UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

THÈSE

pour l'obtention du titre de

Docteur en Sciences

Mention Informatique

présentée et soutenue par

Oleksandra Kulankhina

A framework for rigorous development of distributed components: formalisation and tools

Thèse dirigée par Eric Madelaine
et co-encadrée par Ludovic Henrio

Soutenue le 14 Octobre 2016

Jury

Rapporteurs	Radu Mateescu	Inria Grenoble - Rhône-Alpes
	František Plášil	Charles University, Prague
Examinateurs	Frédéric Mallet	Université Nice Sophia Antipolis
	Simon Bliudze	EPFL
Directeur de thèse	Eric Madelaine	Inria Sophia Antipolis
Co-directeur de thèse	Ludovic Henrio	CNRS
Invité	Rabéa Ameur-Boulifa	Télécom ParisTech

…

Résumé

Dans cette thèse, nous proposons une approche rigoureuse pour la conception et le développement de systèmes à base de composants hiérarchiques distribués. L'idée de base du travail présenté est de combiner les techniques de conception de logiciels dirigées par les modèles, bien connues des programmeurs, avec des méthodes de vérification formelles puissantes, capables d'assurer les propriétés fonctionnelles d'un système distribué et de détecter les erreurs dès le stade de la conception.

Tout d'abord, nous introduisons un formalisme graphique basé sur UML pour l'architecture et le comportement des composants hiérarchiques de modélisation. Deuxièmement, nous spécifions formellement un ensemble de contraintes qui assurent la correction de la composition des composants, en mettant l'accent sur la séparation entre les aspects fonctionnels et non-fonctionnels. Troisièmement, nous expliquons comment nos modèles graphiques peuvent être traduits automatiquement dans le formalisme d'entrée d'un model-checker. Nous nous concentrons ensuite sur le codage des fonctionnalités avancées de composants distribués, comme communications de 1 vers N, la reconfiguration et les communications asynchrones basées sur les appel de procédures distants.

Enfin, nous mettons en œuvre cette approche dans une plateforme intégrée orienté modèle qui comprend un ensemble d'éditeurs graphiques, un module de validation de la décision correcte de l'architecture statique, un module traduisant le modèle conceptuel dans une entrée pour la plateforme de vérification CADP, et enfin un générateur de code exécutable

Abstract

In this thesis we introduce an approach for rigorous design and development of distributed hierarchical component-based systems. The core idea of the presented work is to combine the well-known among the programmers techniques for model-driven software design and the powerful formal verification methods able to ensure the functional properties of a distributed system and to detect errors at the early design stage.

First, we introduce a UML-based graphical formalism for modelling architecture and behaviour of hierarchical components. Second, we formally specify a set of constraints that ensure the correct components composition with a focus on separation between the functional and non-functional aspects. Third, we explain how the graphical models can be automatically translated into an input for a model-checker. For this aim, we rely on a formally specified intermediate structure encoding the semantics of components behaviour as a network of synchronised parametrised label transition systems. We focus here on encoding the advanced features of distributed components such as one-to-many communications, reconfiguration, and asynchronous communications based on request-reply.

Finally, we implement the approach in an integrated model-driven environment which comprises a set of graphical editors, an architecture static correctness validation plug-in, a plug-in translating the conceptual model into an input for a verification toolsuite CADP, and a generator of the implementation code.

Acknowledgements

Table of Contents

List of Figures **xiii**

List of Listings **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation and objectives 1

1.2 Contribution 7

1.3 Outline 10

2 Context **13**

2.1 The Grid Component Model 14

2.1.1 GCM overview 14

2.1.2 GCM/ADL 17

2.1.3 GCM/ProActive 18

2.2 Parameterised networks of synchronised automata 24

2.2.1 Term algebra and notations 24

2.2.2 The pNets model 25

2.2.3 Observation and flow of information 27

2.2.4 Adequacy of pNets for modelling GCM components 28

2.3 CADP 29

2.4 The Fiacre specification language 32

2.5 Model-Driven Engineering 33

2.5.1 Unified Modelling Language 34

2.5.2 Eclipse Modeling Framework 35

2.5.3 Obeo Designer 36

2.6 VerCors 37

3 An overview of the VerCors platform 41
3.1 The core functionalities of VerCors 41
3.2 Diagrams for architecture and behaviour specification 44
3.2.1 An illustrative example 44
3.2.2 Architecture specification 45
3.2.3 Behaviour specification 48
3.3 The architecture of VerCors 51
3.4 Discussion 55

4 Well-formed component architecture 59
4.1 Formalisation of component structure 60
4.2 Auxiliary functions 61
4.3 Interceptors 63
4.4 Well-formed component architecture 65
4.4.1 Core 65
4.4.2 Non-functional aspects 68
4.4.3 Collective communications 70
4.4.4 Additional rules 71
4.5 Properties 71
4.6 Architecture static analysis in VerCors 74
4.7 Discussion and Related work 74

5 Verification and execution of distributed components 79
5.1 From application design to pNets 80
5.1.1 Semantics of primitive components 81
5.1.2 Semantics of composite components 92
5.1.3 Implementation 100
5.2 From pNets to CADP 108
5.2.1 Preparing the input: generating Fiacre, EXP and auxiliary scripts 108
5.2.2 Model-checking with CADP 112
5.3 Code generation and execution 115
5.3.1 ADL generation 116
5.3.2 Java generation 118
5.3.3 Code execution 123
5.4 Discussion 124
5.4.1 On the verification 124
5.4.2 On the executable code generation 125

6 Advanced features **127**
6.1 Non-functional components and interceptors . . . 129
6.1.1 From application design to pNets . . . 129
6.1.2 Implementing pNet generation and integration with CADP . . 134
6.1.3 Code generation . . . 134
6.2 Component attributes and attribute controllers . . . 135
6.2.1 Graphical specification . . . 136
6.2.2 From application design to pNets . . . 137
6.2.3 Implementing pNet generation and integration with CADP . . 138
6.2.4 Code generation . . . 139
6.3 Reconfigurable multicast interfaces . . . 140
6.3.1 Graphical specification . . . 141
6.3.2 From application design to pNets . . . 142
6.3.3 Implementing pNet generation and integration with CADP . . 155
6.3.4 Code generation . . . 157
6.4 Reconfiguring multicasts from NF components . . . 157
6.4.1 Graphical specification . . . 157
6.4.2 From application design to pNets . . . 158
6.4.3 Implementing pNet generation and integration with CADP . . 158
6.4.4 Code generation . . . 159
6.5 Examples . . . 161
6.5.1 Composite pattern . . . 161
6.5.2 Springoo . . . 171
6.6 Discussion . . . 173

7 Related work **177**
7.1 The SOFA 2 project . . . 178
7.2 The BIP Component Framework . . . 181
7.3 Rebeca formal modelling language and development tools . . . 183
7.4 ABS . . . 188
7.5 Other frameworks . . . 191
7.5.1 Component models and tools . . . 192
7.5.2 Verification platforms . . . 196
7.6 Summary . . . 200
7.6.1 On the verification tools . . . 200
7.6.2 On the component development frameworks . . . 200

8 Conclusion **203**

8.1 Summary 203

8.2 Perspectives 206

8.2.1 Modelling and analysis of parameterised architectures 206

8.2.2 Modelling and analysis of multi-threaded components 207

8.2.3 Modelling and analysis of reconfigurable systems 208

8.2.4 Extending the pNet generator 210

8.2.5 Properties specification and visualising the results of model-checking 211

8.2.6 Static analysis and type-checking of state machines 211

8.2.7 Other ideas of the future work 213

List of Figures

2.1 A GCM application 15
2.2 Request-reply by futures 19
2.3 Request treatment by GCM/ProActive components 20
2.4 UML class diagram 35
2.5 UML state machine diagram 35
2.6 EMF example 36

3.1 VerCors workflow 42
3.2 Screenshot of VerCors 43
3.3 VerCors component diagram 46
3.4 A component diagram of Peterson's leader election use-case example . 48
3.5 VerCors class diagram 49
3.6 State machine diagram 51
3.7 Scenario state machine 51
3.8 Architecture of VerCors 52

4.1 Internal interfaces of a membrane 63
4.2 An input chain of interceptors 64
4.3 Examples of architecture constraint violations 73
4.4 Architecture static correctness validation in VerCors 75

5.1 An example of a primitive component 82
5.2 pNet for the PrimExample component from Figure 5.1 83
5.3 Graphical representation of the behaviour of the Body 88
5.4 pLTSs for the Future Proxies and Proxy Managers 89
5.5 A state machine and its translation to a pLTS 91
5.6 An example of a composite component 93
5.7 pNet for the composite component from Figure 5.6 94
5.8 Auxiliary processes proxy and delegate of composite components . . . 96
5.9 pNets meta-model (simplified) 102

5.10 Construction of a pNet of a primitive component 105
5.11 Construction of a pNet of a composite component 107
5.12 Fiacre code of a body 109
5.13 The workflow of implementation code generation 116
5.14 A primitive with an attached UML class 121
5.15 A simple state machine 122
5.16 Code execution 123

6.1 Bindings in a membrane 130
6.2 pNet of a component with a componentised membrane 131
6.3 Graphical specification of a component attribute 136
6.4 An attribute controller pLTS 137
6.5 A primitive component with a multicast Interface 141
6.6 A composite component with multicast internal and external interfaces 141
6.7 A Group Manager 143
6.8 A Group Proxy for a method of a primitive component 144
6.9 pNet model for Figure 6.5 144
6.10 Dynamic Connector for a Multicast Interface 145
6.11 The Proxy of a multicast interface inside a composite component . . 149
6.12 pNets for the Composite component with multicast internal and external interfaces 150
6.13 A multicast of a sub-component sends an external request 154
6.14 A Group Manager for a void method 155
6.15 Modelling binding reconfiguration 158
6.16 VerCors model of the composite pattern 163
6.17 The class diagram of the composite pattern 164
6.18 addSubcomp method 165
6.19 addAnyUnbound method 166
6.20 Scenario for the composite pattern application 167
6.21 Springoo application modelled in VerCors 172

7.1 The BIP design flow [2] 181
7.2 Verification workflow of KeY-ABS[117] 190

List of Listings

2.1 ADL example 18
2.2 A Java class of a GCM/ProActive primitive 22
2.3 A GCM/ProActive component construction and access 23
2.4 An example of synchronization vectors in .exp 30
2.5 A Fiacre process 33

5.1 An example of SVL script 111
5.2 An example of a JAXB-based class 116
5.3 An XML file generated by JAXB 116
5.4 An Acceleo template translating VerCors record type into a Java class 118
5.5 Java code of a record type generated by VerCors 119
5.6 Generated Java code of a primitive component 120
5.7 Generated Java code of a state machine 122

6.1 Generated ADL file of a composite with a componentised membrane . 135
6.2 A Java class implementing the behaviour of a primitive component with an attribute 139
6.3 An ADL specification of a component attribute 139
6.4 An ADL attribute which stores reconfigurable interfaces 160
6.5 Java code of a component-controller 160

List of Tables

4.1 The formalization of GCM architecture 61
4.2 Auxiliary functions 62
4.3 Interceptor predicates 66
4.4 Core predicates 66
4.5 Non-functional predicates 68

5.2 Server and client-side synchronisation vectors for primitive components 85
5.3 Server and client-side synchronisation vectors of a pNet of a composite component 97
5.4 Binding synchronisation vectors of a pNet of a composite component 98
5.5 Behaviour graph files (all with Queue size of 3) 112

6.1 Attribute controller synchronisation vectors for primitive components 138
6.2 Synchronisation vectors for multicast client interfaces in primitive components 147
6.3 Synchronisation vectors for multicast interfaces in composite components 151
6.4 Binding synchronisation vectors for multicast interface 153

7.1 Verification tools 201

Chapter 1

Introduction

Contents

1.1 Motivation and objectives 1

1.2 Contribution 7

1.3 Outline 10

1.1 Motivation and objectives

In recent years the amount of data to be processed has grown exponentially, presenting new challenges to the software developers and scientists. It is often too large to be processed on a single machine. Distributed computing is the approach able to support the efficiency of large applications operating on big data. According to it, a program can be split into several interacting parts which are executed on different computational nodes. Programming such systems is a difficult task because the developer has to ensure not only the correctness of the behaviour of each individual module but also the correctness and consistency of their composition. The collaboration of several processes distributed over multiple machines and the synchronisation between them make the computational logic more complex. In this thesis we target checking the correctness of the computational logic of distributed applications.

Programming distributed components

A popular approach for the development of large-scale distributed applications is the component-oriented programming where a software system is split into separate modules (components) with well-defined interfaces which they use for interacting with each other. The approach enforces a clear design of the applications and provides

a solid basis for safe and modular development of complex systems. There exists a variety of component models [1, 2, 3] defining how an application should be designed, implemented, and deployed. They often use different vocabularies but in general, all of them rely the notions of *components*, *interfaces* (sometimes called *ports*), and *bindings* (sometimes called *connectors*). A component can be seen as a block of a software which provides some functionality. Components use interfaces as the communication points to expose their services and to access the services of each other. The bindings are used to establish the communications between the interfaces. One of the advantages of the component-oriented approach is the re-usability of components: when the developer writes a new program, he can re-use some existing components as he knows statically their provided and required functionalities. In addition to the flat composition, some models allow for the development of hierarchical systems, i.e. a component can "wrap" other components. In this case, the former is called a *parent* component or a *container*, and the latter are its *sub-components*. Such an approach allows the programmer to hide the complexity of the internal implementation of a part of the system.

Even with the help of the component-oriented programming, the development of large-scale distributed applications is challenging for three main reasons. First, such software systems often rely on asynchronous requests. This means that when a component sends a request to another component, the sender does not have to block its execution waiting for the reply. As a result, two components can execute their services in parallel. This increases the efficiency of the application because serving requests in parallel can be much faster than the sequential processing and because the computational resources of the sender do not stay idle while waiting for the result of a remote method invocation. On the other hand, the asynchrony makes the development of distributed systems more complicated. The reason is that the behaviour of such components is not easy to predict at the programming stage as it is impossible to know when exactly the result of a remote computation will arrive and when it can be used.

Another challenge is presented by the evolution of a distributed system at runtime: in order to adapt to the current task or to the changes in the environment, an application often needs to be reconfigured during its execution. This may include, for instance, adding or removing components depending on the system workload. The programmer has to take care of all possible configurations of a software application and to make sure that for each of them the system will behave correctly. Managing a reconfigurable system becomes even more complex in the case of hierarchical applications because the changes applied to a parent component can often affect its

content.

Finally, when programming a distributed system, it is not always easy to keep separated the functional and non-functional aspects while allowing them to communicate. The former is responsible for the business logic of an application: it defines how the system behaves within the given problem domain. The latter controls the application: it measures the necessary performance metrics that are often based on the functional behaviour, plans and executes the reconfiguration. It takes care of the security aspect and the other aspects not related to the application logic. The non-functional part should not depend on the concrete system domain: whenever the software is overloaded, a new computational node should be added no matter whether it processes bank transfers, multiplies matrices, or renders a game graphics. Implementing separately the functional and non-functional parts is important for the safety and re-usability of the software components; it allows for clear definition of the objective of each part. The fact that components have well-defined provided and required interfaces, makes programming systems with strong separation of concerns easier. Moreover, sometimes the separation of concerns is enforced by a component model: some component models define functional and non-functional components and interfaces. The issue is that the two different parts of an application are often influenced by each other and often have to interact with each other, and the developer has to program these communicating parts so that they are still clearly separated.

We have defined a set of challenges that the developer of a distributed application has to face. We can see that there is a need to help the programmer to address them by providing techniques and tools which can assist in the design, analysis, and implementation of distributed system.

The Grid Component Model

Among all the existing component models, we focus on the Grid Component Model (GCM) [4] because its has the following features.

First, it allows for specifying *distributed hierarchical* components: at the leaves of hierarchy it has so-called primitive components which encapsulate some business code and represent the units of distribution. Then components are assembled hierarchically using compote components.

Second, the reference implementation of GCM provided by the GCM/ProActive [5] middleware allows for programming *loosely-coupled* components that communicate only via *asynchronous requests* with futures. More precisely, whenever a component sends a remote request, it creates a future object which is a placeholder for the reply. As opposed to the synchronous communications, the sender continues

its execution as long as it does not require the result of the remote method invocation. When the result is needed, the sender either uses the value that was received by the future object, or waits till it is computed. Such communications are still asynchronous in the sense that the requester does not get blocked immediately after a remote method call, but they are much easier to control than the fully asynchronous message-passing. The communications based on futures increase the level of parallelism as the sender can continue its execution while its remote request is being processed by another component. The absence of the shared memory makes components loosely-coupled: each component has its own local memory and thus only it is responsible for its own state and execution. This makes the model well-adapted to the distributed setting. Moreover, the usage of futures is transparent in GCM/ProActive: the programmer does not need to use any specific instructions for manipulating the futures.

The third advantage of GCM is its *reconfiguration capabilities*. A GCM component can be added or removed, started or stopped, and the bindings between components can be modified at run-time.

Another strong point of GCM is that it enforces *separation of concerns*: it defines the notions of functional and non-functional interfaces and functional and non-functional components which can have hierarchical structure. In addition, GCM provides techniques for modelling not only one-to-one but also *one-to-many* and *many-to-one* communication styles which are widely used in the distributed systems.

Model-driven software engineering

A number of techniques which leverage the documentation, the development, and the maintainance of the large complex software are provided by the model-driven-engineering [6] approach which has become a de-facto standard for the development of industrial projects. In particular, it allows the programmer to design a model of the future application before writing its code, and thus, to plan in advance the structure and the behaviour of each component involved in the system. There exist dozens of textual and graphical notations developed in the industry and academia for the specification of the application design from various viewpoints. One of the most popular graphical languages for obejct and component-oriented systems is the Unified Modelling Language (UML) [7]. It allows for designing a software application as a set of diagrams that describe its architecture, behaviour, the interactions with the user, etc. The diagrams can be used not only for documenting the project but also as an input for the code generation. Indeed, there exist a number of tools [8, 9, 10] which partially translate a software model into executable code so that the programmer

does not need to write it from scratch. In addition, the diagrams can be statically analysed to check the absence of errors that could occur in the implementation code. The earlier an error is detected, the lower the cost of its correction. Model-driven engineering became popular in the industry also because it facilitates the maintenance of large applications. Whenever a new functionality has to be added, instead of modifying directly the implementation code, the developer can, first, introduce it in the design of the application in order to see its impact on the rest of the system.

