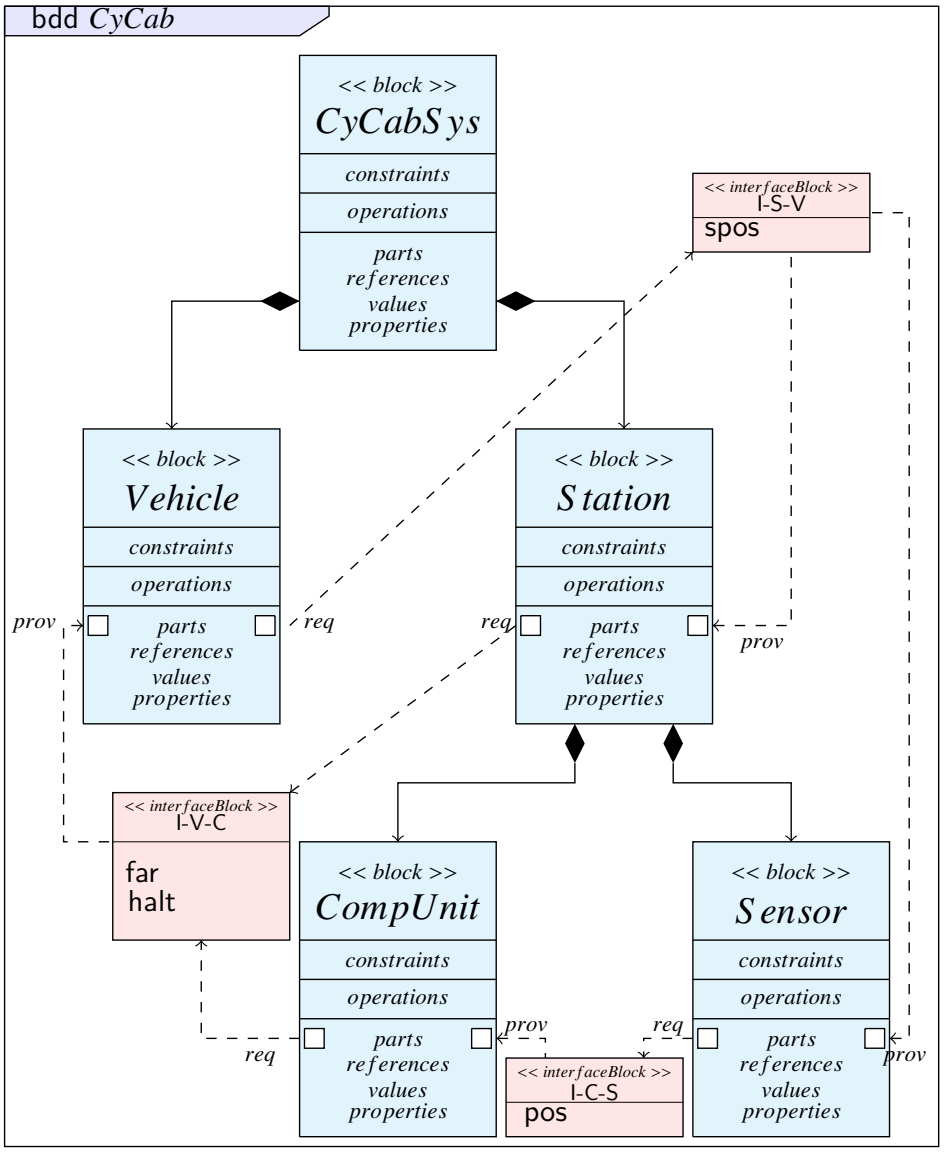


Figure 6.9: Block Definition Diagram of CyCab.

6.5/ Case Study: CyCab

CyCab [BGMPG99] is a new means of electrical transportation, it is conceived basically for free-standing port services. It is controlled by a computer system. The CyCab system has two major parts: the station and the vehicle. The vehicle is guided by the information received from the station, which allows situating the vehicle.

In this case study, we are only interested by the 'station' part. The station has a sensor that receives signals from vehicle giving the vehicle position (pos?). The station has also a computing units that sends a signal (far! or halt!) to the vehicle to indicate if it is far from the station or not. In Figure 6.9, we present the architecture of the Cycab System using the SysML BDD.

The interactions of the sensor and the computing-unit blocks are represented as sequence diagrams (see Figure 6.10 and Figure 6.11). To draw sequence diagrams, we have used the

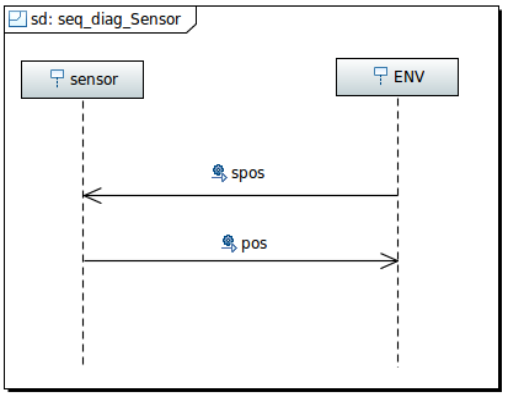


Figure 6.10: SD of Sensor.

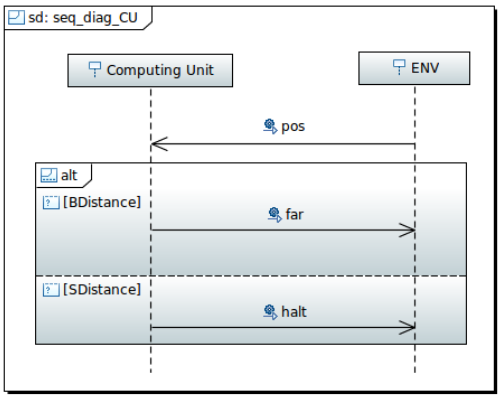


Figure 6.11: SD of Computing-Unit

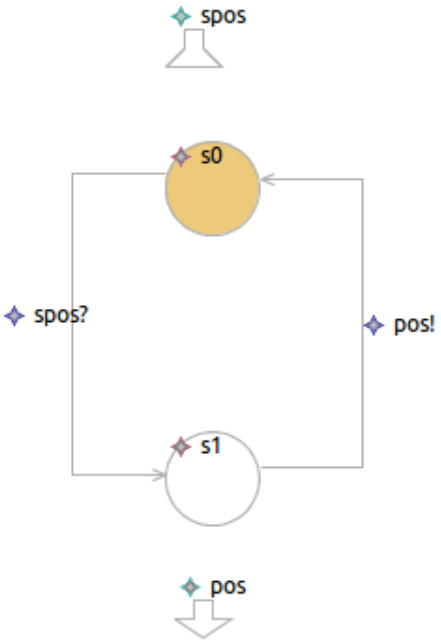


Figure 6.12: IA of Sensor

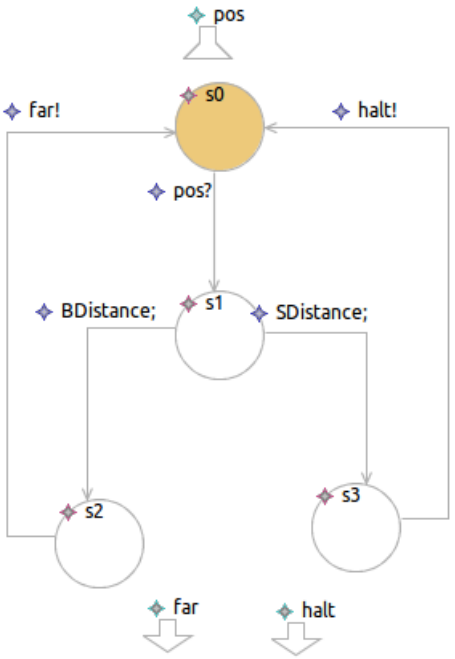


Figure 6.13: IA of Computing-Unit

papyrus editor.

By applying our ATL rules on the sequences diagrams of the sensor and the computing unit, we have obtained their equivalents of interface automata. In Figure 6.12 and Figure 6.13, we present the resulted interface automata in our graphical editor.

By applying the Acceleo templates, that we have defined to generate Ptolemy specification, we have obtained these files.

- Ptolemy file of Sensor block:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="sensor" class="ptolemy.domains.modal.kernel.ia.InterfaceAutomaton"><port
```

```
name="pos" class="ptolemy.actor.TypedIOPort">
   <property name="output"/>
</port>

<port name="spos" class="ptolemy.actor.TypedIOPort">
   <property name="input"/>
</port>

<entity name="s1" class="ptolemy.domains.modal.kernel.State"></entity>

<entity name="s0" class="ptolemy.domains.modal.kernel.State">
   <property name="isInitialState" class="ptolemy.data.expr.Parameter"
      value="true"></property>
</entity>

<relation name="relation" class=
   "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="spos?"></property>
</relation>

<relation name="relation2" class=
    "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="pos!"></property>
</relation>

<link port="s1.outgoingPort" relation="relation"/>
<link port="s0.incomingPort" relation="relation"/>
<link port="s0.outgoingPort" relation="relation2"/>
<link port="s1.incomingPort" relation="relation2"/>
</entity>
```

- Ptolemy file of Computing-Unit block:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="Computing Unit" class="ptolemy.domains.modal.kernel.ia.InterfaceAutomaton">
<port name="far" class="ptolemy.actor.TypedIOPort">
   <property name="output"/> </port>
<port name="halt" class="ptolemy.actor.TypedIOPort">
   <property name="output"/> </port>
<port name="pos" class="ptolemy.actor.TypedIOPort">
   <property name="input"/> </port>
<entity name="s0" class="ptolemy.domains.modal.kernel.State">
   <property name="isInitialState" class="ptolemy.data.expr.Parameter"
      value="true"> </property> </entity>

<entity name="s2" class="ptolemy.domains.modal.kernel.State">  </entity>

<entity name="s3" class="ptolemy.domains.modal.kernel.State">  </entity>

<entity name="s1" class="ptolemy.domains.modal.kernel.State">  </entity>

<relation name="relation" class=
      "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="pos?"> </property>
</relation>

<relation name="relation2" class=
   "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
```

```
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
       value="BDistance;"> </property>
</relation>

<relation name="relation3" class=
   "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="SDistance;"> </property>
</relation>

<relation name="relation4" class=
   "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="far!"> </property>
</relation>

<relation name="relation5" class=
   "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
   <property name="label" class="ptolemy.kernel.util.StringAttribute"
      value="halt!"> </property>
</relation>

<link port="s0.outgoingPort" relation="relation"/>
<link port="s1.incomingPort" relation="relation"/>
<link port="s1.outgoingPort" relation="relation2"/>
<link port="s2.incomingPort" relation="relation2"/>
<link port="s1.outgoingPort" relation="relation3"/>
<link port="s3.incomingPort" relation="relation3"/>
<link port="s2.outgoingPort" relation="relation4"/>
<link port="s0.incomingPort" relation="relation4"/>
<link port="s3.outgoingPort" relation="relation5"/>
<link port="s0.incomingPort" relation="relation5"/>
</entity>
```

Using Ptolemy tool, we can use these two files to verify the consistency and the compatibility of the sensor and the computing unit blocks.

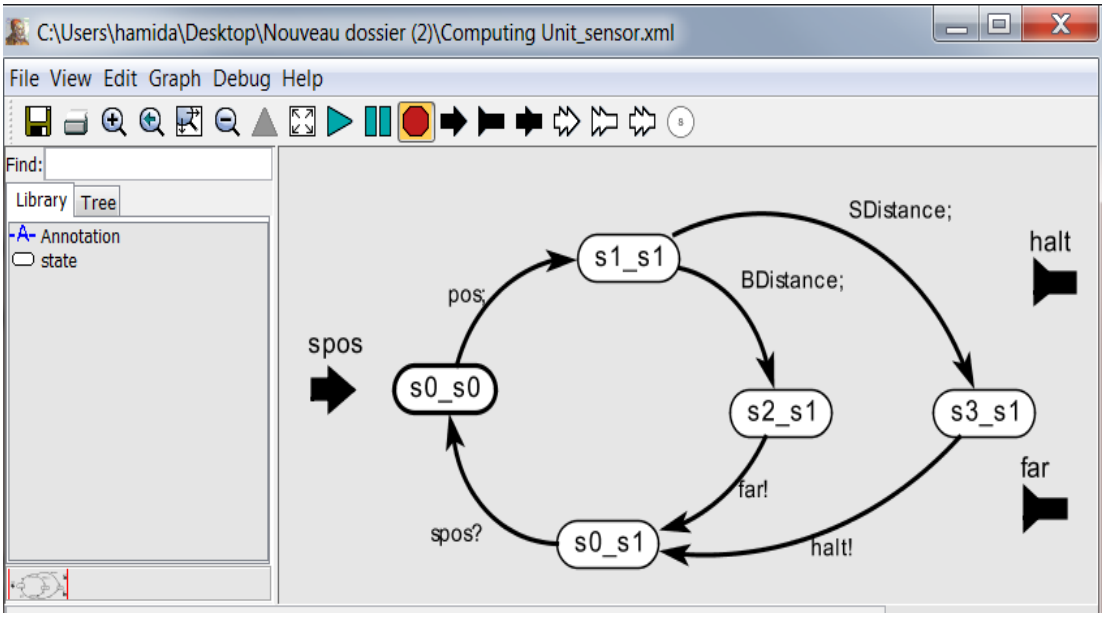


Figure 6.14: Parallel composition of Control Unit and Sensor.

In Figure 6.14, we present the result of composing the two interface automata using Ptolemy tool. Because the composition is not empty, we deduce that the control unit and the sensor blocks are consistent and compatible. If we assemble them in the same system, we obtain a system part that interacts with the rest of the system according to the scenarios modelled as an interface automaton in Figure 6.14.

6.6/ Conclusion

The point that we have addressed in this chapter is how can we prepare the SysML blocks interactions for verification. Thus, our proposed approach is based on specifying the correspondences between the blocks sequence diagrams and interface automata. The goal of this chapter is to present how it's possible to automatize the transformation from sequence diagrams to interface automata using ATL. We have shown the transformation of the basic constructs of sequence diagrams. We have also given the ATL rules to transform the alternative combined fragment. The second objective of this chapter concerns Ptolemy tool, which is used to verify the interface automata compatibility, and to compute their parallel composition. To discharge the user from redrawing the resulted interface automata using the Ptolemy user interface, which can be considered as a source of errors, we have proposed a set of Acceleo templates to generate the entry code of Ptolemy. We have also given an overview of how can we use the generated files to verify the compatibility of blocks. To illustrate our approach, we have applied it on a CyCab case study.

7

Exploiting The Hierarchy to Verify Blocks Compatibility

Generally, the high level modelling languages as SysML, adopt some principles to manage the complexity of system's representation and development. In SysML, the decomposition and the hierarchical organization constitute the major principles used to handle complexity. The utility of the decomposition and the hierarchy appears clearly through the structural and the behavioural specification of the system.

In SysML, the interactions between blocks are modelled with Interaction Block Diagram (IBD) and Sequence Diagram (SD). However, these interactions are modelled by the IBD only as architectural links. In other hand, a block can participate in multiple use cases, which makes its interaction protocol divided into a set of sequence diagrams. For these reasons, there is a lack of global view of the interaction protocol related to a given block.

In this chapter, we propose HPSM, to model the interaction protocol of a block. The proposed HPSM differs from the UML Protocol State Machine (PSM). In UML, each interface of a class can be associated with a PSM. PSM of UML presents the pre and the post conditions of events allowing the enabling of transitions. However, HPSM as we define it, is associated with a block and expresses its states and the transitions between them. Each transition can be labelled with a reception of an ask for a service of the block, and a set of requests that the current bock sends for asking some services of the environment. We note that HPSM uses also the composite states to benefit from the clarity added by the hierarchy.

Contents

One of the advantages of proposing HPSM is to allow modelling all the interactions of a given block in one diagram, and so enabling the verification of compatibility between blocks. In our study, the architecture of the system is given by an IBD. The interactions inside the system take the form of an HPSMs set, each HPSM is assigned to a block. This permits to graphically describe the system architecture and the interaction protocols of its blocks. However, this specification is not thoroughly formal to be adapted for verification. That is why, it is necessary to present the semantic of HPSM in an another model more suitable for verification. For that, we define hierarchical interface automata with Inter-Level Transitions (HIA-ILT), a variant of interface automata [dAH01], we express the HPSM semantics in term of HIA-ILT, and we use HIA-ILT to verify the compatibility between SysML blocks. In our verification approach, we avoid the flattening of the entire HIA-ILT by proposing a preliminary phase that allows selecting the composite states to flatten. The aim behind this is to contribute for reducing the complexity of compatibility verification.

The remainder of this chapter is organized as follows : In section 7.1, we introduce the HPSM. Next, in section 7.2, we present the HIA-ILT the variant of interface automata. In Section 7.3, we present our approach for verifying blocks compatibility which benefit fro the hierarchy of HIA-ILTs. Next, we illustrate our approach by a case study in section 7.4. Finally, in Section 7.5, we conclude.

7.1/ Hierarchical Protocol State Machine (HPSM)

SysML uses the SD diagram to represent interaction protocols of system blocks, each SD is

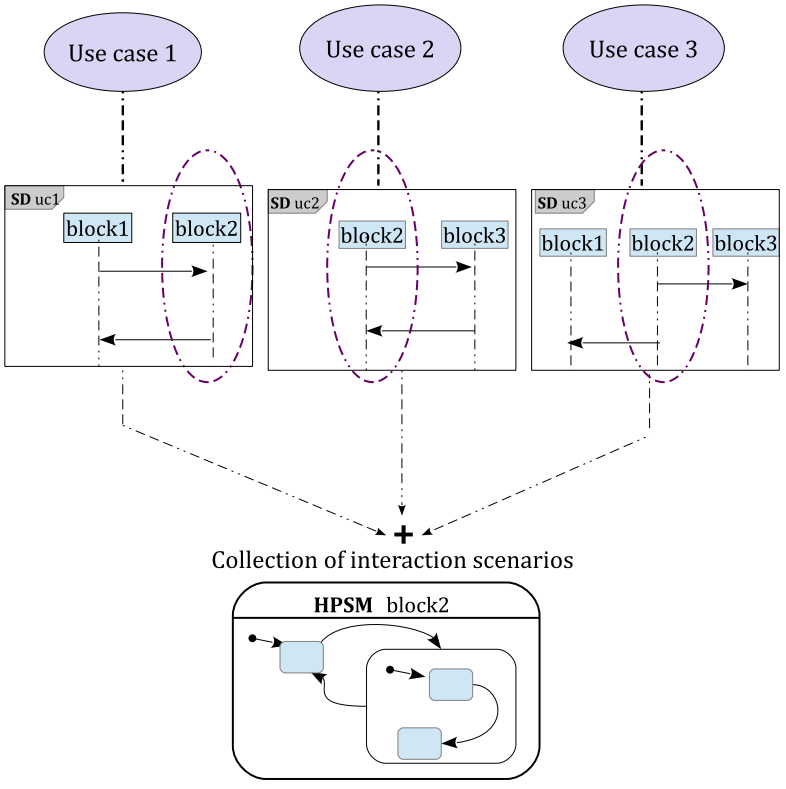


Figure 7.1: Relation between *SD* and *HPSM*.

associated with a use case. When a block participates in multiple use cases, its interaction protocol will be divided into several SDs (see Figure 7.1).

We propose the model HPSM (Hierarchical protocol state machine), which allows us to represent the interaction protocol of a block using one diagram. HPSM adopts the structure of hierarchical state machine, it bases on simple states, composite states and transitions between them. This structure allows us to benefit from hierarchical aspect by hiding details of states when we don't need to visualize it.

Definition 16: HPSM

We define an HPSM of a block **B** as follows :

$$\mathtt{HPSM}_B = \langle\ \mathtt{SS}_B,\ \mathtt{CS}_B,\ \mathtt{I}_B,\ \mathtt{T}_B,\ \mathtt{Prov\_Serv}_B,\ \mathtt{Req\_Serv}_B\ \rangle$$

Where:

- $\mathtt{SS}_B$ is a set of simple states.
- $\mathtt{CS}_B$ is a set of composite states.
- We define by $\mathtt{S}_B = \mathtt{SS}_B \cup \mathtt{CS}_B$ a set of all states of the HPSM.
- $\mathtt{I}_B$ is a set of initial states, $\mathtt{I} \subseteq \mathtt{S}_B$.
- $\mathtt{T}_B$ is a set of transitions.
- $\mathtt{Prov\_Serv}_B$ is a set of services that the block B offers to its environment.
- $\mathtt{Req\_Serv}_B$ is a set of services that the block B requires from its environment.

The set of transitions $\mathtt{T} \subseteq \mathtt{S}_B \times \mathtt{L} \times \mathtt{S}_B$, where L is the set of all transitions labels. A label $\mathtt{l} \in \mathtt{L}$ takes the following form:

$$\mathrm{l}= \textbf{REC}\ \langle \mathrm{ps} \rangle\ /\ \textbf{SND}\ \langle \{\mathrm{rs}_i\}_{i=1..n} \rangle$$

Where:

- REC represents the reception of a request, and SND represents the emission of a request to the environment.
- $\mathtt{ps} \in \mathtt{Prov\_Serv}_B$.
- $\{\ \mathtt{rs}_i\}_{i=1..n} = \mathcal{P}(\mathtt{Req\_Serv}_B)$ is the set of a required services.

When the block B receives a request for its service ps from its adjacent blocks, it may have a need to call some services of other blocks. We model the fact that the block B send requests to these blocks using the directive SND.

7.2/ Hierarchical Interface Automata with Inter-Level Transitions (HIA-ILT)

In this section, we propose Hierarchical Interface Automata with Inter-Level Transitions (HIA-ILT). The model that we use in our approach to formalize HPSMs of blocks. Compar-

ing with IA, HIA-ILT introduces the concept of composite states, and allows the inter-level transitions. In fact, in HIA-ILT the source state and the target state of a transition can belong to two different composite states and to two different levels of hierarchy, which makes HIA-ILT able to represent more complex interactions of a component with a small model.

Definition 17: HIA-ILT

We define an Hierarchical Interface Automata with Inter-Level Transitions *'HA'* by:

$$\langle SS_{HA}, CS_{HA}, I_{HA}, \sum^{I}_{HA}, \sum^{O}_{HA}, \sum^{H}_{HA}, \delta_{HA} \rangle$$

where:

- $\mathtt{SS}_{HA}$ is a set of simple states.
- $\mathtt{CS}_{HA}$ is a set of composite states.
- $\mathtt{I}_{HA}$ is a set of initial states, we have $\mathtt{I}_{HA} \subseteq \mathtt{SS}_{HA} \cup \mathtt{CS}_{HA}$.
- $\sum^{I}_{HA}$ is a set of input actions.
- $\sum^{O}_{HA}$ is a set of output actions.
- $\sum^{H}_{HA}$ is a set of hidden actions.
- δ_{HA} is a set of transitions.
  - $\delta_{HA} \subseteq (\mathtt{SS}_{HA} \cup \mathtt{CS}_{HA}) \times \sum_{HA} \times (\mathtt{SS}_{HA} \cup \mathtt{CS}_{HA})$, where $\sum_{HA} = \sum^{I}_{HA} \cup \sum^{O}_{HA} \cup \sum^{H}_{HA}$
  - $(\mathtt{s}_1, \mathtt{a}, \mathtt{s}_2)$ is an inter-level transition if s_1 and s_2 don't belong to the same composite state or if they belongs to two different level of hierarchy.

The HIA-ILT is as the IA, it is enclosed with a box whose ports correspond to the input and the output actions.

The abstract synchronous product between two HIA-ILTs HA_1 and HA_2, takes all the composite states of HA_1 and HA_2 as abstract states.

Definition 18: Abstract state

An abstract state is a composite state $\mathtt{s} \in \mathtt{CS}_{HA}$, but its internal states and relations between them are ignored

Let HA_1 and HA_2 two HIA-ILTs, we can compute the abstract synchronous product of HA_1 and HA_2 if they are composable.

- HA_1 and HA_2 are composable :

  $\Sigma^{I}_{HA_1} \cap \Sigma^{I}_{HA_2} = \Sigma^{O}_{HA_1} \cap \Sigma^{O}_{HA_2} = \Sigma^{H}_{HA_1} \cap \Sigma_{HA_2} = \Sigma_{HA_1} \cap \Sigma^{H}_{HA_2} = \emptyset$.

- Any composite state of HA_1 or HA_2 have inside a transition labelled with a shared action.

Definition 19: Abstract Synchronous Product of HIA-ILT

we define the abstract synchronous product of HA_1 and HA_2 as:

$$HA_1 \otimes_a HA_2 = \langle \; SS_{HA_1\otimes_a HA_2}, \; CS_{HA_1\otimes_a HA_2}, \; I_{HA_1\otimes_a HA_2}, \; \Sigma^I_{HA_1\otimes_a HA_2}, \; \Sigma^O_{HA_1\otimes_a HA_2}, \; \Sigma^H_{HA_1\otimes_a HA_2}, \; \delta_{HA_1\otimes_a HA_2} \; \rangle$$

- $SS_{HA_1\otimes_a HA_2} = SS_{HA_1} \times SS_{HA_2}$.
- $CS_{HA_1\otimes_a HA_2} = CS_{HA_1} \times CS_{HA_2} \cup CS_{HA_1} \times SS_{HA_2} \cup SS_{HA_1} \times CS_{HA_2}$.
- $I_{HA_1\otimes_a HA_2} = I_{HA_1} \times I_{HA_2}$.
- $\Sigma^I_{HA_1\otimes_a HA_2} = (\Sigma^I_{HA_1} \cup \Sigma^I_{HA_2}) \setminus Shared(HA_1, HA_2)$.
- $\Sigma^O_{HA_1\otimes_a HA_2} = (\Sigma^O_{HA_1} \cup \Sigma^O_{HA_2}) \setminus Shared(HA_1, HA_2)$.
- $\Sigma^H_{HA_1\otimes_a HA_2} = \Sigma^H_{HA_1} \cup \Sigma^H_{HA_2} \cup Shared(HA_1, HA_2)$.
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{HA_1\otimes HA_2}$ if
  - $a \notin Shared(HA_1, HA_2) \wedge (s_1, a, s'_1) \in \delta_{HA_1} \wedge s_2 = s'_2$
  - $a \notin Shared(HA_1, HA_2) \wedge (s_2, a, s'_2) \in \delta_{HA_2} \wedge s_1 = s'_1$
  - $a \in Shared(HA_1, HA_2) \wedge (s_1, a, s'_1) \in \delta_{HA_1} \wedge (s_2, a, s'_2) \in \delta_{HA_2}$.

We define by $\texttt{Shared(HA}_1\texttt{,HA}_2\texttt{)} = (\Sigma^I_{HA_1} \cap \Sigma^O_{HA_2}) \cup (\Sigma^O_{HA_1} \cap \Sigma^I_{HA_2})$ the set of the shared actions between HA_1 and HA_2.

We deduce the synchronous product $\texttt{HA}_1\otimes\texttt{HA}_2$ of HA_1 and HA_2 from the abstract synchronous product $\texttt{HA}_1\otimes_a\texttt{HA}_2$ as follows (an example is given in Figure 7.2):

- $\forall (s_1, s_2) \xrightarrow{t} (s_1, s'_2) \in \sigma_{HA_1\otimes_a HA_2}$

  if $s_1 \in CS_{HA_1} \Rightarrow$ replace the transition $(s_1, s_2) \xrightarrow{t} (s_1, s'_2)$ by this set of transitions:

  $\forall (s_{1i}, s_{2j}) \in (s_1, s_2)$ create new transition $(s_{1i}, s_{2j}) \xrightarrow{t} (s_{1i}, s'_{2k})$

  This means that when the state of the block B_2 is changed from s_2 to s'_2 by crossing the transition t, the block B_1 must still in the same composite state s_1 and also in the same simple state s_{1i}. If the state $s'2$ is a composite state, then the state s'_{2k} must be the initial state of s'_2, otherwise the state s'_{2k} is the state s'_2.

  If t is an inter-level transition in HA2 :

  - if $\exists n (s_{2n} \xrightarrow{t} s'_2 \in \sigma_{HA_2} \Rightarrow s_{2j} = s_{2n})$
  - if $\exists m (s_2 \xrightarrow{t} s'_{2m} \in \sigma_{HA_2} \Rightarrow s'_{2k} = s'_{2m})$
  - if $\exists n, m (s_{2n} \xrightarrow{t} s'_{2m} \in \sigma_{HA_2} \Rightarrow s_{2j} = s_{2n} \wedge s'_{2k} = s'_{2m})$

- $\forall (s_1, s_2) \xrightarrow{t} (s'_1, s_2) \in \sigma_{HA_1\otimes_a HA_2}$

  if $s_2 \in CS_{HA_2} \Rightarrow$ replace the transition $(s_1, s_2) \xrightarrow{t} (s'_1, s_2)$ by this set of transitions:

  $\forall (s_{1i}, s_{2j}) \in (s_1, s_2)$ create new transition $(s_{1i}, s_{2j}) \xrightarrow{t} (s'_{1k}, s_{2j})$

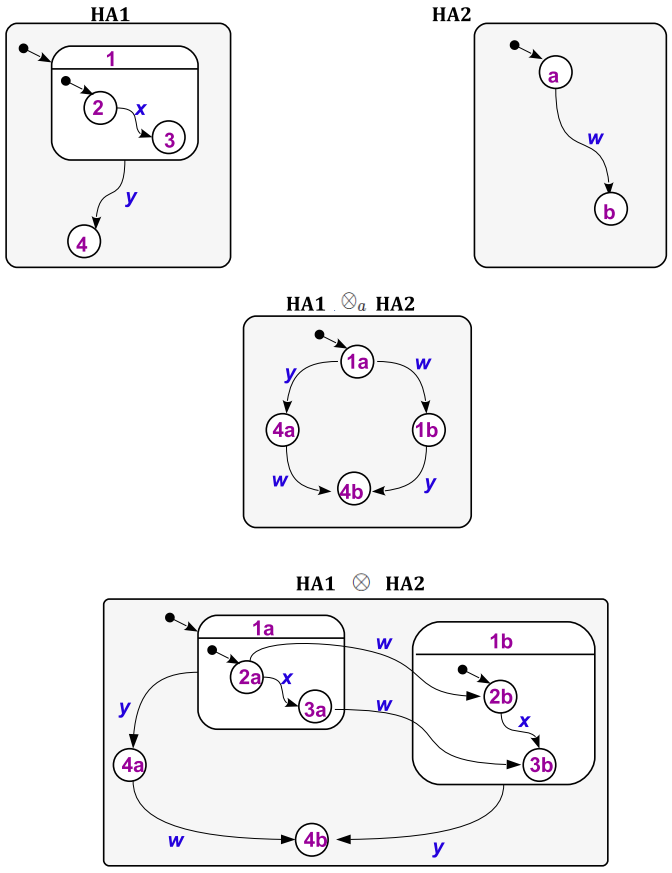


Figure 7.2: Example of abstract synchronous product.