Formal methods

A set of powerful techniques for the specification and analysis of complex software is based on formal methods. They allow one to model a system as a composition of mathematical entities and to prove certain properties on it. There exist a number of approaches [11] that differ in the underlying mathematical model, in the level of automation (some techniques are fully automatic, the others require guidance from the user), and in the addressed aspects of the input model (e.g. the timed behaviour, the safe composition of components, the interactions between concurrent processes). One technique for fully automatic and exhaustive analysis of a system behaviour is provided by *model-checking*. It relies on building a model of an application behaviour and on exploring its state-space in order to verify the formula which models the desired property of the input model. It can be, for instance, the reachability of a particular behaviour, or the absence of a deadlock. One of the advantages of model-checking is that it performs the exhaustive analysis of the input model which allows identifying the erroneous "rare" scenarios that are not always covered by the software tests. The success of the approach is highlighted by its application to the large-scale projects developed by the leading modern companies and institutes such as for instance, the web-services of Amazon [12], the spacecraft controllers of NASA [13], and the flight control systems of Airbus [14].

However, model-checking can be only applied to an abstraction of the real system, encoded in a finite state manner. Such an abstraction can be obtained by analysis of the source code, but this can be costly. We prefer to associate the formal methods with the model-driven engineering approach: starting from a high-level model of an application, we can generate an abstract state-space for model-checking and some executable code. If needed, the generated code can be refined to get the final detailed implementation.

Formal methods allow one to detect a huge variety of errors in a software system at the design stage but their use in the industry is still very low. There are two key reasons for that. First, the application of such techniques can be enormously costly:

for instance, sometimes the generated state-space of a model-checked system is so huge that it is just impossible to verify it exhaustively. To address this challenge, the formal method community is working on the techniques for the state-space reduction such as partial order reduction [15], symmetry reduction [16], etc. Another reason why the formal methods are not widely used by the software engineers is the complexity of their practical usage. Mastering formal methods often requires significant background in mathematics and professional trainings. This second issue is addressed in this dissertation.

Objectives and positioning

This work aims at including systematic verification of behavioural properties in the industrial development process of component-based distributed applications. For this purpose we want to provide the developers of distributed component-based systems with a set of model-driven tools supporting rigorous design and implementation of safe applications. Our tools should guide the user through all crucial phases of component software development: from application design specification to verification of the modelled architecture and of the properties of its behaviour as well as automated code generation. More precisely, we want to:

- design a user-friendly language for the specification of hierarchical asynchronous component-based systems with reconfiguration capabilities;
- help the developers to ensure formally that the application design is statically correct and that the functional components of the modelled system are properly separated from the non-functional ones;
- develop an approach to automatically translate the user-defined specification into an input for a powerful model-checker in order to verify the properties of the modelled application;
- develop an approach to automatically translate the designed application into executable code;
- integrate all these techniques into a single framework for modelling, verification, and code generation for distributed component-based systems.

GCM features a complex programming model, and includes many advanced mechanisms. Its operational semantics has been well-defined and formalised in previous works of the Oasis team [17], and the middleware implementation respects the model

and the semantics. However, we need a behavioural semantics (not the classical operational one) in order to allow for model-checking temporal properties. The challenge here is to define this semantics in a way that respects closely the semantics of GCM/ProActive components and the middleware implementation, ensuring that the properties proven by the model-checker will be respected by the generated code.

In this thesis we target modelling and verification of distributed hierarchical component-based systems with strong separation between functional and non-functional concerns, reconfiguration capabilities, and communications based on futures. The originality of this work lies in the combination of all these features. In fact, there exist a number of component models supported by development platforms that feature some of these elements. For instance, the BIP [2] component model allows specification of hierarchical asynchronous systems but does not provide the reconfiguration capabilities. The components of Rebeca [18] are asynchronous and highly dynamical but not hierarchical. The components of SOFA 2 [1] provide all the targeted features except that the non-functional part of a component can be only flat and the future-based communications are not supported. We present a deeper overview and comparison of the component models and tools related to the work presented in this thesis in Chapter 7.

1.2 Contribution

Overall, this work aims at integrating in a single framework techniques for modelling, analysis, and generation of hierarchical distributed component-based systems with asynchronous communications and reconfiguration capabilities. We would like to provide software developers with a single platform where one can model a distributed application in a user-friendly and easy-to-learn graphical language, analyse the conceptual model by applying powerful formal verification techniques, and generate the implementation code. We try to automatise as much as possible the analysis of conceptual models and code construction because we would like our framework to be used by non-experts in formal methods. We describe below the main contributions of this thesis.

Graphical specification language. We introduce a graphical language for the specification of the architecture and behaviour of component-based distributed systems. The language allows expressing complex hierarchical structures comprising both business logic components (functional components) and components responsible for the control and management of an application (non-functional components).

At the same time, the specifications of the two aspects are graphically separated from each other. The notations of our language extensively reuse UML elements [7] which are well-known among programmers. This makes the graphical formalism user-friendly and easy-to-learn. Several formalisms for GCM components have already been discussed in [19, 20], but none of them has been properly integrated in a software platform. Based on the previous works, in this thesis we present the first version of the specification approach for coherent and complete definition of GCM component architecture and behaviour. It integrates a domain-specific language designed for GCM components with the UML meta-model.

Formalisation of component architecture and static correctness rules. We provide a formal model for component architectures including a flexible set of constructs for the definition of the non-functional part of an application which has never been formalised before. The business logic and control parts of a modelled system are strongly separated. Based on the architecture formalisation, we define a number of predicates insuring the static correctness of the component composition. After validation of these rules, we guarantee that the application possesses a number of properties which are necessary prerequisite for the correct execution and the reconfigurability of the system. Those properties include uniqueness of naming, separation of concerns, and communication determinacy. The properties will be also crucial during the analysis phase and the executable code generation.

Generation of behavioural models. We formalise a set of semantic rules for the transformation of the designed conceptual models into a semantic formalism allowing us to specify the details of the application behaviour and communications between components. The behavioural models are generated in terms of parameterised networks of synchronised automata [21]. They encode future-based communications, hierarchical components, some aspects of architecture reconfiguration, and one-to-many communications. We also show how the generated structures can be automatically transformed into an input for a model-checker (we use the model-checker of CADP [22]) in order to verify the requirements of the designed application expressed as logical properties. Prior to this work, the approach presented in this thesis has been partially manually tested on several use-cases [23, 24]. However, this is the first work where the fully automatic translation of the graphical models into an input for a model checker is implemented as a software. The fully automatised process has also been tested on several use-case examples, and some of them are included in this dissertation.

Generation of executable code. Once the conceptual model has been proven statically correct, and its functional properties have been model-checked, we generate the executable code of the designed application. The produced code includes an XML-based file encoding the system architecture and a set of Java classes implementing the behaviour of the components. This way, we are able to run distributed applications that are proven safe. In this thesis we will explain how the generation process is organised and demonstrate several experiments of running an application produced by our framework.

Integration of graphical designer, model-checker, and execution platform. Finally, we implement and integrate the front-end graphical designer with a model-checker, and an execution middleware in a single model-driven Eclipse-based framework called VerCors. Using the front-end editor, the user can design the conceptual model of his application in our graphical specification language. Then, he can check that the defined architecture is statically correct with respect to the properties we formalised. Next, VerCors fully automatically generates a behavioural model of the designed application and transforms it into an input for a model-checker. As we will show in this thesis, the model-checker can verify a number of properties on the produced structure. This includes both generic properties (e.g. absence of deadlocks) and application-specific properties (e.g. reachability of a particular system state, inevitability, etc). This work does not focus on the assistance to property specification but we provide a number of examples of properties verified for our use-case models. Finally, the platform automatically produces the implementation code of the designed system which can be executed in a distributed environment[1].

The work presented in this thesis is included in the following publications[2]:

- Ludovic Henrio, Oleksandra Kulankhina, Dongqian Liu, Eric Madelaine; "Verifying the correct composition of distributed components: Formalisation and Tool" FOCLASA, Sep 2014, Rome, Italy. The paper presents the formalisation of component architecture and static correctness properties.

- Tatiana Aubonnet, Ludovic Henrio, Soumia Kessal, Oleksandra Kulankhina, Frédéric Lemoine, Eric Madelaine, Cristian Ruz, Noëmie Simoni; "Management

[1] I encoded most of the functionalities of the VerCors platform; I coordinated the work of the students and engineers who helped me with several specific tasks; Regarding the re-used code, I integrated an existing UML editor into the front-end of VerCors; I integrated and enhanced a part of a generator of the executable code (about 40%) from the previous versions of the platform.

[2] I am the main author of the publications at FOCLASA'14 and FASE'16. All the authors have equal contribution to the paper in the JISA journal. Regarding the paper submitted to the JLamp journal, I contributed to the correction and refinement of the presented formalisation, I implemented the software constructing the behavioural models according to the formalised rules.

of service composition based on self-controlled components"; Journal of Internet Services and Applications, Springer, 2015, 6 (15), pp.17. In this paper we use the VerCors platform in order to model and generate a service-oriented distributed application.

- Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, Eric Madelaine; "Integrated environment for verifying and running distributed components"; Fundamental Approaches to Software Engineering (FASE), Apr 2016, Eindhoven, Netherlands; Springer, Lecture Notes in Computer Science, 9633, pp.66-83, 2016, Fundamental Approaches to Software Engineering. The paper provides a general overview of the graphical formalism, behaviour generation, and executable code generation processes presented in this thesis.

- Rabéa Ameur-Boulifa, Ludovic Henrio, Oleksandra Kulankhina, Eric Madelaine, Alexandra Savu; "Behavioural Semantics for Asynchronous Components" (submitted to JLamp). The paper introduces detailed formalisation of the generation of the behavioural models for GCM components.

1.3 Outline

The thesis is organised as follows:

Chapter 2 presents the technical background this work is based on. We describe the main component model we rely on: GCM (Grid Component Model) with its reference implementation in the GCM/ProActive middleware. We introduce the formalism which we use for encoding component behaviour and the verification platform which we use for model-checking. Then, we introduce the basic notions of model-driven engineering and the tools we used for the implementation of the VerCors platform. Finally, we make a short overview of the history of VerCors and discuss what are the contributions of this thesis with respect to the previous versions of the platform.

Chapter 3 makes an overview of the version of VerCors implemented in this thesis. We discuss its core functionalities from the user point of view, the graphical formalisms of the front-end designer, and the implementation architecture.

Chapter 4 presents the formalisation of component architecture and static correctness predicates. We introduce the notion of component well-formedness and a set

of properties insured by the validation of the predicates. We also explain how the static correctness check was implemented in VerCors. Finally, we discuss applicability of the provided architecture formalisation and predicates to the component models other than GCM, and their relation to the similar previous studies.

Chapter 5 describes the core contribution of this thesis: the semantic rules defining the construction of the behavioural models and the generation of the implementation code. We start by explaining the transformation of the specification of GCM-based architecture and behaviour conceptual models into an intermediate structure encoding system behaviour at a low lever (in terms of networks of parameterised automata). Then, we discuss how the generation process is implemented in VerCors, and how the constructed structures are transformed into an input for the model-checker of CADP. We also present the generation of the executable code of an application designed and verified in VerCors.

Chapter 6 extends the graphical formalism, the generation of the behavioural models and of the executable code with the constructs necessary for modelling, verifying, and running component-based systems with advanced features. They include the non-functional components, attribute controllers, and reconfigurable one-to-many communications.

Chapter 7 presents the related works. We discuss the state-of-the-art frameworks for modelling and verification of distributed component-based systems, and we position our work with respect to the other studies. Then, we make an overview of several verification platforms and we explain why we chose the model-checker of CADP for our work.

Chapter 8 provides the conclusion and a discussion on the future work.

Chapter 2

Context

Contents

2.1 The Grid Component Model 14

2.1.1 GCM overview 14

2.1.2 GCM/ADL 17

2.1.3 GCM/ProActive 18

2.2 Parameterised networks of synchronised automata . . . 24

2.2.1 Term algebra and notations 24

2.2.2 The pNets model 25

2.2.3 Observation and flow of information 27

2.2.4 Adequacy of pNets for modelling GCM components 28

2.3 CADP 29

2.4 The Fiacre specification language 32

2.5 Model-Driven Engineering 33

2.5.1 Unified Modelling Language 34

2.5.2 Eclipse Modeling Framework 35

2.5.3 Obeo Designer 36

2.6 VerCors 37

This chapter discusses the background required for reading this thesis. We start by an overview of the component model this work relies on and its reference implementation. Second, we present an intermediate formalism that we use in order to encode the components behaviour at low level. Then, we introduce the verification

toolbox that we apply to model-check the functional properties of the component-based systems. In order to apply the model-checker we rely on an intermediate text language for the behaviour specification; the language is also discussed in this chapter. Next, we introduce the model-driven engineering paradigm which will be used in this thesis for tools implementation. Finally, we make a short overview of the history of the VerCors platform and compare this work to its previous versions.

2.1 The Grid Component Model

The approach presented in this work relies on the Grid Component Model (GCM). GCM was designed by the CoreGrid European network of Excellence [4] as an extension of the Fractal model [25] dedicated to distributed systems, including a set of control capabilities. The framework targets the design, implementation, execution, and deployment of hierarchical reconfigurable large-scale component-based applications. In this section, first, we provide an overview of GCM. Then, we describe an XML-based language for the GCM architecture specification. Finally, we introduce the ProActive middleware providing an implementation of GCM components based on Active Objects.

2.1.1 GCM overview

A GCM-based application consists of components, interfaces and bindings. We describe each of the elements below and illustrate them in Figure 2.1. We do not explain the graphical notations here as they will be presented in details in Section 3.2.

Hierarchical components. There exist two types of components in GCM depending on the level of observation: primitive and composite components, we will call them "primitives" and "composites" correspondingly. A *composite* encompasses inner components which are called *sub-components*. A composite (`Application` in Figure 2.1) is separated into two parts: a *membrane* (the grey part) containing all the subcomponents dealing with the application management and control, and a *content* (the white part) which comprises the subcomponents implementing the business logic. Another component type - *primitive* (`TaskDistributor, Worker1` in Figure 2.1) - could be seen as a black-box view on a component that encapsulates the implementation code and provides some functionality. Each primitive also has a componentised membrane responsible for control and management. A primitive can comprise attributes of primitive type that will be accessible from outside of the component. Each component is characterised by a name and a set of interfaces.

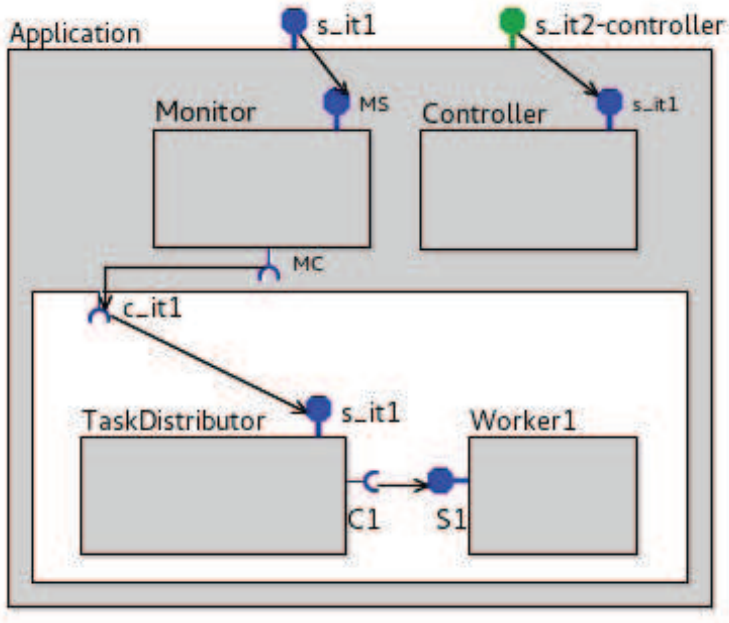


Figure 2.1 – A GCM application

Interfaces. The GCM interfaces are the communication points for the components. The communication between components is performed in the form of method invocation. An interface able to invoke methods and to receive the invocation results is called a *client* interface (e.g. `C1`), and interfaces that accept method invocations and send back the results are called *server* interfaces (e.g. `S1`). The interfaces that call or serve methods implementing the business logic are called *functional* while the ones dealing with the application control are called *non-functional*. The interfaces that communicate are connected by *bindings* (the black arrow from C1 to S1). An interface accessible from outside of a component is said to be *external*. An *internal* interface is reachable only from inside of a component.

Collective communications. In order to facilitate parallel programming, GCM defines the *cardinality* of an interface which can be *singleton*, *multicast* or *gathercast*. A *singleton* is the simplest interface used for one-to-one communications. A client singleton can be connected to only one target interface at each time in order to ensure the deterministic behaviour. However, a singleton server interface can be bound to several client interfaces, but a request from each client is processed separately. A *multicast* provides an abstraction mechanism for the one-to-N communications. It is a client interface that transforms a single method invocation into several requests and sends them to multiple target interfaces at the same time. If the method is supposed to return a result, the replies from the target components are collected in a single

structure and given back to the caller. An abstraction for N-to-one communication is provided by the gathercast interfaces. A *gathercast* is a server interface that can receive calls from several clients at the same time, the calls are assembled in a single request that is processed by the gathercast interface owner.

The separation of concerns. The functional part of a GCM application can be separated from the control part thanks to the separation of a composite component into a membrane and a content and to the usage of functional and non-functional interfaces. This separation ensures, as much as possible, the independence of the business code from the management code.

The non-functional aspect. The GCM was design to support wide range of non-functional capabilities which can be implemented thanks to the three core elements: predefined controllers, non-functional components, and reconfiguration mechanisms.

The GCM specification describes a set of predefined entities used for an application management including:

- A *Lifecycle controller* is included in every component and serves to stop and start a component.
- A *Attribute controller* is used to configure the values of the component attributes.
- A *Binding controller* is used to bind or unbind singleton interfaces.
- *Multicast and gathercast controllers* are used to reconfigure (bind/unbind) multicast and gathercast interfaces.
- *Content and membrane controllers* are used to add subcomponents in a content or a membrane correspondingly.

Additionally, the user can define his custom component-controllers (also referred as non-functional components) and place them in the membrane of the managed component (e.g. `Controller` in Figure 2.1). The component-controllers can be primitives or composites; they have access to the predefined controllers (Lifecycle controller, Binding controller, etc.) and to the host component. The programmer can make a component-controller accessible from outside of the composite through non-functional interfaces, or use interfaces and bindings in order to reach the content subcomponents from the component-controller. In fact, specification of the non-functional part of a composite is as flexible as the design of the business logic.

Thanks to the set of features described above, a GCM application architecture can evolve at runtime; this includes binding reconfiguration, component start and stop, adding or removing subcomponents at different levels of hierarchy. The reconfiguration capabilities provide a mechanism for application adaptation: a system can analyse the current state and change its structure depending on the current needs as demonstrated in [26].

Interceptors. Sometimes, the information should be shared between the functional part and the controllers of the application. Interceptors are specific components inside the membrane that can intercept the flow of functional calls in order to trigger reaction from the non-functional aspects. For example, in the application from Figure 2.1, the `Monitor` component monitors the number of requests sent to the `TaskDistributor` and forwards the information to the `Controller`. Then, the controller can, for example, add more workers to the system if the amount of requests is greater than a given threshold. `Monitor` illustrates very well what is an interceptor. Several interceptors can sequentially intercept the same functional call. The interceptors are discussed in details in Section 4.3.

2.1.2 GCM/ADL

A GCM-based application architecture can be specified in an XML-based format called architecture description language (ADL). Listing 2.1 demonstrates an ADL file of the application depicted on the Figure 2.1. Its root element *definition* represents the root component of the modelled system and in our example corresponds to a composite. The specification of a composite includes the external functional interfaces (line 2), the subcomponents of the content (lines 3-14), the bindings of the content (lines 22-23) and the description of the membrane (lines 15-21) which includes the non-functional part. The definition of a primitive is demonstrated by lines 3-11. It is very similar to a composite except that instead of components nested in a content it has a reference to implementation class (line 6). The specification of a functional interface should include a reference to all interceptors that monitor its calls if there are any. The component attributes reachable from outside should be declared in ADL (lines 7-9). The definition of subcomponents can be alternatively given in a separate ADL file and referenced from the root component. Lines 2, 4-7 include references to Java classes and interfaces which will be detailed in the next section.

```
<definition name="Application">
  <interface name="s_it1" role="server" signature = "ServerInterface" interceptors="Monitor.MS"/>
  <component name="TaskDistributor">
    <interface name="S1" role="server" signature = "ServerInterface"/>
      <interface name="C1" role="client" signature = "ClientInterface"/>
      <content class="TaskDistributorClass"/>
      <attributes signature="TDAttributeController">
          <attribute name="_myId" value="1"/>
      </attributes>
      <controller desc="primitive"/>
  </component>
  <component name="Worker1">
    ...
  </component>
  <controller desc="composite">
    <interface name="s_it2−controller" role="server"...>
    <component name="Controller">
    ...
    </component>
    <binding client="this.s_it2−controller" server="Controller.s_it1"/>
  </controller">
  <binding client="TaskDistributor" server="TaskDistributor.s_it2"/>
  ... other bindings
</definition>
```

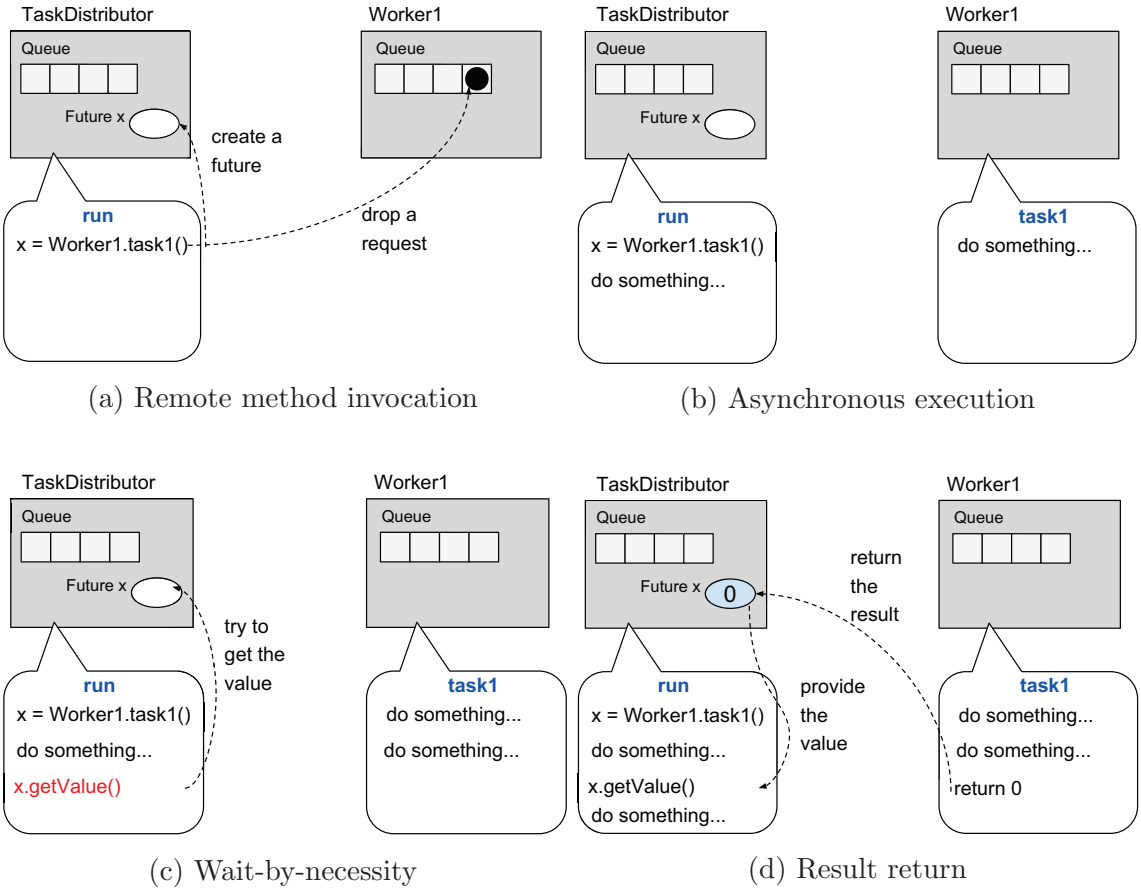
Listing 2.1 – ADL example

2.1.3 GCM/ProActive

The ProActive platform [5] is a Java middleware for programming distributed and concurrent applications. The core notions of this framework are the active objects and communication paradigm based on request-reply by futures [26]; we discuss both of them below. The ProActive platform is important for this work because it provides a reference implementation of GCM which is used in order to run the executable the code generated by VerCors.

Active objects and request-reply by futures. An active object [27] is a unit of distribution in ProActive and represents a normal Java object that additionally has a queue of pending requests, a body that has the ability to decide in which order the requests are served, and a control thread. It is an independent activity that can be addressed remotely and encapsulates its state: only the unique thread of the active object can modify its state. The active objects communicate asynchronously and we describe in details the communication paradigm below.

We rely on an example shown in Figure 2.2 which illustrates the behaviour of two active objects. Here, `TaskDistributor` and `Worker` are active objects but not primitives as in the previous example. Whenever TaskDistributor invokes a method



(a) Remote method invocation (b) Asynchronous execution

(c) Wait-by-necessity (d) Result return

Figure 2.2 – Request-reply by futures

on Worker (Figure 5.5a), a short rendez-vous occurs and the caller gets blocked until the corresponding request is dropped in the queue of the callee. This rendez-vous is used to ensure a causal ordering of requests, The calls between active objects are asynchronous meaning that the requester can continue its execution. If the method is supposed to return a result, the requester creates a so-called *future* object that does not store any value initially but is ready to receive the result of the remote method invocation. The requester continues its execution as long as it does not need the value of the future (Figure 2.2d). Once the value is required, the requester checks whether the result was obtained or not. If the result is available, the caller can use it and proceed the execution. Otherwise, the requester gets blocked waiting for the actual value (Figure 2.2c); such mechanism is called *wait-by-necessity*. If an active object calls one of its own methods (a *local* method), the invocation is synchronous.

In fact, the communications in GCM/ProActive can rely on so-called *first-class futures*. A first-class future is a future object which can be passed as an argument of a remote method call even if its value is not known. In this thesis we do not consider

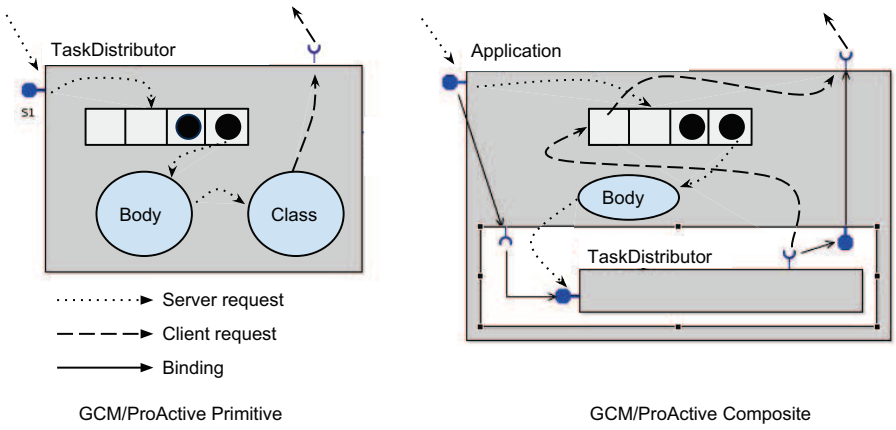


Figure 2.3 – Request treatment by GCM/ProActive components

the first-class futures.

From active objects to components. From the high-level point of view, a GCM/ProActive primitive is implemented as an active object that comprises all the features discussed above. By default, the incoming requests are served in FIFO order, but this can be modified by the programmer. Figure 2.3 illustrates treatment of requests by the GCM/ProActive components. The body of a primitive selects a request from the queue and forwards it to its Java class that implements the corresponding behaviour of the primitive. When the request has been processed, the body selects the next request from the queue. If a primitive invokes a remote method, the call is forwarded from the Java class to the client interface and then to the target component. If the method is supposed to return a result, the corresponding future object is created. A composite is also an active object and it has an implementation class that can be used for configuring some specific parameters, but it does not implement any logic. All requests to the external server interfaces are first dropped in the queue of the composite and then the body forwards them to the subcomponents that are bound to the corresponding server interface. The client requests from the subcomponents to outside of the composite are also dropped in the queue of the composite and then distributed among the corresponding client interfaces.

More technically, in order to wrap an active object into a GCM primitive, the programmer has to decide which methods of the active object can be called externally, or, in terms of GCM, which methods of the primitive will be accessible on the external server interfaces. Those methods are then composed into Java interfaces. Each Java interface will be later associated to possibly multiple GCM interfaces and there is no correlation between their names. Then, the programmer writes a Java

class implementing the behaviour of the primitive. The class is supposed to implement the Java interfaces associated with the GCM server interfaces of the primitive and a set of auxiliary interfaces. An example of a possible implementation of the `TaskDistributor` primitive from Figure 2.1 is given in the Listing 2.2. The class should at minimum include the following definitions:

- the references to the client interfaces (line 7);
- the local variables;
- the attributes that will be exposed by the Attribute Controllers (line 5);
- the methods to access the attributes (lines 35-36);
- the methods that are exposed on the server interfaces (lines 10-15);
- the auxiliary methods dealing with the client interfaces and bindings (lines 17-32);
- the local methods

Multi-threaded active objects and primitive components. Significant work has been done on the theoretical and practical aspects of the multi-threaded active objects [28, 29] which are already implemented in ProActive and can serve as the basis for the multi-threaded GCM primitives. The work presented in this thesis supports only single-threaded primitives, but specification and analysis of the multi-threaded primitives is kept for the future work.

Constructing and using components. There are two ways to construct a GCM application in ProActive: either manually or using a factory. In the first case, the programmer invokes the GCM/ProActive API to create components with all necessary characteristics, to add subcomponents in the composites and to connect interfaces with bindings. This approach is not applied in this thesis. Instead, we rely on the second method where an application is constructed by a dedicated *factory*. The factory takes an ADL file with the system architecture, a set of Java classes implementing primitives' behaviour, a set of Java interfaces declaring the methods of the GCM interfaces and creates automatically all necessary elements including components at different levels of hierarchy controllers and bindings.

When we explained the structure of an ADL file, we did not mention how it is linked to the actual implementation because this is related to the underlying middleware. In the case of a GCM/ProActive ADL (example in the Listing 2.1), the

```
1 public class TaskDistributorClass implements  Serializable, BindingController,
2  TDAttributeController, ServerInterface{
3
4   //an attribute accessible from outside
5   public int myId;
6   //client interfaces
7   public ClientInterface C1;
8
9   //methods of the server interfaces
10  public int run() {
11    IntWrapper x = C1.task1();
12    //do something
13    int y = x.getValue();
14    return y;
15  }
16
17  //auxiliary methods for the management of bindings and interfaces
18  public void bindFc(String myClientItf, Object serverItf) {
19    switch(myClientItf) {
20    case "C1": C1 = serverItf;
21  ...}
22  public String[] listFc() {
23    return new String[1]{"C1"};
24  ...}
25  public Object lookupFc(String myClientItf) {
26    switch(myClientItf) {
27    case "C1": return C1;
28  ...}
29  public void unbindFc(String myClientItf) throws NoSuchInterfaceException {
30    switch(myClientItf) {
31    case "C1": C1 = null;
32  ...}
33
34  //methods to manage the attributes
35  public void set_myId(int newId) {...}
36  public int get_myId() {...}
37 }
```

Listing 2.2 – A Java class of a GCM/ProActive primitive

programmer has to associate a primitive and a Java class implementing its behaviour (line 6), a GCM interface and a Java interface with its method signatures (lines 4, 5), a list of attributes of a primitive with a Java interface providing access to them (line 7).

Given an ADL file, Java classes and interfaces, the GCM/ProActive factory returns an instance of the root component as illustrated in the Listing 2.3. Here, line 1 gets an instance of the factory which will create the component. The factory is asked to construct a new component at line 5; it takes two arguments: the `context` which stores the description of the deployment infrastructure (we omit the details here) and the name of the ADL file. Next, the constructed component should be "started". GCM/ProActive has an implementation of the Lifecycle controller able to start a component as illustrated at line 6. The user can then get a server interface of the constructed component by name (line 7) and invoke its methods (line 8). Additionally, the user can use the standard API to modify the root component and bind it to the other components.

```
1 Factory f = FactoryFactory.getFactory();
2 String adl = "Application.adl";
3 Map<String, Object> context = new HashMap<String, Object>();
4 //fill the context with the deployment data
5 Component application = (Component) f.newComponent(adl, context);
6 GCM.getLifeCycleController(mainComponent).startFc();
7 RunItf itf = (RunItf)(component.getFcInterface("s_it1"));
8 itf.run();
```

Listing 2.3 – A GCM/ProActive component construction and access

Overall, specifying GCM/ProActive applications manually can sometimes require significant effort as the programmer has to take care of the coherency between the ADL description, Java classes and interfaces. In order to facilitate the development process, VerCors generates all elements of the application automatically from the user-defined graphical specification.

Distributed deployment. The GCM components can be deployed in a distributed manner where a primitive component is a unit of distribution. The underlying infrastructure is specified as a composition of *virtual nodes* that express the abstract references to the resources where the components will be deployed. The virtual nodes are then associated to the exact physical infrastructure. The information about virtual nodes can be specified either with Java API or in the ADL file. The physical infrastructure should be provided in an XML-based file and given to the GCM factory when the component is being constructed (lines 3-5, Listing 2.3.

Collective communications. GCM/ProActive provides an API for multicast and gathercast interfaces with the corresponding controllers and several policies on dispatching request arguments and assembling results. At the current stage, the VerCors platform deals *only with the multicast interfaces for which an outgoing request is replicated and sent to all the bound targets and the results are collected in a list and given back to the requester*. Implementing other policies will be necessary in the future work.

Non-functional aspect. The predefined controllers are implemented in GCM/ProActive and the user has access to them. For example, line 6 of the Listing 2.3 demonstrated invocation of a Lifecycle controller that starts a component. Except from that, the user has two ways to implement his customized controllers: as object-controllers or as component-controllers. Both of them should be inserted in the component membrane and specified in the ADL file. As opposed to the objects, the component-controllers can have hierarchical structure and communicate with the

other subcomponents. ProActive implements the reconfiguration primitives defined in the CM specification.

2.2 Parameterised networks of synchronised automata (pNets)

In this section we present the parameterised networks of synchronised automata (pNets) - an intermediate formalism that we will use in this work to encode the behaviour of GCM components.

2.2.1 Term algebra and notations

In the following definitions, we extensively use indexed structures (maps) over some countable indexed sets. The indices will usually be integers, bounded or not. Such an indexed family is denoted as follows: $a_i^{i\in I}$ is a family of elements a_i indexed over the set I. Such a family is equivalent to the mapping $(i \mapsto a_i)^{i\in I}$. To specify the set over which the structure is indexed, indexed structures are always denoted with an exponent of the form $i \in I$ (arithmetic only appears in the indices if necessary). Consequently, $a_i^{i\in I}$ defines first I the set over which the family is indexed, and then a_i the elements of the family.

For example $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3; exceptionally, such mappings with only a few entries will also be denoted $(3 \mapsto a)$. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over I" when formally we should speak of multisets, and "$x \in A_i^{i\in I}$" to mean $\exists i \in I.\, x = A_i$. An empty family is denoted []. To simplify equations, an indexed set can be denoted $\overline{M}$ instead of $M_l^{l\in L}$ when L is not meaningful.

In all forthcoming definitions, we suppose that we have a fixed set of variables, used to construct the expressions of our term algebra. Our models rely on the notion of parameterised actions. We leave unspecified the constructors of the algebra that will allow building actions and expressions used in our models. Let us denote Σ the signature of those constructors, and $\mathcal{T}$ be the term algebra of Σ over the set of variables P. We suppose that we are able to distinguish inside $\mathcal{T}$ a set of *action terms* (over variables of P) denoted $\mathcal{A}$ (*parameterised actions*), a set of *data expression terms* (disjoint from actions) denoted $\mathcal{E}$, and, among expressions, a set of *boolean expressions* (guards) denoted $\mathcal{B}$. For each term $t \in \mathcal{T}$ we define $vars(t)$ the set of variables of t.

The countable indexed sets can also depend upon variables, and we denote $\mathcal{I}$ the

set of indexed sets using variables of P. There must exist an inclusion relationship $\subseteq$ over the indexed sets of $\mathcal{I}$, with the natural guarantee that this operation ensures set inclusion when one replaces variables by their values. In practice we will mostly use intervals for which the upper bound depends on the variables of P: $\mathcal{I} = [1..n]$ where n is an integer.

Let $\uplus$ (disjoint union) be a union operator on indexed sets requiring that the two sets are indexed over disjoint sets, we do not take care here of set re-indexing that should be performed to avoid collisions. The elements of the union are thus accessed by using an index of one of the two joined families.