This means that when the state of the block B_1 is changed from s_1 to s'_1 by crossing the transition t, the component B_2 must still in the same composite state s_2 and also in the same simple state s_{2j}. If the state $s'1$ is a composite state, then the state s'_{1k} must be the initial state of s'_1, otherwise the state s'_{1k} is the state s'_1.

If t is an inter-level transition in HA1 :

- if $\exists n(s_{1n} \xrightarrow{t} s'_1 \in \sigma_{HA_1} \Rightarrow s_{1i} = s_{1n})$
- if $\exists m(s_1 \xrightarrow{t} s'_{1m} \in \sigma_{HA_1} \Rightarrow s'_{1k} = s'_{1m})$
- if $\exists n, m(s_{1n} \xrightarrow{t} s'_{1m} \in \sigma_{HA_1} \Rightarrow s_{1j} = s_{1n} \wedge s'_{1k} = s'_{1m})$

Definition 20: Abstract Parallel Composition

The abstract parallel composition of HA_1 and HA_2 $(HA_1 \|_a HA_2)$ bases on eliminating from the product $HA_1 \otimes_a HA_2$ the illegal states and all states reached from these illegal states by enabling output and internal actions.
We define illegal states as follows:

$$Illegal(HA_1, HA_2) = \left\{ \begin{array}{c} (s_1, s_2) \in \sigma_{HA_1 \otimes_a HA_2} \mid \exists a \in Shared(HA_1, HA_2). \\ \left(\begin{array}{c} a \in \Sigma^O_{HA_1}(s_1) \wedge a \notin \Sigma^I_{HA_2}(s_2) \\ \vee \\ a \in \Sigma^O_{HA_2}(s_2) \wedge a \notin \Sigma^I_{HA_1}(s_1) \end{array} \right) \end{array} \right\}$$

Definition 21: Compatibility of HIA-ILTs

HA_1 and HA_2 are compatible *iff* $\mathtt{HA}_1 \parallel_c \mathtt{HA}_2 \neq \emptyset$

Theorem 1:

If we have an abstract state (x,y) $\in S_{HA_1 \otimes_a HA_2}$ where (x,y) is an illegal state, this implies that all states insides are illegal states.

proof :

a. (x,y) $\in S_{HA_1 \otimes_a HA_2}$ is an abstract state $\Rightarrow$ x is an abstract state in HA_1 or y is an abstract state in HA_2.

b. (x,y) is an illegal state $\Rightarrow \exists a \in Shared(HA_1, HA_2) \wedge (\exists x \xrightarrow{a!} x' \in \delta_{HA_1} \wedge \nexists y \xrightarrow{a?} y' \in \delta_{HA_2}$ or $\exists y \xrightarrow{a!} y' \in \delta_{HA_2} \wedge \nexists x \xrightarrow{a?} x' \in \delta_{HA_1})$

- If we suppose that x is an abstract state in S_{HA_1}, and (x,y) is an illegal state because $\exists x \xrightarrow{a!} x' \in \delta_{HA_1} \wedge \nexists y \xrightarrow{a?} y' \in \delta_{HA_2}$:

  $\exists x \xrightarrow{a!} x' \in \delta_{HA_1} \Rightarrow \forall s \in x, \exists s \xrightarrow{a!} x' \in \delta_{flatten(HA_1)}$

  $\Rightarrow \forall (s,y) \in (x,y), \exists s \xrightarrow{a!} x' \in \delta_{flatten(HA_1)}$, and we have $\nexists y \xrightarrow{a?} y' \in \delta_{HA_2}$

  $\Rightarrow \forall (s,y) \in (x,y)$, (s,y) is an illegal state.

- If we suppose that x is an abstract state in S_{HA_1}, and (x,y) is an illegal state because $\exists y \xrightarrow{a!} y' \in \delta_{HA_2} \wedge \nexists x \xrightarrow{a?} x' \in \delta_{HA_1}$:

  $\nexists x \xrightarrow{a?} x' \notin \delta_{HA_1} \Rightarrow \forall s \in x, \nexists s \xrightarrow{a?} x' \in \delta_{flatten(HA_1)}$

  $\Rightarrow \forall (s,y) \in (x,y), \nexists s \xrightarrow{a?} x' \in \delta_{flatten(HA_1)}$, and we have $\exists y \xrightarrow{a!} y' \in \delta_{HA_2}$

  $\Rightarrow \forall (s,y) \in (x,y)$, (s,y) is an illegal state.

7.3/ The Proposed Approach

Our approach includes four steps (Figure 7.3): The first step is a mapping from HPSM to HIA-ILT of the two blocks to verify their compatibility, the second step is to verify the consistency between HIA-ILTs associated with blocks, the third step is to select composite states to flatten, and the last step is for verifying the compatibility between the two blocks.

7.3.1/ The Mapping Between HPSM and HIA-ILT

In this section, we give the rules to translate HPSM to HIA-ILT. During this transformation, each simple state in the HPSM must be transformed to a simple state in the HIA-ILT and each composite state in the HPSM must be copied as a composite state in the HIA. The difference between HPSM and HIA-ILT resides in the labels of the transitions. A transition in HPSM can be decomposed into a set of HIA-ILT transitions. In Figure 7.4, we see that a provided service on a transition of HPSM ($\xrightarrow{ps}$) must be translated into a transition in HIA-ILT, labelled with an input action ($\xrightarrow{ps?}$). The set of required services on an HPSM

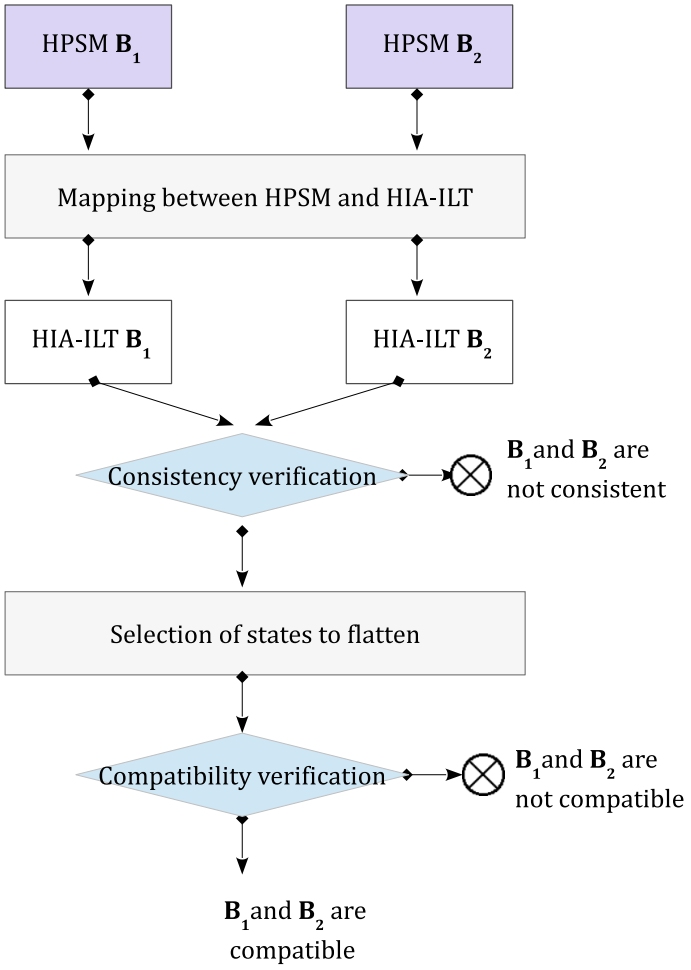


Figure 7.3: Our approach of using hierarchy to verify blocks compatibility.

transition ($\overset{/rs_1,...,rs_n}{\longrightarrow}$) must be translated to a sequence of HIA-ILT transitions, labelled with output actions ($\overset{rs_1!}{\longrightarrow}...\overset{rs_n!}{\longrightarrow}$).

To implement the correspondences, we have defined the meta-model of the HPSM (Figure 7.5) and the meta-model of the HIA-ILT (Figure 7.6) using EMF (Eclipse Modelling Framework). Figure 7.5 presents the HPSM as a set of states, a set of transitions, a set of provided services and a set of required services. A state can be simple or composite, it includes other states only if it is a composite state. A transition may be labelled with a reception of request and a set of required services. It must have a source state and a target one. The meta-model of HIA-ILT in Figure 7.6, describes the hierarchical interface automata as a set of states, a set of transitions, a set of in-ports and a set of out-ports. If a transition takes as label an input or an output action, the input action must correspond to an in-port and the output action must correspond to an out-port.

We have implemented the correspondences as an ATL grammar 'TransformHPSM2HIA-ILT'. This ATL grammar takes as source the meta-model of HPSM and it has as target the meta-model of HIA-ILT. It includes a set of rules.

In Figure 7.7, we give an extract of the grammar 'TransformHPSM2HIA-ILT'. The first rule creates the HIA-ILT element from the HPSM element. However, the second rule initializes

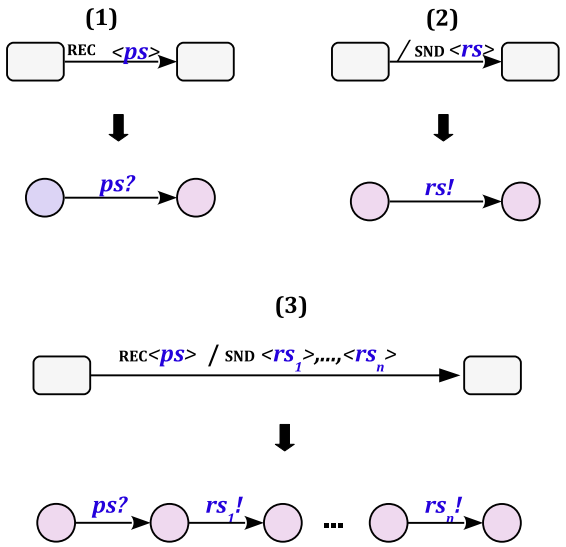


Figure 7.4: Correspondences between HPSM and HIA-ILT.

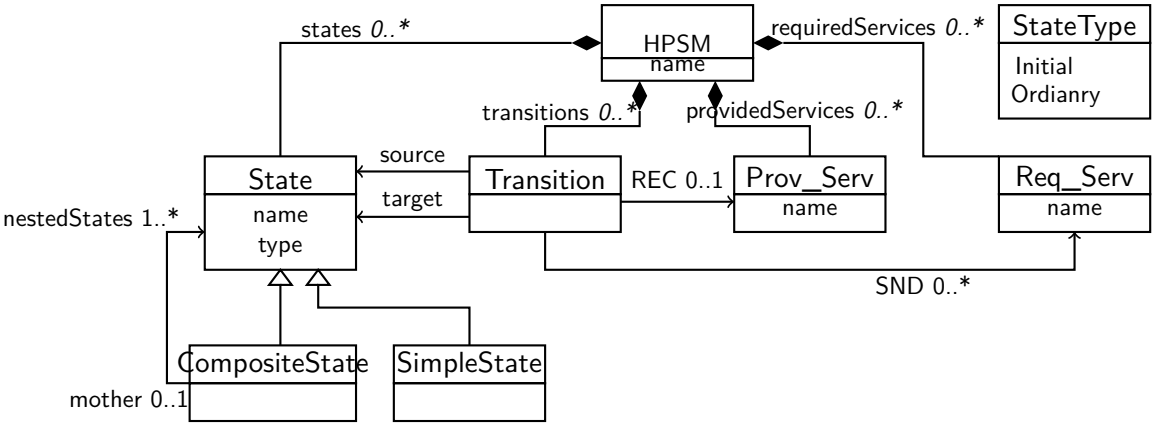


Figure 7.5: Meta-Model of HPSM.

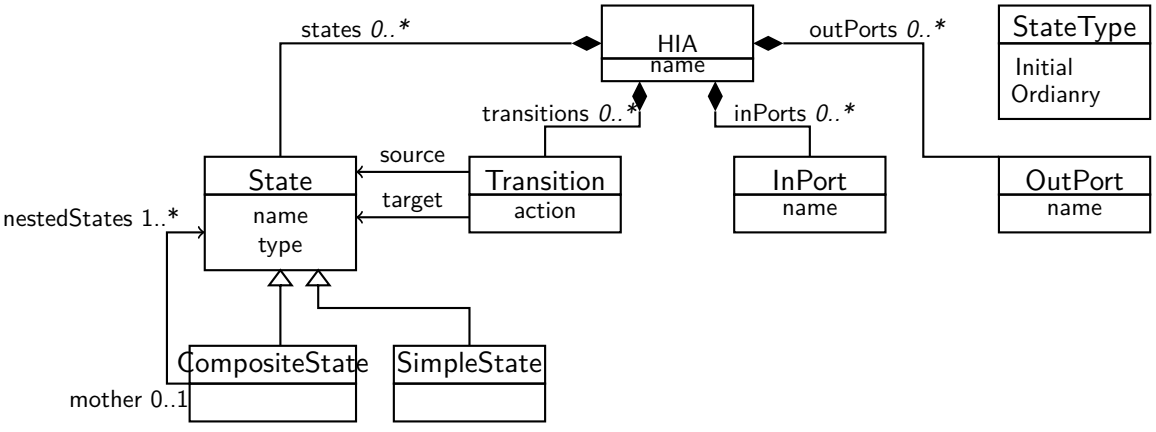


Figure 7.6: Meta-Model of HIA-ILT.

the simple states of HIA-ILT from the simple states of HPSM.

ATL Rule 1: HPSM2HIA

```
rule HPSM2HIA {
from hpsm : HPSM!HPSM
to   hia : HIA!HIA (
name <-hpsm.name,
states <-HIA!State.allInstances(),
transitions <-HIA!Transition.allInstances(),
inports <- HIA!InPort.allInstances(),
outports <- HIA!OutPort.allInstances()
) }
```

ATL Rule 2: SimpleState2SimpleState

```
rule SimpleState2SimpleState {
from s1 : HPSM!SimpleState
to   s2 : HIA!SimpleState (
name <-s1.name,
type <-if s1.type=#Initial then s1.type else #Ordinary endif,
mother <- thisModule.resolveTemp(s1.mother, 'cs2');
) }
```

Figure 7.7: Rules ATL.

7.3.2/ The Consistency Verification of Blocks

This step must ensure that the two blocks B1 and B2 associated with respectively HA_1 and HA_2, their HIA-ILTs, respect the condition of composability. This means that the block B1 and B2 haven't shared provided services and shared required services, and there aren't an overlap between the hidden actions of a block and the set of actions of the other block. The relation of composability ζ between two blocks B1 and B2 is defined as follows:

$B1\ \zeta\ B2 \Leftrightarrow$

$\Sigma^I_{HA_1} \cap \Sigma^I_{HA_2} = \Sigma^O_{HA_1} \cap \Sigma^O_{HA_2} =$

$\Sigma^H_{HA_1} \cap \Sigma_{HA_2} = \Sigma_{HA_1} \cap \Sigma^H_{HA_2} = \emptyset.$

7.3.3/ The Selection of Composite States to Flatten

In this step, we must construct the set of shared actions between HA_1 and HA_2 of B1 and B2.

$$\text{Shared}(HA1,HA2) = (\Sigma^I_{HA1} \cap \Sigma^O_{HA2}) \cup (\Sigma^O_{HA1} \cap \Sigma^I_{HA2})$$

The existence of a shared action means that there is an interaction between these two blocks. Thus, we look over all composite states in HA1 and HA2, if there is a composite state C having inside a transition labelled with an action which belongs to the set

Shared($HA1$,$HA2$) then this composite state C must be flattened. It is mandatory to flatten these composite states to allow the synchronization of their transitions with the transitions of the other HIA-ILT.

For flattening, we refer to works which have been already proposed for this purpose (i.e [KC09, DM01]).

7.3.4/ The Compatibility Verification Between Blocks

The compatibility verification between two blocks B1 and B2 is obtained by verifying the compatibility between their interface automata HA_1 and HA_2. To verify the compatibility between two blocks B1 and B2, we adopt the approach in [dAH01] which verifies if there is an environment where it is possible to correctly assemble B1 and B2. Thus, we assume that this environment will never led one of the blocks B1 or B2 to a deadlock state, means that the environment will never allow to the parallel execution of B1 and B2 to reach an illegal state.

The relation of compatibility ζom between two blocks B1 and B2 have as interaction protocol models HA_1 and HA_2 is defined by:

B1 ζom B2 $\Leftrightarrow$ $HA_1 \|_a HA_2$ has at least one reachable state.

7.4/ Case Study

In this section, the case study concerns a robotic vacuum called Roomba. In our case study, we consider that Roomba is controlled by human operator. To allow this control, we consider that a kinect is placed between them. It is used to communicate the operator positions to the coordinator in the form of images. Then, the coordinator analyses these images and extracts actions required by the operator and conveys them to the Robot.The Robot includes a receiver and Roomba vacuum. The receiver captures actions wanted by the operator, transmitted through a WIFI connection, and codes them in the form of sci commands. Roomba can work using two methods: autonomous(SAFE) or non-autonomous(FULL).

In the autonomous method (SAFE), there are essentially three modes : **Clean mode** is the normal cleaning program, starting in a spiral and then following a wall, until the room is determined to be clean. **Spot mode** cleans a small area. **Max mode** runs the standard cleaning algorithm until the battery is depleted.

In manually method (FULL), the operator specifies the direction and movement of roomba in real time. He can ask for these actions : **ADVANCE** is to forward, **LEFT** is a counter-clockwise rotation, and **RIGHT** is the same like Left but the rotation is in the other sens.

We focus on the part Robot of the system. We consider that the receiver ensures only a manual control of roomba. The IBD of assembling the receiver and roomba is given in Figure 7.9, and the protocols of interaction of the receiver and roomba are exposed in the form of HPSMs in Figure 7.10.

In Figure 7.9, we have two blocks : Receiver and Roomba. The block receiver has two ports: a provided port 'robotServices' and a required port 'sci-cmd-Req'. The provided port includes services that the receiver can perform to the coordinator, and the required port

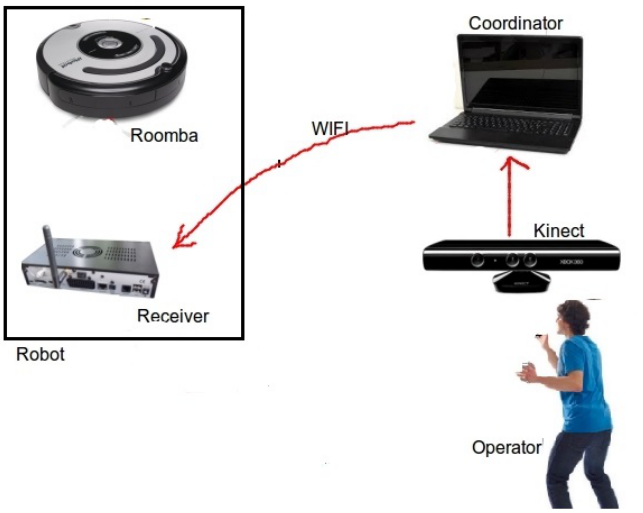


Figure 7.8: Case Study.

represents the services that the receiver can request from the block Roomba. The block Roomba has one provided port 'sci-cmd-Prov', through this port roomba offers its services.

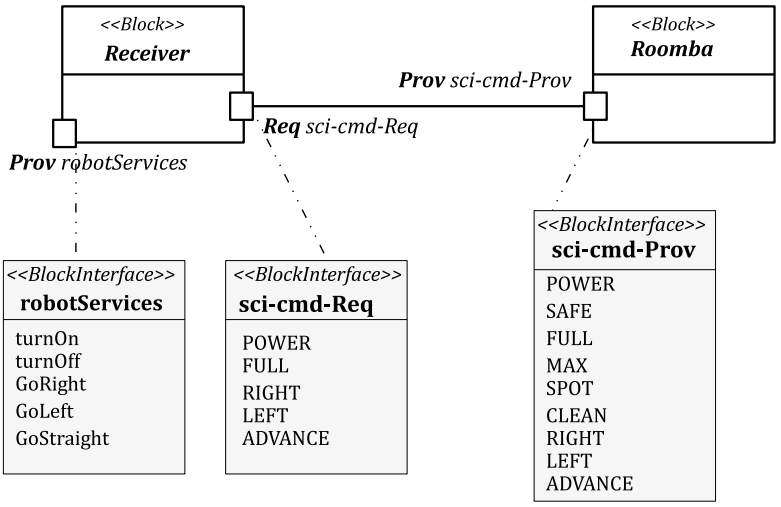


Figure 7.9: IBD of assembling the receiver and roomba.

In Figure 7.10, we present the HPSM of the receiver in the top and the HPSM of roomba in the bottom. The receiver plays the role of a converter between the coordinator and roomba. Initially, roomba is in state "OFF". When it receives the 'POWER' command, it goes to state "Wait". At this state, if roomba receives the 'SAFE' command, its state will be changed to "Autonomous". However, if it receives the 'FULL' command, it passes to the state "No-autonomous" and exactly to the state "Adv". At the state "Adv", if the receiver passes the 'ADVANCE' command to roomba, roomba remains in the same state. If it receives the 'LEFT' command, it changes its movement to a counter-clockwise rotation and it goes to state "LRot". Otherwise, by receiving the 'RIGHT' command, roomba leaves the state "Adv" and it goes to state "RRot". The same reactions will happen , when roomba is in the state "LRot" or in the state "RRot".

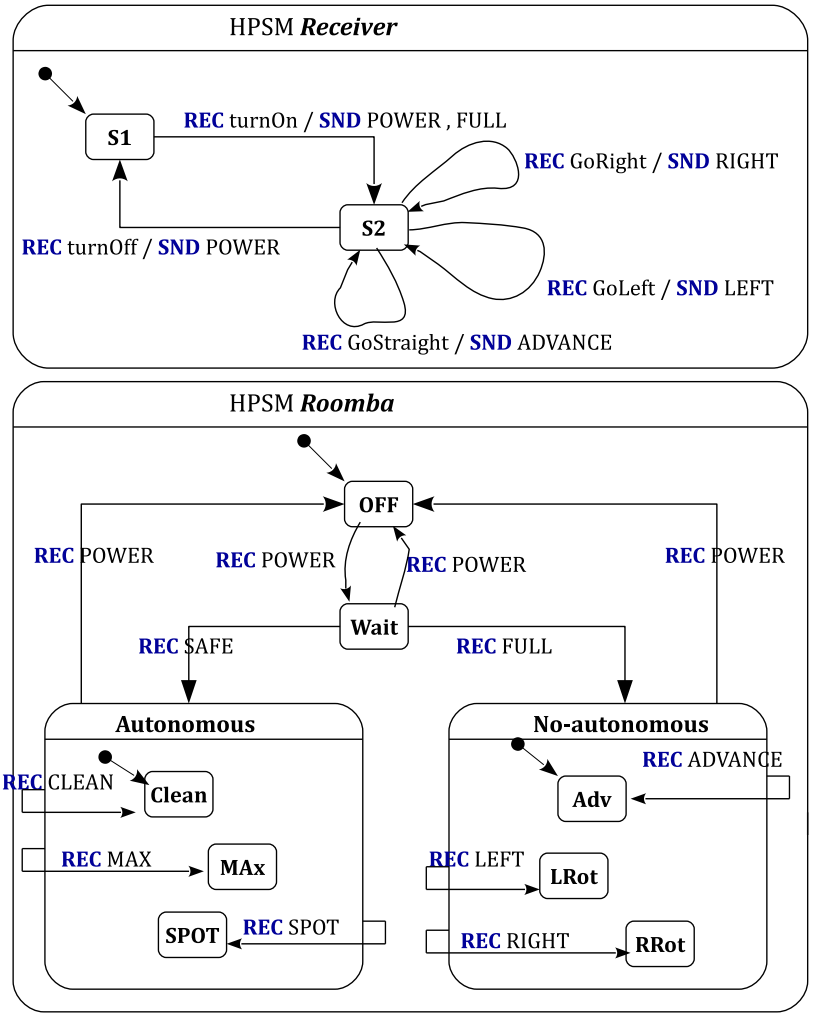


Figure 7.10: HPSM of the receiver and roomba.

The 1st step : Mapping between HPSM and HIA-ILT

Figure 7.11 shows the result of transforming HPSMs of the receiver and roomba into theirs equivalents of HIA-ILTs. The use of inter-level transitions $\overset{ADVANCE?}{\longrightarrow}$, $\overset{LEFT?}{\longrightarrow}$, $\overset{RIGHT?}{\longrightarrow}$ make the model obvious and small. For example, if we don't have the possibility of using inter-level transitions, the inter-level transition $No-autonomous \overset{ADVANCE?}{\longrightarrow} 6$ must be replaced by three transitions $6 \overset{ADVANCE?}{\longrightarrow} 6$, $7 \overset{ADVANCE?}{\longrightarrow} 6$ and $8 \overset{ADVANCE?}{\longrightarrow} 6$. the same for the two other inter-level transitions.

For clarity, we don't show the detail of the state Autonomous. Because it has the same structure like the No-autonomous state.

The 2nd step :Consistency verification between blocks

- $\Sigma^{I}_{receiver}$= {turnOn, turnOff, GoRight, GoLeft, GoStraight}
- $\Sigma^{O}_{receiver}$ = { POWER, FULL ,RIGHT, LEFT, ADVANCE}

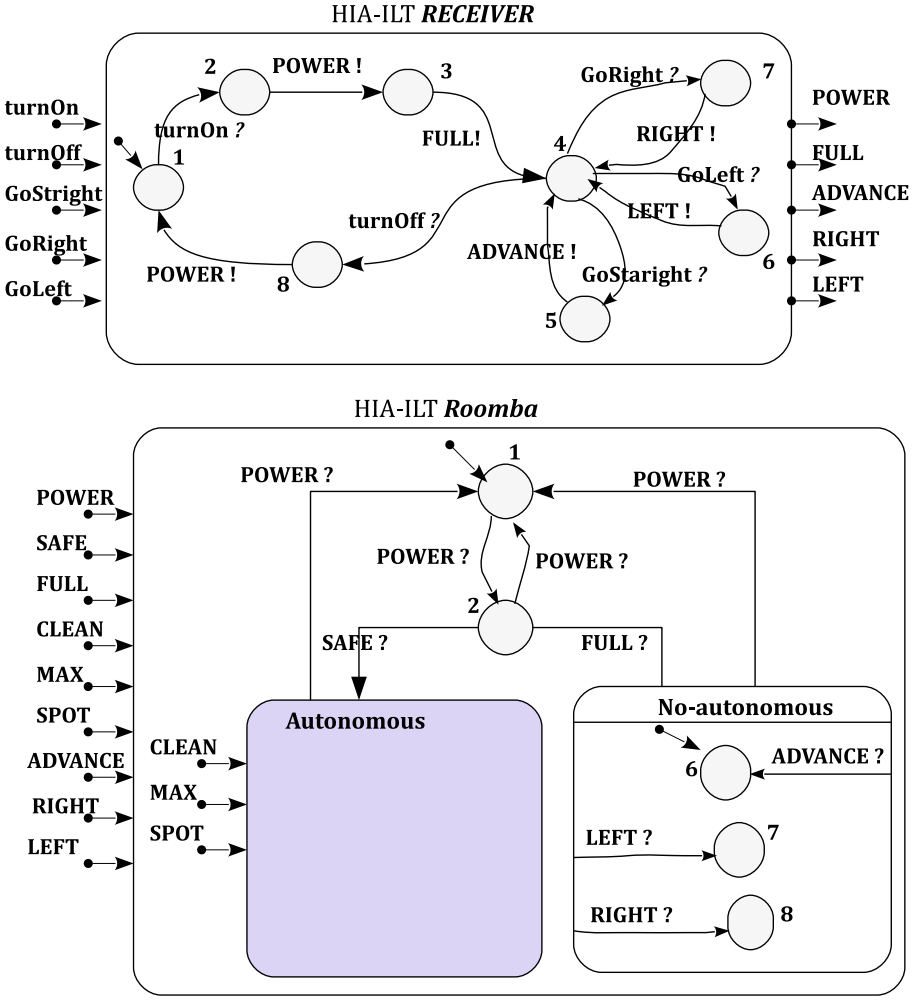


Figure 7.11: HIA-ILT of the receiver and roomba.