2.2.2 The pNets model

pNets were first formalised in [30]. pNets are hierarchical structures made of parameterised labelled transition systems (pLTSs) at their leaves, synchronised by synchronisation vectors. In this section, we define the structure of pLTSs, pNets and Queues. The formal properties of pNets have been further studied in [21, 31].

pLTS. A pLTS is a labelled transition system with variables; a pLTS can have guards and assignments of variables on transitions. Variables can be manipulated, defined, or accessed inside actions, guards, and assignments.

We first identify the actions a pLTS can use. Let a range over action labels, op are operators, and x range over variable names. The set $\mathcal{A}$ of action terms used in pLTSs is defined as follows:

$\alpha \in \mathcal{A}$	::=	$a(p_1, \ldots, p_n)$	action terms
p_i	::=	$?x \mid Expr$	action parameters (input variables or expression)
$Expr$	::=	$Value \mid x \mid op(Expr_1, .., Expr_n)$	Expressions

We additionally suppose that each input variable does not appear anywhere else in the same action term: $p_i = ?x \Rightarrow \forall j \neq i.\, x \notin vars(p_j)$

For $\alpha \in \mathcal{A}$ we also suppose that there is a function $iv(\alpha)$ that returns a subset of $vars(\alpha)$ which are the input variables of α, i.e. the variables preceded by a "?" in the action label.

Definition 1 (pLTS)**.** *A parameterised LTS is a tuple* $pLTS \triangleq \langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle$ *where:*

- S *is a set of states.*
- $s_0 \in S$ *is the initial state.*

- *L is the set of labels of the form $\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle$, where $\alpha \in \mathcal{A}$ is a parameterised action, $e_b \in \mathcal{B}$ is a guard, and the variables $x_j \in P$ are assigned the expressions $e_j \in \mathcal{E}$. Variables in $iv(\alpha)$ are assigned by the action, other variables can be assigned by the additional assignments.*
- *$\rightarrow \subseteq S \times L \times S$ is the transition relation.*

Note that we make no assumption on finiteness of S or of branching in $\rightarrow$.

pNet. pNets are constructors for hierarchical behavioural structures: a pNet is formed of other pNets, or pLTSs at the bottom of the hierarchy tree. Request queues can also appear in leaves of a pNet system. A composite pNet consists of a set of pNets exposing a set of actions, each of them triggered by internal actions in each of the sub-pNets. The synchronisation between global actions and internal actions is given by *synchronisation vectors*: a synchronisation vector synchronises one or several internal actions, and exposes a single resulting global action. Actions involved at the pNet level do not need to distinguish input variables. The set $\mathcal{A}_S$ of action terms used in pNets is defined as follows:

$$\alpha \in \mathcal{A}_S \quad ::= \quad a(Expr_1, \ldots, Expr_n)$$

Definition 2 (pNets)**.** *A pNet is a hierarchical structure where leaves are pLTSs (or queues defined below), and nodes are synchronisation artefacts:*
$pNet \triangleq pLTS \mid Queue(\overline{m}) \mid \langle\!\langle pNet_i^{i\in I}, SV_k^{k\in K}\rangle\!\rangle$ where

- *$I \in \mathcal{I}$ is the set over which sub-pNets are indexed, $I \neq \emptyset$.*
- *$pNet_i^{i\in I}$ is the family of sub-pNets.*
- *$SV_k^{k\in K}$ is a set of synchronisation vectors ($K \in \mathcal{I}$). $\forall k \in K, SV_k = \alpha_j^{j\in J_k} \rightarrow \alpha'_k$ where $\alpha_j, \alpha'_k \in \mathcal{A}_S$. Each synchronisation vector verifies: $J_k \in \mathcal{I}$ and $\emptyset \subset J_k \subseteq I$.*

A synchronisation vector SV_k of the form $\alpha_j^{j\in J_k} \rightarrow \alpha'_k$ means that if each sub-pNet j in J_k performs synchronously the action α_j; this results in a global action labelled α'_k. For example, the synchronisation of the same action in two processes indexed i and j corresponds to the synchronisation vector $(i \mapsto a, j \mapsto a) \rightarrow \tau$ (recall that we identify indexed sets and mappings, giving us a convenient notation for synchronisation vectors). Brute-force unification of the sub-pNets actions with the corresponding vector actions, possibly followed by a simplification step, allow us to identify the exact actions of the sub-pNets that should be synchronised.

When $I = [1..n]$, it is equivalent to use tuple notations instead of indexed sets. In that case, we denote the pNet as $\langle\!\langle pNet_1, \ldots, pNet_n, SV \rangle\!\rangle$, and each synchronisation vector as: $\langle \alpha_1, \ldots, \alpha_n \rangle \rightarrow \alpha$. In that case, elements not taking part in the synchronisation are denoted by a dash $(-)$ as in: $\langle -, -, \alpha, -, - \rangle \rightarrow \alpha$.

Queues. There also exists a particular pNet construct called $Queue(\overline{m})$; it models the behaviour of a FIFO queue, with $\overline{m}$ the set of enqueue-able elements. We assume that the term algebra has two generic constructors iQ and $Serve$ such that, $\forall m_i \in \overline{m}.\, Serve_m_i \in \mathcal{A}_S \wedge iQ_m_i \in \mathcal{A}_S$. Then the queue pNet offers the following actions: $L = \{iQ_m_i | m_i \in \overline{m}\} \cup \{Serve_m_i | m_i \in \overline{m}\}$. The behaviour of a queue is only FIFO en-queueing/de-queueing of requests. It could be encoded as an infinite pLTS.

Sort. For each pNet, we define a sort function ($Sort : pNet \rightarrow \mathcal{A}$). The sort of a pNet is its signature: the set of actions that it can perform. For a pLTS we do not need to distinguish input variables. More formally[1]:

$$\begin{aligned}
&\mathrm{Sort}(\langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle) = \{\alpha\{\!\!\{x \leftarrow ?x | x \in iv(\alpha)\}\!\!\} | \langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle \in L\} \\
&\mathrm{Sort}(\langle\!\langle \overline{pNet}, \overline{SV} \rangle\!\rangle) = \{\alpha'_k |\, \alpha_j^{j \in J_k} \rightarrow \alpha'_k \in \overline{SV}\} \\
&\mathrm{Sort}(Queue(\overline{m})) = \{iQ_m_i | m_i \in \overline{m}\} \cup \{Serve_m_i | m_i \in \overline{m}\}
\end{aligned}$$

More notations. A constructor for a pNet is made of an indexed family of pNets. $\overleftrightarrow{\langle\!\langle PN_i^{i \in I} \rangle\!\rangle}$; it takes a family of pNets indexed over a set $I \in \mathcal{I}$ and produces a global pNet. The synchronisation vectors for this family will be expressed at the level above, consequently we "export" all the possible synchronisation vectors that the family could offer, even if only some of them will be used.

$$\begin{aligned}
\overleftrightarrow{\langle\!\langle PN_i^{i \in I} \rangle\!\rangle} &\triangleq \langle\!\langle PN_i^{i \in I}, \{\overline{\alpha} \rightarrow \overline{\alpha} | \overline{\alpha} \in V\} \rangle\!\rangle \\
&\text{where } V = \{\alpha_j^{j \in J} |\, J \subseteq I \wedge \forall j \in J.\, \alpha_j \in \mathrm{Sort}(PN_j)\}
\end{aligned}$$

This supposes that the elements of V are action terms. If all the elements of the family are identical, then we simply write $\overleftrightarrow{\langle\!\langle PN^I \rangle\!\rangle}$.

An operational semantics for pNets is given in [21].

2.2.3 Observation and flow of information

In the context of encoding GCM components with pNets we assume that for each globally synchronised action there is a single pLTS that outputs the value of each

[1] $\{\!\!\{y \leftarrow x\}\!\!\}$ is the substitution operation.

parameter (and potentially several targets). However, several actions will receive one parameter and output another one, without any consequence on the decidability of which actions can be triggered. By convention, we use labels starting by "i" for the input side of the main flow (if there is one) as shown in the diagrams, like iQ and iR.

Finally, we identify the actions that are already synchronised (they will not need further synchronisation). We slightly extend the action algebra with such already synchronised actions (distinguished by underlined labels):

$$\alpha \in \mathcal{A}_S \quad ::= \quad a(Expr_1, \ldots, Expr_n) \mid \underline{a}(Expr_1, \ldots, Expr_n) \quad \text{pNet actions}$$

Synchronised actions are not meant to be used anymore for synchronisation purposes, they should just be visible at the top-level of the pNet hierarchy for the observation purpose. Consequently, we define an operator that takes an indexed set of pNets and returns the synchronisation vectors that should be included in the parent pNets to allow the visibility of synchronised actions:

$$\text{Observe}(pNet_i^{i\in I}) = \{(i \mapsto \underline{\alpha}) | i \in I \wedge \underline{\alpha} \in \text{Sort}(pNet_i)\}$$

In the following, those synchronisation vectors dedicated to observation will be *implicitly* included as synchronisation vectors of all the pNets. This means that for all pNets, Observe($pNet_i^{i\in I}$) is implicitly included in the set of synchronisation vectors (where $pNet_i^{i\in I}$ is the set of sub-pNets of the new pNet). These synchronisation vectors are only useful to observe the internal reductions.

In order to express synchronisation vectors of families of pNets, we must allow families of actions to be considered as actions themselves. More precisely, if a_i is an action, then actions can be of the form $(a_i)^{i\in I}$, or $i \mapsto a$ to allow the sub-pNet at index i to perform an action.

2.2.4 Adequacy of pNets for modelling GCM components

In this work we will present the behavioural semantics of GCM components, expressed as a translation from a component architecture into a hierarchical pNet. Before defining this translation, we explain below why the pNets were chosen as a support for GCM behavioural semantics.

First, our goal is to provide a model adapted to the behavioural verification of the properties of GCM applications. It must be adapted to the generation of a model that can then be verified, typically a finite model that could be model-checked, even if other techniques could be envisioned. In pNets, models can use parameters, both

in the structure and in the LTSs which allows us to give a semantics based on an infinite set of states, but also to easily consider finite instances by restricting each parameter to a finite domain. Thus the first reason for the choice of pNet is that *it is adapted to the definition of infinite models from which a finite instance can easily be extracted.*

Second, the semantics of communication and asynchrony of pNets fits closely to the one of GCM. Indeed, to guarantee causal ordering of requests, GCM components communicate by a rendez-vous mechanism. In GCM/ProActive the request sending and its arrival in the queue of the destination component occur synchronously. The rest of the execution is entirely asynchronous. pNets have a similar semantics: they are made of independent pLTSs or pNets interacting by synchronous communications. On the contrary, futures are a too high-level construct to be part of pNet definition. They will have to be encoded by a set of pLTSs. Next sections will show that the parameterised nature of pNets and the synchronisation vectors allow for encoding futures in a precise and generic way. Thus the second reason why we chose pNet is that *it provides a communication model similar to GCM for requests and give enough expressive power to encode futures.*

From another point of view, we mentioned earlier that we want our behavioural models to represent the structure of the application (e.g. to allow the encoding of reconfigurations). On that aspect and more generally when encoding communication channels, π-calculus might seem to be a reasonable approach. However, we think that channels *à la* π-calculus are too powerful. Indeed, in GCM/ProActive bindings are not first-class entities and can only be reconfigured by an application manager. Additionally, pNets are much better adapted to the verification techniques we target, i.e., finite state model-checking, than π-calculus.

Finally, *the hierarchical structure of pNets fits well with hierarchical components.* The different levels of hierarchy of the ADL will lead to the same hierarchical levels in pNets, even if additional pLTSs and pNets will be defined to encode specific features (e.g., future proxies).

2.3 CADP - a toolbox for construction and analysis of distributed processes

CADP (for Construction and Analysis of Distributed Processes)[32] is a toolsuite for the formal modelling, simulation and analysis of parallel asynchronous systems; it has been proven efficient by multiple case-studies [33, 34, 35]. The framework relies

```
1  < a1, b1, − > −> g1,
2  < −, −, c1 > −> g2
3  A | B | C
```

Listing 2.4 – An example of synchronization vectors in .exp

mainly on LOTOS [36] and LotosNT [37] as system specification languages and LTSs as the abstraction model. We describe below some of the tools included in CADP and we start by the modules applied in this work.

Caesar and Caesar.adt are compilers that translate the behavioural and the data parts of a LOTOS specification correspondingly either into an LTS that can be verified or into a C program to be executed or simulated.

BCG (Binary-Coded-Graphs) is both a format for LTSs and a set of dedicated libraries. The advantage of the format is that compared to the LTS represented in ASCII, BCG graphs can take up to twenty times less space. Among variety of tools, the library includes:

- **bcg_draw** - a module for graphical representation of the graphs;
- **bcg_min** - a tool for graph minimization by strong or branching bisimulation;
- **bcg_labels** allows hiding and renaming labels;
- **bcg_info** gathers information about a graph such as its size, the number of states and transitions, the list of labels, etc. and provides it to the user;

Exp.open [38] takes a **.exp** file encoding a network of communicating automata and constructs its global behaviour graph. The automata are represented as .bcg files and composed together in parallel using different techniques including *synchronisation vectors*, on which we rely in this thesis. Listing 2.4 demonstrates an example of an .exp file with two synchronisation vectors (lines 1-2). Line 3 lists the names of the .bcg files of the operating in parallel LTSs that will be composed. A LOTOS action label is divided into two parts: *gate* is the action name and *offers* is the list of action parameters. The vector at line 1 should be interpreted as follows: if the LTS `A` performs an action with gate `a1`, it should synchronise with `B` performing an action with gate `b1` and the behaviour of `C` is not taken into account. This will result in a global action with gate `g1`. The offers of synchronised actions should have parameters with exactly the same types and values. Those parameters will be added to the global action.

The model checking language (MCL) is a regular alternation-free μ-calculus with actions, predicates and expressions over action sequences equipped with parametrised fixed-point operators and regular expression. The formalism allows direct specification of branching-time logics formulas like ACTL [39] and CTL [40] and of regular logics formulas like PDL [41] interpreted over LTSs. A strong advantage of MCL is the ability to encode manipulations with data variables. Three types of formulas can be encoded with the language. First, action formulas are built upon action predicates and boolean operators; second, regular formulas are constructed from action formulas and regular expression operators (such as choice, counting, repetition and others). Finally, state formulas are built from boolean operators, modalities, fixed point operators, and data-handling constructs. In [42] the authors demonstrate the expressiveness of the formalism by encoding safety, liveness (potential reachability, inevitability, deadlock freedom), fairness properties. In order to facilitate formula specification, MCL is equipped with a library of property patterns [43] such as absence, precedence, bounded existence, and others. The language allows one to construct his own reusable libraries of temporal operators.

Evaluator [42] is an on-the-fly model-checker and it is the core CADP engine used in this work. The tool takes as input an LTS expressed in various formats including binary graphs, LOTOS, LotosNT, EXP, and a temporal logic property to be checked in MCL [44]. Then, the input is translated into a boolean equation system [45] which is solved using algorithms explained in [42].

Evaluator answers either TRUE or FALSE depending on whether the property is satisfied and possibly provides diagnostics which can be either an example or a counterexample.

SVL (Script Verification Language) [46] is a high-level scripting language aiming at facilitating program verification with CADP. The language includes operators for model-checking, label hiding, renaming, and state-space reduction.

Ocis is an interactive graphical simulator. It takes an automaton as an input and visualises the execution tree of a system, the execution traces and the communication between the parallel processes. The user can manually step-by-step navigate through the execution scenario and save the simulation results as a .bcg graph.

State-space reduction is implemented in CADP at two levels and employs the notion of so-called "tau-transitions" or "hidden transitions", i.e. the transitions that exist in the behaviour graph but do not need to be observed during model-checking.

The first approach called *partial reduction* simply compresses and merges such transitions. The second approach - *total reduction* consists in merging several states of a partially reduced LTS into one state by applying bisimulation techniques. Such minimisation approach is particularly useful for hierarchical systems like GCM applications because it allows one to benefit from hiding some details of the behaviour of the processes at the low-levels of hierarchy. Also, as we will explain later, many processes inside GCM components have bisimular behaviour that can be merged during state space minimisation.

The application of CADP tools in this work. To wrap-up, in this work we will apply the discussed CADP tools as follows. We will express LTSs in `.bcg` and synchronisation vectors in `.exp`, we will apply *exp.open* for the composition of automata. We will use the *bcg_min* to minimise the state space as we will hide a lot of communications that should not be observed during verification. We will specify system properties in *MCL* and model-check them with *evaluator*. Finally, we will take advantage of the user-friendly *svl* script for invoking various CADP modules and debug our systems with *bcg_info*, *bcg_draw*, *bcg_labels*, and *ocis*.

The following two modules are not applied in this thesis but we plan experimenting with them in the future work.

Distributor [47] is a tool for distributed state-space construction, it splits the generation over N machines and each of them builds a fragment of the resulting LTS. The fragments are then assembled by **bcg_merge**.

Projector abstracts the behaviour of a graph by synchronizing it with the restricted version of the behaviour of its interfaces.

2.4 The Fiacre specification language

Fiacre (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [48] is a user-friendly language for modelling distributed and embedded compositional systems. Fiacre can be translated into input formats for various verification tools including CADP.

An example of a Fiacre program is illustrated in the Listing 2.5. Its core element is a *process* (named `TaskDistributor_run`, line 3) that consists of a set of states (declared at line 8), operates on variables (declared at line 9) and constant values (line 1). The variable data types include built-in integer, natural numbers, boolean

```
1 const max:nat is 1
2 type Interval is 0..2
3 process TaskDistributor_run
4 [task1: out Interval,
5 r_task1: in Interval,
6 return_res: out boolean]
7 is
8 states s0,s1,s2
9 var x:Interval, y:nat
10 from s0
11   run!x, ls; to s1
12 from s1
13   r_run? y; to s5
14 from s2
15   if  y < max then
16    return_res !true; to s0
17   else
18    return_res !false; to s0
19 TaskDistributor_run
```

Listing 2.5 – A Fiacre process

and user-defined lists, integer intervals (line 2), records (C-like structs). Each state is associated with a sequence of performed instructions that can include standard if-else statements, non-deterministic choices, assignments, communication events followed by a transition to another state. A communication event has a name and may have a list of either input (prefixed by ?) or output (prefixed by !) parameters. The programmer should specify the signatures of all communication events as illustrated at lines 4-6.

Another element of a Fiacre program - a *component* - can be used to assemble several processes that operate in parallel and have shared variables; it will not be used in this thesis.