- Σ^I_{roomba} = {POWER, SAFE, FULL, MAX, SPOT, CLEAN, RIGHT, LEFT, ADVANCE}
- $\Sigma^O_{roomba} = \emptyset$
- $\Sigma^I_{receiver} \cap \Sigma^I_{roomba} = \Sigma^O_{receiver} \cap \Sigma^O_{roomba} = \Sigma^H_{receiver} \cap \Sigma_{roomba} = \Sigma_{receiver} \cap \Sigma^H_{roomba} = \emptyset$.

$\Rightarrow$ *receiver* ζ *roomba* , The receiver and roomba are composable.

The 3^{dth} step : The Selection of Composite States to Flatten

Shared(*receiver*,*roomba*) =
$(\Sigma^I_{receiver} \cap \Sigma^O_{roomba}) \cup (\Sigma^O_{receiver} \cap \Sigma^I_{roomba}) =$
$\{POWER, FULL, RIGHT, LEFT, ADVANCE\}$

In HA_{roomba}, we see that the composite state "autonomous" has no transition labelled with an action belong to Shared(*receiver*,*roomba*), so it will not be flattened. However, it is not the case for the composite state "no-autonomous". In Figure 7.12, we expose the result of flattening state "no-autonomous" of roomba. For visibility, we don't show the internal detail of the composite state 'autonomous'. By considering the state "autonomous" as

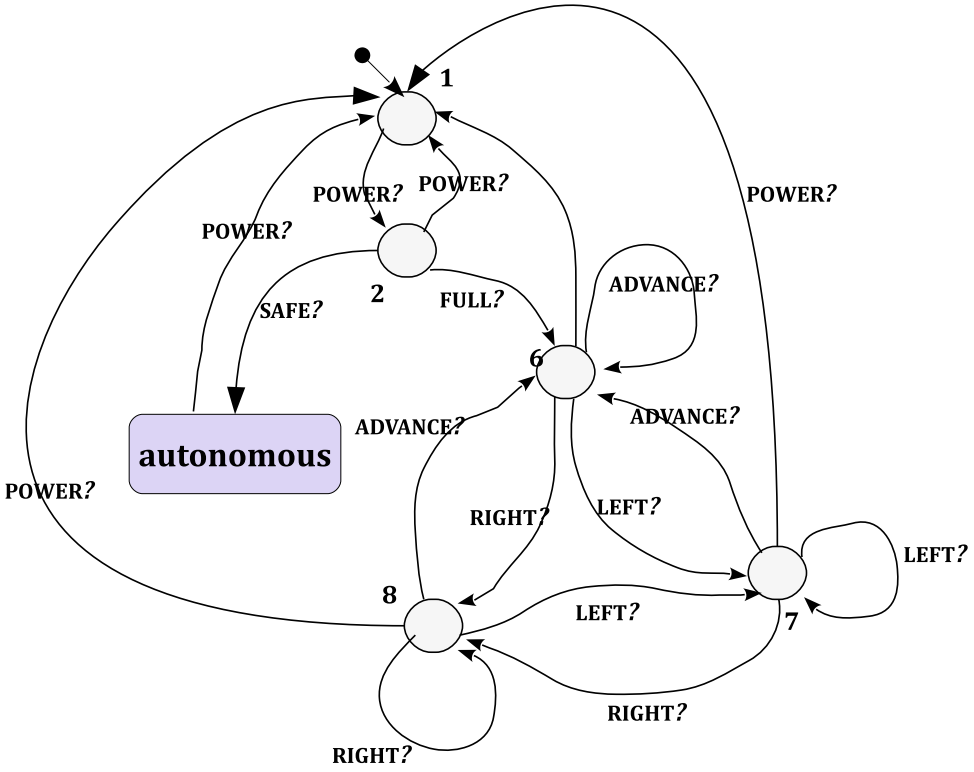


Figure 7.12: HIA-ILT of roomba after flattening no-autonomous state.

an abstract state, we avoid adding three states and nine transitions to the automaton of "roomba" which alleviates the next step of compatibility verification.

The 4^{th} step : Compatibility verification between blocks

In Figure 7.13, we show the abstract synchronous product $HA_{receiver} \otimes_a HA_{roomba}$. The state '3-auto', is an abstract state, it contains all the internal interaction of the composite state 'autonomous'. This abstraction allows handling simultaneously all internal states of 'autonomous' with their mother. To compute the composition $HA_{receiver} \|_a HA_{roomba}$, we must delete the state 3-auto because it is a deadlock state for the parallel execution of the receiver and roomba and it's an illegal state because full $\in$ Shared(*receiver*,*roomba*) and full $\in \Sigma_3^O$ and full $\notin \Sigma_{auto}^I$. All the internal states of 3-auto are illegal states (see theorem 1)

$HA_{receiver} \|_a HA_{roomba}$ has at least one reachable state, so HA_{roomba} is compatible with $HA_{receiver}$. Therefore, this receiver and this roomba can be assembled together.

7.5/ Conclusion

We have presented in this chapter the HPSM, a new convivial model for representing the interaction protocol of a SysML block. We have also presented HIA-ILT, a variant of IA, which allows the use of the composite states and the inter-level transitions. We have given rules to formalize the HPSM using HIA-ILT, and we have shown how to exploit the hierarchy and the abstraction present in the HIA-ILT to verify the compatibility between blocks.

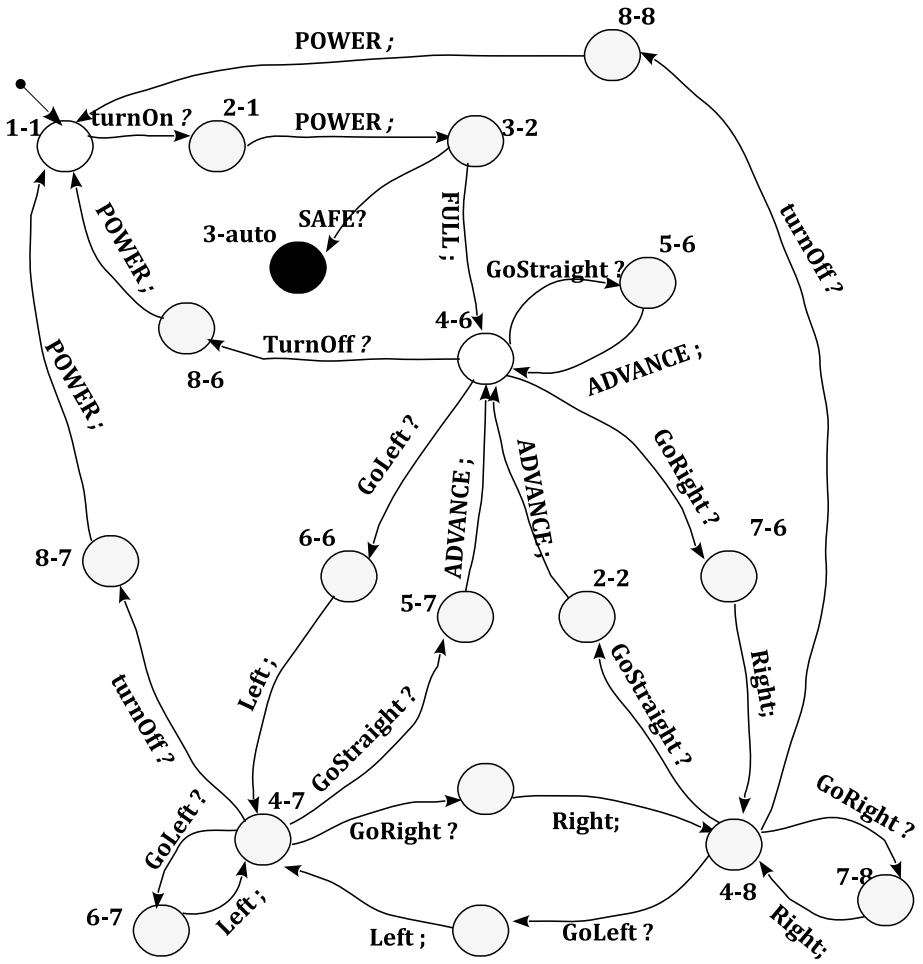


Figure 7.13: $\mathsf{HA}_{receiver} \otimes_a \mathsf{HA}_{roomba}$.

This verification allows the detection of the anomalies during the interaction. Our major objective was to alleviate the verification phase, by considering abstract states in HIA-ILT and flattening only some ones. The states to flatten are those having inside transitions that must synchronize with the other blocks transitions. So we gain, compared with the classical interface automata approach, on the size of the product automata HIA-ILT to analyse when we verify the compatibility between blocks. Two blocks are considered compatibles if the composition of their HIA-ILTs is not empty.

8

SysML Blocks Adaptation

How to assemble components designed in isolation? That is the major question on which CBSE domain tries to give more precise and adequate answers. As we have mentioned before, CBSE is considered as a natural consequence to the object oriented paradigm and the emergence of platforms of components (i.e CORBA, CCM). Its major goal is to build a market of software components (the so called COTS: Commercial-Off-The-Shelf), in which the developer finds the adequate components to integrate to its application.

System engineering also adopts the principle of using the component as the development unit. This appears clearly through SysML [OMG12b], a language that is used to design systems that include software and hardware. The System Modelling Language (SysML), through its diagrams, fosters the view point that takes the system as a set of components (the so called blocks). The Block Definition Diagram (BDD) of SysML can be seen as a tree of blocks, where the leaf nodes are the concrete blocks and the rest nodes until the root are abstract blocks. The abstract ones are called composite blocks, they are composed by assembling a set of blocks located in a less level of hierarchy.

In this chapter, we propose a bottom-up approach to build the system by adapting SysML blocks. Starting from a specification of a system part, which we consider as a SysML composite block 'B' to be built, the architect selects some SysML blocks, and adapts them using our method to meet the specification of B. In the next step of the development, the composite block B and another set of blocks will be used to achieve the specification of their parent. Thus, in our approach, we build an adapter per a composite block, the sub-blocks use this adapter to interact with the rest of the system.

Contents

The adaptation concerns the interaction protocols of the blocks. In this chapter, we start from the SysML Sequence Diagrams (SDs) that model the interactions of each block with its environment. Due to the fact that SysML SDs are not formal, we can't base on them to define the adaptation rules. Thus, we propose to translate SDs on their equivalents of Interface Automata (IA). Thus, we use the interface automata [dAH01] as formalism to formally specify the interaction protocol of the reused blocks (sub-blocks), the adapter block and the specification of the part to be built (the parent block).

As we have mentioned in the related works part, our notion of the adapter differs from the notion used in the existing works [IT03b, PST07, CMM12, CPS06a, BBC05a], which define the adapter as a protocol converter. In fact, in our approach the adapter has two roles. It plays its role as a converter between the reused blocks on the one hand, and between the reused blocks and their future parent block on the other hand. It plays the second role as a complement by performing to the reused blocks what they require and it's not planned to be required by their parent, and to offer what the parent must provide and it's not provided by any part of it.

The remainder of this chapter is organized as follows: In section 8.1, we present our approach of adapting SysML blocks, starting from the selection of the reused blocks until the generation of the adapter, and in section 8.2, we illustrate our approach through a case study.

8.1/ Our Incremental Approach for Adapting SysML Blocks

In this section, we explain our bottom-up approach to assemble and adapt SysML blocks which are designed separately. In Figure 8.1, we give the different steps of our method.

- We start by specifying the interaction protocol of the part of the system that we want to develop and integrate to our system. We model this part as a SysML composite block B which will contain the reused blocks and their adapter.
- After that, we can select the set of blocks $\{B_i\}$ to reuse. During the selection of these blocks, we take into consideration the specification that we want to fulfil.
- Next, basing on the reused blocks and the specification modelled by the composite block B, we can deduce if it is possible that these blocks may participate in meeting the specification of B. If it is the case, we compute the interaction protocol of the adapter and its structure.
- Finally, we integrate the adapter block with the selected blocks to build the BDD and the IBD of the parent block B.

In the next step of the system development (see Figure 8.2), the composite block B will be used to meet the specification of its parent block. Thus, the unit used to construct the system is the composite block, and we build an adapter per a composite block.

8.1.1/ The First Phase: Defining a Specification for the Part to Develop

At this phase, we must specify the structure and the interaction protocol of the part B that we want to develop. Structurally, we model this part as a SysML block. On other hand,

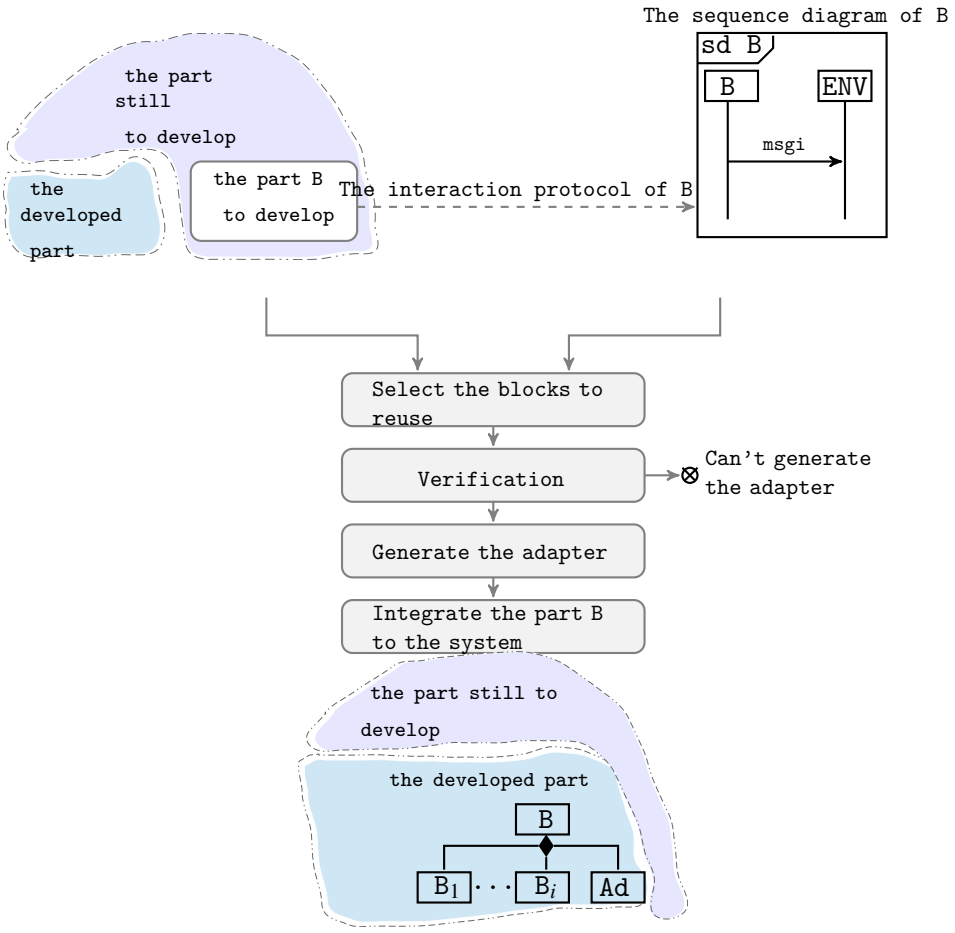


Figure 8.1: The proposed approach.

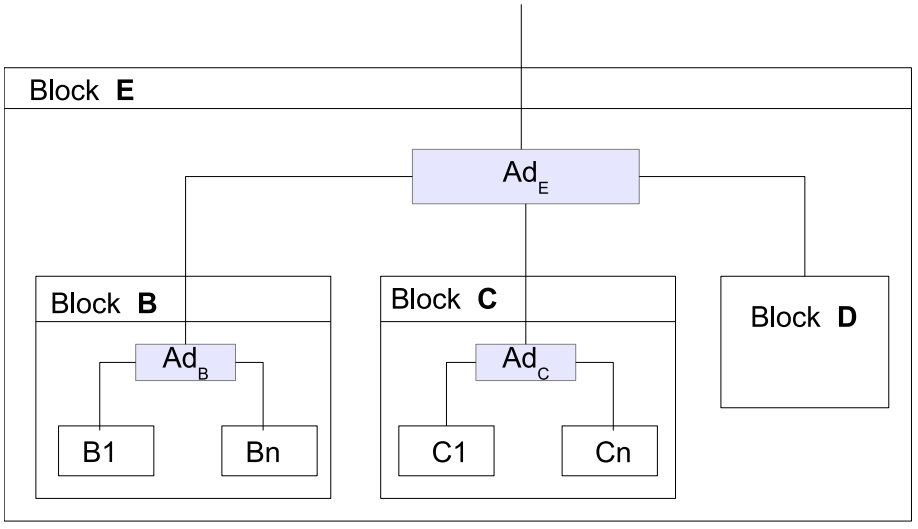


Figure 8.2: Incremental approach.

we make use of the SysML Sequence Diagram (SD) to specify the different interactions of our part with its environment. It is easy to model these interactions with SD. However, formal verification is still inapplicable directly on SDs, because they are expressed in a

semi-formal modelling language. Hence, to apply a verification method, it is necessary to transform the SysML SDs to formal models. In our work, we have used as formal model Interface Automata (IAs). To perform this transformation, we make reference to chapter 6, where we have defined the correspondences between these two models.

8.1.2/ The Second Phase: The Selection of the Reused Blocks $\{B_i\}$

In this phase, the architect can select a set of blocks $\{B_i\}$ that will participate to meet the specification of the composite block B (constructed in the previous phase). Each selected block must be equipped with a sequence diagram describing its interactions with its environment. The result of this phase will be a set of SysML blocks to reuse with their sequence diagrams and a contract C that specifies the correspondences between the services of the blocks.

To specify some conditions on these blocks and the contract format, we need to define each block B_i by three sets. To define these sets, we have based on the formalization of SysML diagrams given in chapter 5.

- PS_{B_i} : the set of provided services,

  PS_{B_i}={ps |∃ p∈ Ports(B_i), ps ∈ p.type.Op ∧ p.Direction=provided}

- RS_{B_i} : the set of required services,

  RS_B={rs |∃ p∈ Ports(B_i), rs ∈ p.type.Op ∧ p.Direction=required}

- IOp_{B_i} : the set of internal operations,

  IOp_B= {o | o∈ operations(B_i)}

The adaptation contract C is constructed incrementally. After adding a new block B_i, the architect must specify the correspondences between the services of B_i and the services of the specification of B on the one hand, and between the services of B_i and the services of the blocks already chosen ($\{B_j\}_{j<i}$) on the other hand. These correspondences represent the contract C = $\{v_i\}_{i=1..m}$.

Each element v_i of the adaptation contract C takes the format of a synchronous vector: ⟨ a_1, a_2, ... , a_n, s ⟩ , where:

s ∈ PS_{spec} ∪ RS_{spec} ∪ {ε} ∧ a_i ∈ PS_{B_i} ∪ RS_{B_i} ∪ {ε}.

Each vector contains two elements a_i and a_j which are different from epsilon, this means that the service a_i of the block B_i corresponds to the service a_j of the block B_j. We can see that the adaptation contract C is the union of two sub-contracts elements: C= $C_{subBlocks}$∪ C_{spec} , where:

- $C_{subBlocks}$: specifies the correspondences between the reused sub-blocks $\{B_i\}$, $C_{subBlocks}$={⟨a_1, a_2, ..., a_n, s⟩}, where s=ε ∧ a_i∈PS_{B_i} ∪ RS_{B_i} ∪ {ε}

- C_{spec}: specifies the correspondences between the reused sub-blocks $\{B_i\}$ and the specification of the parent block,
  C_{spec}={⟨a_1, a_2, ..., a_n, s⟩}, where s≠ε ∧ a_i∈PS_{B_i} ∪ RS_{B_i} ∪ {ε}

8.1.3/ The Third Step: the Contract and the Reused Blocks Verification

The contract C must respect some validity conditions. The contract C is valid if its sub-contracts are valid as well:

Because in our approach, we are interested with mapping of type one-to-one, our sub-contract $C_{subBlocks}$ must verify the condition 1 and the sub-contract C_{spec} must verify the condition 2.

condition 1 ($C_{subBlocks}$ validity):

condition 1.1: A required service of a block corresponds at most to one provided service of another block.

$\forall\ v_i = \langle\ e_{i1}, \ldots, e_{in}, \varepsilon\ \rangle \in C_{subBlocks}\ (e_{ik} = a, a \in RS_{B_k}$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, \ldots, e_{jn}, \varepsilon\ \rangle \in C_{subBlocks}, e_{jk} \neq a)$

condition 1.2: A provided service of a block corresponds at most to one required service of another block.

$\forall\ v_i = \langle\ e_{i1}, \ldots, e_{in}, \varepsilon\ \rangle \in C_{subBlocks}\ (e_{ik} = a, a \in PS_{B_k}$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, \ldots, e_{jn}, \varepsilon\ \rangle \in C_{subBlocks}, e_{jk} \neq a)$

The sub-contract C_{spec} must verify condition 2:

condition 2 (C_{spec} validity):

condition 2.1: A provided service **a** of the specification can correspond at most to one provided service **b** of the sub-blocks.

$\forall\ a \in PS_{spec}, \forall\ v_i = \langle\ e_{i1}, \ldots, e_{in}, a\rangle \in C_{spec}\ (e_{ik} = b$
$\Rightarrow b \in PS_{B_k} \wedge \forall\ v_{j \neq i} = \langle\ e_{j1}, \ldots, e_{jn}, c\ \rangle \in C\ , c \neq a \wedge e_{jk} \neq b)$

condition 2.2: A required service a of the specification can correspond at most to one required service b of the sub-blocks.

$\forall\ a \in RS_{spec}, \forall\ v_i = \langle\ e_{i1}, \ldots, e_{in}, a\rangle \in C_{spec}\ (e_{ik} = b$
$\Rightarrow b \in RS_{B_k} \wedge \forall\ v_{j \neq i} = \langle\ e_{j1}, \ldots, e_{jn}, c\rangle \in C\ , c \neq a \wedge e_{jk} \neq b)$

The reused blocks and the specification must also verify the conditions of consistency.

condition 3 (Consistency verification of the selected sub-blocks and the parent block): This condition must be verified by the parent block (B) that represents the specification of the part to develop and the reused blocks ($\{B_i\}$), that will be children blocks of B.

- A provided service of a sub-block can not be a required service of the parent block: $\forall\ a \in PS_{B_i}, a \notin RS_{A_{spec}}$
- A required service of a sub-block can not be a provided service of the parent block: $\forall\ a \in RS_{B_i}, a \notin PS_{A_{spec}}$

8.1.4/ The Fourth Step: Generating the Adapter

To generate the adapter, we need to compute the global interaction protocol of the reused blocks. To do that, we need to transform the sequence diagrams of blocks into their equivalents of interface automata. To compute the parallel composition of interface automata the reused blocks, we need to adapt the notions of synchronous and parallel composition to take into consideration the contract and the corresponding actions instead of the

shared actions. Thus, we have defined in [BCHM15], the notions of contract-based synchronous product ($\otimes_c$) and contract-based parallel composition ($\|_c$).

Definition 1 (Contract-based synchronous product):

The contract-based synchronous product is possible between two interface automata A_i and A_j, if they are composable ($\Sigma^I_{A_i} \cap \Sigma^I_{A_j} = \Sigma^O_{A_i} \cap \Sigma^O_{A_j} = \Sigma^H_{A_i} \cap \Sigma_{A_j} = \Sigma_{A_i} \cap \Sigma^H_{A_j} = \emptyset$), and the adaptation contract is valid (it verifies the condition 1).

Before defining the contract-based synchronous product between two interface automata A_i and A_j, we need to define $Corresponding(A_i, A_j)$, the set of corresponding actions between the interface automata A_i and A_j, and the function corresp(a) that returns the action that corresponds to the action a by referring to the adaptation contract:

$\mathtt{Corresponding(A}_i\mathtt{,A}_j\mathtt{)=}$
$\mathtt{\{a}\in\Sigma^I_{A_i} \cup \Sigma^O_{A_i} \cup \Sigma^I_{A_j} \cup \Sigma^O_{A_j} \mid \exists \mathtt{v=}\langle \mathtt{e}_1,\ldots,\mathtt{e}_n,\varepsilon\rangle\in \mathtt{C}_{subBlocks}\mathtt{,\ e}_k\mathtt{=a\}}$
$\mathtt{corresp(a)=\{a'|}\exists \mathtt{v}\in\mathtt{C,\ }\exists\ \mathtt{(i,\ j)}\in \mathbb{N}^2\mathtt{,\ v=}\langle \mathtt{a}_1,\ldots,\mathtt{a}_n\rangle \wedge \mathtt{a}_i\mathtt{=a}\wedge\ \mathtt{a}_j\mathtt{=a'\}}$

Definition 22: Contract-based synchronous product

We define the contract-based synchronous product of A_i and A_j as:

$$\mathtt{A}_i\otimes_c\mathtt{A}_j=\langle \mathtt{S}_{A_i\otimes_c A_j},\ \mathtt{I}_{A_i\otimes_c A_j},\ \Sigma^I_{A_i\otimes_c A_j},\ \Sigma^O_{A_i\otimes_c A_j},\ \Sigma^H_{A_i\otimes_c A_j},\ \delta_{A_i\otimes_c A_j}\rangle$$

- $\mathtt{S}_{A_i\otimes A_j} = \mathtt{S}_{A_i} \times \mathtt{S}_{A_j}$ and $\mathtt{I}_{A_i\otimes_c A_j} = \mathtt{I}_{A_i} \times \mathtt{I}_{A_j}$;
- $\Sigma^I_{A_i\otimes_c A_j} = (\Sigma^I_{A_i} \cup \Sigma^I_{A_j}) \setminus$ Corresponding($\mathtt{A}_i$, $\mathtt{A}_j$);
- $\Sigma^O_{A_i\otimes_c A_j} = (\Sigma^O_{A_i} \cup \Sigma^O_{A_j}) \setminus$ Corresponding($\mathtt{A}_i$, $\mathtt{A}_j$);
- $\Sigma^H_{A_i\otimes_c A_j} = \Sigma^H_{A_i} \cup \Sigma^H_{A_j} \cup$ Corresponding($\mathtt{A}_i$, $\mathtt{A}_j$);
- $((\mathtt{s}_i,\mathtt{s}_j),\mathtt{a},(\mathtt{s}_i\mathtt{'},\mathtt{s}_j\mathtt{'})) \in \delta_{A_i\otimes_c A_j}$ if:
  - $\mathtt{a} \notin$ Corresponding($\mathtt{A}_i$,$\mathtt{A}_j$) $\wedge$ ($\mathtt{s}_i$,a,$\mathtt{s'}_i$) $\in \delta_{A_i} \wedge\ \mathtt{s}_j = \mathtt{s'}_j$
  - $\mathtt{a} \notin$ Corresponding($\mathtt{A}_i$,$\mathtt{A}_j$) $\wedge$ ($\mathtt{s}_j$,a,$\mathtt{s'}_j$) $\in \delta_{A_j} \wedge\ \mathtt{s}_i = \mathtt{s'}_i$
  - $\mathtt{a} \in$ Corresponding($\mathtt{A}_i$,$\mathtt{A}_j$) $\wedge\ \mathtt{a} \in \Sigma^O_{A_i}$ $\wedge$ ($\mathtt{s}_i$,a,$\mathtt{s'}_i$) $\in \delta_{A_i}$ $\wedge$ ($\mathtt{s}_j$,corresp(a),$\mathtt{s'}_j$) $\in \delta_{A_j}$
  - $\mathtt{a} \in$ Corresponding($\mathtt{A}_i$,$\mathtt{A}_j$) $\wedge\ \mathtt{a} \in \Sigma^O_{A_j}$ $\wedge$ ($\mathtt{s}_j$,a,$\mathtt{s'}_j$) $\in \delta_{A_j}$ $\wedge$ ($\mathtt{s}_i$,corresp(a),$\mathtt{s'}_i$) $\in \delta_{A_i}$

This product absorbs the transitions $(s_i, s_j) \xrightarrow{a!} (s'_i, s_j) \xrightarrow{corresp(a)} (s'_i, s'_j)$ and the transitions $(s_i, s_j) \xrightarrow{a!} (s_i, s'_j) \xrightarrow{corresp(a)} (s'_i, s'_j)$ by replacing them by a single transition $(s_i, s_j) \xrightarrow{a;} (s'_i, s'_j)$. This absorption is helpful when we need to compute the synchronous product between multiple IAs having corresponding actions. It allows the atomic execution of the emission of an action and the reception of its corresponding action.

The contract-based parallel composition between two interface automata A_i and A_j is defined as:

Definition 23: Contract-based parallel composition

$A_i \| _c A_j = A_i \otimes_c A_j$ after removing illegal states and all states reached from these illegal states by enabling output and internal actions. The set of illegal states is defined as:

$$\texttt{Illegal(A}_i\texttt{,A}_j\texttt{)=} \left\{ \begin{array}{c} (s_i, s_j) \in S_{A_i} \times S_{A_j} \mid \exists a \in Corresponding(A_i, A_j). \\ \left(\begin{array}{c} a \in \Sigma^O_{A_i}(s_i) \wedge corresp(a) \notin \Sigma^I_{A_j}(s_j) \\ \vee \\ a \in \Sigma^O_{A_j}(s_j) \wedge corresp(a) \notin \Sigma^I_{A_i}(s_i) \end{array} \right) \end{array} \right\}$$

Thus, the global interaction protocol $\texttt{A}_G$ of the sub-blocks $\{B_i\}_{i=1..n}$ is obtained by composing their interface automata $\{A_i\}_{i=1..n}$ using the contract based parallel composition:

$$\texttt{A}_G \texttt{ = A}_1 \|_c \texttt{ A}_2 \|_c \texttt{ ... } \|_c \texttt{ A}_n$$

At each given stage i of computing the composition, we must compute the composition between the interface automaton $\texttt{A}_c$ (where $\texttt{A}_c \texttt{ = A}_1 \|_c \texttt{ ... } \|_c \texttt{ A}_{i-1}$) and the interface automaton $\texttt{A}_i$ of the block $\texttt{B}_i$. At each stage i, we must verify the condition 4.