Fiacre is not a native input language of CADP but it has a compiler to Lotos. In this work we will encode pLTSs with Fiacre and then use the **flac** [49] compiler to translate them to bcg graphs that will be model-checked by CADP. We prefer using Fiacre because it is very easy to model pLTSs as Fiacre processes and because of the simplicity and readability of the language.

2.5 Model-Driven Engineering

Model-driven engineering (MDE)[50] is a software development paradigm that allows constructing an abstract model of an application which can be analysed at the design stage, transformed into the actual implementation, and used as a documentation and for automatic testing. Models can describe a system at different levels of abstraction and from different viewpoints such as hardware/software requirements, application architecture or behaviour.

The pivotal concept of MDE - a *model* - is often defined as an abstraction of a system under study. Another key notion - a *meta-model* - specifies how the model should be described, i.e. the building blocks of a model and the relations between them. For example, if we would like to create a model of a family, the meta-model would specify that the model consists of family members who have various properties (for example, name and age) and can be related as children, spouses, cousins, etc. Models are often associated with *views* that are their graphical representations.

Constructing a conceptual model before starting the implementation is good because it provides a clear view on system requirements and can be used as a source of documentation, but there are other advantages of the MDE approach. Multiple analysis techniques can be applied to a model in order to evaluate the quality of the future application and detect errors at the design stage. The static correctness of a model can be ensured by static analysis techniques; formal methods such as model-checking can be applied to check functional properties. Also, conceptual models can be used for an application performance predication at the design stage. Additionally model-to-model and model-to-text transformation techniques can be used to automatically transform a model into another model or textual representation that could be an executable code.

Models are specified in modelling languages. The **domain-specific languages** (DSLs) are used to design systems for a particular domain, and there exist hundreds of DSLs in the world. However, sometimes it is a good idea to design a software in a language that will be easily understood by a wide audience, and this is what is provided by the Unified Modelling Language (UML).

2.5.1 Unified Modelling Language

UML [7] is a general-purpose modelling language created and supported by the Object Management Group (OMG). The language is mainly dedicated to the design of object-oriented applications and widely used in industry and academia. UML allows describing a system at different levels of abstraction as a set of diagrams. The examples of diagrams could be: use-case diagrams illustrating how the user should use the application, sequence diagrams that depict the interactions between entities, and activity diagrams that describe step-by-step behaviour.

The two UML diagrams used in this work are the class diagrams and the state machine diagrams. The **class** diagrams define program classes and interfaces with their attributes and method signatures, and relations between them that include inheritance, aggregation and composition. Figure 2.4 illustrates an example of a class diagram with class `A` that implements an interface `Itf`, owns two attributes `at1`

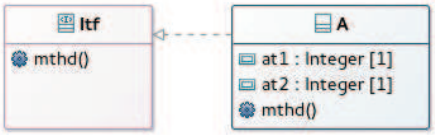


Figure 2.4 – UML class diagram

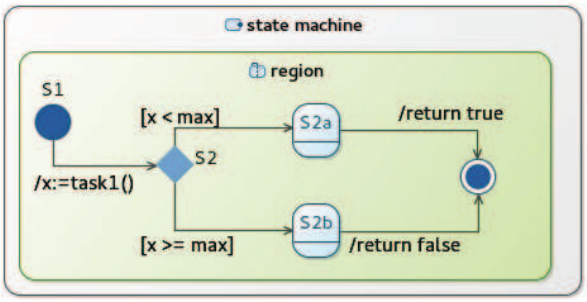
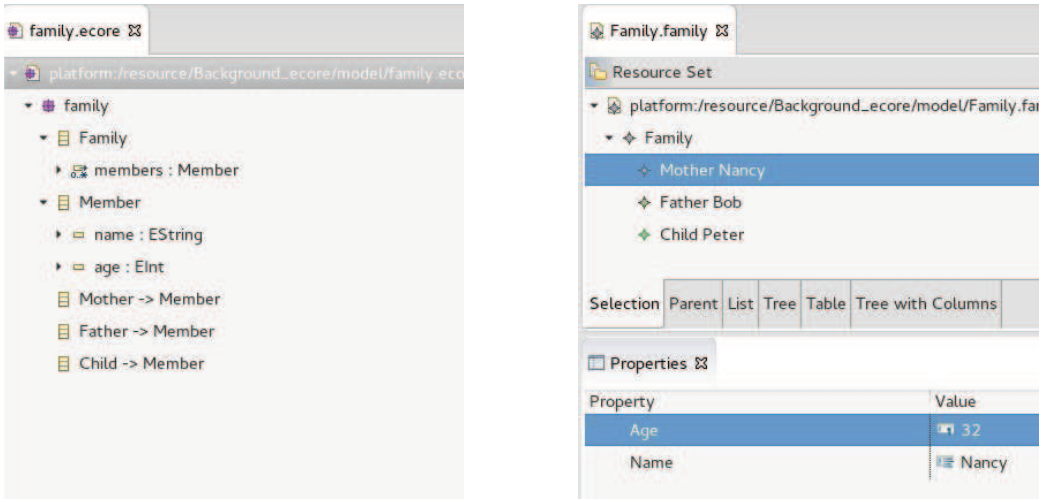


Figure 2.5 – UML state machine diagram

and `at2`, and defines a method `mthd`. The behaviour of an entity (class, method, or even a system as a whole) can be described with possibly hierarchical **state machine** diagrams (also known as state charts). A state machine diagram (we will call them state machines for short) illustrates the sequence of events that an entity goes through and their impact on the entity state. An example is illustrated in Figure 2.5. The model is composed of different type of states, e.g. choice (`S2`), initial (`S1`), fork, and join states, connected by transitions that describe possibly guarded events. States can be assembled in regions. In this work we will not use hierarchical state machines and we will have one single region per state machine.

2.5.2 Eclipse Modeling Framework

The Eclipse development environment [51] provides a rich ecosystem for model-driven engineering; its core technology is the **Eclipse Modeling Framework** (EMF) [52] which includes the following features. The first element is a set of tools for constructing a meta-model (so-called *ecore* model). The construction is supported by a tree-like editor and a static validator. From the ecore model, EMF automatically generates the Java API of the designed model, an editor that represents a model in a tree-like viewer (we will call it an *EMF-editor* in the next chapters) and a standard Eclipse property editor for the parameters of the modelled elements. The models are stored in an XML-based format. Figure 2.6 illustrates an example of an ecore meta-model of a family on the left and the corresponding generated EMF editor with a model of a family and properties editor on the right.



(a) Ecore model (b) EMF-based editor

Figure 2.6 – EMF example

A graphical editor for an EMF-based model can be implemented with the help of the Graphical Modeling Framework (GMF) [53]. Both EMF and GMF editors are distributed as Eclipse plug-ins, i.e. a software that can be installed on top of Eclipse. For example, a UML ecore and dedicated editors are implemented as an Eclipse plug-in [54].

2.5.3 Obeo Designer

EMF and GMF provide a large variety of instruments for building domain-specific model editors. We will not get into details here, but one who has an experience of working with GMF knows that the technology is powerful, but using it for constructing and maintaining a large graphical designer requires significant effort. A platform that allows one to implement easily a GMF-based modeller "without knowledge of GMF" is Obeo Designer (it has an open-source version Sirius also known as "Obeo Designer Community") [55, 56] that was developed by the Obeo company. The framework is built on top of EMF and GMF and provides techniques for specifying a domain-specific language, various model representations such as a diagram, a table or a matrix, and tools to edit the model. We would like to highlight the following features of Obeo Designer:

- a technique to extend the modeller with external Java code which allows implementing complex computations;
- built-in layout managers and possibility to implement a custom layout manager;

- wide range of predefined graphical representations of diagram elements and a mechanism to implement a custom graphical view;
- a mechanism for diagram validation: the programmer can declare the rules to be checked and Obeo Designer will automatically generate the validation engine and mark the erroneous elements;
- for any graphical editor, Obeo Designer automatically provides instruments to export a view in various graphical formats, to hide and show representation elements, to undo/redo actions, and many other features.

A graphical editor for UML models based on Obeo Designer is implemented in [57]. It is an open-source project that can be easily extended.

Another framework developed by Obeo is a model-to-text translator **Acceleo** [58] which is integrated with Obeo Designer. It can be used to transform an EMF-based model into any kind of code. The technology is based on templates: the programmer has to define textually the template of an input element and the corresponding output text; the programmer can invoke external Java services during text generation. From the given input, Acceleo constructs the generator that can be then distributed as an Eclipse plug-in.

In this work we will use Obeo Designer to define our own DSL that relies on UML and to implement a graphical editor. We will apply model static validation to ensure that the models are statically correct and use Acceleo model-to-text transformation to generate executable Java code.

2.6 VerCors

The VerCors platform has already undergone several major generations, with significant evolutions for the underlying semantic model, as well as the modelling platform and the specification formalisms.

The very first version of VerCors was based on a textual description of component architecture. The original version of the graphical front-end editor called **CTTool** [20] was using UML component structures for describing the application architecture and activity diagrams for behaviour modelling. CTTool was generating LOTOS specification that could be given to CADP for model-checking. The tool was extensively used in the CoCome case-study [59]. The problem was that, at that time, the authors already aimed at using pNets as an intermediate format and they were not

able to implement properly all the constructs with CTTool. Additionally, the authors realised that the UML components were too far from GCM needs.

Hence, a new DSL for component structure and a new graphical formalism were defined and implemented in a tool called **VCE** (VerCors Component Editor) [19]. At the same time, aiming at better support for maintenance and usability, the platform was moved to an Eclipse-based environment and EMF. The new version included a graphical designer, an architecture static validation engine, an ADL generator, and a module transforming ADL into graphical models. The generated ADL could be given to **ADL2N** tool that transformed it into a pNet in **fc2** format [60]. The signatures of methods had to be specified in Java interfaces. Additionally, with the help of dedicated GUI, the user had to abstract the data domains. Then, a generated pNet and a set of manually written fc2 files with server methods behaviour could be given to **FC2Parametrized** [61] tool which instantiated the system and produced EXP files.

A series of publications [62, 63] described the support for several features of distributed component-based systems, including group communications and futures, but the toolchain from the specification to behavioural model generation was incomplete and relied on several manual steps. Moreover, generation of the behavioural models including some particular features (e.g. group communications) has been discussed and illustrated by examples in the previous works, but has never been implemented.

The work presented in this thesis has significant changes regarding the previous tools. It is the first version of VerCors that gathers all bits and pieces of the previous works and implements fully automatised generation of behavioural models and executable code. The key improvements are listed below::

- This is the first version of VerCors that integrates the GCM architecture DSL with UML and allows specifying all core features of GCM. We extended the existing architecture DSL with references to UML classes and interfaces that include method signatures. We integrated UML state machine editor for behaviour specification and we defined precisely the state machine structure and semantics in the context of GCM. The graphical formalisms are explained in details in Section 3.2. In fact, there was no graphical editor for behaviour specification in the previous versions.

- Second, we changed the underlying platform to Obeo Designer and we benefit from its rich infrastructure for building and maintaining graphical editors. We discuss the architecture of the latest version of VerCors in Section 3.3.

- We refined and extended the existing set of architecture static validation con-

straints to deal with the non-functional aspect. The formalisation of the validation rules and the implementation of their verification are given in Chapter 4.

- Next, since the latest version of VerCors, the semantics of GCM components in pNets has significantly evolved, and all the changes are taken into account in the new version. We generate pNets directly from the graphical representation. Moreover, this is the first version of the platform which automatically constructs the behaviour of server and local methods. We explain in details the latest version of the pNets encoding the GCM component semantics and the VerCors pNets generator in Section 5.1. We address the advanced features (the non-functional aspect, the group communications) in Chapter 6. Still, we reuse some code for pNets construction from the previous versions.

- This is the first version of VerCors where the user can specify graphically a reconfigurable system, automatically translate it into an input for the model-checker and generate its Java code. For more details, we refer to Section 6.4.

- We almost fully reuse the engine generating ADL. We improved it significantly by implementing the non-functional part generation.

- This is the first version of VerCors that produces executable Java code from the graphical model. The generator is discussed in Section 5.3.

In the rest of this thesis we will first make a overview of the current version of the VerCors platform in Chapter 3. We will present it from the user point of view, explain its core functionalities, and describe its architecture which relies model-driven technologies. Then, in Chapter 4 we will formalise the architecture of GCM components and the notion of component well-formedness. Next, in Chapter 5 we will formalise how the basic behaviour of GCM components can be encoded in pNets, we will explain how the formalised generators are implemented in VerCors, and how the constructed pNets can be then given as an input to the model-checker of CADP. The generation of the executable code which can run on top of ProActive is also presented in Chapter 5.

Chapter 3

An overview of the VerCors platform

Contents

3.1 The core functionalities of VerCors 41

3.2 Diagrams for architecture and behaviour specification . 44

3.2.1 An illustrative example 44

3.2.2 Architecture specification 45

3.2.3 Behaviour specification 48

3.3 The architecture of VerCors 51

3.4 Discussion 55

The chapter provides an overview of the core features of VerCors. The version of the platform presented in this chapter is one of the contributions of this thesis. It was published in [64]. We do not detail the differences between the current version and the previous ones as they have already been discussed in Section 2.6. We start by describing the platform capabilities from the user point of view. Then, we present the graphical languages used to design component system architecture and behaviour in VerCors. Finally, we introduce the architecture of the framework and give a brief overview of the implementation.

3.1 The core functionalities of VerCors

Now, when the reader is familiar with all underlying technologies and formalisms, we can introduce the global view of the VerCors workflow which is illustrated in Figure

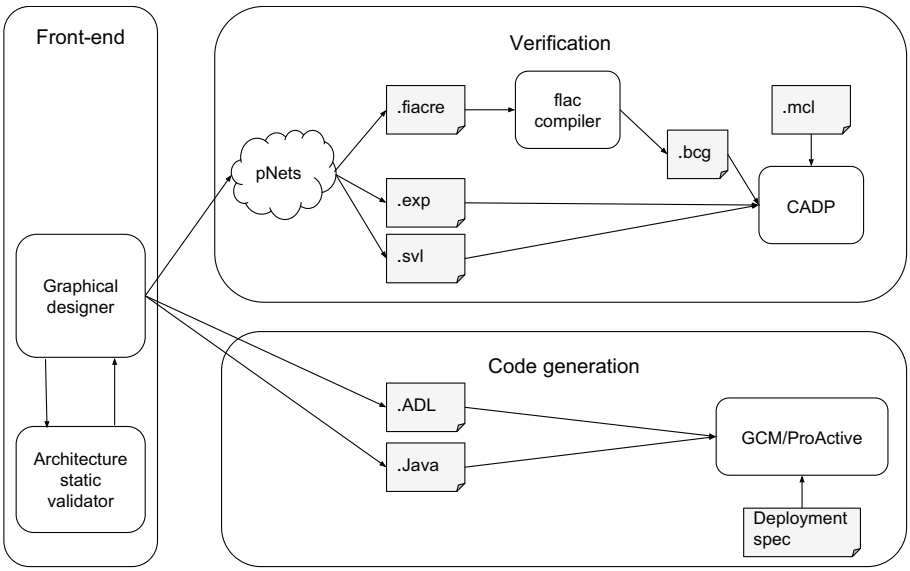


Figure 3.1 – VerCors workflow

3.1. The platform consists of three core parts: the front-end graphical designer, the verification module, and the executable code generation plug-in. We discuss below each element from the user point of view and we briefly mention the VerCors installation procedure first.

Installation. The procedure is extremely simple: the user can follow the installation process offered by the standard Eclipse software install manager [1] in order to install the platform on top of Obeo Designer (or Eclipse which already has an Obeo Designer installed).

The Front-end. Once VerCors is installed, the user can create a VerCors-based project ("VCE-project") using a standard Eclipse project-creation wizard. By default, a newly created VCE-project contains a set of empty models where the user will specify the architecture, classes, behaviour of his component-based system, and type definitions. The models can be modified using the standard EMF editors and/or diagram designers which rely on the graphical languages described in the Section 3.2. Among several type of diagrams that can be modelled in VerCors, the following four are used by the verification and code generation modules: component (architecture), UML class, UML state machine and type diagrams. Additionally, the user can model a number of other UML views such as use-case, sequence, package diagrams, etc. All graphical designers are equipped with the standard functionalities provided by Obeo

[1] http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-124.htm

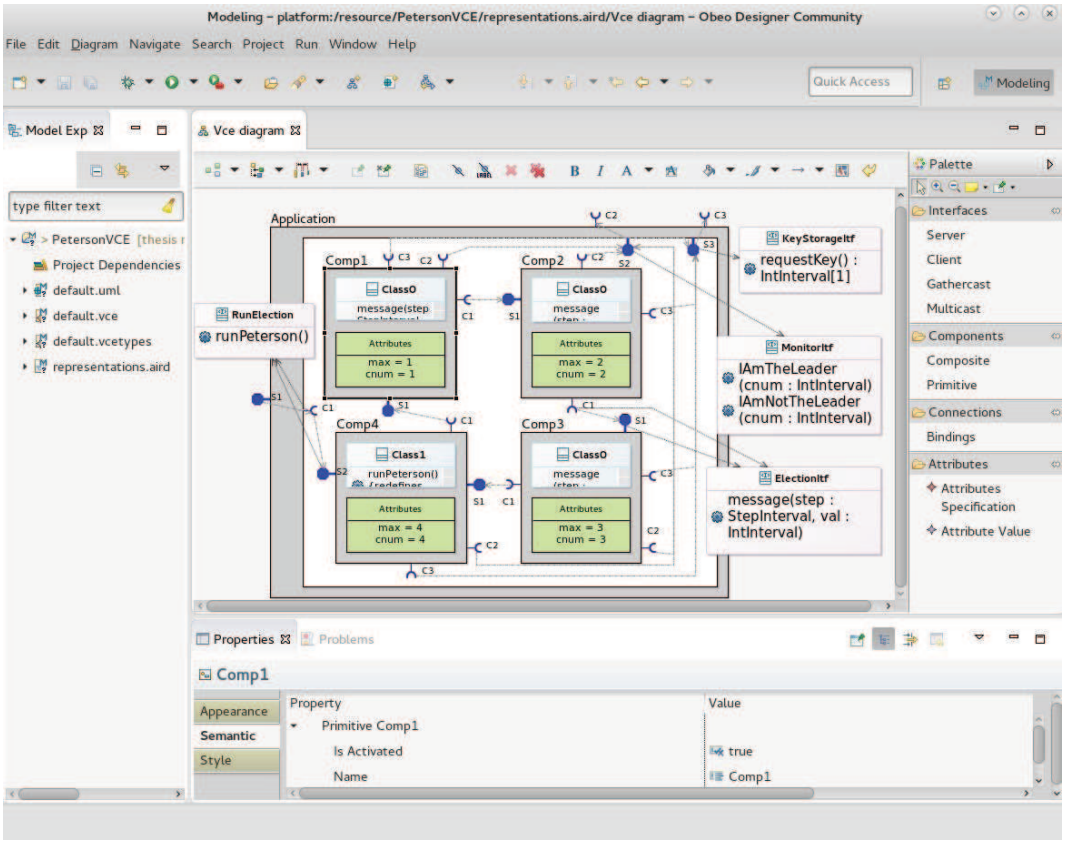


Figure 3.2 – Screenshot of VerCors

Designer; for instance, the user can show and hide graphical elements, export his diagrams in a number of formats, layout the elements. The static correctness of the modelled component architecture can be verified with respect to a set of predefined properties which are explained and specified formally in Chapter 4.