Condition 4: (The blocks must be compatible) $\texttt{A}_c$ must be not empty.

Now, we can deduce the interaction protocol of the adapter by using the interface automaton A_G and basing of this relation:

$$\texttt{A}_{spec} \geq \texttt{A}_G \|_c \texttt{ A}_{ad}$$

It means that the automaton resulting from composing interface automata of the blocks $\{\texttt{B}_i\}$ with the adapter automaton, must refine the interface automaton of the part B. Thus, to deduce A_{ad}, we refer to the formula proposed in [BRO8]. To compute the most general solution R where $Q \geq P \| R$, the authors in [BRO8] prove that $R = mirror(P \| mirror(Q))$, where P, R and Q are interface automata, and mirror(Q) is the interface automaton Q with inputs and outputs interchanged. We define formally the notion of mirror as follows:

$$\begin{aligned} \texttt{mirror(Q)=\{Q' } \mid & \forall \texttt{ (s, a!, s')} \in \delta_Q, \exists \texttt{ (s, a?, s')} \in \delta_{Q'} \wedge \\ & \forall \texttt{ (s, a?, s')} \in \delta_Q, \exists \texttt{ (s, a!, s')} \in \delta_{Q'} \wedge \\ & \forall \texttt{ (s, a;, s')} \in \delta_Q, \exists \texttt{ (s, a;, s')} \in \delta_{Q'}\texttt{\}} \end{aligned}$$

Thus, in our case, because we have corresponding actions between automata instead of shared actions, the $\texttt{A}_{ad}$ must be computed as follows:

$$\begin{aligned} \texttt{A}_{ad} \texttt{ = mirror(A}_G \|_c \texttt{ mirror(A}_{spec}\texttt{))} = \\ \texttt{mirror(A}_1 \|_c \texttt{ ...} \|_c \texttt{ A}_n \|_c \texttt{ mirror(A}_{spec}\texttt{))} \end{aligned}$$

Condition 5: (A_G and $mirror(A_{spec})$ must be compatible) A_{ad} is not empty

If the condition 5 is verified, we can deduce the real interaction protocol of the adapter by applying the algorithm 2, which allows to return transitions absorbed in the contract based synchronous product.

Algorithm 2 Deduce the interaction protocol of the adapter

INPUT: A_{ad} =⟨ S_{ad}, I_{ad}, Σ^I_{ad}, Σ^O_{ad}, Σ^H_{ad}, δ_{ad} ⟩, C
OUTPUT: $A_{adapter}$ =⟨$S_{adapter}$, $I_{adapter}$, $\Sigma^I_{adapter}$, $\Sigma^O_{adapter}$, $\Sigma^H_{adapter}$, $\delta_{adapter}$⟩

1: - Create a copy $A_{adapter}$ of A_{ad}.
2: - Construct the set T of all transitions ($s \xrightarrow{a;} s' \in \delta_{adapter}$), where a appears in the contract C.
3: - Replace all $s \xrightarrow{a;} s' \in \delta_{adapter}$ where $s \xrightarrow{a;} s' \in T$, by $s \xrightarrow{a?} s'' \xrightarrow{corresp(a)!} s'$.

According to the contract based synchronous product, the transitions labelled with internal actions a; in A_{ad}, which appear in the contract, represent the transitions where the adapter plays the role of a converter: so each transition of this set must be replaced by two transitions. The first is labelled with the input action a? and the second by the corresponding action corresp(a)!. This means that the adapter receives the action a? from a block, after that, it converts it to the suitable input of another block and it conveys it using an output action corresp(a)!. The transitions which aren't selected by the algorithm 2 are those where the adapter plays the role of a complement and not a converter.

Now, we can construct the architecture of the SysML adapter block $B_{adapter}$. We use the algorithm 3 to deduce the set of ports of $B_{adapter}$. To build the BDD and the IBD of the part B,we apply the Algorithm 3 and the algorithm 5. The role of algorithm 4 is to establish the composition relations between the parent block B and its sub-blocks $\{B_i\}$, and a composition relation between the parent block B and the adapter block B_{ad}. We use the algorithm 5 to generate the IBD of the block B. It bases on relying the adapter block ports with the ports of the sub-blocks $\{B_i\}$ and the parent block B.

Algorithm 3 Construct the SysML adapter block

INPUT: $A_{adapter}$=⟨$S_{adapter}$,$I_{adapter}$,$\Sigma^I_{adapter}$,$\Sigma^O_{adapter}$,$\Sigma^H_{adapter}$,$\delta_{adapter}$⟩
OUTPUT: $B_{adapter}$ =⟨ *'Adapter'*, V, O, C, P, *Ports* ⟩

1: -Create the adapter block $B_{adapter}$=⟨ *'Adapter'*, ∅, ∅, ∅, ∅, ∅ ⟩
2: // CREATE THE LIST OF PORTS OF THE ADAPTER THAT MUST BE LINKED TO THE PORTS OF THE PARENT BLOCK.
3: if $\Sigma^I_{adapter} \cap \Sigma^I_{spec} \neq \emptyset$ then
4: -create a new provided port p which offers the services $\Sigma^I_{adapter} \cap \Sigma^I_{spec}$
5: -add p to the ports list of $B_{adapter}$
6: if $\Sigma^O_{adapter} \cap \Sigma^O_{spec} \neq \emptyset$ then
7: create a new required port p which requires the services $\Sigma^O_{adapter} \cap \Sigma^O_{spec}$
8: add p to the ports list of $B_{adapter}$
9: // CREATE THE LIST OF PORTS OF THE ADAPTER THAT MUST BE LINKED TO THE PORTS OF SUB-BLOCKS $\{B_i\}$
10: for all B_i in the list of sub-blocks $\{B_i\}$ do
11: if $\Sigma^I_{adapter} \cap \Sigma^O_{B_i} \neq \emptyset$ then
12: create a new provided port p which offers the services $\Sigma^I_{adapter} \cap \Sigma^O_{B_i}$
13: add p to the ports list of $B_{adapter}$
14: if $\Sigma^O_{adapter} \cap \Sigma^I_{B_i} \neq \emptyset$ then
15: create a new required port p which requires the services $\Sigma^O_{adapter} \cap \Sigma^I_{B_i}$
16: add p to the ports list of $B_{adapter}$

Algorithm 4 Construct the BDD of the parent block B

INPUT: B, $\{B_i\}$, $B_{adapter}$
OUTPUT: $BDD_B = \langle B, R \rangle$

- Set the value of the blocks set of the BDD_B to: B= $\{B_i\}_{i=1..n} \cup \{B, B_{adapter}\}$
- Create a composition relation r_i between the parent block B and each block B_i where: SourceOf(r_i) = B, TargetOf(r_i) = B_i
- Create a composition relation r_{ad} between the parent block B and the adapter block $B_{adapter}$ where: SourceOf(r_{ad}) = B, TargetOf(r_{ad}) = $B_{adapter}$
- Set the value of the relations set of BDD_B to: R= $\{r_i\}_{i=1..n} \cup \{r_{ad}\}$

Algorithm 5 Construct the IBD of the parent block B

INPUT: B, $\{B_i\}$, $B_{adapter}$
OUTPUT: $IBD_B = \langle Parts, Ports, Connectors \rangle$

- Create instances $\{part_i\}_{i=1..n}$ of the blocks Set $\{B_i\}_{i=1..n}$.
- Create an instance *'ad'* of the adapter block $B_{adapter}$.
- Set the set *Parts* of IBD_B to: $\{part_i\}_{i=1..n} \cup$ {ad}
- Set the set *Ports* of IBD_B to: $\{Ports(part_i)\}_{i=1..n} \cup$ Ports(ad)

```
//CREATE CONNECTORS BETWEEN THE ADAPTER AND {part_i}_{i=1..n} .
for all part_i ∈ {part_i}_{i=1..n} do
  for all port p ∈ Ports(ad) do
    if ∃p' ∈ Ports(part_i) ∧ (p.type.Op ∩ p'.type.Op ≠ ∅) then
       create a connector between p and p'
//CREATE DELEGATION CONNECTORS BETWEEN THE ADAPTER AND THE PARENT BLOCK B.
for all port p ∈ Ports(ad) do
  if ∃p' ∈ Ports(B) ∧ (p.type.Op ∩ p'.type.Op ≠ ∅) then
     - create a connector between p and p'
```

8.2/ Case Study

To illustrate our SysML blocks adaptation approach, we give a simple example of a robot which is guided by a station. To simplify, we consider that the corresponding actions have the same name and we differentiate between them by adding the first letter of the block's name to each action.

Remark:

In each figure of the reused blocks, we present the architecture of the block with its interfaces, its sequence diagram, and the result of transforming its sequence diagram into an interface automaton.

8.2.1/ Generate the Adapters

To build this system, we start by building the robot (see Figure 8.3). We want that our robot receives a request to move. After that, it can either receive a request to stop, or to communicate its location. In this last case, the robot send the location data to its environment.

Thus, to build this robot, we have reused a motor and a controller (see Figure 8.4). At this step, we use the contract $C_{Contr \leftrightarrow Mot}$=

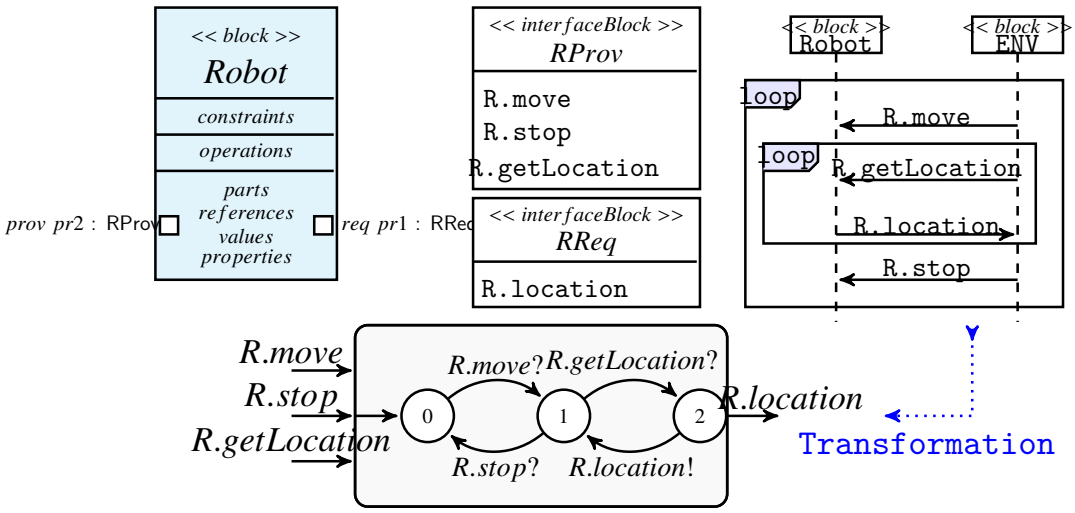


Figure 8.3: The Robot.

$\{\langle$C.on, M.on, $\varepsilon\rangle$, $\langle$C.off, M.off, $\varepsilon\rangle$, $\langle$C.move, ε, R.move$\rangle$, $\langle$C.stop, ε, R.stop$\rangle\}$

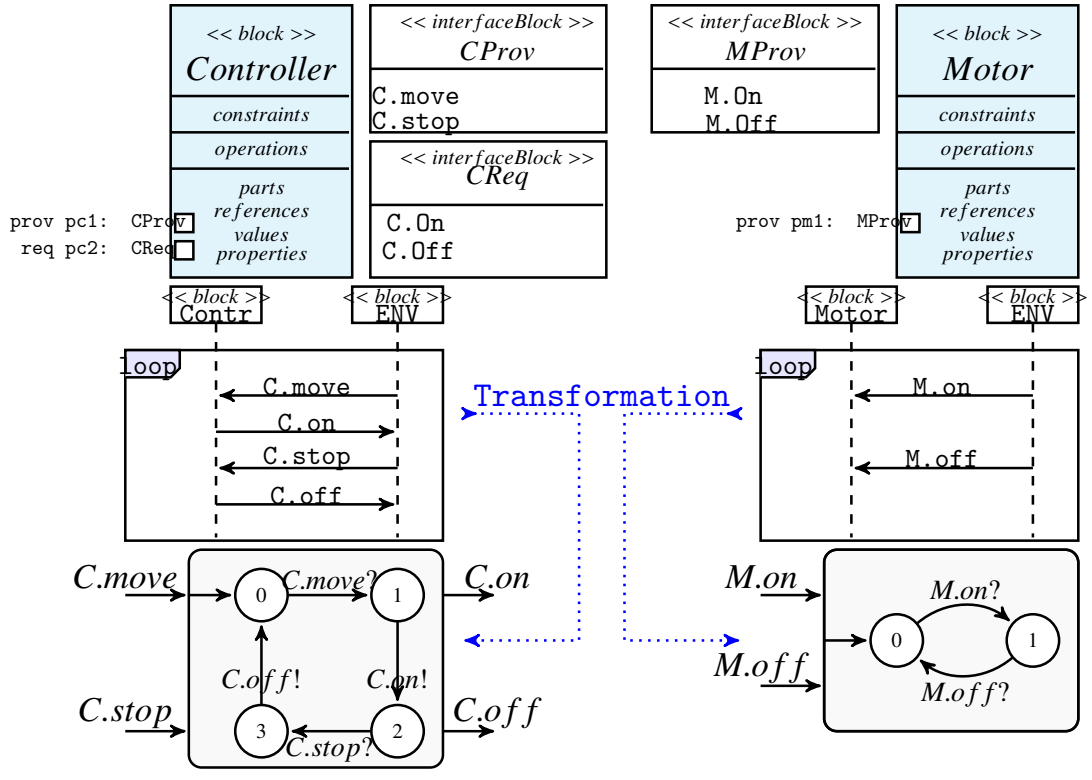


Figure 8.4: The Controller and the Motor blocks.

Basing on the contract $C_{Contr \leftrightarrow Mot}$, we adapt these two blocks to meet the specification of the robot by introducing the adapter (a converter-complement) $Ad_{Contr \leftrightarrow Mot}$ (see Figure 8.5), where the interaction protocol of the adapter $Ad_{Contr \leftrightarrow Mot}$ is represented using an interface automaton and it is computed by applying the algorithm 2 on the result of this formula:

$\mathtt{IA}_{Ad_{Contr \leftrightarrow Mot}} = \mathtt{mirror}(\mathtt{IA}_{Controller} \parallel_c \mathtt{IA}_{Motor} \parallel_c \mathtt{mirror}(\mathtt{IA}_{Robot}))$

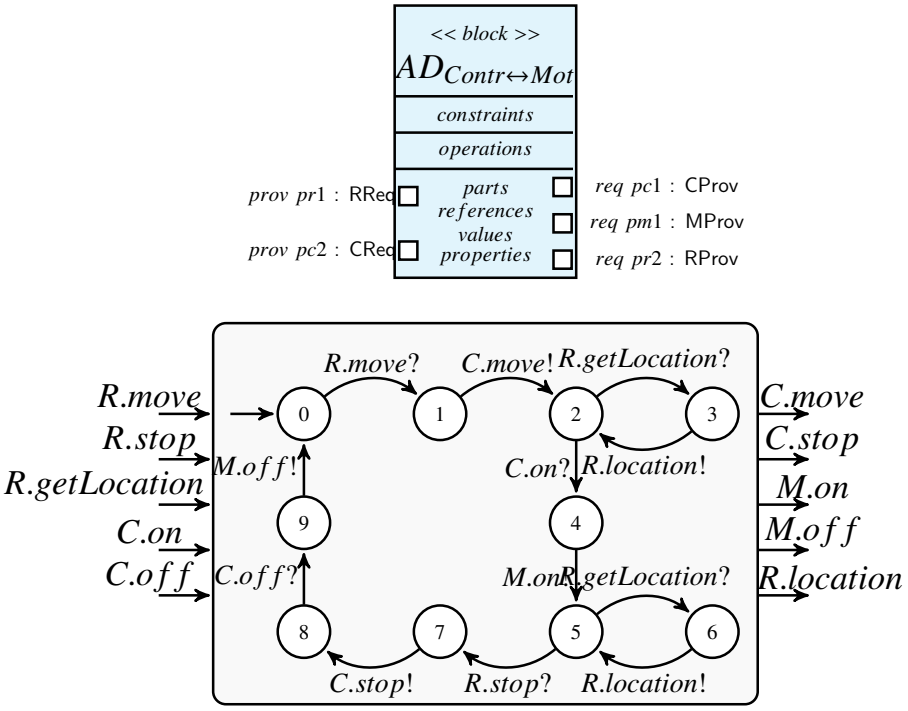


Figure 8.5: The adapter $AD_{Contr \leftrightarrow Mot}$.

Our robot must be guided by a station. That is why, we have reused the station at Figure 8.6, which is modelled using SysML and interface automata.

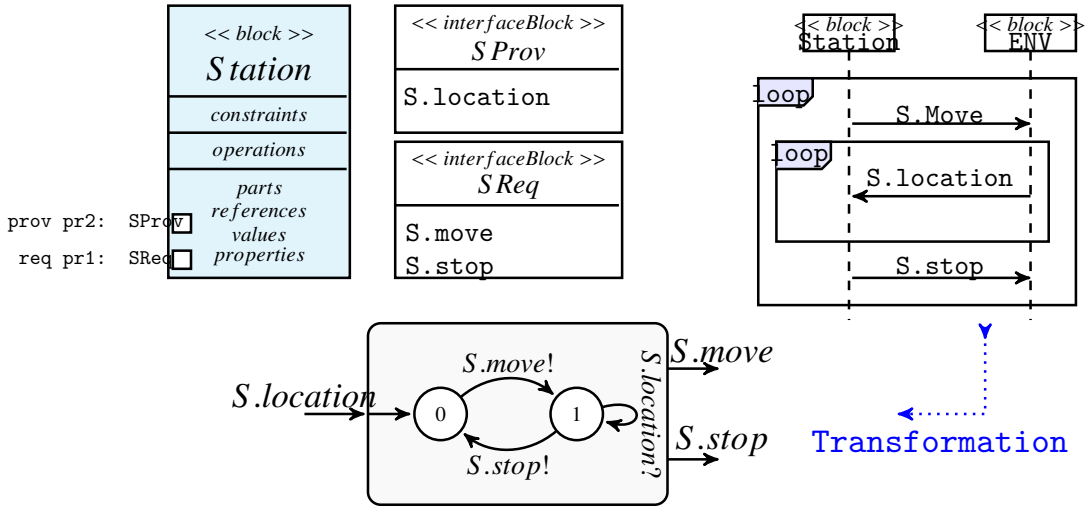


Figure 8.6: The station.

To adapt the station and the robot, we have used the contract $C_{Rob \leftrightarrow Sta}$:

$\{ \langle$ R.move, S.move, $\varepsilon \rangle, \langle$ R.stop, S.stop, $\varepsilon \rangle, \langle$ R.location, S.location, $\varepsilon \rangle\}$

At this step, we have a closed system, which cannot interact with its environment. Thus, the interface automaton which specifies the interactions of our station and our robot with their environment will be empty. Thus, to compute the adapter $Ad_{Rob \leftrightarrow Sta}$ (see Figure 8.7), we need just the contract $C_{Rob \leftrightarrow Sta}$, where, in Figure 8.7, the interaction protocol of the

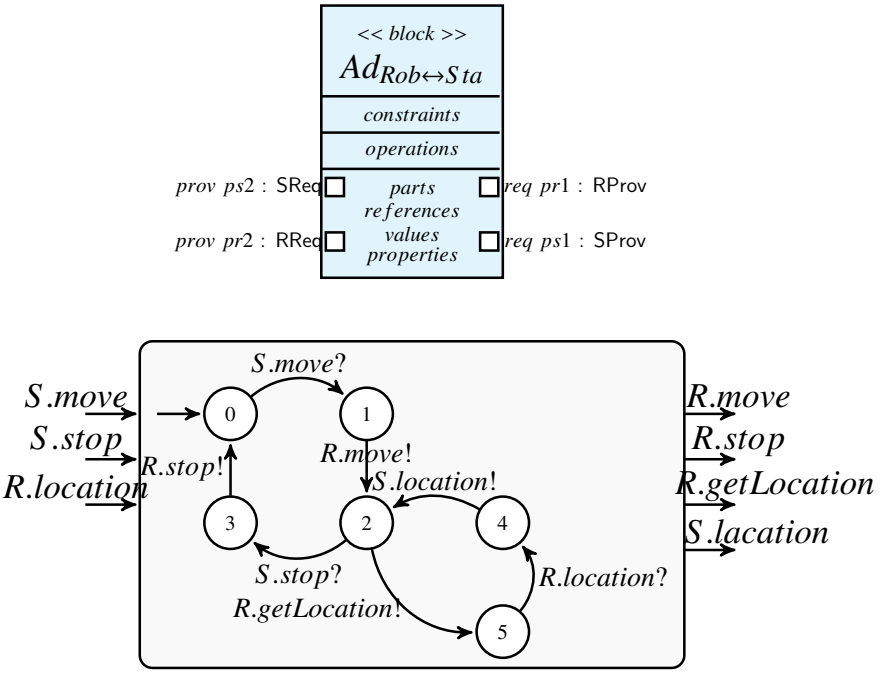


Figure 8.7: The adapter $Ad_{Rob \leftrightarrow Sta}$.

adapter $Ad_{Rob \leftrightarrow Sta}$ is represented using an interface automaton and it is computed by applying the algorithm 2 on the result of this formula:

$\mathtt{IA}_{Ad_{Rob \leftrightarrow Sta}} = \mathtt{mirror}(\mathtt{IA}_{Robot} \parallel_c \mathtt{IA}_{Station})$

8.2.2/ Deduce the BDD and the IBDs of the Composite Blocks

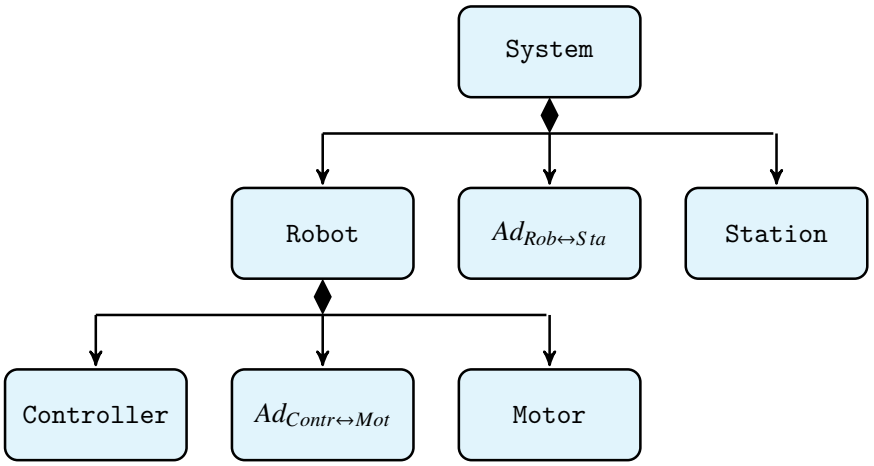


Figure 8.8: The Block definition diagram of the system.

In Figure 8.8, we represent the blocks of the system and the composition relations between them, we have used Algorithm 4 to generate this BDD. However, we have applied Algorithm 5 to generate the internal structure of each composite block: the system and the robot blocks (see Figure 8.9).

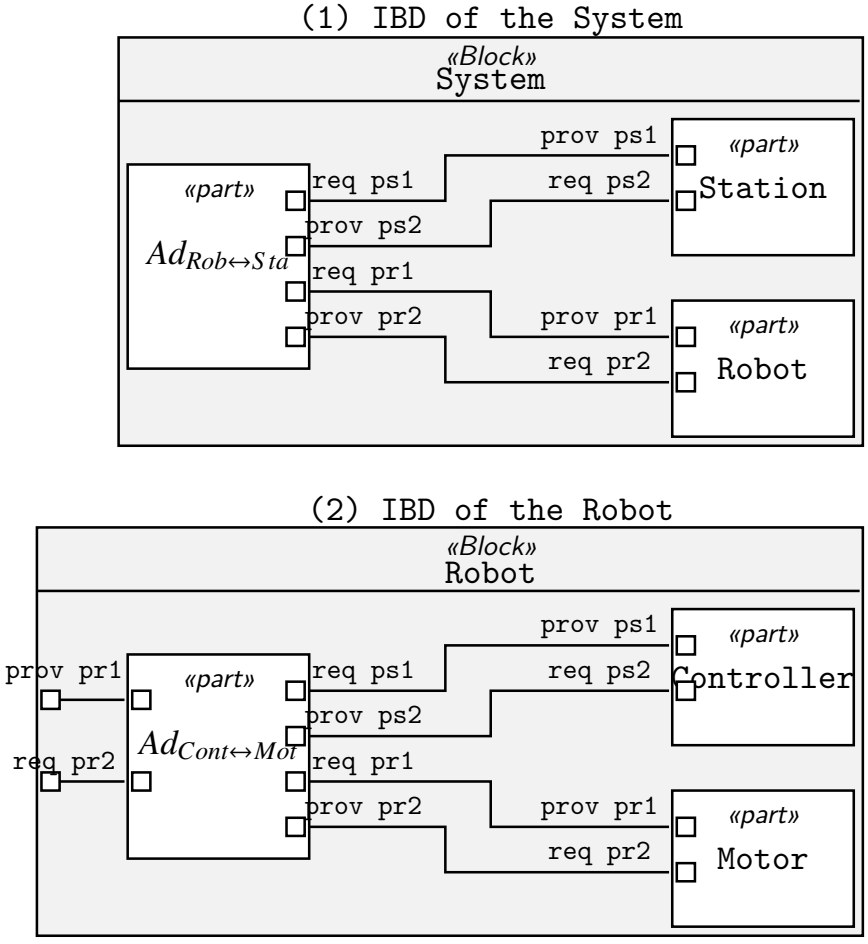


Figure 8.9: The internal block diagram.

8.3/ Conclusion

We have presented in this chapter, a bottom-up approach to build a system, based on its partial specifications. The approach is based on reusing and adapting SysML blocks using a converter-complement block. Starting from a specification of a system's part that we consider as a SysML composite block, the architect tries to meet this specification by reusing existing blocks. In our present work, we have given a set of conditions that this set of blocks must verify, and also, we have given some constraints to be respected by the contract specified by the architect. We have used the interface automata as formalism to specify formally the interaction protocols of blocks. By defining the new notion of contract based synchronous product and basing on the relation of refinement between interface automata, we deduce the interaction protocol of the converter-complement block, when the reused blocks respect the adaptation conditions. In our approach the adapter has two roles. It plays its role as a converter between the reused blocks on the one hand, and between the reused blocks and their future parent block on the other hand. It plays the second role as a complement by performing to the reused blocks what they require and its not planed to be required by their parent, and to offer what the parent must provide and its not provided by any part of it.

9

Incremental Verification of System Requirements

It is very interesting to make call for formal methods to verify the result of assembling a set of components to construct a system. In our context, the verification of the resulted system after adaptation targets the temporal properties as in [CCM14]. These properties can concern the order of components services invocation, or it can verify if when a component requests a service, the other components can always answer this request by offering the corresponding service.

In our approach, we take advantage from our adaptation method, to tackle the problem of state explosion. For a given requirement, we must know its location in the system hierarchy to specify the blocks involved in the verification of this requirement. The set of selected blocks will be the minimal set that allows ensuring that if the property, which represents the requirement, is verified on this set, it will be verified on the whole of the system.

In this chapter, we extend the adaptation (presented in chapter 8) by a verification approach which allows verifying SysML requirements on only a partial part of a system in order to decide on its verification on the whole system. In this phase, we exploit our manner of defining the adapter, to avoid the verification of the initial requirements satisfied by the adapted blocks on the totality of the system, and thus, we avoid the state space explosion. To allow the verification of the properties that specify these requirements, we have based our work on SPIN model checker, we have generated the Promela code starting from the interface automata of blocks, and we have expressed the set of requirements using LTL properties.

Contents

In the remainder of this chapter, we expose our approach for requirement verification, where we present how we represent the requirements and how we verify them. Next, we illustrate our approach through an extended version of the case study presented in chapter 8.

9.1/ Our Approach

9.1.1/ Requirements Specification

In this work, we consider that each functional requirement is related to the provided (PS) and required (RS) services of the block (B) on which is defined, and it expresses constraints and the order of executing these services.

To model the behaviour of our blocks, we have used interface automata. We have mentioned, early in this paper, that the required services (RS) of a block correspond to the output actions of its interface automaton, and the provided services (PS) correspond to the input actions. This means that, we can consider that each requirement **r** which is defined on a Block B is also specified using the input and output actions of interface automaton of B.

In our work, we will translate the interface automata of blocks to Promela processes, and we will write the requirements using LTL in order to verify these requirements using SPIN model checker as in [CCM14]. Thus, a block B satisfies a requirement r if the Promela program describing the block behaviour (interface automaton) satisfies the LTL property p specifying the requirement r.

In order to verify whether a component satisfies a LTL property, which describes the order of executing a component services, in [LTM+09], the authors have proposed to use a series of flags in Promela processes to keep track of who is sending/receiving what message to/from whom at any time of the execution. These flags are updated together at each send/receive event using a d_step statement. After defining the flags to track the execution state of the system, LTL properties can be written as boolean expressions over the flags.

Thus a property p which is defined on the block B can be expressed as a formula defined with the flags (the flags takes the value true or false) belonging to this set:

$$\{\text{send, receive}\} \cup \{\text{actionFlag(a)} | \ a \in \Sigma_B^I \cup \Sigma_B^O\} \cup \{\text{blockFlag(B), blockFlag(ENV)}\}$$

where:

- `actionFlag()` is the function that returns the flag which is associated to a given action.
- `blockFlag()` is the function that returns the flag which is associated to a given block, or to the environment.

Thus, for example, if we want to express that, when the block Bi receives a request to execute the service y or the service z, it must send a call for the service x, as a LTL property using flags, we do it as follows:

```
□ ((f_Bi && f_receive && ( f_y || f_z)) →
                                ◊ (f_Bi && f_send && f_x))
```

where:

- f_Bi is the flag associated to Bi, it takes the value true or false.
- f_x, f_y and f_z are the flags associated to services x, y and z.
- f_send and f_receive are the flags that specify if the executed action is an emission or a reception.

All the flags must be updated after each action of a block. Where, the flags that represent the executed action, its type (send or receive) and the block which executes this action, must be updated to true. However, the other flags must become equal to false.

9.1.2/ Problem definition

Our approach aims to alleviate the verification phase of functional requirements of the adapted system by exploiting our manner of adapting system blocks. Through our approach, we will explain how can we reduce the problem of the state explosion during the verification, by benefiting from our adaptation approach and the composition relations between blocks and between requirements. Our adaptation mechanism, presented before, generates for each set of reused blocks an adapter block. This adapter plays the role of an orchestral conductor for the adapted blocks.

We can resume our problem of verification as follows:

If we have a LTL property p which is verified on a block B_1, how can we check whether p is verified or not on the result of assembling B_1 and adapting it with other blocks $\{B_i\}_{i=2..n}$.

The first idea that comes to the mind is to verify p on the parallel execution of the blocks $\{B_i\}_{i=1..n}$ and their adapters $\{Ad_j\}$ (we can't verify p only on the interaction protocol of B_i because, after adaptation, there is a possibility that some interaction scenarios of B_i will be eliminated and others will be created due to the parallel composition). This first idea is constrained with the problem of the state explosion because the system behaviour obtained after the composition of several blocks is generally complex and voluminous. For this reason, in our approach, to verify p, we focus on only the generated adapters $\{Ad_j\}$ and the mirror of the property p . We generate the mirror of the property p because p is initially defined on the input and output actions of the reused block B_i, while we want to verify it on the adapter, and we also know that the input (resp. output) actions of B_i are the output (resp. input) actions of the adapter.

In our approach, we distinguish between two cases:

9.1.3/ The First Case : The Low Level Verification

We mean by low level verification (see Figure 9.1), the stages where we verify a property p on a block B containing a set a blocks $\{B_i\}_{i=1..n}$ and an adapter Ad, where the property p is initially defined on a child block $B_i \in \{B_i\}_{i=1..n}$.

Taking into account that the adapter Ad mediates all interactions between the blocks $\{B_i\}_{i=1..n}$, we can only focused on its interaction protocol to verify if a property p which is verified on B_i before adaptation steals verified after adapting B_i in the new system, instead of verifying p on the parallel execution of the blocks $\{B_i\}_{i=1..n}$ and the adapter Ad.

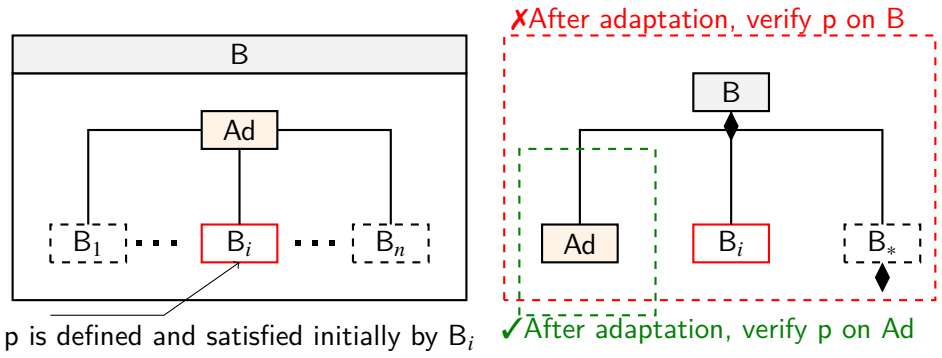


Figure 9.1: The first case: the low level verification.

Theorem 2:

$\forall$ B= $B_1 \|^b \ldots \|^b B_n \|^b$ Ad, $\forall$ $B_i \in \{B_j\}_{1..n}$, $\forall$ p∈Properties(B_i),
$\forall$ p'=PropertyMirror(p)
($B_i \models p \wedge$ Ad $\models$ p' $\Rightarrow$ B $\models$ p)

Theorem 3:

$\forall$ B= $B_1 \|^b \ldots \|^b B_n \|^b$ Ad, $\forall$ $B_i \in \{B_j\}_{1..n}$, $\forall$ p∈Properties(B_i),
$\forall$ p'=PropertyMirror(p)
($B_i \models p \wedge$ Ad $\not\models$ p' $\Rightarrow$ B $\not\models$ p)

We define by:

- $\|^b$: the blocks assembling operation.
- $B_i \models p$: means that all the execution scenarios of B_i satisfy the temporal order specified at the level of the property p.
- Properties(B_i): is the set of all properties defined and satisfied initially by B_i before the adaptation.
- PropertyMirror(p): is the function that transforms each input action in p into an output action and each output action into an input action.

Proof:

- We have p is a property satisfied by the block B_i. $B_i \models p$
- we have $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ represent the sets of shared actions between the interface automaton of the adapter A_{Ad} and, respectively, the interface automata of the blocks $A_{B1}, A_{B2}, \ldots, A_{Bn}$: $\Sigma_i = \{a \in \Sigma_{Bi} \mid \exists a \in \Sigma_{Ad}\}$.

- we have $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$.
- The Traces set of an automaton A represents the set of all execution scenarios $\{\sigma\}$ of this automaton, where each $\sigma \in Traces(A) = a_1 a_2 ... a_i ... a_m (\forall i = 1..m, a_i \in \Sigma_A)$.
- TraceMirror(σ) is the function that takes a trace σ and transforms each input action into an output action, and each output action into an input action (e.g. TraceMirror(a!b?c!)=a?b!c?).
- reduce(σ, Σ) is the function that takes a trace σ and eliminates from it all the actions which do not belong to the set Σ.
- Our definition of the adapter as a converter-complement in a synchronous system implies that:
  - all emission of a message a! by the adapter Ad (where $a \in \Sigma_i$) must be followed directly (synchronous system) by its reception a? by the block B_i, and each reception of a message a? by the adapter (where the adapter waits this message from B_i) must be preceded directly by an emission of this message a! by a block B_i... **(1)**

Proof 1:

- We consider that the property p'= PropertyMirror(p) is satisfied by the adapter Ad. $Ad \models p'$...**(2)**
- From **(1)**, we have:

$$- \ \forall \sigma \in Traces(A_{Ad} \otimes A_{B1} \otimes ... \otimes A_{Bn}) \Rightarrow \exists \sigma' \in Traces(A_{Ad}), \quad \sigma' = TraceMirror(reduce(\sigma, \Sigma)) \ ...\textbf{(3)}$$

- We interest only with the actions of the block B_i because p is defined on B_i, thus we can restrict **(3)** as follows:

$$- \ \forall \sigma \in Traces(A_{Ad} \otimes A_{B1} \otimes ... \otimes A_{Bn}) \Rightarrow \exists \sigma' \in Traces(A_{Ad}), \quad \sigma' = TraceMirror(reduce(\sigma, \Sigma_i)) \ ...\textbf{(4)}$$

- Our properties specify the order of executing the actions...**(5)**
- From **(4)** and **(5)**, we can deduce that: if an action a $\in \Sigma_i$ is followed (resp. is not followed) by an action b$\in \Sigma_i$ in all the executions σ' of Ad, then it will be the case for all executions σ of B (because according to **(4)**, Traces(B) are included in Traces(Ad) by considering only actions that belong to Σ_i : $Traces(A_B) \underset{\Sigma_i}{\subset} Traces(A_{Ad})$)...**(6)**

  Thus, from **(2)** and **(6)**, we can deduce that $B \models p$.

Proof 2:

We consider that the property p'= PropertyMirror(p) is not satisfied by the adapter Ad. $Ad \not\models p'$...**(2)**

- From **(1)**, we deduce that:

$$- \forall \sigma \in Traces(A_{Ad}) \Rightarrow \exists \sigma' \in Traces(A_{Ad} \otimes A_{B1} \otimes ... \otimes A_{Bn}),$$
$$\sigma' = TraceMirror(reduce(\sigma, \Sigma)) \text{ ...}\mathbf{(3)}$$

- We interest only with the actions of the block B_i because p is defined on B_i, we can thus restrict **(3)** as follows:

$$- \forall \sigma \in Traces(A_{Ad}) \Rightarrow \exists \sigma' \in Traces(A_{Ad} \otimes A_{B1} \otimes ... \otimes A_{Bn}),$$
$$\sigma' = TraceMirror(reduce(\sigma, \Sigma_i)) \text{ ...}\mathbf{(4)}$$

- Our properties specify the order of executing the actions...**(5)**

- From **(4)** and **(5)**, we can deduce that: The scenarios of the adapter are included in the scenarios set of the father block B by considering only actions which belong to Σ_i. Thus, if an actions order is not verified by at least one of the scenarios of adapter then this implies that this order will not be verified by at least one scenario of the father block **B**...(6)

  Thus, from**(2)** and **(6)**, we can deduce that $B \not\models p$.

9.1.4/ The Second Case : The High Level Verification

We mean by high level verification, the stages where a property p is initially satisfied by a block B_{ij} (where i is the level of the block in our system hierarchy and j is the identifier of this block in its level), and we try to verify it on B_m (the ancestor m-i of B_{ij}) (see Figure 9.2).

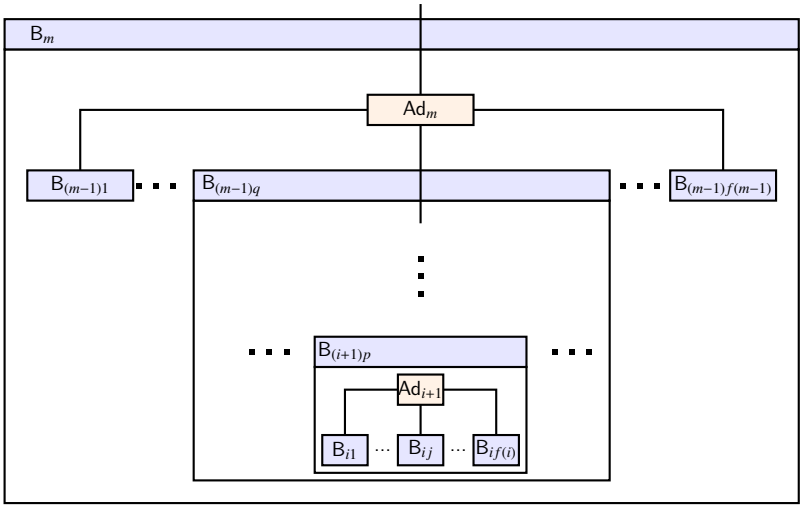


Figure 9.2: Incremental adaptation.

We can define the problem as follows: We have a temporal property p satisfied by the block B_{ij}, where this property concerns the actions of the block B_{ij}. We have applied a succession of adaptations on this block. For example, at the first stage of adaptation of B_{ij} (we are at the level i of our system hierarchy) the block B_{ij} interacts with the adapter Ad_{i+1}, where B_{ij} and Ad_{i+1} has the same father block $B_{(i+1)p}$. when we are at the stage m of adaptation of B_{ij}, we will have the adapter Ad_m which adapts a set of blocks and the block $B_{(m-1)p}$ (the

$ancestor_{m-i}$ of B_{ij}). According to our method of adaptation, the block Ad_m interacts with the sub-blocks of $B_{(m-1)}p$ through the adapter Ad_{m-1}.The question here is how can we verify p at the level m without taking into account the parallel execution of all blocks inside the ancestor block B_m of B_{ij}?

Theorem 4:

$\forall$ i=1..m, $\forall$ j=1..f(i), B_m=$Ad_m$$\|^b$ $B_{(m-1)1}$$\|^b$ ... $\|^b$ $B_{(m-1)f(m-1)}$,
$\forall$ p $\in$ Properties(B_{ij}), $\forall$ p'=PropertyMirror(p)
(B_{ij} $\models$ p $\wedge$ $\|^b_{k=(i+1)..m}Ad_k$ $\models$ p' $\Rightarrow$ B_m $\models$ p)

Theorem 5:

$\forall$ i=1..m, $\forall$ j=1..f(i), B_m=$Ad_m$$\|^b$ $B_{(m-1)1}$$\|^b$ ... $\|^b$ $B_{(m-1)f(m-1)}$,
$\forall$ p $\in$ Properties(B_{ij}), $\forall$ p'=PropertyMirror(p)
(B_{ij} $\models$ p $\wedge$ $\|^b_{k=(i+1)..m}Ad_k$ $\not\models$ p' $\Rightarrow$ B_m $\not\models$ p)

Where:

- $\|^b$ is the blocks assembling operation.
- i is the identifier of the hierarchy level,
- j is the identifier of the block inside the level,
- f(i) is the function that returns the number of the children blocks at the level i,
- B_m is the ancestor $m-i$ of B_{ij}.

Proof:

To simplify, we consider two levels, the level 1 and the level 2. This means that we have adapted the block B_{1j} and after we have adapted its father. So the problem will be as in Figure 9.3.

- we have Σ represents the sets of shared actions between the interface automaton of the block B_{1j} and the interface automaton of the adapter Ad2. $\Sigma = \{a \in \Sigma_{B_{1j}} \mid \exists a \in \Sigma_{Ad2}\}$.
- We have $A_{B3} = A_{Ad3} \otimes A_{B_{21}} \otimes ... \otimes A_{B_{2p}} \otimes ... \otimes A_{B_{2m}}$
- So (we mean by $\sigma \underset{\Sigma}{=} \sigma'$ that $\sigma = \sigma'$ when we consider only the actions belong to Σ):

$$Traces(A_{B3}) \underset{\Sigma}{=} Traces(A_{Ad3} \otimes A_{B_{21}} \otimes ... \otimes A_{B_{2p}} \otimes ... \otimes A_{B_{2m}})...\textbf{(1)}$$

- In Our adaptation mechanism:
  - the adapted blocks are in interaction only with their adapter. Thus, the block B_{2p} interacts only with the adapter Ad3...**(2)**
  - the other adapted blocks by Ad3 can't block the interaction of the adapter Ad3 with the block B_{2p}...**(3)**

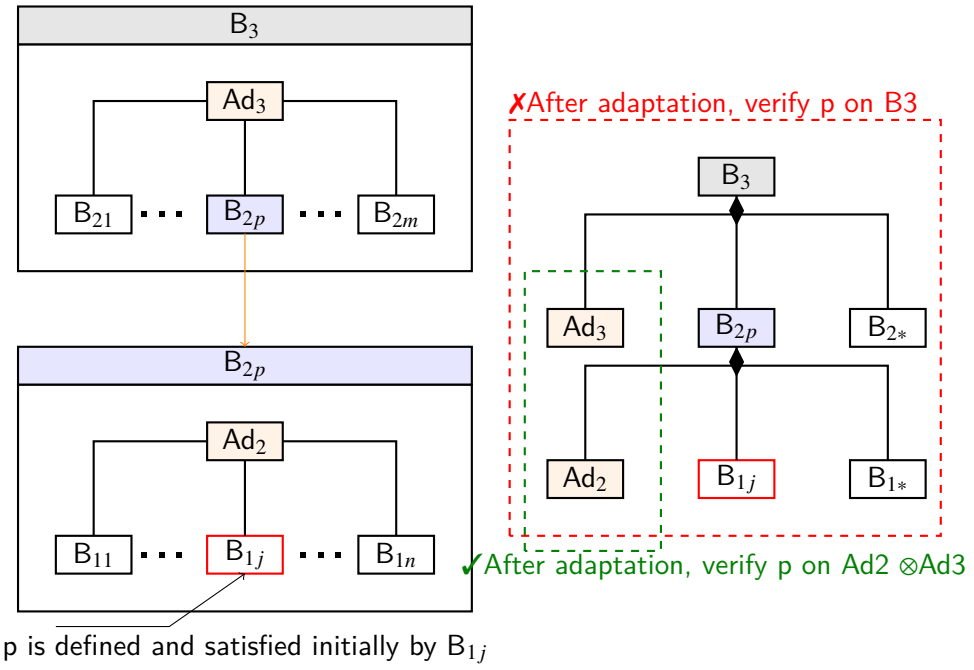


Figure 9.3: The second case: The high level verification.

- from **(1)**, **(2)** and **(3)**, we can deduce that:

$$Traces(A_{B3}) \underset{\Sigma}{=} Traces(A_{Ad3} \otimes A_{B_{2p}}) \ldots \textbf{(4)}$$

- We have also:
  - the block B_{ij} interacts only with the adapter Ad2...(5)
  - the other adapted blocks by Ad2 can't block the interactions of the Ad2 with the block B_{ij}...(6)
- from **(3)**, **(5)** and **(6)**, we can deduce that:

$$Traces(A_{Ad3} \otimes A_{B_{2p}}) \underset{\Sigma}{=} Traces(A_{Ad3} \otimes A_{Ad2} \otimes A_{B_{ij}}) \ldots \textbf{(7)}$$

- from **(4)** and **(7)**, we deduce that:

$$Traces(A_{B3}) \underset{\Sigma}{=} Traces(A_{Ad3} \otimes A_{Ad2} \otimes A_{B_{ij}}) \ldots \textbf{(8)}$$

- Our definition of our adaptor implies that: all emission of a message a! by the adapter Ad2 (where $a \in \Sigma_1$) must be followed directly (synchronous system) by its reception a? by the block B_{1j}, and each reception of a message a? by the adapter (where the adapter waits this message from B_{1j}) must be preceded directly by an emission of this message a! by a block B_{1j}... **(9)**
- from **(8)** and **(9)**, we deduce that:

$$\forall \sigma \in Traces(A_{B3}) \Rightarrow \exists \sigma' \in Traces(A_{Ad2} \otimes A_{Ad3})$$
$$\sigma' = TraceMirror(reduce(\sigma, \Sigma)) \ldots \textbf{(10)}$$

- If the mirror of a property p which concerns the actions order of the block B_{1j} is satisfied by each execution of $A_{Ad2} \otimes A_{Ad3}$, from **(10)**, we can deduce that this order will be preserved in all traces of A_{B3}, and thus, p will be satisfied by the block B3.

  The same for a property p, when its mirror is not satisfied by at least one scenario of $Ad2 \otimes Ad3$, then from **(10)**, we can deduce that this property will not be satisfied by B3.

9.1.5/ The Verification Algorithm

Our verification algorithm takes respectively as inputs the block B_m and the requirement diagram ReqD. B_m represents the block on which we want to verify the properties defined and satisfied initially by its children. However, ReqD represents the requirement diagram. ReqD helps us to deduce the properties satisfied by the children blocks of B_m. We must mention that at the beginning of the adaptation, the requirement diagram takes the form of separated requirements sets, where each set is satisfied by a reused block. Thus, the role of this algorithm is to synthesize the verification steps described above. Firstly, it extracts all the requirements satisfied by the sub-blocks of the input block B_m. After, for each sub-block, it constructs the set of its ancestors until arriving to B_m. Next, Basing on this last set, it can decide if it will apply the low level or the high level verification. The output of the verification phase is a set of requirements which are satisfied by the new system B_m. Next, we use this set, the hierarchy of blocks and the composition relation between requirements to update the requirement diagram.

Algorithm 6 verification of temporal properties

INPUT: B_m, *ReqD*

```
1: //Pv REPRESENTS THE SET OF PROPERTIES SATISFIED BY B_m
2: Pv ← ∅
3: //Sub_Blocks(Bm) REPRESENTS THE BLOCKS SET THAT HAVE B_m
4: //IN THEIR ANCESTORS SET, INCLUDING B_m
5: for all B_i ∈ Sub_Blocks(Bm) do
6:    // SELECT, FROM THE REQUIREMENT DIAGRAM, ALL THE ATOMIC REQUIREMENTS
7:    //satisfied by B_i
8:    REQ← {req| ∃ r ∈ SRel(ReqD) ∧ r.source= B_i ∧ r.target=p}
9:    -P represents the properties set specifying the requirements set REQ
10:   // Ancestors(B_i) REPRESENTS THE ANCESTORS LIST OF THE BLOCK B_i
11:   Ancestors(B_i)=[B_{i+1}, ..., B_m]
12:   //Ad_of_Ancesters(B_i) REPRESENTS THE LIST OF ADAPTERS OF THE
13:   //ANCESTORS BLOCKS OF B_i
14:   Ad_of_Ancesters(B_i)=[Ad_{i+1}, ..., Ad_m]
15:   for all p ∈ P do
16:      //CREATE THE MIRROR PROPERTY OF p
17:      -Create p' a copy of p
18:      -Replace all the flags 'send' in p' with the flags 'receive'.
19:      -Replace all the flags 'receive' in p' with the flags 'send'.
20:      if size(Ancestors(B_i))=1 then //THE LOW LEVEL VERIFICATION
21:         -Verify p' on the Adapter Ad_{i+1}.
22:      else//THE HIGH LEVEL VERIFICATION
23:         -π =⊗_{j=(i+1)..m} Ad_j
24:         -Verify p' on the Adapter π.
25:      end if
26:      if p' is verified then
27:         Pv ← Pv ∪ {p}
28:      end if
29:   end for
30: end for
31: //MODIFY THE REQUIREMENT DIAGRAM
32: -REQ_S represents the requirements set specified by the properties in Pv
33: //REPLACE SOME ATOMIC REQUIREMENTS IN Rs BY THEIR ANCESTORS
34: for all r ∈ REQ_S do
35:    -R_Sibl represents the set of the sibling requirements of r,
36:    including r.
37:    if R_Sibl ⊂ REQ_S then
38:       -delete all requirements in R_Sibl from REQ_S and replace them by
39:       parent(r)
40:    end if
41: end for
42: -Create a new requirement req.
43: for all req_i ∈ REQ_S do
44:    -Create a composition relation cr between req and the
45:     requirement req_i, where : source(cr)=req ∧ targets(cr)=req_i
46: end for
47: -Create a satisfaction relation «satisfy» sr between the block B_m and the
    requirement req, where: source(sr)=B_m ∧ target(sr)= req
```

9.2/ Case Study

We extend the case study of the previous chapter as follows:

During the selection of the blocks to reuse for building our system, we have verified if these blocks satisfy our initial requirements. For example in Figure 9.4, we give the requirements on those we have based to select the controller, the motor and the station blocks. During the construction of the robot, we have taken into consideration that the reused controller must satisfy the requirement R_{c1}. However, during the adaptation of the robot with the station to construct the global system, we have verified that the station satisfies the requirement R_{s1}. Also, from the specification of the robot, we have deduced that the robot satisfies the requirement R_{r1}.

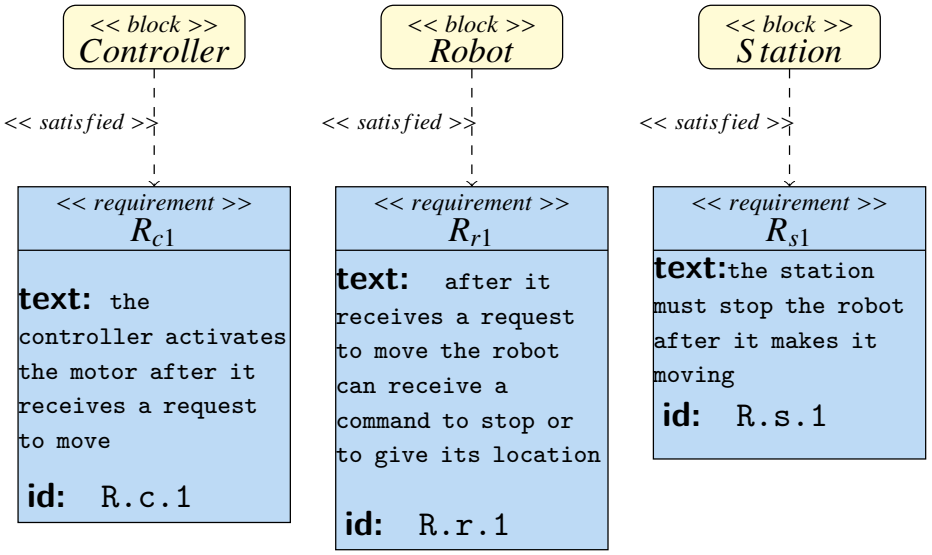


Figure 9.4: The basic requirements.

After the adaptation (adapting the controller and the motor), we must verify if the requirements still verified in the parallel execution of our system blocks.

The requirements specified on the station and on the robot can be verified directly on the their adapter $Ad_{Rob \leftrightarrow Sta}$ by following these steps (low level verification):

1. we must write the interface automaton of $Ad_{Rob \leftrightarrow Sta}$ as a Promela process. We must also specify its environment ,which is the mirror of $Ad_{Rob \leftrightarrow Sta}$ interface automaton, as a Promela process.
2. we add the processes flags f_proc, which are the flag associated to the adapter f_ad, and another flag which is associated to the environment f_env.
3. we add send f_send and receive flags f_rec.
4. we add the actions flags f_act, where each flag f_act is associated to an action of the adapter $Ad_{Rob \leftrightarrow Sta}$. This action must belong to the shared actions between this adapter and the concerned blocks (robot and station).
5. we write the requirements as LTL properties, which are specified on the flags already mentioned, as follows:

```
/*Variable global*/
bit near=0;
/*Messages declaration*/
mtype={S_move, R_move, S_location, R_location, R_getLocation, S_stop, R_stop };
/*Channels declaration*/
chan ch_Ad_Env[10]=[0] of {mtype};
/***FLAGS***/
/*Last performed action by the adapter block*/
bit send=0;bit receive=0;
/*process that performed last action*/
bit f_ad=0; bit f_Env=0;
/*action performed at a given execution step*/
bit f_S_move=0; bit f_R_move=0; bit f_S_location=0; bit f_R_location=0;
bit f_R_getLocation=0; bit f_S_stop=0; bit f_R_stop=0;
proctype Ad_Rob_Sta() {
do
::atomic{ ch_Ad_Env[0]?S_move;
       d_step{send=0;receive=1; Ad=1; Env=0; f_S_move=1; f_R_move=0; f_S_location=0;
              f_R_location=0; f_R_getLocation=0; f_S_stop=0; f_R_stop=0; }}
::atomic{ ch_Ad_Env[0]!R_move;
       d_step{send=1;receive=0; Ad=1; Env=0; f_S_move=0; f_R_move=1; f_S_location=0;
              f_R_location=0; f_R_getLocation=0; f_S_stop=0; f_R_stop=0; }}
  do
  ::near=0 ->
   ::atomic{ ch_Ad_Env[0]!R_getLocation;
    d_step{send=1;receive=0; Ad=1; Env=0; f_S_move=0; f_R_move=0; f_S_location=0;
        f_R_location=0; f_R_getLocation=1; f_S_stop=0; f_R_stop=0; }}
   ::atomic{ ch_Ad_Env[0]?R_location;
    d_step{send=0;receive=1; Ad=1; Env=0; f_S_move=0; f_R_move=0; f_S_location=0;
        f_R_location=1; f_R_getLocation=0; f_S_stop=0; f_R_stop=0; }}
   ::atomic{ ch_Ad_Env[0]!S_location;
    d_step{send=1;receive=0; Ad=1; Env=0; f_S_move=0; f_R_move=0; f_S_location=1;
        f_R_location=0; f_R_getLocation=0; f_S_stop=0; f_R_stop=0; }}
  ::else ->break
  od;
::atomic{ ch_Ad_Env[0]?S_stop;
       d_step{send=0;receive=1; Ad=1; Env=0; f_S_move=0; f_R_move=0; f_S_location=0;
              f_R_location=0; f_R_getLocation=0; f_S_stop=1; f_R_stop=0; }}
::atomic{ ch_Ad_Env[0]!R_stop;
       d_step{send=1;receive=0; Ad=1; Env=0; f_S_move=0; f_R_move=0; f_S_location=0;
              f_R_location=0; f_R_getLocation=0; f_S_stop=0; f_R_stop=1; }}
od;}
proctype Env() {
do
:
:

od;}
/*System Instantiation*/
init{atomic{run Ad_Rob_Sta(); run Env();}}
 /*LTL properties*/
ltl Mirror_R_s1 {[]((f_ad && f_receive && f_S_move) ->
                    <> (f_ad && f_receive && f_S_stop))}
ltl Mirror_R_r1 {[]((f_ad && f_send && f_R_move) ->
        <> (f_ad && f_send && (f_R_stop || f_R_getLocation)))}
```

Figure 9.5: SPIN system for the adapter $\mathtt{Ad}_{Rob \leftrightarrow Sta}$ and its environment.

- $\mathtt{R}_{s1}$: $\Box$ ((f_station && f_send && f_S.move) $\rightarrow$
  $\Diamond$ (f_station && f_send && f_S.stop))
- $\mathtt{R}_{r1}$: $\Box$ ((f_robot && f_receive && f_R.move) $\rightarrow$

◊ (f_robot && f_receive && (f_R.stop || f_R.getLocation)))

6. we must specify the mirror of these properties:

   - Mirror_R$_{s1}$: □ ((f_ad && f_receive && f_S.move)→
     ◊ (f_ad && f_receive && f_S.stop))
   - Mirror_R$_{r1}$: □ ((f_ad && f_send && f_R.move)→
     ◊ (f_ad && f_send && (f_R.stop || f_R.getLocation)))

7. we verify the properties Mirror_R$_{s1}$ and Mirror_R$_{r1}$ on the SPIN system composed of the two Promela process of the adapter and the environment.