Figure 3.2 illustrates a screenshot of VerCors with a VCE-project structure on the left, a diagram in the middle and tool palettes on the top and on the right. The bottom panel can be used to modify the semantic properties of the modelled system.

Verification. After checking the static correctness, for any component in the conceptual model, the user can launch the generation of an input for the model-checker. The selected component is called a *root*, and VerCors produces the behaviour graph both for the root component and for all its sub-components. The generated graph can be given to the CADP model-checker to prove the behaviour correctness, As an additional input, the user should specify the queue size for each component, the communications that will be hidden during the model-checking and, optionally, a scenario restricting the environment behaviour. VerCors analyses the input data and generates the pNets encoding the behaviour of the modelled GCM components. The

pNets are not visible to the user, they are used as an intermediate formal. Then, VerCors transforms the pNets into .fiacre and .exp files with the data types abstracted according to the type diagrams. The platform also creates the auxiliary scripts for managing the verification workflow. Flac compiler transforms .fiacre files into BCG format which can be then given to CADP together with the .exp and the auxiliary scripts. Finally, the user can specify the properties that he wants to check on the generated graph and run Evaluator of CADP (see the details in Section 5.2).

Code generation. The Java code of the modelled components can be automatically generated as it is presented in Section 5.3. VerCors translates the designed architecture into an ADL file and produces a set of Java classes and interfaces including the signatures of the methods, attributes, and the implementation code of the methods that were modelled by the user. The user can choose to keep the behaviour of some methods undefined, in this case VerCors produces an empty body. The enumeration types and records are also translated into Java enumerations and classes respectively. The generated code can be executed on the GCM/ProActive platform.

3.2 Diagrams for architecture and behaviour specification

In this section we present the diagrams for the design of a component system architecture and behaviour in VerCors. The section is illustrated by a small size use-case example - a GCM-based application implementing Peterson's leader election algorithm [65]. We start by introducing the algorithm. Then, we present the four core types of diagrams used in VerCors: component diagrams, UML class diagrams, UML state machine diagrams and type diagrams. We show how the four graphical languages are integrated together in order to provide an expressive formalism for component-based application specification.

3.2.1 An illustrative example

Distributed processes often need to select a unique leader; Peterson's election algorithm can be used for this purpose. The participants are organised in a unidirectional ring of asynchronous processes. Every process participating in the elections has a FIFO queue and the order of sent messages is preserved by the communication channels. Each process can be either in active mode if the process participates in the election, or in passive mode if it only forwards messages. Initially, every process stores

a unique number that will be modified during the election. The processes exchange two rounds of messages so that every active process learns the numbers stored by the two nearest active processes preceding it. If the maximum of the two values of the nearest active processes and the value held by the current process is the value received from the nearest predecessor of the process, then the active process takes this value as its own value; otherwise the process becomes passive. The rounds of messages and local decision steps are repeated until a process receives its own number, this process is the leader.

In details, every process P stores variables $max(P)$ and $left(P)$. $Max(P)$ is the number stored by P. $Left(P)$ is the number of the active process on the left of P. Processes exchange messages of the form $M(step, value)$ where *step* is the phase of the algorithm. At the *preliminary phase*, each process P_i sends $M(1, max(P_i))$ to its neighbour. Then, if an active process P_i receives a message $M(1, x)$ and x is equal to its own number, the process is the leader, otherwise it assigns x to $left(P_i)$ and sends $M(2, x)$ to its neighbour. When an active process P_i receives $M(2, x)$ it compares $left(P_i)$ to x and $max(P_i)$. If $left(P_i)$ is greater than both values, P_i assigns $left(P_i)$ to $max(P_i)$ and sends $M(1, max(P_i))$; otherwise P_i becomes passive.

In [66] the authors prove that "if the algorithm ever terminates, it does so correctly" in the sense that one and only one leader is elected.

3.2.2 Architecture specification

Component diagrams. The component diagrams are used to define the architecture of a component-based application, i.e. to design the composite and primitive components with their interfaces and relations between them. Figure 3.3 illustrates an example of a simple component diagram. A primitive is depicted as a grey box (`Prim1` in the figure) while a composite (`Composite` in the figure) is illustrated as a rectangle divided into two parts: the grey part shows its membrane while the white part corresponds to the content. A GCM interface can be either attached to the border of its host component (e.g. `C-ext`), or to the border of a content in the case of internal interfaces (e.g. `S-int`). An interface has a set of characteristics that impact its graphical representation. The icon of an interface changes depending on whether it is a server (e.g. `S1`) or a client interface (e.g. `C1`), a singleton or a collective interface, a functional or a non-functional interface. The bindings are shown as black arrows that go from the requesting interfaces to the serving ones.

Graphical distinction between functional and non-functional aspects The specification of a GCM application functional part can be separated from the non-functional one thanks to the separation of a composite into membrane and content and the dis-

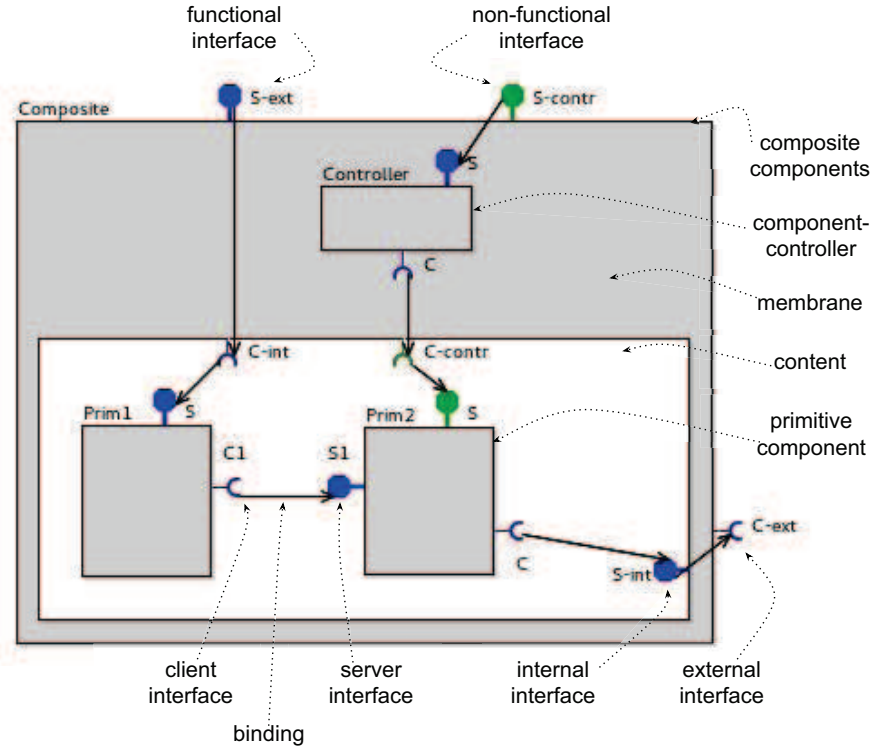


Figure 3.3 – VerCors component diagram

tinction between functional and non-functional interfaces. The non-functional subcomponents (e.g. `Controller`) are located in the membrane of a composite while the functional ones (e.g. `Prim1` and `Prim2`) are placed in the content. The non-functional interfaces (such as `S1-contr`) have green color whether the functional ones are blue. Note that the non-functional component `Controller` has two functional interfaces `S` and `C` that fulfil a non-functional role at the level of the composite.

UML class diagrams. Another kind of diagrams used for the conceptual model specification in VerCors is UML class diagrams. The platform relies on the following elements: interfaces, classes and generalisation relations, methods, and attributes.

The *UML interfaces* are used to define the list of methods that can be called and served by client and server GCM interfaces correspondingly. The UML interfaces attached to GCM ones appear on the corresponding component diagrams.

Every primitive should have an attached *UML class* that appears on the component diagram and defines the list of methods implemented by the component and the list of owned attributes. The user can specify generalisation relations between classes.

A *UML attribute* is characterised by its type, its name and a default value. There exist two ways to define the default value: either in the class specification, or in the

component definition. The former will assign the same default value to the attribute for all components using this class or its successor; the latter is the approach to specify the default value of an attribute belonging to a particular primitive. If the default value is given both in the class specification and in the component definition, the second one has priority.

Every *UML method* is described by a name, a list of input parameters and an output type. Additionally, any method belonging to a class can redefine an interface method; this means that the class method implements the corresponding method of an interface. The methods that do not redefine anything are considered as local ones, i.e. they are used for the component internal computations. For every class attribute the user should provide the signature of its set and get methods in order to access the attribute value. It would not be difficult to implement an automatic generation of those methods in the future versions of VerCors.

Type diagrams. The data types used by UML methods and attributes should be declared in a type diagram. The user can define integer intervals, enumerations, record types similar to C-like structs and arrays of fixed size. Additionally, there exist number of built-in types that do not appear on type diagram: boolean, integer and natural number types.

The use-case example The Component diagram representing the architecture of our use-case model of Peterson's leader election algorithm is shown in Figure 3.4. It relies on the UML class diagram illustrated in Figure 3.5

`Application` is a composite; it includes four primitives that participate in the leader election process. The primitives are connected in a ring topology and have similar structure. The entry point of the system is the `runPeterson()` method of `Application` server interface `S1`. This request is forwarded to `Comp4` that triggers the election process. During the election, components invoke method `message` on their client interfaces `C1`. As defined in Section 3.2.1, each message transmits two parameters: *step* and *val*. The message is transmitted to the server interface `S1` of the called component. The signature of *message* is specified in a UML interface `ElectionItf`. If a component decides to become a leader or a non-leader, it reports its decision to the environment by invoking an `IAmTheLeader(cnum)` or an `IAmNotTheLeader(cnum)` method on its client interface `C2`. These methods take the identifier of the component as a parameter.

In order to illustrate the futures mechanism, the leader component triggers an external computation that returns a result; here we consider getting a key for en-

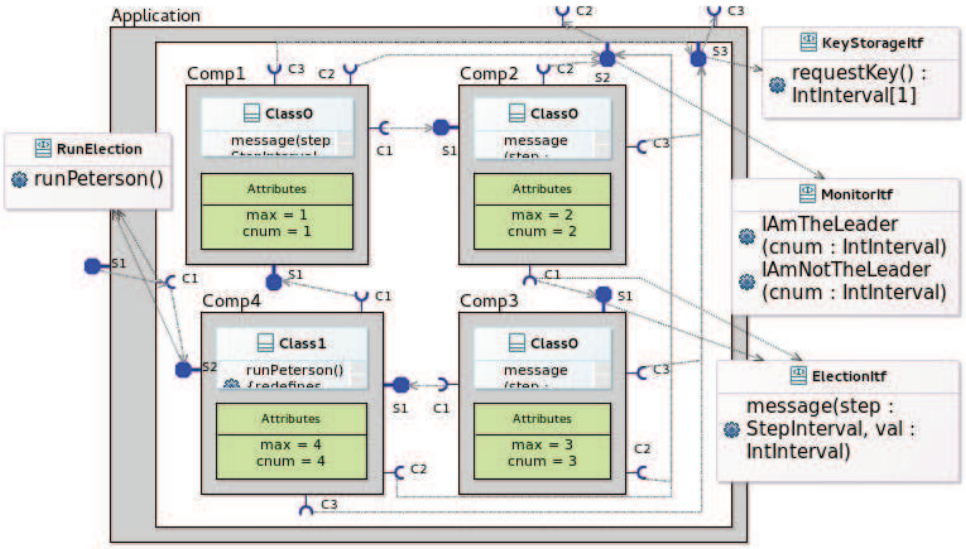


Figure 3.4 – A component diagram of Peterson's leader election use-case example

cryption process. We extend each primitive with a local method `encrypt(key)` and a client interface `C3` with a method `requestKey()` which should be served outside of the composite and which returns an encryption key. As we will show in the behaviour specification, the component who claims itself as the leader requests the encryption key by invoking `requestKey()` on its client interface, and performs the encryption once the key is obtained.

All four primitives have the same set of attributes. They have the `message(...)` method implementing the leader election algorithm and a list of methods to access local attributes. `Comp4` has an additional interface providing the method `runPeterson()` which triggers the election process. `Comp1`, `Comp2`, and `Comp3` are implemented by `Class0` while `Comp4` uses `Class1` that extends `Class0` with `runPeterson()` method. Initially, the components should have different default values of attribute `max` and `cnum`. `cnum` is a static unique identifier of a component. To specify the values of those attributes for every component individually, we define them in the **Attributes** field represented as a green box in each primitive definition.

3.2.3 Behaviour specification

State machine diagrams UML state machine diagrams are used for behaviour specification in VerCors. Each state machine defines the behaviour of a single method of a UML class.

A state machine has a set of states connected by transitions. In order to make the further analysis less complicated, we rely on flat state machines (they do not have

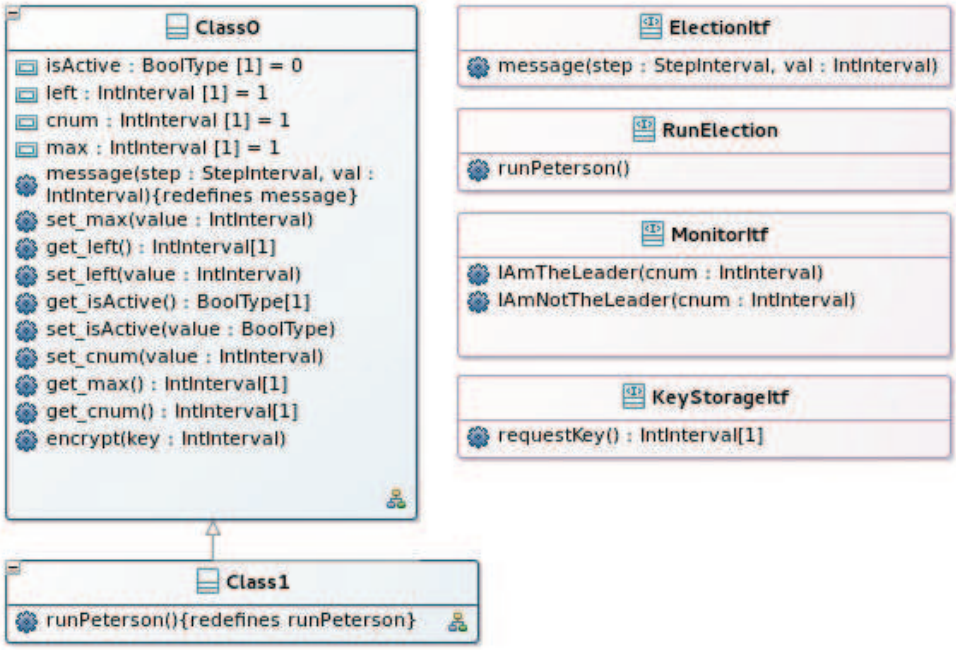


Figure 3.5 – VerCors class diagram

hierarchical states) with only one region each. A state stores its name, while the logic code is specified on the transitions. The UML specification does not provide any strict syntax for the state machine labels, but since we would like to be able to analyse component behaviour and to perform the model-checking, we have to define a specific syntax. A transition has a label of the form `[guard]/action1....actionN` where `guard` is a boolean expression and an `action` is an assignment or a method call. An expression can be constructed from variables, constant values (integer and boolean values, enumeration elements), access to array elements, and arithmetic and logical operators. An assignment includes a name of a variable (or an array element) whose value is modified, and an expression or a method call whose value is copied into the variable. The syntax for a method invocation is `owner"."method_name"("possibly list of argument)")"` where an `owner` is either a name of a client interface in the case of a remote method invocation, or "`this`" if the called method is local to the component. When constructing the grammar we tried to keep it coherent with the UML specification, expressive, and simple enough so that the labels can be analysed easily.

State machine transitions should not include any variable declarations, instead, the local variables of a state machine must be declared in a special area. For each variable the user can specify its name and its type. A state machine has access to its own local variables, to the client interfaces and to the local methods of the component which behaviour the state machine describes; the value of the component attributes

can be accessed only through getters and setters.

The labels prefixed by "//" are ignored during model-checking and generated in the executable code. The programmer can use them in order define instructions which cannot be analysed by VerCors, and which do not have impact on the computations but serve for monitoring purposes. We often used them in our experiments in order to lot messages from the generated Java program. However, the programmer should be very careful with such kind of instructions because VerCors does not check whether they have impact on the computations. Hence, if they, in fact, influence the control flow, VerCors cannot guarantee the model-checked properties in the generated code.

The usage of futures is transparent: the designer does not need to specify explicitly which variables are futures; whenever a state machine invokes a method on a client interface and assigns the result to a given variable, the variable is interpreted as a future. This set of constructs is sufficient to encode any behaviour of distributed objects; the control structures (like do-while, if-else) have to be encoded as guards on transitions.

The use-case example Figure 3.6 illustrates the state machine of the `message` method of Peterson's leader election algorithm. It uses six variables where *step* and *val* are input parameters of the method. The initial state is illustrated with a blue circle (`Initial`). First, `Choice6` checks the phase of the election algorithm. If the algorithm is in the preliminary (zero) phase either the component is active – it already participates in the election – or the component triggers the election process on its neighbour and performs the preliminary phase described in Section 3.2.1. If it is not the preliminary phase, either the component is passive and the message is forwarded to the neighbour `[isActive==false]/C1.message(step,val)`, or the actions of the state machine correspond to the two cases $M(1, x)$ or $M(2, x)$ depending on the value of *step* (see Section 3.2.1).

As it was mentioned earlier, to illustrate future-based communications, we extend our use-case as follows. If a component decides to become the leader, it sends a `requestKey()` invocation on its client interface (see the transition from `State10` to `State12`). The request is forwarded to outside of `Application`. Then, the component claims itself as the leader by sending an `IamTheLeader(cnum)` request. Finally, the component calls its local method `encrypt(key)` using the result of `requestKey()` as a parameter. The component should be able to claim itself as the leader before it receives the result of `requestKey()`. However, it cannot execute `encrypt(key)` if the `key` is not obtained. The VerCors user does not need to explicitly model future-based communications. Whenever a state machine has a non-void client method invocation, it is interpreted as a future-based one.