To verify the requirement, which is specified on the controller, on the global system, we need to follow these steps (high level verification):

1. We must compute the synchronous product (π)of the two adapters Ad$_{Rob\leftrightarrow Sta}$ ⊗ Ad$_{Contr\leftrightarrow Mot}$.
2. we write the resulted interface automaton π as a Promela process PI. We must also specify its environment ,which is the mirror of π interface automaton, as a Promela process.
3. we add the process flag f_proc, which are the flag associated to the adapter product f_PI, and another flag which is associated to the environment f_env.
4. we add send f_send and receive flags f_rec.
5. we add the actions flags f_act, where each flag f_act is associated to an action of the adapter π. This action must belong to the shared actions between π and the controller.
6. we write the requirement R$_{c1}$ as an LTL property, which is specified on the flags already mentioned, as follows:

   - R$_{c1}$: □ ((f_PI && f_receive && f_C.move) →
     ◊ (f_PI && f_send && f_C.on))

7. we must specify the mirror of this property:

   - Mirror_R$_{c1}$: □ ((f_PI && f_send && f_C.move) →
     ◊ (f_PI && f_receive && f_C.on))

8. we verify the properties Mirror_R$_{c1}$ on the SPIN system composed of the two Promela process of PI and the environment.

In Figure 9.6, we show the synchronous product of the two adapters $Ad_{Rob\leftrightarrow Sta}$ ⊗ $Ad_{Contr\leftrightarrow Mot}$. After translating it into Promela code and verifying the property $Mirror\_R_{c1}$ using SPIN model checker, we found that the SPIN system satisfies this property. Thus, according to theorem3, the global system satisfies the property R_{c1}.

Thus, the new requirement diagram will be as it is shown in Figure 9.7.

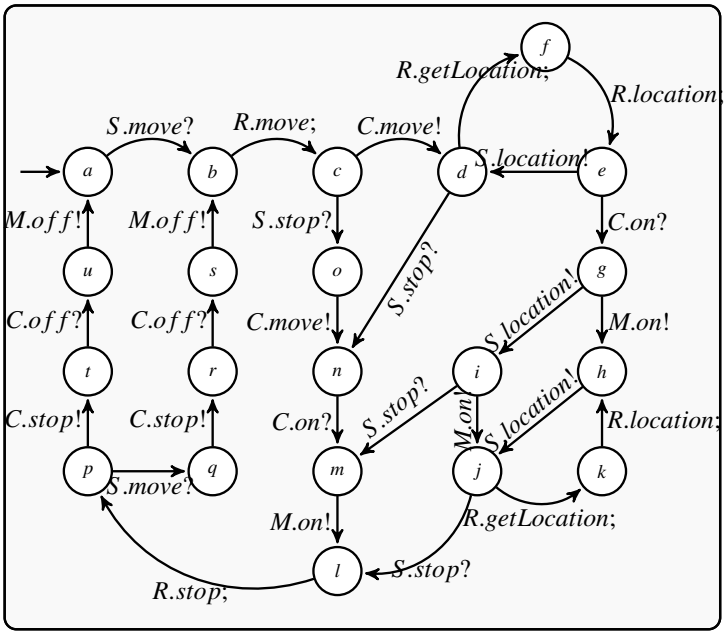


Figure 9.6: $\pi = Ad_{Rob \leftrightarrow Sta} \otimes Ad_{Contr \leftrightarrow Mot}$.

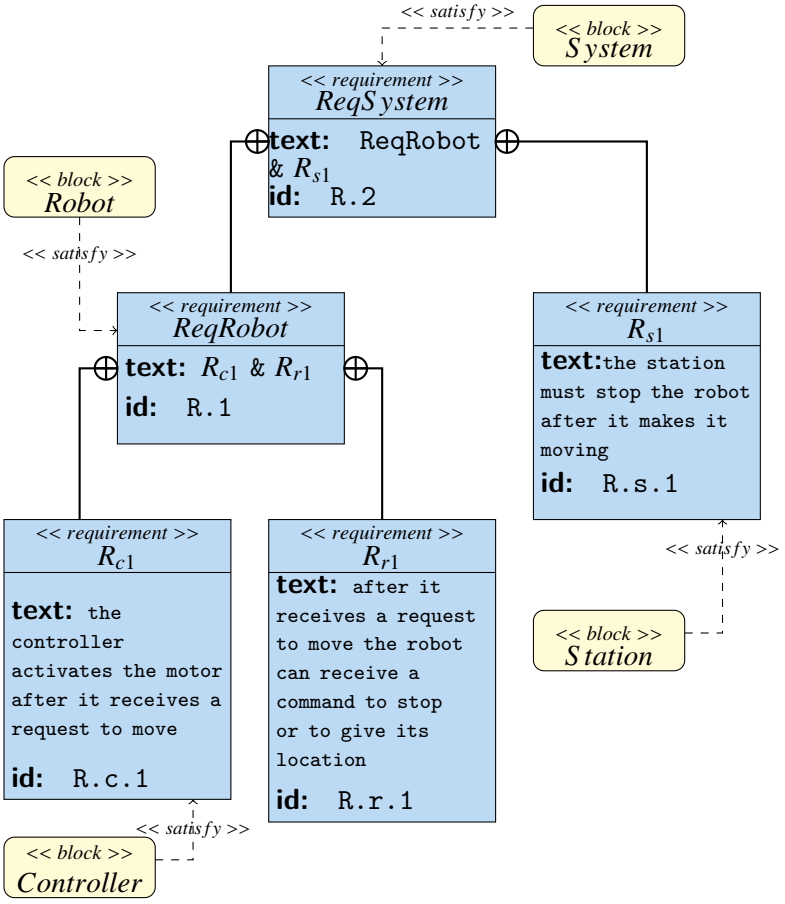


Figure 9.7: The requirement diagram of the system.

9.3/ Conclusion

Our approach of verification presented in this chapter, is proposed to complement the adaptation approach presented in the previous chapter. Its objective is to alleviate the verification of functional requirements of the adapted system by exploiting our manner of adapting system blocks. Thus, we have proposed a verification approach that tackles the state explosion problem by reducing the state space of the verification of SysML requirements, thanks to the adapters blocks. To allow a verification of these requirement on the interaction protocol of blocks modelled using interface automata, we have proposed to write the interface automata as SPIN processes and to specify the requirements using LTL properties. Our verification is performed using SPIN model-checker.

10

Adaptation with Reordering of SysML Block Services

When assembling two components developed separately, there is a high probability of encountering the problem of mismatches between them. These mismatches can concern the name of services (as the problem processed in chapter 8), as well as the order in which the component asks (resp. offers) for environment services (resp. its services). In this chapter, we propose our approach for adapting a set of reused blocks to meet an initial specification given by the designer to solve the problems mentioned before. During our process, this specification will become the parent block that will include the reused blocks. The interactions of each block with its environment are modelled using SysML Sequence Diagrams (SDs), an adaptation contract is defined and used to guide the adaptation by specifying atomic and no-atomic correspondences between block services.

In the previous chapter 8, in the same context of this work, we have proposed a bottom-up approach that bases on interface automata [dAH01] to adapt SysML blocks, but with different inputs and objectives. The major difference resides on that the adapter as we will define it in this chapter can solve more problems such as the reordering of services to eliminate livelock between blocks, it can also solve more types of mismatches ('one-to-many' rather than only 'one-to-one'). In fact, in our adaptation approach, the system can consider the messages of blocks as resources, where it is possible to capture a message call of a block and deliver it to the concerned block when this last can receive this call. Thus, our adapter must authorize the reordering of the service calls and deliver them to the concerned blocks when they will be ready. Representing this information and implementing these operations using interface automata will be very difficult. We have taken that into consideration during the adaptation phase, that is what justifies our choice of

Contents

Petri Nets (PNs). The Petri nets are characterized by their richness compared with interface automata. They easily allow to represent the reordering of service calls.

However, using simple PNs (as in [CPS06a, DBM14]) implies the introduction of many empty transitions (tau transitions that serve just for connecting places) that increase the state space of the system. That's why we have opted for Coloured Petri Nets(CPNs) where the colours represent the services of blocks. In our method, we use tau transitions only in some cases when we have no-atomic correspondences between blocks services. Thus, in this chapter, we will analyse the different correspondences between the blocks by taking into account the existed hierarchy relation between the reused blocks and the specification of the parent block made initially by the designer, and we will define a CPNs rule for each kind of correspondences.

Unlike the works already done which rely only on formal models, our method bases on SysML models as inputs, which allows for designers to deal with graphical convivial models. After, to generate the formal models that we will use for adaptation, we have proposed a meta-level approach which bases on meta-modelling and models transformation.

In the remainder of this chapter, we will present our approach in Section 1. After that, in Section 2, we will show how we apply our approach through a case study.

10.1/ Our Adaptation Approach

Our approach is an incremental bottom-up approach. It consists in constructing the system by developing a part of the system at each increment. In our approach this part is considered as SysML composite block **B**. After that, the developer selects some blocks to build **B**. These blocks will become the sub-blocks of **B**, but they need to be adapted to fulfil the tasks expected from the environment of **B**.

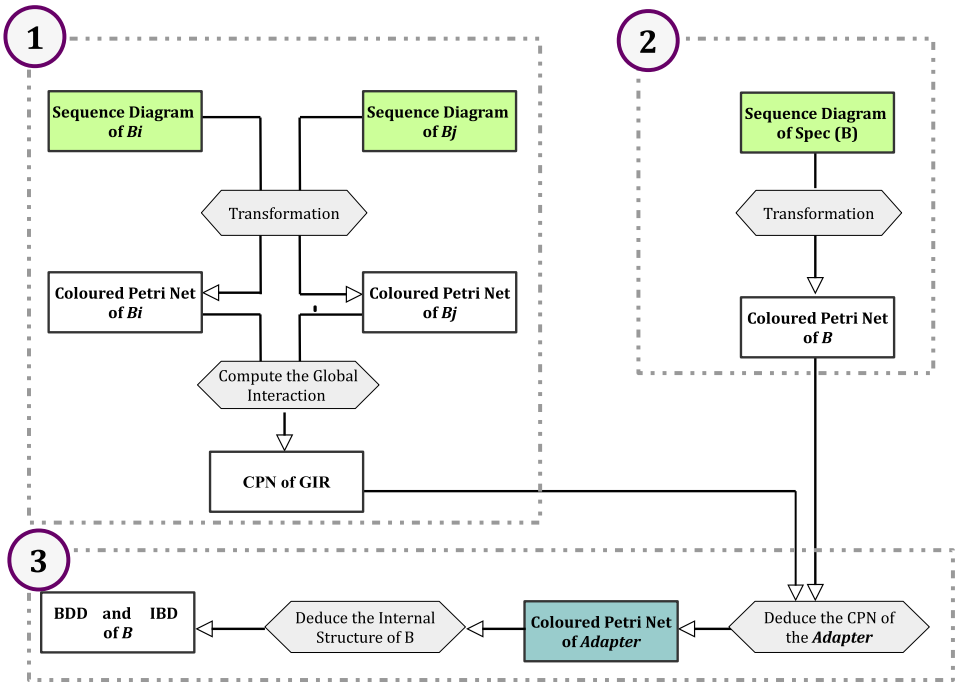


Figure 10.1: Our approach of adaptation with reordering.

Our approach of adaptation can be divided into three phases (see Figure 10.1). The first phase consists in computing the global interaction protocol of the reused blocks (future sub-blocks). During this computation, we authorize the reordering of receiving the service calls. To do that, we need to transform the sequence diagrams that describe the interaction protocols of the reused blocks into their equivalents of Coloured Petri Nets (CPNs), and synthesize them basing on the adaptation contract, where the contract helps to define the correspondences between the block services.

The second phase, which can be done in parallel with the first one, consists in transforming the sequence diagram, that specifies the interactions of the future parent block **B** with its environment, into Coloured Petri Net (CPN_B). After computing the global interaction protocol of the reused blocks ($\{B_i\}$) by synthesising their Petri nets using the adaptation contract, we synthesize this resulted CPN with the CPN of the specification (CPN_B) basing always on the adaptation contract. We apply some modifications on the resulted CPN and we obtain the CPN of the adapter ($CPN_{adapter}$). Finally, we model the internal structure of the new part B using the SysML BDD and IBD.

10.1.1/ Computing the Global Interaction Protocol of the Reused Blocks GIR

As we have mentioned before, the objective of this phase is for deducing the different scenarios of the parallel execution of the reused blocks. To do that, we need to transform their sequence diagrams (SDs) into Coloured Petri Nets (CPNs). This transition from SDs to CPNs is necessary, because sequence diagrams are informal models, and they do not offer the necessary tools to execute the wanted operations. However, in our adaptation method, we need to synthesis the interactions of the blocks and to integrate the informations presented in the adaptation contract using one CPN. We want also that the system can consider the messages of blocks as resources, where it is possible to capture a message call of a block and deliver it to the concerned block when this last can receive this message. All these informations can be represented easily using coloured Petri nets.

Transformation SD → CPN:

In the first step (see Figure 10.2),

We associate to each event on the life line of each reused block a place. Each emission and reception of a message is transformed into a transition labelled with the message name and a mark that specifies the type of the event (! for emission, ? for reception, and ; for emitting and receiving the same message by the same block). At this stage, we need just one colour 'idle'. The existence of a token idle in a place of a block means that this block is ready to execute the next actions.

The algorithm 7 ensures the transformation of the basic interaction of sequence diagrams. Its takes as inputs the block B and its sequence diagram SD, and it generates as output its CPN. We represent by EventSet(B) the list of the all events of emission and reception associated to the life line of B.

The complexity of Algorithm 7 is computed in function of the number of messages in the

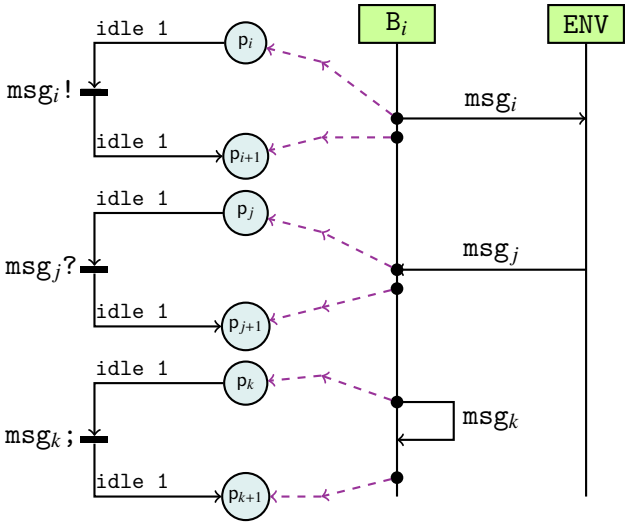


Figure 10.2: Transformation SD→CPN.

Algorithm 7 SD2CPN

INPUT: B, SD_B

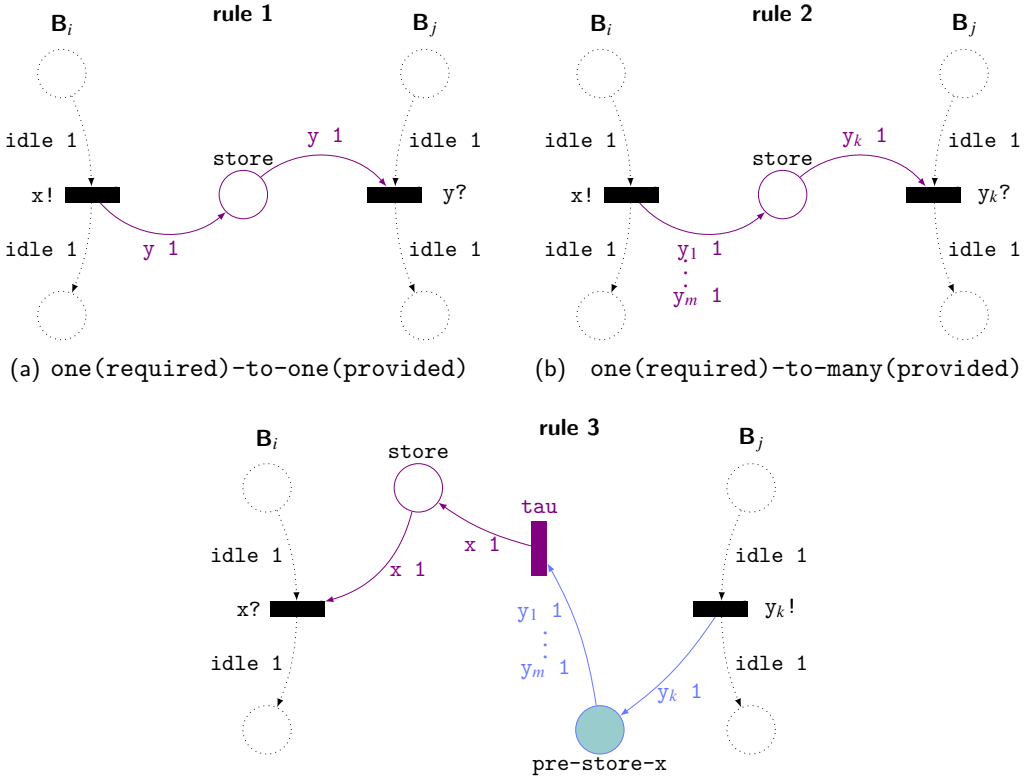
OUTPUT: CPN_B

```
1:  procedure SD2CPN(B, SD_B)
2:      -create an empty CPN_B
3:      for all evt_i ∈ EventSet(B) do
4:          -Create a place p_evt_i in CPN_B
5:          if evt_i is the first event in EventSet(B) then
6:              -add a token idle to the place p_evt_i
7:          end if
8:      end for
9:      for all message evt_i --mesi--> evt_i' ∈ SD_B do
10:         if (evt_i ∈ EventSet(B) ∧ (evt_i' ∈ EventSet(B)) then
11:             -create a transition t with the label 'mes_i;'
12:             -create an arc from the place p_evt_i to t
13:             -create an arc from t to the place p_evt_i+2
14:         else
15:             if (evt_i ∈ EventSet(B) then
16:                 -create a transition t with the label 'mes_i!'
17:                 -create an arc from the place p_evt_i to t
18:                 -create an arc from t to the place p_evt_i+1
19:             else
20:                 -create a transition t with the label 'mes_i?'
21:                 -create an arc from the place p_evt'_i to t
22:                 -create an arc from t to the place p_evt'_i+1
23:             end if
24:         end if
25:     end for
26:     return CPN_B
27: end procedure
```

sequence diagram of the block B passed in parameters (m_B). Thus, $C(\text{Algorithm 7}) = O(m_B)$.

Synthesizing the CPNs of the reused blocks:

To compute the global interaction protocol of the reused blocks, we synthesize their CPNs basing on the adaptation contract. In this case, we consider the services (messages) as resources, where each service (message) will be represented using a token colour. To do this synthesis, we create a place called store that plays the role of a store for service calls. When a block sends a call for a service, its corresponding services (according to the adaptation contract) will be automatically created as tokens, and they will be stored in the store place. However, to be able to receive a call for a service, the block needs to verify that this call (message) is available in the store place.



(a) one(required)-to-one(provided) (b) one(required)-to-many(provided)

(c) one(provided)-to-many(required)

Figure 10.3: Rules for synthesizing the reused blocks.

Thus, the CPNs, which schedule the actions of the reused blocks, are glued using the store place and a set of transitions (which translate the adaptation contract). In the following, through these rules, we explain how we glue them.

- **rule 1: one(required)-to-one(provided).** We apply the rule 1 (presented in Figure 10.3(a)), if we have in the adaptation contract, a vector:

  $v=\langle e_1, \ldots, e_n, \varepsilon \rangle$, where $e_i=\{x!\}$, $e_j=\{y?\}$

  and $\forall$ f=1..n, $f \neq i$, $f \neq j$, we have $e_f = \varepsilon$.

v specifies that the required service x of block B_i corresponds to the provided service y of block B_j. In this case, when the block B_i executes the transition x!, it generates the corresponding action y as a token, which will be consumed later by the block B_j, when it tries to execute the action y?.

- **rule 2: one(required)-to-many(provided).** We apply the rule 2 (presented in Figure 10.3(b)) if we have, in the adaptation contract, a mapping vector:

  $v=\langle e_1, \ldots, e_n, \varepsilon \rangle$, where $e_i = \{x!\}$, $e_j = \{y_1?, \ldots, y_m?\}$

  and $\forall f = 1..n, f \neq i, f \neq j$, we have $e_k = \varepsilon$.

  v specifies that the required service x of block B_i corresponds to the provided services $y_1,\ldots,y_m$ of block B_j. In this case, when the block B_i executes the transition x!, it generates the corresponding actions $y_1,\ldots,y_m$ as tokens, which will be consumed later by the block B_j when it tries to execute an action $y_k \in \{y_1,\ldots,y_m\}$.

- **rule 3: one(provided)-to-many(required).** A correspondence of type One(provided)-to-many(required) between B_i and B_j means that the block B_i can execute the service mentioned in 'one(provided)' only after when the block B_j sends requests for all services specified in 'many(required)'. This correspondence can be specified at the level of the adaptation contract by a vector:

  $v=\langle e_1, \ldots, e_n, \varepsilon \rangle$, where $e_i = \{x?\}$, $e_j = \{y_1!, \ldots, y_m!\}$

  and $\forall f = 1..n, f \neq i, f \neq j$, we have $e_f = \varepsilon$.

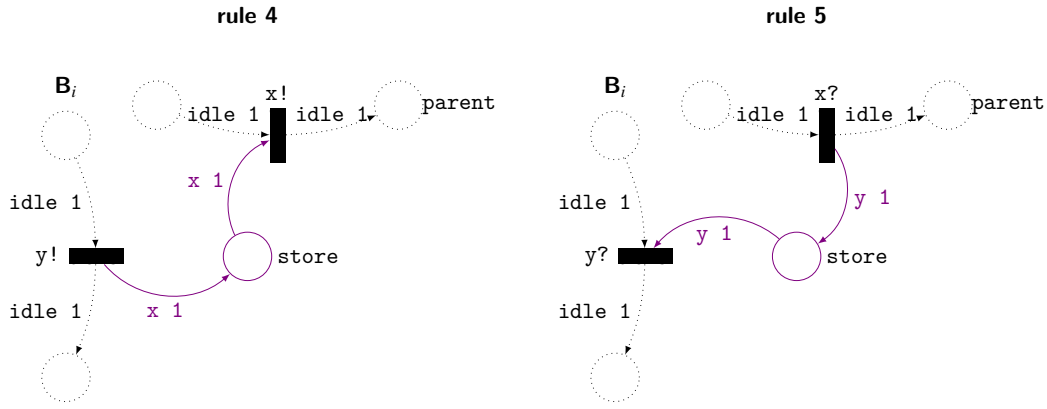
  To represent this vector using CPNs, we apply the rule 3 (see Figure 10.3(c)). Thus, we create a place 'pre-store-x'. This place stores calls for the services that correspond to x. Then, we link all transitions labelled by y_k! where k=1..m to the place 'pre-store-x'. After, we must create a transition tau that has an incoming arc which starts from the place 'pre-store-x'. This arc is labelled by $[y_k\ 1]_{k=1..m}$. We must also create an arc which starts from transition tau and ends at the store place, where this arc must be labelled by 'x 1'. Finally, to allow to the block B_i to execute the service x, we link the store state with the transition x?.

10.1.2/ Introducing the Specification of the Future Parent Block

In this phase, we must introduce the rules that link the reused blocks and the future parent block. The specification of the parent block B allows to define what the environment expects (resp. offers) from (resp. to) the parts of B. Because we have reused the sub-blocks of B, we must adapt them to the specification of their future parent which is made initially with respect to the part of the system already developed. So, to represent the part of the adaptation contract that specifies the relations between B and their sub-blocks, using CPNs, we need firstly to apply Algorithm 1 on the SD of the future parent block B to obtain its CPN. After, we apply the delegation rules (presented later) to synthesize the CPN of B and the CPN which represents the global interactions of the reused blocks.

Because we will represent the delegation between the set of blocks $\{B_i\}$ and their father, in these rules the correspondences are expressed between services of the same type. This means that a required service can not correspond to a provided service.

- **The correspondences of type one(parent)-to-one(child).** This correspondence can be specified at the level of the adaptation contract by a vector:



(a) one(parent)-to-one(child)[required] (b) one(parent)-to-one(child)[provided]

Figure 10.4: The correspondences of type one(parent)-to-one(child).

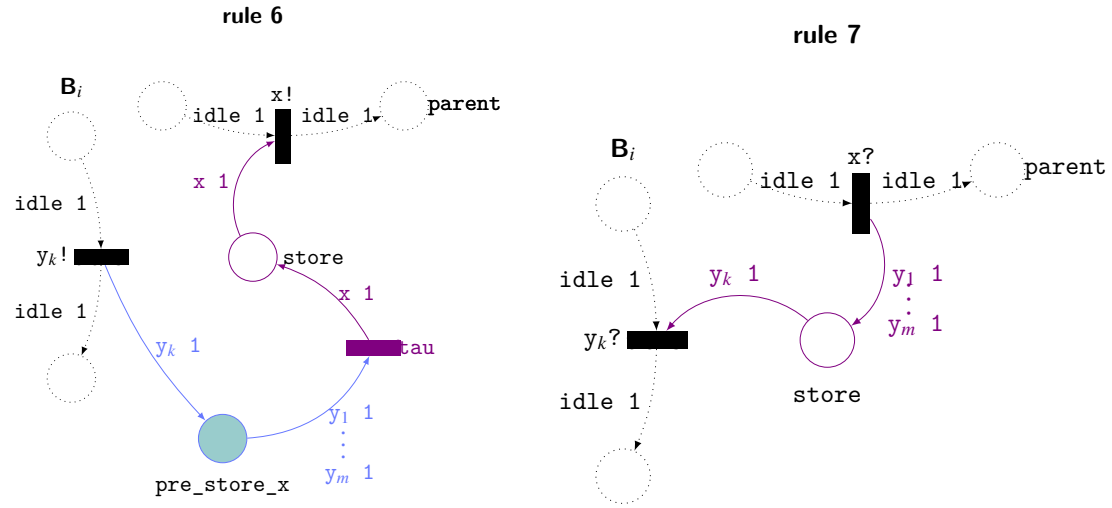
v=⟨ e_1, ...,e_n, s ⟩, where (s={x!} ∧ e_i={y!}) ∨ (s={x?}∧e_i={y?})
and ∀ f=1..n, f≠ i, we have e_f = ε.

In this case where we have v=⟨ ε,..., {y},...,ε, {x} ⟩, it means that the environment recognizes the service y of the block B_i as service x. Thus, we can differentiate two cases:

- **rule 4: one (parent)-to-one (child)[required].** This correspondence can be specified at the level of the adaptation contract using a vector:
  v=⟨ e_1, ...,e_n, s ⟩, where (s = {x!}) ∧ (e_i = {y!}),
  and ∀ f=1..n, f≠ i, we have e_f = ε.
  To represent this vector using CPNs, we apply the rule 4 presented in Figure 10.4(a). This rule consists in transforming the call for the service y of the block B_i to a call for the service x of the environment.
- **rule 5: one (parent)-to-one (child)[provided].** This correspondence can be specified at the level of the adaptation contract by a vector:
  v=⟨ e_1, ...,e_n, s ⟩, where (s = {x?}) ∧ (e_i = {y?})
  and ∀ f=1..n, f≠ i, we have e_f = ε.
  To represent this vector using CPNs, we apply the rule 5 (see Figure 10.4(b)). This rule plays its role in the other direction of rule 4 (i.e. when the block B_i offers the service y, the environment recognizes this service as x). In this case, we must rename each reception of x to y to be received after by the block B_i.

- **The correspondences of type one(parent)-to-many(child).** We distinguish two cases:

  - **rule 6: one (parent)-to-many (child)[required].** In this case, we have one required service of the parent (which is a provided service of the environment), corresponds to many required services of the child block B_i. This correspondence is represented in the contract using a vector v:
    v=⟨ e_1,..., e_n, s ⟩, where (s={x!}∧ e_i ={y_1!,...,y_m!})
    and ∀ f=1..n, f≠ i , we have e_f = ε.



(a) one(parent)-to-many(child)[required] (b) one(parent)-to-many(child)[provided]

Figure 10.5: The correspondences of type one(parent)-to-many(child).

To represent this vector using CPNs, we apply the rule in Figure 10.5(a). The block B_i sends calls for the services y_k ($\forall$ k=1..m) at the level of k different transitions. These calls will be stored in the place 'pre-store-x'. When, we have all the token y_1,..., y_m present in the place 'pre-store-x', the parent block can consume them to pass a call for the service x which is implemented by the environment.

– **rule 7: one (parent)-to-many (child)[provided].** In this case, we have one provided service of the parent (which is a required service of the environment) corresponds to many provided services of the child block B_i. This correspondence is represented in the contract using a vector:

$v=\langle e_1,\ldots, e_n, s\rangle$, where $(s = \{x?\}) \wedge (e_i = \{y_1?,\ldots,y_m?\})$, and $\forall$ f=1..n, $f \neq i$, we have $e_f = \varepsilon$.

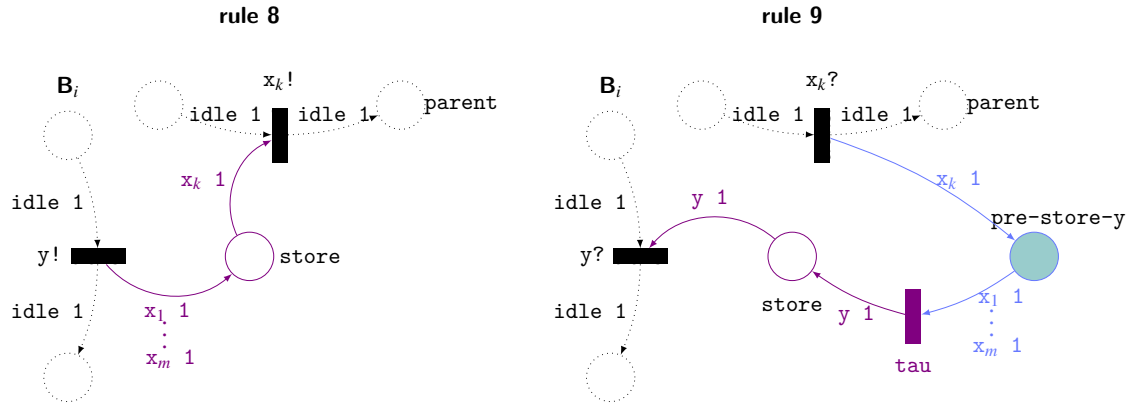
To represent this vector using CPNs, we apply the rule in Figure 10.5(b). When the parent block receives a call for the service x, we transform it to a call for the services y_1?,...,y_m? which are implemented by the block B_i, and we store them in the store place. when the block B_i will be ready to execute a service $y_k \in \{y_k|$ k=1..n}, it consumes the corresponding token from the store place.

- **The correspondences of type one(child)-to-many(parent).** In this case, we can also distinguish two sub-cases:

  – **rule 8: one (child)-to-many (parent)[required].** In this case, we have one required service of the child block B_i corresponds to many required services of its parent block B (which are provided services of the environment). This correspondence is represented in the contract using a vector:

  $v=\langle e_1,\ldots, e_n, s\rangle$, where $(s = \{x_1!,\ldots,x_m!\}) \wedge (e_i = \{y!\})$, and $\forall$ f=1..n, $f \neq i$, we have $e_f = \varepsilon$.

  To represent this vector using CPNs, we apply the rule in Figure 10.6(a). When the block B_i sends a call for the service y, we generate calls for its corresponding services x_1,..., x_m, and we store them in the store place. When the parent block



(a) one(child)-to-many(parent)[required] (b) one(child)-to-many(parent)[provided]

Figure 10.6: The correspondences of type one(child)-to-many(parent).

will be able to send the request for a service x_k which is implemented by the environment, the corresponding token will be consumed from the store place.

– **rule 9: one (child)-to-many (parent)[provided].** In this case, we have one provided service of the child block B_i corresponds to many provided services of its parent block B (which are required services of the environment). This correspondence is represented in the contract using a vector:

  v=⟨ e_1,..., e_n, s ⟩, where (s = {x_1?,...,x_m?} ∧ (e_i = {y?}),

  and ∀ f=1..n, f≠ i, we have e_f = ε.

  To represent this vector using CPNs, we apply the rule in Figure 10.6(b). When the parent block receives a call for the service x_k, we store this call in the pre-store-y place. When we have all the calls for services (x_1?,...,x_m?) present as tokens in the place pre-store-y, we consume them and we transform them to a token y, after we store y in the store place. The token y will be consumed later by the block B_i.

10.1.3/ Deduce the Adapter

The adapter will play the role of a mirror between the reused sub-blocks {B_i}. So each call for a service by a sub-block B_i will be received by the adapter, and each reception of a call for a service by a sub-block B_i must be preceded by a call for this service, this call must be emitted by the adapter. Thus, to generate the adapter, we base on the CPN synthesized in the last phase. Thus, we take this CPN, and we apply the mirror function on some transitions, we transform each call for a service (x!) by a reused block B_i into a reception of this call (x?), and each reception of a call for a service (x?) by a reused block B_i must be transformed to an emission of this call (x!). Therefore this transformation concerns only the transitions of the reused blocks, because the adapter plays the role of mirror only between the reused sub-blocks. Concerning the relation between the adapter and the parent block, it is a delegation relation. Thus, the adapter will delegate the parent to interact with the environment, that's why, we don't need to inverse the actions done at the level of the parent transitions.

At this stage, we have the $CPN_{adapter}$ which represents the interactions of the adapter with

the reused blocks and their environment. To generate the different scenarios of interaction, we compute the reachability graph of $CPN_{adapter}$ using CPNtool.

Algorithm 8 resumes all the previous steps. It takes as parameters the reused blocks $\{B_i\}_{i=1..n}$ with their future parent block B. The other parameters are the SDs that model the interaction protocol of these blocks and the adaptation contract C. The role of the algorithm is to compute the coloured Petri net of the adapter.

Algorithm 8 Construct the CPN of the adapter

INPUT: $\{B_i\}_{i=1..n}$, $\{SD_i\}_{i=1..n}$, B, SD_B, C
OUTPUT: $CPN_{adapter}$

```
1:  for all i ∈ [1...n] do
2:      CPN_i ← SD2CPN(B_i, SD_i)
3:  end for
4:  CPN_B ← SD2CPN(B, SD_B)
5:  CPN_adapter ← concatenation of {CPN_i}_i=1..n and CPN_B
6:  -create and add the store place to CPN_adapter
7:  for all v ∈ C do
8:      apply the rule that corresponds to v
9:  end for
10: for all t ∈ CPN_adapter do
11:     if ∃ t' ∈ CPN_i, t.name=t'.name then
12:         change the name of t to mirror(t.name)
13:     end if
14: end for
```

Algorithm 9 generates the structure of the adapter block basing on its CPN, and the other reused blocks.

Finally, we use algorithm 9 to generate the structure of the new part **B** that we will integrate to our system. The algorithm 10 constructs the blocks definition (BDD_B) diagram and the internal block diagram (IBD_B) of the part B. It takes as parameters the reused blocks ($\{B_i\}$), the parent block **B** and the adapter block.

The complexity of Algorithm 8 is computed as follows: The first loop [lines 1-3] calls the algorithm 7 n-times, where n is the number of the reused blocks. Thus, this loop will have a complexity equal to: $O(\Sigma_{i=1}^{n}\ m_{B_i})$, where m_{B_i} is the number of messages in the sequence diagram of the block B_i.

In line 4, there is a call for Algorithm 7 to construct the CPN of the parent block B. Thus, line 4 has a complexity equal to: $O(m_B)$.

The second loop [lines 7-9] has a complexity equal to $O(w)$, where w represents the number of mapping vector in the adaptation contract.

The complexity of the last loop [lines 10-14] is computed in function of the number of transitions (tn) in the synthesized CPN. Thus, the complexity will be equal to $O(tn)$.

Thus, $C(\text{Algorithm 8}) = O(\Sigma_{i=1}^{n}\ m_{B_i}) + O(m_B) + O(w) + O(tn) =$

$$O(\Sigma_{i=1}^{n}\ m_{B_i} + m_B + w + tn)$$

The complexity of Algorithm 9 is computed as follows: $C(\text{Algorithm 9}) = O(n)$, where n is the number of the reused blocks.

The complexity of Algorithm 10 is computed as follows: The first loop [lines 10-16] has a complexity equal to: $O(n*nport)$, where n is the number of instances of the reused blocks,

Algorithm 9 Construct the architecture of the block adapter

INPUT: $\{B_i\}_{i=1..n}$, B, $CPN_{adapter}=\langle$ P, T, A, Σ, C, N, E, G, I $\rangle$
OUTPUT: $B_{adapter}$

1: -Create the adapter block $B_{adapter}=\langle$ *'Adapter'*, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$ $\rangle$
2: //CREATE THE LIST OF PORTS OF THE ADAPTER THAT MUST BE LINKED TO THE PORTS OF THE PARENT BLOCK.
3: **if** $PS_B \cap \Sigma \neq \emptyset$ **then**
4: -create a new provided port p which offers the services
5: $PS_B \cap \Sigma$
6: -add p to the ports list of $B_{adapter}$
7: **end if**
8: **if** $RS_B \cap \Sigma \neq \emptyset$ **then**
9: create a new required port p which requires the services
10: $RS_B \cap \Sigma$
11: add p to the ports list of $B_{adapter}$
12: **end if**
13: //CREATE THE LIST OF PORTS OF THE ADAPTER THAT MUST BE LINKED TO THE PORTS OF SUB-BLOCKS $\{B_i\}$
14: **for all** B_i in the list of sub-blocks $\{B_i\}$ **do**
15: **if** $RS_{B_i} \cap \Sigma \neq \emptyset$ **then**
16: -create a new provided port p which offers the services
17: $RS_{B_i} \cap \Sigma$
18: **end if**
19: **if** $PS_{B_i} \cap \Sigma \neq \emptyset$ **then**
20: -create a new required port p which requires the
21: services $PS_{B_i} \cap \Sigma$
22: **end if**
23: -add p to the ports list of $B_{adapter}$
24: **end for**

and nport is the number of ports of the adapter.

The second loop has a linear complexity equal to: O(nport)

Thus, C(Algorithm 10)= O(n*nport).

10.1.4/ Tool Support

Modelling. To model the interactions of the blocks with their environment, we base on papyrus tool [pap]. Papyrus offers a set of graphical editors to model systems using UML and SysML diagrams. The editor of sequence diagrams bases on a SysML ecore meta-model part that defines the classes of lifelines, messages, etc.

Transformation. To generate the CPNs that correspond to the sequence diagrams of blocks, we have defined a meta-level model-driven approach which bases on meta-modelling and ATL transformation. Thus, firstly, we have defined the meta-model (.ecore) for CPNs using EMF [EMF]. After, we have defined an ATL grammar that performs the transformation SDs→CPNs. In our adaptation approach, we don't need to show the recursive messages (messages that start and end at the same lifeline). Because the adapter

Algorithm 10 Construct the BDD and the IBD of B

INPUT: $\{B_i\}_{i=1..n}$, B, $B_{adapter}$
OUTPUT: BDD_B, IBD_B

1: //CONSTRUCT THE BDD OF B
2: - Set the value of the blocks set of the BDD_B to: B= $\{B_i\}_{i=1..n} \cup \{B, B_{adapter}\}$
3: - Create a composition relation r_i between the parent block B and each block B_i where: SourceOf(r_i) = B, TargetOf(r_i) = B_i
4: - Create a composition relation r_{ad} between the parent block B and the adapter block $B_{adapter}$ where: SourceOf(r_{ad}) = B, TargetOf(r_{ad}) = $B_{adapter}$
5: - Set the value of the relations set of BDD_B to: R= $\{r_i\}_{i=1..n} \cup \{r_{ad}\}$
6: //CONSTRUCT THE IBD OF B
7: - Set the set *Parts* of IBD_B to: $\{part_i\}_{i=1..n} \cup \{part_{adapter}\}$
8: - Set the set *Ports* of IBD_B to: $\{Ports(part_i)\}_{i=1..n} \cup Ports(part_{adapter})$
9: //CREATE CONNECTORS BETWEEN THE ADAPTER AND THE PARTS THAT INSTANTIATE THE REUSED BLOCKS
10: **for all** $part_i \in \{part_i\}_{i=1..n}$ **do**
11: **for all** port $p \in Ports(part_{adapter})$ **do**
12: **if** $\exists p' \in Ports(part_i) \wedge (p.type.Op \cap p'.type.Op \neq \emptyset)$ **then**
13: -create a connector between p and p'
14: **end if**
15: **end for**
16: **end for**
17: //CREATE DELEGATION CONNECTORS BETWEEN THE ADAPTER AND THE PARENT BLOCK B.
18: **for all** port $p \in Ports(ad)$ **do**
19: **if** $\exists p' \in Ports(B) \wedge (p.type.Op \cap p'.type.Op \neq \emptyset)$ **then**
20: - create a connector between p and p'
21: **end if**
22: **end for**

can't see this kind of actions (messages), thus, we have transformed only the entered and exited messages and we have ignored the recursive messages. To implement this transformation we have defined three ATL rules. The first rule allows initializing the CPN, the second rule is used to generate the places of the CPN, and the third one creates transitions and arcs that link these transitions with the places generated by the previous rule. These ATL rules implement the algorithm 7.

Synthesising the adapter. To synthesis the CPN of the adapter, we have also defined a meta-level model-driven approach which base on meta-modelling and ATL transformation rules. We have defined a meta-model of the adaptation contract using EMF [EMF], and we have generated its graphical editor using GMF [Pro]. Next, basing on the meta-model of CPNs and the meta-model of the adaptation contract, we have defined an ATL grammar that generates the CPN of the adapter.

Generate the scenarios of the adapter. To obtain the interaction scenarios of the adapter, we compute the reachability graph of its CPN. The reachability graph can be computed using CPNtool[too]. We plan to develop an Acceleo [Acc] transformation to generate the entry files of the CPN tool from our CPN meta-model.

10.2/ Case Study

We give an example of a specification of a simple robot which moves according to a specific path. This robot (see Figure 10.7) can receive a command to fix its speed (`R_setSpeed`) and after a command to move (`R_move`). Next, it sends a request to the environment (e.g. a remote control) to display its initial location (`R_DisplayInitialLoc`). After, it is still moving until it receives a request to stop (R_stop). Finally, it sends a request to the environment (e.g. a remote control) to display the final report (`R_DisplayReport`) about its final location and the travelled distance. The interaction of this robot with its environment is modelled using the sequence diagram in Figure 10.7.

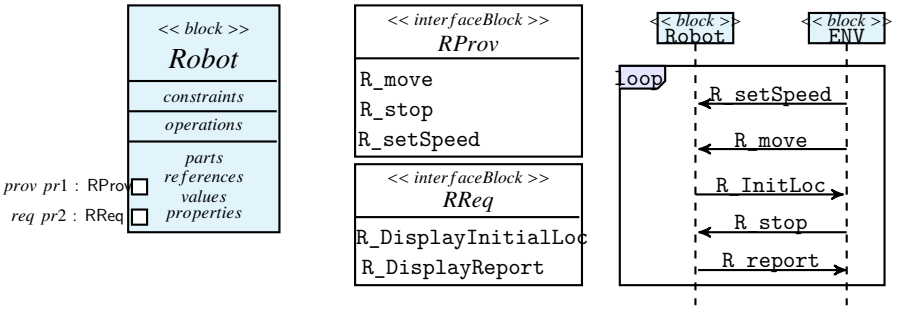


Figure 10.7: `The specification of the robot`

In our case study we want to build this robot. To do that, we have reused two blocks 'Controller' (see Figure 10.8) and 'Moving_System' (see Figure 10.9), their interaction protocols are modelled using sequence diagrams. Thus, the Robot will be the parent block and the reused blocks ('Controller' and 'Moving_System') will be its children.

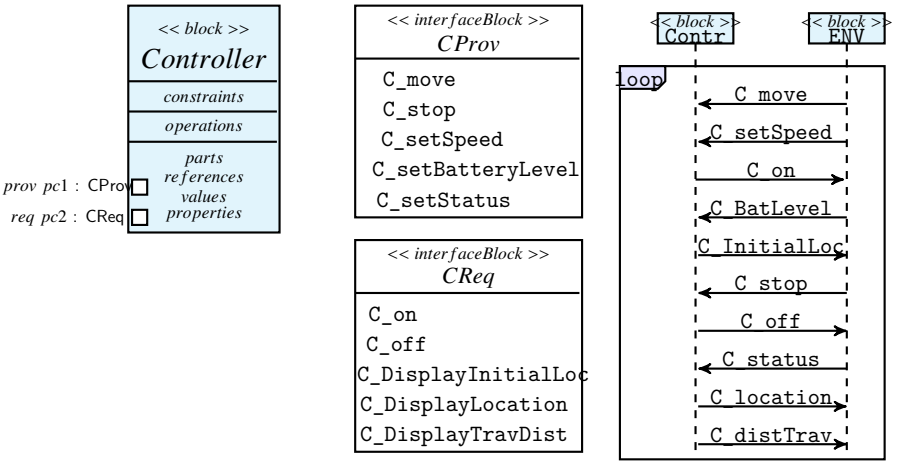


Figure 10.8: The Controller

In Figure 10.8, we can see that our reused controller receives a request to move (`C_move`) before receiving the request to fix the speed (`C_setSpeed`). After, it asks the moving system to go on (`C_on`), and so it receives an information about the battery level (`C_setBatteryLevel`) from the moving system. Next, it communicates the initial location (`C_DisplayInitialLoc`) to the environment (e.g. a remote control). The controller waits for a request to stop (`C_stop`). After receiving this request, it asks the moving system to turn off (`C_off`), and so this last will communicate its status to the con-

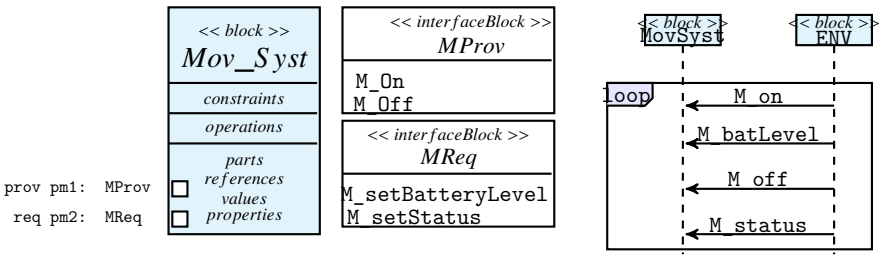


Figure 10.9: The moving system

troller (C_setStatus). Finally, the controller asks the environment to display the location of the engine to which is belong (C_DisplayLocation), and the travelled distance (C_DisplayTravDist).

To simplify, we consider that the corresponding services (one-to-one) have the same names, and we differentiate between them by adding the first letter of the block's name to each service. Thus, to adapt and assemble the controller and the moving system to meet the specification of our robot, we use the following contract (Figure 10.10 represents the contract modelled using our generated editor).

C= { ⟨C_on!, M_on?, ε⟩, ⟨C_off!, M_off?, ε⟩, ⟨C_status?, M_status!, ε⟩, ⟨C_batLevel?, M_batLevel!, ε⟩, ⟨C_move?, ε, R_move?⟩, ⟨C_setSpeed?, ε, R_setSpeed?⟩, ⟨C_stop?, ε, R_stop?⟩, ⟨C_InitialLoc!, ε, R_InitialLoc!⟩, ⟨(C_location!,C_disTrav!), ε, R_report!⟩ }

Thus, The adaptation contract C includes nine vectors, where:

- v_1=⟨C_on!, M_on?, ε⟩
- v_2=⟨C_off!, M_off?, ε⟩
- v_3=⟨C_status?, M_status!, ε⟩
- v_4=⟨C_batLevel?, M_batLevel!, ε⟩
- v_5=⟨C_move?, ε, R_move?⟩
- v_6=⟨C_setSpeed?, ε, R_setSpeed?⟩
- v_7=⟨C_stop?, ε, R_stop?⟩
- v_8=⟨C_InitialLoc!, ε, R_InitialLoc!⟩
- v_9=⟨(C_location!,C_disTrav!), ε, R_report!⟩

By applying the algorithm 8, we have obtained the CPN presented in Figure 10.11.

1. we have transformed the sequence diagrams of the robot (parent block), the controller and the moving system to coloured Petri nets (we have applied the algorithm 7). In Figure 10.11, by ignoring the coloured arcs, and applying the mirror on the transitions of this cpn,we can see that the right part of the cpn represents the cpn of the moving system, the left part represents the cpn of the controller, and the bottom part represent the cpn of the robot.

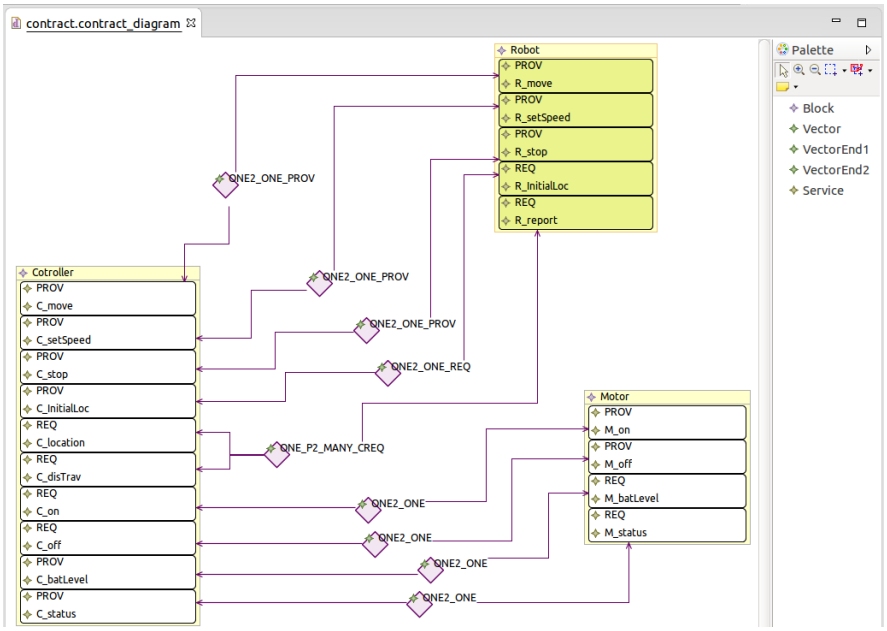


Figure 10.10: Adaptation Contract modelled using our generated editor

2. we have applied the rule 1 to represent the vectors 1, 2, 3 and 4 of the contract.
3. we have applied the rule 5 to represent the vectors 5, 6 and 7 of the contract.
4. we have applied the rule 4 to represent the vector 8 of the contract.
5. we have applied the rule 6 to represent the last vector of the contract.
6. we have inverted the actions of the controller and the moving system (each emission becomes a reception, and each reception become an emission).

By applying the algorithm 10, we obtain the BDD of the robot (in the Figure 10.12) and its IBD (in the Figure 10.13).

10.3/ Conclusion

In this chapter, we have presented a bottom-up approach to adapt SysML blocks for building systems. Our adaptation process takes a part of the system to develop, and generates an adapter for the SysML blocks which are reused to meet the specification of this part. During the adaptation process, we consider the part to develop as a SysML composite block. After, we select a set of blocks to fulfil the specification of this composite block (the specification of the block that will include these reused blocks). We have relied on sequence diagrams of SysML to model the interactions of each block with its environment. Due to the informal aspect of SysML, we have proposed to transform the sequence diagrams of blocks into coloured Petri nets, and we have implemented this transformation using EMF and GMF for meta-modelling and ATL language for transformation. This transformation was guided by our objective of adaptation. After, to generate the adapter, starting from an adaptation contract, we have proposed the staple rules to link the coloured Petri nets of blocks. Our adaptation contract can specify atomic or non-atomic

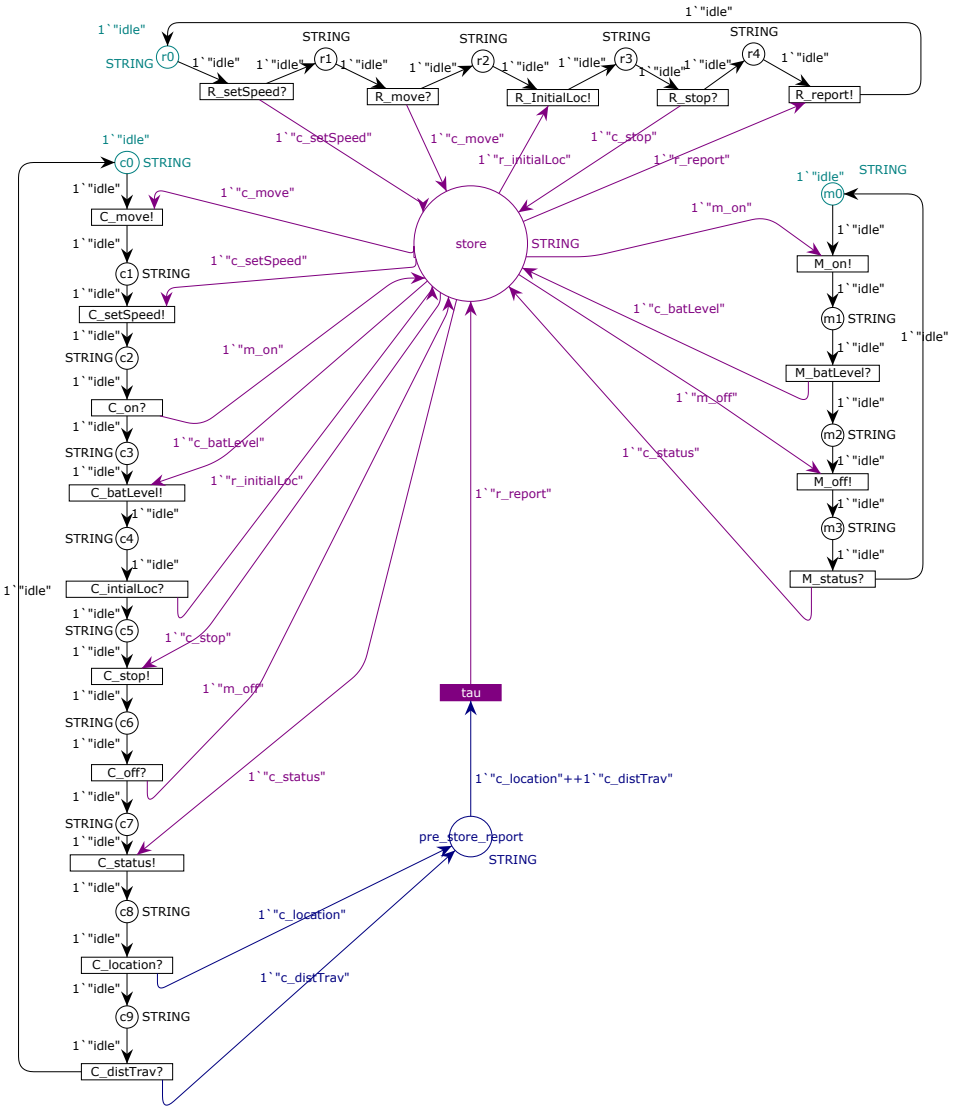


Figure 10.11: CPN$_{adapter}$

correspondences between the reused blocks or between the reused blocks and their future parent block (the specification of the adapter built in function of the developed part of the system and its part still to develop). To let the designer deal with graphical elements for modelling, we have proposed a meta-model for the adaptation contract, and we have developed its graphical editor using GMF. We have also implemented the ATL rules that represent our rules of adaptation (translation of the contract).

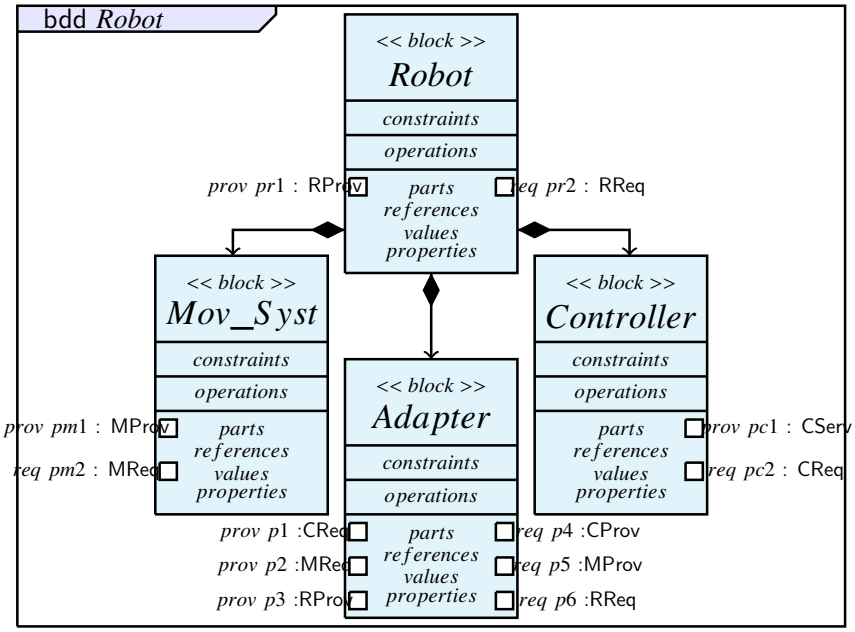


Figure 10.12: BDD of the Robot

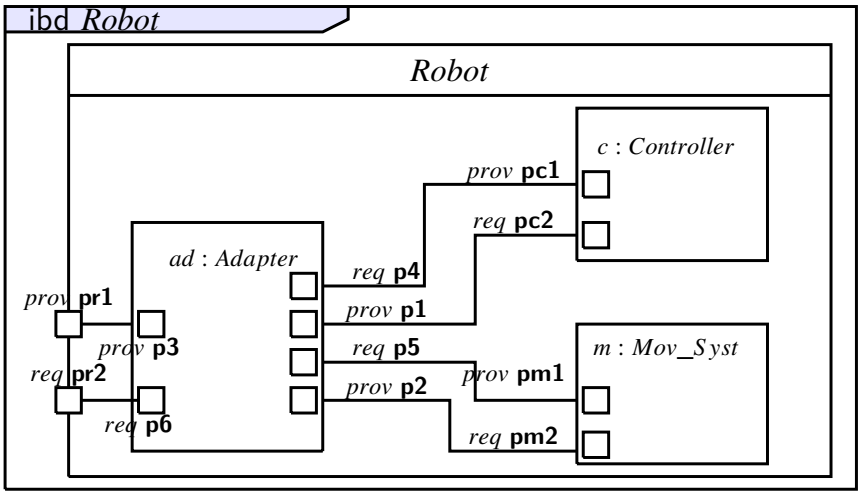


Figure 10.13: IBD of the Robot

III

Conclusion

11

Conclusion and Perspectives

11.1/ Conclusion

The component-based development (CBD) focuses on the decomposition of the system into individual functional components that represent well-defined communication interfaces. The primary objective of CBD is to ensure component re-usability. CBD techniques involve procedures for developing systems by choosing ideal off-the-shelf components, adapting and assembling them using well-defined adapters.