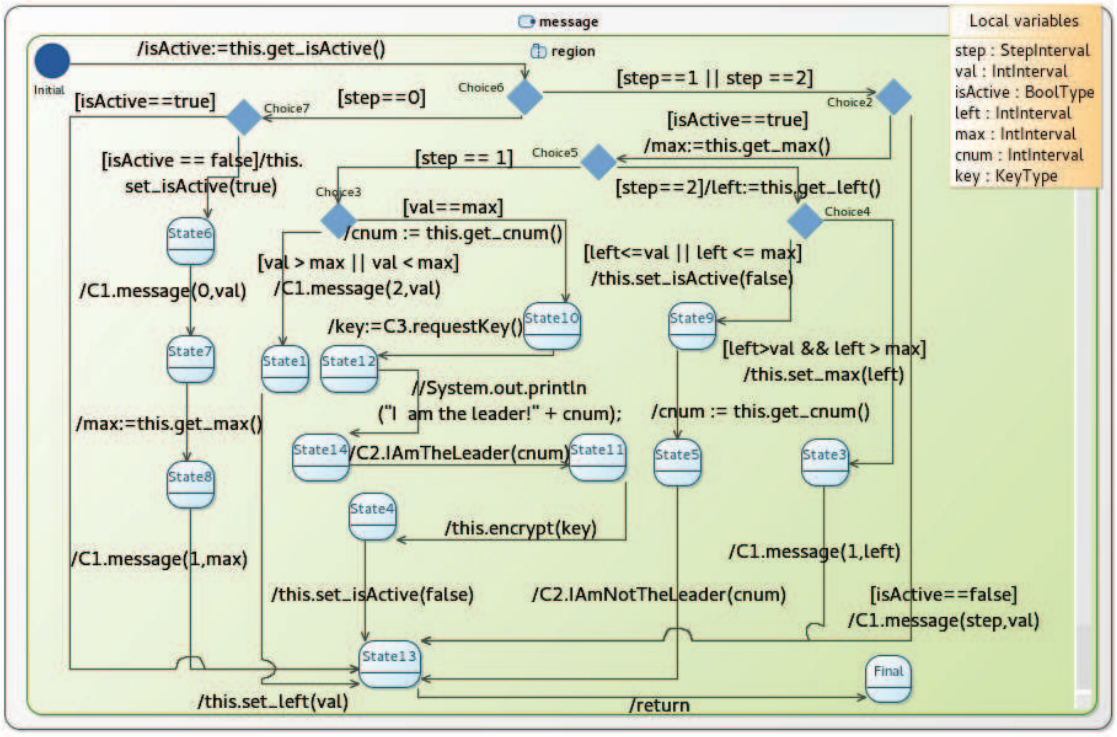


Figure 3.6 – State machine diagram

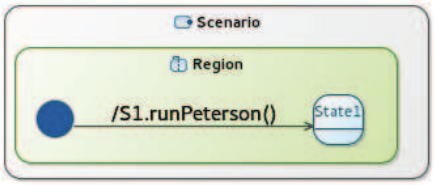


Figure 3.7 – Scenario state machine

Scenario specification

The user can additionally model the calls that the system will receive from its environment. For this purpose, the user specifies a state machine which has its own local variables and access to the server interfaces of the modelled application root component. A scenario can have loops and choice states as any other state machine, but it does not use the wait-by-necessity mechanism. We modelled a very simple scenario (in Figure 3.7 for our use-case example which invokes the method `runPeterson` triggering the election process once.

3.3 The architecture of VerCors

VerCors is implemented as a set of plug-ins for Eclipse; its architecture is illustrated in Figure 3.8. The modules of the platform can be divided into four categories based

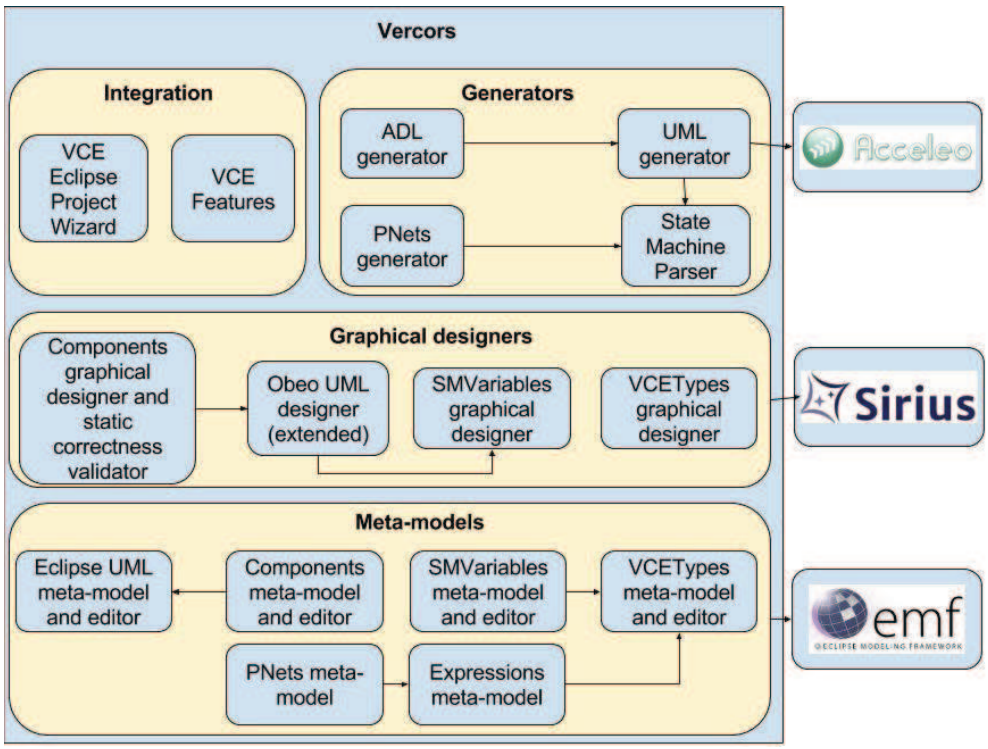


Figure 3.8 – Architecture of VerCors

on their functionality: meta-models with their EMF editors, graphical designers, generators, and integration plug-ins.

Meta-models. VerCors relies on six meta-models based on the EMF ecore technology. The **Components** meta-model is used for the component-based system architecture specifications. Its structure reflects the GCM components structure. The Components meta-model references the **Eclipse UML** [54] meta-model for the UML classes, UML interfaces and components behaviour specification (state machines). Additionally, we implemented the **SMVariables** meta-model for the variables declaration on the UML state machines. The types used by the state machine variables and UML method signatures are based on the **VCETypes** meta-model. Its root element *VCEType* extends UML *Type* which allows using the types declared by the user in the specification of the UML elements. The **pNets** meta-model is used for the pNets construction. Finally, the pLTS' labels are based on the **Expressions** meta-model. Its structure reflects the grammar of the UML state machine labels.

Graphical designers. The four graphical designers provided by VerCors are fully based on the Obeo Designer platform and rely on the meta-models described above. The core editor is the **Components** graphical designer where the user can graphi-

cally specify the architecture of his/her application. The component designer includes a **static correctness validator** which checks the correctness of the user-defined models with respect to a set of rules formalised in Chapter 4. The **Obeo UML** graphical designer is integrated into VerCors and can be used to define classes that implement components, UML interfaces that define signatures of methods of component interfaces, and state machines that specify component behaviour. Obeo UML designer includes a number of other UML diagram editors (e.g. Use-case and Activity diagrams). We extended Obeo UML state machine diagram editor with tools and graphical representation for the **Variable declarations**. Finally, **VCETypes** designer can be used for the specification of the types built from integer intervals, enumerations, records, arrays of fixed size, and boolean.

In addition to the graphical designers, there exist so-called EMF editors that represent a model as a tree-like structure and allow its modification. We generated the EMF editors for those structures that can be edited by the users of VerCors (i.e. components, state machine variables, and VCETypes). An EMF editor for the UML models is included in the UML Eclipse plug-in.

Generators. The core part of the VerCors platform is the GCM/ADL+Java generator which produces the implementation code of a modelled system and a pNet generator that constructs the input for the model-checker. Both construction processes involve the analysis of a component behaviour modelled with UML state machines. More precisely, the behavioural instructions are specified as a state machine labels, and in order to interpret them, both generators invoke a dedicated **Parser** which takes a state machine, parses all its transition labels in the context of a given component, and returns a map from a state machine transition to an instantiation of the `Expression` meta-model classes corresponding to the parsed label. The context is used to establish references to the signatures of the methods invoked by the state machine. We implemented the parser using the combination of the Cup [67] and JFlex [68] technologies. They allow one to specify textually the BNF grammar of the parsed text and to map the grammar symbols and expressions into a sequence of actions which will be performed each time when the parser recognises a symbol or an expression. The actions in our case include the instantiation of the `Expression` meta-model classes. From the given specification, Cup and JFlex produce the Java code of the parser.

The **ADL generator** takes a Component diagram and a package name as an input and produces an XML-based (GCM/ADL) file with the given architecture. The package name corresponds to the package where the Java classes and interfaces

will be produced. Then, the ADL generator invokes a **UML generator** that analyses the UML, VCETypes, and Component models and uses Acceleo templates in order to produce Java classes and interfaces. Both generators are explained in Section 5.3. For every set/get method of a UML class, the UML generator produces the corresponding template-based Java code. For every method which behaviour is defined by a state machine diagram, UML generator translates the parsed version of the state machine into Java.

The **PNets generator** takes the following input: component architecture, referenced UML elements, scenario state machine if there is any, queue size for each component, and interactions that should be hidden during model-checking. Then, it processes the input as follows:

1. **Pre-processing**. The pre-processor analyses each composite component in the model being generated and gathers auxiliary information. For each server interface it finds the sub-component that will process the requests. For every client interface it finds the sub-components that can send the request. Then, the state machine Parser is invoked to parse labels of all state machines of the primitive components and to gather information about local and remote methods invoked by each state machine.

2. **PNets generation**. Starting from the root component, a pNets generator recursively produces a pNet encoding the behaviour of each component. More precisely, for a composite, it generates pLTSs of internal processes (body, queue, etc), produces a set of synchronisation vectors, and triggers the pNet generation for each subcomponent. For a primitive, it produces pLTSs of internal processes and a set of synchronisation vectors. The formalisation and the implementation details are given in Section 5.1. The scenario state machine is also translated into a pLTS. Synchronisation vectors of the root component include synchronisation with the scenario.

3. **Fiacre generation**. Every constructed pLTS is translated into a `.fiacre` file. This and the following two generators are presented in Section 5.2.

4. **EXP generation**. A set of synchronisation vectors of each pNet is translated into an EXP file.

5. **Auxiliary scripts generation**. For every pNet we generate a script assembling its sub-nets into a common structure with respect to the synchronisation given in the corresponding `.exp` file. The scripts also hide communications that should not be observed during model-checking. More precisely, VerCors

generates one `.svl` file for each pNet encoding the behaviour of a component which should be model-checked. The script does the necessary renaming in the sub-nets, invokes EXP.OPEN in order to construct an automaton, hides some of the communications in the generated LTS, and finally reduces the state space to obtain the final model of the component behaviour. In addition, VerCors creates a `.sh` file which calls the Flac compiler on each generated `.fiacre` file and triggers the execution of each produced SVL script. The order in which the instructions are executed is important: we have to make sure that when an SVL script corresponding to a given component invokes EXP.OPEN to construct the component behaviour, all automaton synchronised by the `.exp` file have already been built. Hence, the construction should start from components at the lowest levels of hierarchy, and the `.svl` file corresponding to the root component should be invoked in the last step.

Integration. Finally, integration modules are used to integrate VerCors in Eclipse. The **VCEWizard** plug-in implements a wizard creating a VerCors project with the Component, UML and VCETypes model files and one diagram illustrating each model. The user can then add other models and diagrams. The **VCE Features** module makes VerCors installation/update accessible via the standard Eclipse plug-in installation/update wizard.

3.4 Discussion

The approach and the software platform presented in this dissertation are the result of not only one doctoral work but rather more than ten years of experience of the researchers and engineers involved in the project. In this section we discuss some of the choices that we had to make while designing and implementing the current version of VerCors based on our own experience and on the previous works.

On the core functionalities and the workflow. The most interesting part of the VerCors workflow is the integration between the front-end editor and CADP. It involves two main steps: the construction of pNets and the generation of .fiacre and .exp files. Using pNets as an intermediate format has already been discussed in [19] and we still believe that it is highly beneficial for two main reasons. First, because it does not limit our framework to finite systems: as a parameterised structure the pNets can represent models with infinite state-space. Hence, in the future we can experiment with translating them into an input for the infinite state-space model-

checkers. Second, the gap between the GCM and UML models designed in the front-end and the networks of communicating automata accepted by CADP is very large and the pNets are able to fill this gap: they are at the same time close to the automata accepted by CADP and adequate for modelling the semantics of GCM components as discussed in Section 2.2.4.

On the graphical formalisms. When designing the specification formalism of the front-end VerCors editors we targeted two essential features of the specification language: it should be user-friendly and it should be easy to learn.

First, we had to choose whether the core specification formalism should be textual of graphical. While several textual languages for GCM components have been already defined in [69, 70, 71] we decided to extend and implement the existing graphical formalism as we believe that it is more illustrative and more user-friendly. However, we still plan to develop a tool for reverse-engineering the ADL description into the graphical models. Another question is the level of details: what should be illustrated on the diagrams and what should not, so that the diagrams are not overloaded but at the same time provide all necessary information. In the current version our intuition is to show all those elements that are involved both in the model-checking and in the code generation, and to keep the details necessary only for one of the phases for the dedicated wizards (e.g. queue size for the verified components, the name of the package for the implementation classes). In the perfect case we would like the user to be able to switch between several viewpoints on the model: one dedicated to the model-checked and another one illustrating the generated implementation.

The second challenge was designing such a specification language that could be easily mastered by the programmers. An obvious solution would be relying on a subset of UML models, but it appeared to be difficult to adapt them for the specification of the GCM component architecture. There are several important differences between GCM components and the meta-model of UML composite structures. In particular, UML components have bidirectional ports comprising input and output interfaces, while GCM has interfaces (i.e. the same interface cannot both emit and receive requests). Another difference is that the structure of GCM components is much richer than UML composite structures, and we would have had to extend it with many concepts (multicast/gathercast interfaces, membrane, etc). Hence, it was decided to create a DSL for the GCM components that would reuse some of the UML elements. Indeed why would we invent our own formalism for the specification of the primitive's implementation classes while there already exist UML classes whose notation is well-known among the software engineers?

The version of VerCors presented in this thesis is actually the first version that includes UML diagrams integrated with the GCM DSL; and how exactly the UML elements should be used for the GCM architecture and behaviour specification was not so obvious at the beginning. For example, in one of the first prototypes we used classes to define the signatures of the GCM interfaces. Soon, we realised that using UML interfaces for this purpose is much more natural. Another example could be the choice of the formalism for the behaviour specification, and the possible solutions could be using sequence, activity or state machine diagrams. The sequence diagrams are good for illustrating the message exchange between processes while we wanted to model the behaviour of each component separately. Moreover, the sequence diagrams are quite far from the pLTSs and this could cause difficulties for the generation of the pNets. This is why we decided that the sequence diagrams are not a good choice. Still, it could be interesting to automatise their extraction from the conceptual models designed in VerCors in order to illustrate the communication between the processes. Regarding the choice between the activity and state machine models, it is still discussable as the two representations are very close to each other. Another question was whether we should model one state machine per primitive or one state machine per method. We chose the second option for two reasons. First, it introduces better modularity in the code and allows using the same state machine for different components. Second, it is more coherent with the further transformation steps: one state machine will be translated into one pLTS and one Java operation.

On the platform architecture and implementation. The choice of the underlying implementation platform was not so obvious at the beginning. In 2008 [19] it was already decided to develop VerCors on top of the Eclipse Modeling Project and the Graphical Modeling Framework as they are integrated in the popular Eclipse IDE and provide rich infrastructure for developing graphical editors. It was clear that the VerCors platform was going to be large and feature-rich, and maintaining it only with the help of GMF would require significant effort. Hence, we were looking for some additional technology that would facilitate the creation and maintenance of our model-driven environment. Also, we did not want to implement the UML designer from scratch as there already existed several of them on top of Eclipse. At the beginning of 2013 we experimented with implementing a prototype of the VerCors platform on top of the Papyrus Modeling environment [72]. The idea seemed to be good as the framework already had a UML designer and we even found a tutorial for extending it with the custom diagram editors. However, after a couple of months of experiments we realised that implementing VerCors on top of Papyrus would be fea-

sible but complicated. At that time, our colleague Julien DeAntoni introduced us the Obeo Designer framework which was exactly what we needed: an EMF-based tool for developing graphical modellers with an open-source UML designer implemented on top of it. As we mentioned in Section 2.5.3 we could benefit from multiple advantages and features of the technology, although, it had the only drawback - its license was not free for the industrial users. Anyway, we started the development of VerCors on top of Obeo Designer under academic license and in 2014 we ported it to the recently released open-source version of Obeo Designer which is called Sirius.

In this chapter we made an overview of the VerCors platform: we discussed its capabilities, the basic workflow and the implementation choices. We present in the following chapters each of the functionalities and the underlying theory, starting by the formalisation and static verification of the GCM-based application architecture in Chapter 4. Then, in Chapter 5 we formally define how a pNet model encoding the behaviour of GCM components can be generated from the component architecture and server method behaviour specifications. Then, we explain how the pNet generation process is implemented in VerCors and how the produced pNets serve as an input for the model-checker. We also discuss in Chapter 5 the implementation code generation from VerCors. Finally, in Chapter 6 we explain the construction of pNets and the generation of the executable code for the advanced features of the GCM components such as attribute controllers, group communications, and the non-functional aspects.

Chapter 4

Formalisation of the static correctness rules for component architecture

Contents

4.1 Formalisation of component structure 60

4.2 Auxiliary functions 61

4.3 Interceptors 63

4.4 Well-formed component architecture 65

4.4.1 Core 65

4.4.2 Non-functional aspects 68

4.4.3 Collective communications 70

4.4.4 Additional rules 71

4.5 Properties 71

4.6 Architecture static analysis in VerCors 74

4.7 Discussion and Related work 74

Before implementing an application, a programmer has to make sure that its conceptual model is statically correct as this can prevent from issues at the deployment and execution stages. We would like to help the developers of component-based systems to check that the component assembly is statically well-defined and satisfies a range of properties such as correct typing, separation between business logic and application management, proper component encapsulation. For this purpose, we gathered a number of constraints that should be satisfied by a component assembly.