The main purpose of this thesis is the proposition of a formal approach to build incrementally complex systems by assembling and adapting a set of components, where their structure and behaviour are modelled using SysML diagrams. When assembling these blocks, we must take into consideration the fact that these components may present some incompatibilities. A decision about the compatibility or not of these blocks must be taken onto several different levels. The first level concerns the signature of component services. When the blocks expose a problem of mismatches between the name of corresponding services, this problem can be easily solved by inserting an adapter that aligns the names of block services and thus allows the communication between the reused blocks. However, the incompatibility at the level of the interaction protocols cannot be solved in some cases. The feasibility of solving this kind of incompatibility depends on the authorized operations during the adaptation (reordering the calls for services of blocks or not). It may depend also on the requirements that we want to satisfy by reusing and composing these blocks.

In our thesis, we have exploited the SysML diagrams to specify the requirements, the architecture and the interactions of the blocks, and we have applied the transformations on them in a model-driven process to extract their equivalents of formal models. This process is based on meta-modelling and model transformations. We have also adapt the existing approaches of adaptation of components to be applied in an incremental approach by exploiting the notion of SysML blocks refinement.

Thus, the main contributions of our thesis are the followings:

1. We have introduced SysML sequence diagrams that model component protocols, into a model-driven process, that is based on meta-modelling and model transformations, to obtain their equivalent of interface automata. Thus, we have based on the part of UML/SysML meta-model, that concerns the constructs of the sequence diagrams, as source meta-model of the ATL grammar that we have defined. To draw sequence diagrams, we have based on the graphical editor of Papyrus which

is built on the UML/SysML meta-model. The target meta-model concerns interface automata formalism. Thus, we have used EMF (Eclipse Modelling Framework) to define its meta-model, and GMF (Graphical Modelling Framework) to generate its graphical editor. We have established the mapping rules between the source and the target meta-models and we have implemented them using ATL (Atlas Transformation Language). Since the objective was to verify the compatibility of blocks basing on the optimistic approach of Hizenger that is defined on interface automata, we have used Ptolemy tool that implements the parallel composition of interface automata to decide on the compatibility or not of interface automata. To discharge the user from redrawing the interface automata using the Ptolemy user interface, we have proposed a set of Acceleo templates to generate automatically the Ptolemy entry specification.

2. When a block has a complex interaction with its environment, a use of an hierarchical model appears worthwhile. In our thesis, we have proposed to use HPSM (Hierarchical Protocol State Machine), the model that we have defined basing on the notions of provided and required services of the blocks and that allows us to establish a scheduling of the interactions of the block with its environment. This model uses the composite states to hide the details of some block's states. Since our objective was to alleviate the verification of the compatibility of blocks basing on the abstraction manifested by their interaction models, we have proposed to transform the HPSMs of blocks to hierarchical interface automata. After that, by analysing the relation between the provided services (input actions) and the required services (output actions) of the blocks, we can decide on the set of composite states of interface automata that must be considered as abstract states, i.e., the composite states that we will ignore their contained actions, and thus we consider them as simple states. The rest of composite states will be flattened, and the compatibility verification will be applied using Ptolemy tool.

3. In this stage, we process the problem of name mismatches between the services of the interacting blocks. Thus, we proposed an approach to define a block adapter that allows the interaction between the blocks that present mismatches on their protocols. We have adapted the synchronous product and the parallel composition of interface automata to consider corresponding actions of automata rather than shared actions. Thus, we have defined the notions of contract-based synchronous product and contract-based parallel composition. It is at the level of the adaptation contract where we define the corresponding actions. That is what justify the introduction the adaptation contract as a third parameter of these operations. In this case an introduction of an adapter block is mandatory, but it is not always possible. That is why we have defined some conditions on these blocks. Also, our adaptation is defined as an incremental approach, where we give the specification of the future parent block of the reused blocks. The generation of the adapter is based on the refinement relation that exists between these reused blocks and their parent. But, in this stage we haven't authorized the reordering of services calls, i.e., we cannot adapt the block that asks for services, implemented by sibling blocks or the environments of its parent, in a given order, however these latter offer there services in another order that is different from the first one.

4. To verify the preservation of the requirements initially satisfied by the reused blocks, we have defined our approach to focus only on the generated adapters to perform

the verification, which allows to avoid the problem of state explosion. Our approach prevents the computation of the synchronous product of all the system blocks, and focus only on the interaction protocol of a sub-set of adapters. We have used SPIN model checker and Promela, its input specification, to verify the properties that specify the SysML requirements. Thus, we were inspired from the works proposed by V. Lima et al. to generate Promela-based models from UML interactions expressed in Sequence Diagrams, and uses conjointly the SPIN model checker in order to simulate and verify properties written in Linear Temporal Logic(LTL) on a set of flags that represent the exchanged messages.

5. In the same context (incremental adaptation), to make the adapter able to solve more type of mismatches (not only one-to-one), and to allow the reordering of the calls for services, we have defined another approach which bases on Coloured Petri Nets (CPNs). The Petri nets are characterized by their richness compared with interface automata. They easily allow to represent the reordering of service calls. Thus, we have analysed the different relations that may exist between the reused blocks and their parent, and we have defined a CPN rule for each relation. We have based on EMF, GMF and ATL to implement our approach.

11.2/ Perspectives

Our approach is suitable for systems where the temporal order of actions is important. These actions can represent either a request or a reception of a request for component services. This information represents the interface of the components and it is available at the level of the contract (here we don't talk about the adaptation contract, but the contract that specify the information related to the input/output of the component) offered with the component. This contract respects the principle of black-box components, thus it mentions only the manifested information from the component to its environment. To adapt our approach to other types of systems, the contract must perform more information. For example, Timed interface automata was proposed by Hinzinger, thus it will be interesting to consider this formalism with our approach which is based on SysML models.

Our approach of verification is suitable for temporal properties because the contract associated with components mentions only the temporal order of executing the actions. To allow a verification of the functional properties that concern for example the value of a variable, we must enrich the contract (i.e. the model used to represent the behaviour of components). We can for example extend the contract of the component by pre and post conditions . After that, using these information, we associate to each state of the component a set of flags that specify the value of component variables. Because the adapter is generated from the parallel composition, we can deduce the values of each variable of the adapted blocks at the level of the adapter states. Then, when we generate the Promela code, we extend the set of flags associated to actions by the set of flags associated to the variables.

We can consider these two points as perspectives of our work.

Bibliography

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: The kobra approach. In Patrick Donohoe, editor, Software Product Lines, volume 576 of The Springer International Series in Engineering and Computer Science, pages 289--309. Springer US, 2000.

[Acc] Acceleo. http://www.eclipse.org/acceleo/.

[AEC14] Mouna Aouag, Raida Elmansouri, and Allaoua Chaoui. From uml 2.0 diagrams to aspect oriented diagrams using graph transformation. International Journal of Computer Aided Engineering and Technology, 6(2):200--233, 2014.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. ACM Trans. Softw. Eng. Methodol., 6(3):213--249, July 1997.

[atl] ATL: Atlas Transformation Language. https://eclipse.org/atl/.

[AYAM11] Mouna Ait_Oubelli, Nadia Younsi, Abdelkrim Amirat, and Ahcene Menasria. From uml 2.0 sequence diagrams to promela code by graph transformation using atom 3. CIIA'11, 2011.

[BBB+11] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. IEEE Software, 28(3):41--48, 2011.

[BBC05a] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A Formal Approach to Component Adaptation. J. Syst. Softw., 74(1):45--54, January 2005.

[BBC05b] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. Journal of Systems and Software, 74(1):45--54, 2005.

[BBJ+10] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010, pages 209--218, 2010.

[BCHM15] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings, chapter SysML Blocks Adaptation, pages 417--433. Springer International Publishing, Cham, 2015.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java.

In Proceedings of the 7th International Symposium on Component-based Software Engineering, Lecture Notes in Computer science, pages 7--22, Edinburgh, UK, 2004. Springer.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper., 36(11-12):1257--1284, September 2006.

[BCP04] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural types and component adaptation. In Algebraic Methodology And Software Technology, pages 42--56. Springer, 2004.

[BCS+08] Nikola Benes, Ivana Cerná, Jiri Sochor, Pavlína Vareková, and Barbora Zimmerova. A case study in parallel verification of component-based systems. Electr. Notes Theor. Comput. Sci., 220(2):67--83, 2008.

[BGMPG99] G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. The INRIA Rhone-Alpes CyCab' , INRIA, 1999.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In Software Engineering Research, Management and Applications, 2006. Fourth International Conference on, pages 40--48, Aug 2006.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In Proceedings of the 6th International Workshop on Software and Performance, WOSP '07, pages 54--65, New York, NY, USA, 2007. ACM.

[BR08] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. Formal Asp. Comput., 20(2):205--224, 2008.

[BSBM04] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible? In Technologies for E-Services, 5th International Workshop, TES 2004, Toronto, Canada, August 29-30, 2004, Revised Selected Papers, pages 15--28, 2004.

[CCM12a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Formalizing and verifying compatibility and consistency of SysML blocks. In UML & FM 2012, 5-th Int. workshop on UML and Formal Methods, volume 37-4 of ACM Software Engineering Notes, pages 1--8, Paris, France, August 2012.

[CCM12b] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Formalizing and Verifying Compatibility and Consistency of SysML Blocks. ACM SIGSOFT Software Engineering Notes, 37(4):1--8, 2012.

[CCM14] 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, FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers, volume 8348 of LNCS, pages 79--99, Nanchang, China, 2014. Springer.

[CEP00] L.A. Cortes, P. Eles, and Zebo Peng. Verification of embedded systems using a Petri net based representation. In System Synthesis, 2000. Proceedings. The 13th International Symposium on, pages 149--155, 2000.

[CESK09] Allaoua Chaoui, Raida ElMansouri, Wafa Saadi, and Elhillali Kerkouche. From uml sequence diagrams to ecatnets: a graph transformation based approach for modelling and analysis. In proceedings of The 4th International Conference on Information Technology ICIT, 2009.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, Cambridge, MA, USA, 1999.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. Model checking. MIT Press, 2001.

[CH03] K. Czarnecki and s. Helsen. Classification of model transformation approaches. In OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, Anaheim, USA, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. IBM Systems Journal, 45(3):621--645, 2006.

[CH11a] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on sysml and interface automata. ISSE, 7(4):265--274, 2011.

[CH11b] Samir Chouali and Ahmed Hammad. Formal Verification of Components Assembly Based on SysML and Interface Automata. ISSE, 7(4):265--274, 2011.

[CIP04] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, pages 221--230, May 2004.

[CK96] Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. ACM Trans. Softw. Eng. Methodol., 5(4):334--377, October 1996.

[CLY+14] Chih-Hung Chang, Chih-Wei Lu, Wen Pin Yang, W.C.-C. Chu, Chao-Tung Yang, Ching-Tsorng Tsai, and Pao-Ann Hsiung. A sysml based requirement modeling automatic transformation approach. In Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International, pages 474--479, July 2014.

[CMM10a] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation des Protocoles des Composants par les Automates d'Interface. In AFADL'10, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, pages 253--266, Poitiers, France, June 2010.

[CMM10b] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting component behaviours using interface automata. In Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on, pages 119--122. IEEE, 2010.

[CMM12] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation sémantique des protocoles des composants par les automates d'interface. TSI, Technique et Science Informatiques, 31(6):769--796, 2012.

[CMP06] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. L'OBJET, 12(1):9--31, 2006.

[CPS06a] Carlos Canal, Pascal Poizat, and Gwen Salaün. Adaptation de composants logiciels une approche automatisée basée sur des expressions régulières de vecteurs de synchronisation. In CAL, pages 31--39, 2006.

[CPS06b] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing Behavioural Mismatch in Software Composition. In Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings, pages 63--77, 2006.

[CPS08] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. IEEE Trans. Software Eng., 34(4):546--563, 2008.

[Crn02] Ivica Crnkovic. Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA, 2002.

[CS14] Carlos Canal and Gwen Salaün. Adaptation of asynchronously communicating software. In Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings, pages 437--444, 2014.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In ESEC / SIGSOFT FSE, pages 109--120, 2001.

[dAH05] Luca de Alfaro and ThomasA. Henzinger. Interface-based design. In Manfred Broy, Johannes Grunbauer, David Harel, and Tony Hoare, editors, Engineering Theories of Software Intensive Systems, volume 195 of NATO Science Series, pages 83--104. Springer Netherlands, 2005.

[Dav03] Frankel S David. Model driven architecture: applying mda to enterprise computing, 2003.

[DBM14] Djaouida Dahmani, Mohand Cherif Boukala, and Hassan Mountassir. A petri net approach for reusing and adapting components with atomic and non-atomic synchronisation. In Proceedings of the International Workshop on Petri Nets and Software Engineering, Tunis, Tunisia, pages 129--141, 2014.

[DHJ+10] Mourad Debbabi, Fawzi Hassaine, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. Verification and validation in systems engineering: assessing UML/SysML design models. Springer Science & Business Media, 2010.

[DIK09] P. David, V. Idasiak, and F. Kratz. Improving reliability studies with sysml. In Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual, pages 527--532, Jan 2009.

[dLVA04] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom3. Software and System Modeling, 3(3):194--209, 2004.

[DM01] Alexandre David and M. Oliver Möller. From HUPPaal to Uppaal : A Transition from Hierarchical Timed Automata to Flat Timed Automata. Technical report, BRICS, University of Aarhus, Denmark, 2001.

[DOP13] Iulia Dragomir, Iulian Ober, and Christian Percebois. Integrating verifiable assume/guarantee contracts in uml/sysml. In Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013., 2013.

[DOP14] Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety contracts for timed reactive components in sysml. In SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, Proceedings, pages 211--222, 2014.

[EMF] Eclipse Modelling Framework EMF. http://www.eclipse.org/modeling/emf/.

[EPK02] Petru Eles, Zebo Peng, and Daniel Karlsson. Formal Verification in a Component-Based Reuse Methodology. In Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002), October 2-4, 2002, Kyoto, Japan, pages 156--161, 2002.

[ES09] Sima Emadi and Fereidoon Shams. Mapping annotated use case and sequence diagrams to a petri net notation for performance evaluation. In Computer and Electrical Engineering, 2009. ICCEE'09. Second International Conference on, volume 2, pages 68--71. IEEE, 2009.

[GBB05] Patrick Graessle, Henriette Baumann, and Philippe Baumann. UML 2.0 in Action. Pearson Higher Education, 2005.

[GBHP13] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. Verification and validation of meta-model based transformation from SysML to VHDL-AMS. In MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development, pages 123--128, Barcelona, Spain, February 2013.

[GBHP15] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. Tooled process for early validation of SysML models using Modelica simulation. In FSEN'15, 6th IPM Int. Conf. on Fundamentals of Software Engineering, volume 9392 of LNCS, pages 230--237, Tehran, Iran, April 2015. Springer.

[GCA13] Fayçal Guerrouf, Allaoua Chaoui, and Ali Aldahoud. A graph transformation approach of mobile activity diagram to nested petri nets. International Journal of Computer Aided Engineering and Technology, 5(1):44--57, 2013.

[GCRJ08] Yue Guo, A. Chakrapani Rao, and R.P. Jones. Architectural and functional modelling of an automotive driver information system using sysml. In Mechtronic and Embedded Systems and Applications, 2008. MESA 2008. IEEE/ASME International Conference on, pages 552--557, Oct 2008.

[GDKR+11] Antonio García-Domínguez, Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Inmaculada Medina-Bulo. Eunit: A unit testing framework for model management tasks. In Model Driven Engineering Languages and Systems, pages 395--409. Springer, 2011.

[Gen] Gentleware. http://www.gentleware.com/.

[GGA+08] M. Grecki, Zheqiao Geng, Gohar Ayvazyan, S. Simrock, and Bahtiar Aminov. Application of sysml to design of atca based llrf control system. In Nuclear Science Symposium Conference Record, 2008. NSS '08. IEEE, pages 44--52, Oct 2008.

[GJ09] Yue Guo and R.P. Jones. A study of approaches for model based development of an automotive driver information system. In Systems Conference, 2009 3rd Annual IEEE, pages 267--272, March 2009.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer, 15(2):89--107, 2013.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 16--27. ACM, 2003.

[Hoa85] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[INC] INCOSE. http://www.incose.org/.

[IS01] Paola Inverardi and Simone Scriboni. Connectors synthesis for deadlock-free component based architectures. In Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on, pages 174--181. IEEE, 2001.

[IT03a] Paola Inverardi and Massimo Tivoli. Deadlock-free software architectures for com/dcom applications. Journal of Systems and Software, 65(3):173--183, 2003.

[IT03b] Paola Inverardi and Massimo Tivoli. Software Architecture for Correct Components Assembly. In SFM 2003, Bertinoro, Italy, pages 92--121, 2003.

[Jal08] Pankaj Jalote. A concise introduction to software engineering. Springer, 2008.

[Jen96] Kurt Jensen. Coloured Petri Nets (2Nd Ed.): Basic Concepts, Analysis Methods and Practical Use: Volume 1. Springer-Verlag, London, UK, UK, 1996.

[JKPB12] Thomas Johnson, Aleksandr Kerzhner, Christiaan JJ Paredis, and Roger Burkhart. Integrating models and simulations of continuous dynamics into sysml. Journal of Computing and Information Science in Engineering, 12(1):011002, 2012.

[JT13] Marcin Jamro and Bartosz Trybus. An approach to sysml modeling of iec 61131-3 control software. In MMAR, pages 217--222, 2013.

[KBSB10] Marouane Kessentini, Arbi Bouchoucha, Houari Sahraoui, and Mounir Boukadoum. Example-based sequence diagrams to colored petri nets transformation using heuristic search. In Modelling Foundations and Applications, pages 156--172. Springer, 2010.

[KC09] Elhillali Kerkouche and Allaoua Chaoui. A Formal Framework and a Tool for the Specification and Analysis of G-Nets Models Based on Graph Transformation. In Vijay Garg, Roger Wattenhofer, and Kishore Kothapalli, editors, Distributed Computing and Networking, volume 5408 of Lecture Notes in Computer Science, pages 206--211. Springer Berlin Heidelberg, 2009.

[KEP07a] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal verification of component-based designs. Design Automation for Embedded Systems, 11(1):49--90, 2007.

[KEP07b] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal Verification of Component-based Designs. Design Autom. for Emb. Sys., 11(1):49--90, 2007.

[LBLP11] Jonathan Lasalle, Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. Sysml to uml model transformation for test generation purpose. SIGSOFT Softw. Eng. Notes, 36(1):1--8, January 2011.

[LCKB08] S. Lafi, R. Champagne, A. B. Kouki, and J. Belzile. Modeling radio frequency front ends using sysml: a case study of a umts transceiver. In First International Workshop on Model Based Architecting and Construction of Embedded Systems, Toulouse, France, pages 115--128, 2008.

[LdSdO06] Marcos Vinicius Linhares, Alexandre Jose da Silva, and Rômulo Silva de Oliveira. Empirical evaluation of sysml through the modeling of an industrial automation unit. In ETFA, pages 145--152, 2006.

[LTM+09] Vitor Lima, Chamseddine Talhi, Djedjiga Mouheb, Mourad Debbabi, Lingyu Wang, and Makan Pourzandi. Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. Electr. Notes Theor. Comput. Sci., 254:143--160, 2009.

[LWMY11] Lei Li, Naichao Wang, Lin Ma, and Qingwei Yang. Modeling method of military aircraft support process based sysml. In Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on, pages 1247--1251, June 2011.

[M. 07] M. Corporation. COM: Component Object Model Technologies. http://www.microsoft.com/com/, 2007.

[MCR+12] F. Mhenni, J. Choley, A. Riviere, Nga Nguyen, and H. Kadima. Sysml and safety analysis for mechatronic systems. In Mechatronics (MECATRONICS) , 2012 9th France-Japan 7th Europe-Asia Congress on and Research and Education in Mechatronics (REM), 2012 13th Int'l Workshop on, pages 417--424, Nov 2012.

[MDA] The Model-Driven Architecture. http://www.omg.org/mda/specs.htm.

[Mer14] Elkamel Merah. Design of atl rules for transforming uml 2 sequence diagrams into petri nets. International Journal of Computer Science and Business Informatics, 8(1), 2014.

[Mic95] microsoft Microsoft. The component object model specification-version 0.9, 1995.

[MMSC13] Elkamel Merah, Nabil Messaoudi, Halima Saidi, and Allaoua Chaoui. Design of atl rules for transforming uml 2 communication diagrams into büchi automata. International Journal of Software Engineering and Its Applications, 7(2):19--34, 2013.

[Mot] Motorola. http://www.motorola.com/.

[Mou11] Sebti Mouelhi. Contribution à la vérification de la sûreté de l'assemblage et à l'adaptation de composants réutilisables. PhD thesis, Franche comté university, 2011.

[MPS12] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Trans. Software Eng., 38(4):755--777, 2012.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. Inf. Comput., 100(1):1--40, September 1992.

[MR07] Marylène Micheloud and Medard Rieder. Programmation orientée objets en C++: une approche évolutive. PPUR presses polytechniques, 2007.

[MTO+11] Yasushi Mae, Hideyasu Takahashi, Kenichi Ohara, Tomohito Takubo, and Tatsuo Arai. Component-based robot system design for grasping tasks. Intelligent Service Robotics, 4(1):91--98, 2011.

[obe] Obeo. http://www.obeo.fr.

[OMG00] OMG. CORBA, Version 2.4. http://www.omg.org/spec/CORBA/2.4/, 2000.

[OMG06] OMG. Corba Component Model 4.0 Specification. Technical Report Version 4.0, 2006.

[OMG12a] OMG. OMG Systems Modeling Language (OMG SysML™) Version 1.3, 2012.

[OMG12b] OMG. OMG Systems Modeling Language (OMG SysML™) Version 1.3. http://www.omg.org, 2012.

[pap] PAPYRUS. https://eclipse.org/papyrus/.

[PBG14] Rui Pais, João Paulo Barros, and Luís Gomes. From sysml state machines to petri nets using ATL transformations. In Technological Innovation for Collective Awareness Systems - 5th IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2014, Costa de Caparica, Portugal, April 7-9, 2014. Proceedings, pages 227--236, 2014.

[Pro] Graphical Modelling Project. Graphical modelling framework (gmf). http://www.eclipse.org/modeling/gmp/.

[PST07] Pascal Poizat, Gwen Salaün, and Massimo Tivoli. An adaptation-based approach to incrementally build component systems. Electr. Notes Theor. Comput. Sci., 182:155--170, 2007.

[PSTV13] Pekka Pihlanko, Seppo Sierla, Kleanthis Thramboulidis, and Mauri Viitasalo. An industrial evaluation of sysml: The case of a nuclear automation modernization project. In ETFA, pages 1--8, 2013.

[Pto] Ptolemy. Ptolemy Project. http://ptolemy.eecs.berkeley.edu/.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. Software Engineering, IEEE Transactions on, 28(11):1056--1076, Nov 2002.

[RC15] Seidali Rehab and Allaoua Chaoui. Tgg-based process for automating the transformation of uml models towards b specifications. International Journal of Computer Aided Engineering and Technology, 7(3):378--400, 2015.

[RF06] Oscar 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 (CPN 2006), pages 237--56. Citeseer, 2006.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. Unified Modeling Language Reference Manual, The (2Nd Edition). Pearson Higher Education, 2004.

[RKBIH15] Messaoud Rahim, Ahmed Kheldoun, Malika Boukala-Ioualalen, and Ahmed Hammad. Recursive ecatnets-based approach for formally verifying system modelling language activity diagrams. Software, IET, 9(5):119--128, 2015.

[Sam76] JeffreyR. Sampson. Theorem proving. In Adaptive Information Processing, Texts and Monographs in Computer Science, pages 160--174. Springer Berlin Heidelberg, 1976.

[Sch04] Klaus Schneider. Verification of Reactive Systems: Formal Methods and Algorithms. SpringerVerlag, 2004.

[SEG09] R. Seguel, R. Eshuis, and P. Grefen. An overview on protocol adaptors for service component integration, 2009.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. Computer, 30(4):110--111, 1997.

[Sun06] Sun Microsystems. Enterprise JavaBeans 3.0 Specification, May 2006.

[sys] SYSML. https://sysml.org/.

[Szy98a] Clemens Szyperski. Component software: beyond object-oriented programming. Pearson Education, 1998.

[Szy98b] Clemens A. Szyperski. Component software - beyond object-oriented programming. Addison-Wesley-Longman, 1998.

[TeS11] Nara Sueina Teixeira and Ricardo Pereira e Silva. Compatibility evaluation of components specified in UML. In 30th International Conference of the Chilean Computer Science Society, SCCC 2011, Curico, Chile, November 9-11, 2011, pages 90--99, 2011.

[TI08] Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. Science of Computer Programming, 71(3):181 -- 212, 2008.

[too] CPN tool. http://cpntools.org/.

[top] TOPCASED. https://www.polarsys.org/topcased.

[Vas09] Parham Vasaiely. Interactive simulation of sysml models using modelica. Bachelor Thesis, Dept of Computer Science, Hamburg University of Applied Sciences, 2009.

[wik] WIKIPEDIA. https://fr.wikipedia.org/.

[WS01] Rainer Weinreich and Johannes Sametinger. Component-based software engineering. chapter Component Models and Component Services: Concepts and Principles, pages 33--48. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[XB03] Fei Xie and James C Browne. Verified systems by composition from verified components. In ACM SIGSOFT Software Engineering Notes, volume 28, pages 277--286. ACM, 2003.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst., 19(2):292--333, March 1997.

[YTL04] Nesrine Yahiaoui, Bruno Traverson, and Nicole Levy. Classification and comparison of adaptable platforms. In Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04), Oslo , Norway, pages 55--61, 2004.

Document generated with LaTeX 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 takes place in the component-based development domain, it is a contribution to the specification, adaptation and verification of component-based systems. The main purpose of this thesis is the proposition of a formal approach to build incrementally complex systems by assembling and adapting a set of components, where their structure and behaviour are modelled using SysML diagrams. In the first stage, we have defined a meta-model driven approach which is based on meta-modelling and models transformation, to verify the compatibility of blocks having their interaction protocols modelled using SysML sequence diagrams. To verify their compatibility, we perform a transformation into interface automata (IAs), and we base on the optimistic approach defined on IAs. This approach consider that two components are compatible if there is a suitable environment with which they can interact correctly. After that, we have proposed to benefit from the hierarchy, that may be present in the interaction protocol models of the blocks, to alleviate the verification of blocks compatibility. In the next stage, we have taken into consideration the problem of names mismatches of type one2one between services of blocks. At this stage, an adapter is generated for a set of reused blocks which have their interaction protocols modelled formally by interface automata. The generation of the adapter is guided by the specification of the parent block which is made initially by the designer. Our approach is completed by a verification phase which allows us to verify SysML requirements, expressed formally by temporal properties, on SySML blocks. In this phase, we have exploited only the generated adapters to verify the preservation of the requirements initially satisfied by the reused blocks. Thus, our approach intends to give more chance to avoid the state space explosion problem during the verification. In the same context, where we have a set of reused blocks and the specification of their parent blocks, we have proposed to use coloured Petri nets (CPNs) to model the blocks interactions and to generate adapters that solve more type of problems. In this case the adapter can solve the problem of livelock by enabling the reordering of services calls.

Keywords: component-based development, SysML, adaptation, verification, Compatibility, interface automata, coloured Petri nets, Requirements

Résumé :

Le travail présenté dans cette thèse a lieu dans le domaine de développement basé sur les composants, il est une contribution à la spécification, l'adaptation et la vérification des systèmes à base de composants. Le but principal de cette thèse est la proposition d'une approche formelle pour construire progressivement des systèmes complexes en assemblant et en adaptant un ensemble de composants, où leur structure et leur comportement sont modélisés à l'aide de diagrammes SysML. Dans la première étape, nous avons défini une approche basée sur la méta-modélisation et la transformation des modèles pour vérifier la compatibilité des blocs ayant leurs protocoles d'interaction modélisés à l'aide de diagrammes de séquence SysML. Pour vérifier leur compatibilité, nous effectuons une transformation en automates d'interface (IAs), et nous utilisons l'approche optimiste définie sur les IA. Cette approche considère que deux composants sont compatibles s'il existe un environnement approprié avec lequel ils peuvent interagir correctement. Après cela, nous avons proposé de bénéficier de la hiérarchie, qui peut être présente dans les modèles de protocole d'interaction des blocs, pour alléger la vérification de la compatibilité des blocs. Dans l'étape suivante, nous avons pris en considération le problème des incohérences de noms de type one2one entre les services des blocs. A ce stade, un adaptateur est généré pour un ensemble de blocs réutilisés qui ont leurs protocoles d'interaction modélisés formellement par des automates d'interface. La génération de l'adaptateur est guidée par la spécification du bloc parent qui est faite initialement par le concepteur. Notre approche est complétée par une phase de vérification qui nous permet de vérifier les exigences SysML, exprimées formellement par les propriétés temporelles, sur les blocs SySML. Dans cette phase, nous avons exploité uniquement les adaptateurs générés pour vérifier la préservation des exigences initialement satisfaites par les blocs réutilisés. Ainsi, notre approche a l'intention de donner plus de chance d'éviter le problème de l'explosion de l'espace d'état au moment de la vérification. Dans le même contexte, où nous avons un ensemble de blocs réutilisés et la spécification de leurs blocs parents, nous avons proposé d'utiliser des réseaux de Petri colorés (CPN) pour modéliser les interactions des blocs et générer des adaptateurs qui résolvent plus de types de problèmes. Dans ce cas, l'adaptateur peut résoudre le problème de blocage en permettant le réordonnancement des appels de services.

Mots-clés : développement basé sur les composants, SysML, adaptation, vérification, compatibilité, automates d'interface, réseaux de Petri colorés, exigences

■ 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