The constraints are based on the existing literature [73, 74, 69], on the ones implemented in the previous versions of VerCors, on the experience of the GCM-based systems engineers, and on our own experience. We specify the validation predicates and discuss how we implement their validation in this chapter.

We start by the first contribution - the formal definition of the core elements of a component-based application architecture Second, we introduce the auxiliary functions that are used by the well-formedness specification. Next, we describe in details the formalisation of so-called interceptors as their definition is a bit more complex than for the other elements and it has not been given in the previous works. Then, we introduce the notion of the well-formed components and explain what kind of properties are guaranteed for an application that satisfies the given constraints. We briefly discuss how the architecture well-formedness is checked in VerCors. We make a brief overview of the works related to the specification of correct components composition. Finally, we discuss how the other component-based frameworks could benefit from our formalisation.

The core of the contribution presented in this chapter has been published in [75]. We extend it with a few validation constraints that are necessary for the further verification.

4.1 Formalisation of component structure

In this section we define the core elements of GCM, namely: Interfaces, Components and Bindings. Their formal definition is given in Table 4.1. We denote $A_i^{i \in I}$ a set of elements A_i indexed in a set I. The formal definition is not provided for some elements of the table, because they are the terminal symbols. Such elements are presented in a different font (e.g. Name, Type, NF).

Every server (*SItf*) or client (*CItf*) interface is characterised by three attributes: *Name* is the name of an interface; *MSignatures* represents the methods which can be served by a server interface or called by a client interface; *Nature* defines if it is a functional or a non-functional interface. Please, note that for the sake of simplicity we omit the cardinality attribute in the core definitions and we assume all interfaces to be singletons. Nevertheless, we will introduce the cardinality and show its impact on the well-formedness specification in Section 4.4.3.

A component is described by its name, the sets of external client and server interfaces, and a membrane. A primitive additionally includes a set of local methods. In practice, a primitive also stores a list of attributes but we omit them in the formal specification as the attributes are not involved in the static validation. A composite

Element	Formalisation
Server interface	$SItf ::= (\mathsf{Name}, MSignature_i^{i \in I}, Nature)_S$
Client interface	$CItf ::= (\mathsf{Name}, MSignature_i^{i \in I}, Nature)_C$
Interface	$Itf ::= SItf \mid CItf$
Method signature	$MSignature ::= \mathsf{MName} \times \mathsf{Type} \rightarrow \mathsf{Type}$
Primitive	$Prim ::= \mathsf{CName} < SItf_i^{i \in I}, CItf_j^{j \in J}, M_k^{k \in K}, Membr >$
Composite	$Compos ::= \mathsf{CName} < SItf_i^{i \in I}, CItf_j^{j \in J}, Membr, Cont >$
Component	$Comp ::= Prim \mid Compos$
Content	$Cont ::=< SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} >$
Binding	$Binding ::= (QName, QName)$ $QName ::= \mathsf{This.Name} \mid \mathsf{CName}.\mathrm{Name}$
Nature	$Nature ::= \mathsf{F} \mid \mathsf{NF}$
Membrane	$Membr ::=< Comp_k^{k \in K}, Binding_l^{l \in L} >$

Table 4.1 – The formalization of GCM architecture

component includes a content which is defined by the sets of internal client and server interfaces, sub-components, and bindings located inside the content.

A binding is described as a couple of names defining its source and target interfaces. Each qualified name consists of two parts: the first part defines the container of the interface, either the name of a sub-component or the identifier *This*; the second part is the name of the interface itself.

Non-functional aspect. The *Nature* property of an interface can be either *functional (F)* or *non-functional (NF)*. A membrane is the part of a component which is responsible for the non-functional aspect of an application. Its formalisation is given at the bottom of Table 4.1. The bindings ($Binding_l^{l \in L}$) are the connections between the interfaces in a membrane. A membrane can contain a set of sub-components ($Comp_k^{k \in K}$). Among them, so-called interceptors have a special use. They are recognised in this set essentially from their binding pattern. Section 4.3 will focus on the definition and identification of interceptors. All the other components in a membrane are component-controllers.

4.2 Auxiliary functions

Based on the formal definitions given in the previous section, we will specify a set of rules for the well-formed components in Section 4.4, but first we should introduce the auxiliary functions that will be used by these rules. The functions are given in

Function name	Function Definition
Sym	$Sym : Itf \rightarrow Itf$
Itf	$GetItf : (Membr \mid Cont \mid Comp) \rightarrow Itf$ $GetItf(X : Comp \mid Cont) ::= SItf(X) \cup CItf(X)$ $GetItf(X : Membr) ::=$ $Sym\Big(Itf(Parent(X)) \cup Itf(Cont(Parent(X)))\Big)$
GetSrc	$GetSrc : Binding \times (Membr \mid Cont) \rightarrow Itf$ $GetSrc((Src, Dst), ctnr) ::= itf$ s.t. $Name(itf) = Name(Src) \wedge$ $\begin{cases} itf \in Itf(ctnr) \wedge Role(itf) = \mathsf{C}, \\ \quad if\ Src = \mathsf{This.Name} \\ itf \in Itf(comp).(comp \in ctnr \wedge Name(comp) = \mathsf{CName}), \\ \quad if\ Src = \mathsf{CName.Name} \end{cases}$
GetDst	$GetDst : Binding \times (Membr \mid Cont) \rightarrow Itf$ $GetDst((Src, Dst), ctnr) ::= itf$ s.t. $Name(itf) = Name(Dst) \wedge$ $\begin{cases} itf \in Itf(ctnr) \wedge Role(itf) = \mathsf{S}, \\ \quad if\ Dst = \mathsf{This.Name} \\ itf \in Itf(comp).(comp \in ctnr \wedge Name(comp) = \mathsf{CName}), \\ \quad if\ Dst = \mathsf{CName.Name} \end{cases}$
Parent	$Parent : (Itf \mid Comp \mid Membr \mid Cont) \rightarrow Membr \mid Cont \mid Comp$

Table 4.2 – Auxiliary functions

Table 4.2 and explained in details below.

First, we use auxiliary functions providing access to the attributes of the interfaces and components. For example, the $Nature(itf : Itf)$ function returns the nature of an interface, $Binding(membr : Membr)$ returns the set of bindings in a membrane. The definitions of such functions are straightforward and we omit them in Table 4.2.

Second, the symmetry function (Sym) takes an interface as an input and returns an interface with exactly the same properties but with an opposite role: a client $(\ldots)_C$ interface becomes a server $(\ldots)_S$ one and symmetrically.

Third, we introduce a *GetItf* function. It takes a container (a component, a membrane or a content) as an input and computes the set of interfaces stored by it. If its argument is a component or a content, then the result is the union of its server and client interfaces sets. The membrane of a component can also have internal interfaces, even though they are not declared explicitly. The set of internal interfaces of a membrane is represented by the union of two other sets: all the external interfaces of the component containing the membrane and all the internal interfaces of the content inside the component. All the interfaces are taken with the same properties, but with an opposite role (e.g. a server interface becomes a client one and vice-verse).

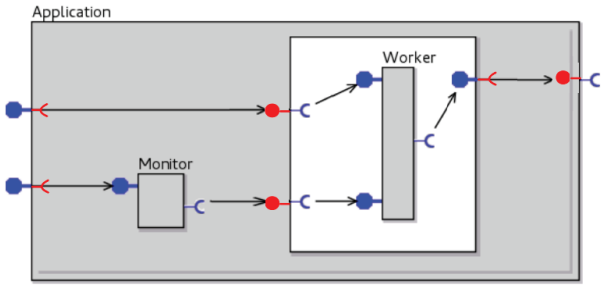


Figure 4.1 – Internal interfaces of a membrane

Figure 4.1 illustrates the (implicit) internal interfaces of a membrane, displayed in red color. In terms of our formal definition, the interfaces of a membrane are obtained by computing the set of the symmetric of the interfaces belonging to the component containing the membrane and its content.

Finally, most of the consistency rules dealing with the bindings will use the auxiliary functions *GetSrc* and *GetDst*. They are able to retrieve the Interface objects which are respectively the source and the destination ends of a binding. If *GetSrc* and *GetDst* functions are applied to a binding inside a membrane, then they may return an internal interface of the membrane or an external interface of one of its sub-components.

Conversely, some rules use the *Parent* auxiliary function, that recovers the container (component, membrane or content) of an interface, a component, a content or a membrane. The parent of a sub-component in a composite can be either the membrane of the content.

4.3 Interceptors

The separation between the functional and non-functional parts of an application is important for the safety and re-usability of software components. However, a clean interaction between these two concerns is highly desirable because they can often influence each other. We formalise the notion of interceptors which are special components used for the interactions between functional and non-functional elements. Interceptors can observe functional invocations and trigger a reaction of the control part of the component. In the other direction, non-functional components can influence the functional behaviour by modifying its behaviour in two ways: either through reconfiguration, i.e. modification at runtime of the component architecture, or by changing parameters, i.e. component attributes, that will influence the functional behaviour. The formalisation of component architecture with interceptors has never been provided before. In this section, we, first, we introduce the non-formal

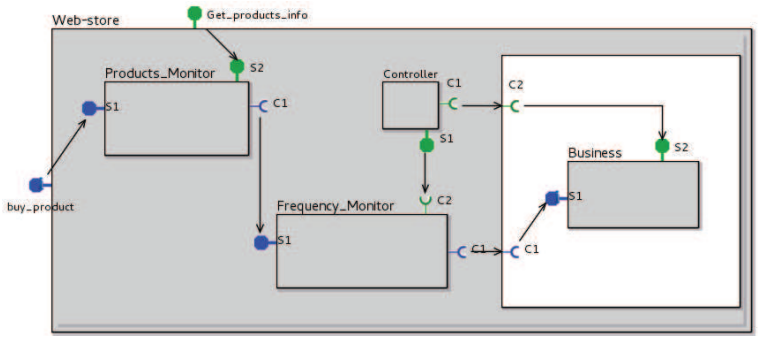


Figure 4.2 – An input chain of interceptors

definition of an interceptor. Then, we present a predicate recognising the interceptors among the other sub-components of a membrane.

An interceptor is a functional component inside a membrane. It is connected to one external and one internal functional interfaces of the membrane's parent. An interceptor "intercepts" a functional call that goes from outside to inside of the membrane (input interceptor) or vice versa (output interceptors). The interceptors are used to monitor an application and its functional calls. The only functional activity of an interceptor should be to intercept and forward the functional calls through its functional client and server interfaces. All the other actions are performed through non-functional interfaces. To allow more modularity in the design, interceptors can be assembled in *chains* inside the membrane. Interceptors in a chain must all be either input or all output; we shall speak of input chains and output chains. An example of an input chain is illustrated at Figure 4.2; here, the chain is formed by two interceptor components: `Products_Monitor` and `Frequency_Monitor`. By contrast, `Controller` is a component-controller: it does not intercept any functional call but communicates with one of the interceptors and with the component in the content.

In principle, it would be possible to relax this definition, and allow for more general interceptor structures, e.g. including some parallelism in the form of multicast client interface in the "chain", allowing more efficient processing. However it is not clear whether this would be useful for real applications, and this is not implemented in the GCM/ProActive middleware, so we prefer to keep the (relatively) simple form in our formal definition.

In order to distinguish formally the interceptors from the other components, we define a predicate *IsInterc* (see Table 4.3). It takes a component and a membrane as an input and returns *true* if the component belongs to a chain of interceptors inside the given membrane. An interceptor chain consists of one or several interceptors. *IsInterc* uses a predicate *IsIntercChain* which identifies if a given sequence of K components is a chain of pipelined interceptors inside the given membrane. $IsIntercChain$ predicate

checks the following features of the indexed set of components given as input:

- all the components are inside the membrane;
- all the components have exactly one functional server and one functional client interface (they can have other non-functional interfaces);
- a functional call can go through the sequence of components. More formally, for any $k \in 2...K$ there is a binding connecting client functional interface of the component number $k-1$ and server functional interface of the component number k;
- the first component of an input chain intercepts a functional call from an external functional interface to the content while the last component forwards the functional call to the content or vice versa for an output chain.

Predicate *IsIntercChain* uses two auxiliary functions: $GetSItf_F(comp)$ (resp. $GetCItf_F(comp)$) returns the sets of all *functional* server (resp. client) interfaces of component *comp*.

The predicate *IsExtEnd* checks, for an input interceptor chain, whether the first interceptor in the chain is connected to a server functional interface of the parent component or, for the output interceptors, whether the last one is connected to a client functional interface of the parent component. Predicate *IsIntEnd* is the symmetric, it checks connection with the content. However, the content is not known for a primitive component; in that case the interfaces of the content are computed by symmetry of the external (functional) interfaces of the component.

4.4 Well-formed component architecture

Now, we can specify the well-formedness requirements. First, we introduce the core rules and predicates used for the definition of the well-formedness; then, we focus on the non-functional aspects and on the collective communications.

4.4.1 Core

Table 4.4 formalises the auxiliary predicates which are used for the definition of the following constraints:

- Component naming constraint ($UniqueCompNames$): all the components at the same level of hierarchy must have different names. This restriction is due the fact that components are referenced by their name; typically, *QName* (see

Predicate name	Predicate Definition
IsInterc	$IsInterc(comp : Comp,\ membr : Membr) \Leftrightarrow$ $\exists ic. IsIntercChain(ic, membr) \wedge comp \in ic;$
IsIntercChain	$IsIntercChain(\{i_1 ... i_K\} : \text{set of } Comp,\ membr : Membr) \Leftrightarrow$ $\{i_1 ... i_K\} \subseteq Comp(membr) \wedge \exists SI_1 ... SI_K, CI_1 ... CI_K.$ $(\forall k \in \{1...K\}).\ (GetSItf_F(i_k) = \{SI_k\} \wedge GetCItf_F(i_k) = \{CI_k\}) \wedge$ $(\forall k \in \{1...K-1\}).(\exists b_k \in Binding(membr).$ $GetSrc(b_k, membr) = CI_k \wedge GetDst(b_k, membr) = SI_{k+1}) \wedge$ $\exists b_0, b_K \in Binding(membr). \exists I_0, I_K : Itf.$ $\big((IsExtEnd(I_0, b_0, SI_1, membr, I_0) \wedge IsIntEnd(CI_k, b_K, I_K, membr, I_K)) \vee$ $(IsIntEnd(I_0, b_0, SI_1, membr, I_0) \wedge IsExtEnd(CI_k, b_K, I_K, membr, I_K))\big)$
IsExtEnd	$IsExtEnd(CI : CItf,\ b : Binding,\ SI : SItf,\ membr : Membr,\ I : Itf) \Leftrightarrow$ $GetSrc(b, membr) = CI \wedge GetDst(b, membr) = SI \wedge Nature(I) = \mathsf{F} \wedge$ $Sym(I) \in GetItf(Parent(membr))$
IsIntEnd	$IsIntEnd(CI : CItf,\ b : Binding,\ SI : SItf,\ membr : Membr,\ I : Itf) \Leftrightarrow$ $GetSrc(b, membr) = CI \wedge GetDst(b, membr) = SI \wedge Nature(I) = \mathsf{F} \wedge$ $If\ IsComposite(Parent(membr))\ then$ $Sym(I) \in GetItf(Cont(Parent(membr)))$ $Else\ Sym(I) \in Sym(GetItf(Parent(membr)))$

Table 4.3 – Interceptor predicates

Predicate name	Predicate Definition
UniqueCompNames	$UniqueCompNames(Comp_i^{i \in I} : \text{set of } Comp) \Leftrightarrow$ $\forall i, i' \in I. i \neq i' \Rightarrow CName(Comp_i) \neq CName(Comp_{i'})$
UniqueItfNames	$UniqueItfNames(Itf_i^{i \in I} : \text{set of } Itf) \Leftrightarrow$ $\forall i, i' \in I.\ i \neq i' \Rightarrow Name(Itf_i) \neq Name(Itf_{i'})$
BindingRoles	$BindingRoles(b : Binding) \Leftrightarrow$ $Role(GetSrc(b, Parent(b))) = \mathsf{C} \wedge$ $Role(GetDst(b, Parent(b))) = \mathsf{S}$
BindingTypes	$BindingTypes(b : Binding) \Leftrightarrow$ $MSignature(GetSrc(b, Parent(b))) \leq$ $MSignature(GetDst(b, Parent(b)))$
CardValidity	$CardValidity(Binding_l^{l \in L} : \text{set of } Binding) \Leftrightarrow$ $\forall l, l' \in L.\ l \neq l' \Rightarrow$ $GetSrc(Binding_l, Parent(Binding_l)) \neq$ $GetSrc(Binding_{l'}, Parent(Binding_{l'}))$

Table 4.4 – Core predicates

Table 4.1) can be of the form CName.Name; and if two components had the same name the functions *GetSrc* and *GetDst* would not return a single deterministic result. The two contents of two different composite components as well as two membranes are considered to be different name-spaces. A membrane and a content are also different name-spaces even if they belong to the same component.

- Interface naming constraint (*UniqueItfNames*): all the interfaces of a component must have different names. This constraint will be checked separately, for the external interfaces of a component, and for the internal interfaces of a content to allow external and internal interfaces to have the same name. This constraint also ensures the determinacy of the functions *GetSrc* and *GetDst*.

- Role constraint (*BindingRoles*): a binding must go from a client interface to a server interface. The predicate uses a function $Role(I)$ that returns C (resp. S) if I is a client (resp. server) interface.

- Typing constraint (*BindingTypes*): a binding must bind interfaces of compatible types. The compatibility of interfaces means that for each method of a client interface there must exist an adequate corresponding method in the server interface. In other words, if a client interface is connected to a server interface and it wants to call some method, then this method must actually exist on the server interface. In general, a corresponding method does not need to have exactly the same signature as the one required, but can use any sub-typing or inheritance pre-order available in the modelling language. We denote $\leq$ such an order between interface signatures.

- Cardinality constraint, *CardValidity*, ensures that a client interface is bound to a single server one. In other words, there is not two bindings going from the same client interface.

We use the previous constraints to specify a well-formedness predicate, denoted *WF*. It is defined recursively on the component architecture, namely on primitive components, on composite components, and on contents.

A GCM primitive component is well-formed if all its interfaces have distinct names and its membrane is well-formed.

$$\begin{aligned} &\text{Let } prim : Prim = \mathsf{CName} < SItf_i^{i \in I}, CItf_j^{j \in J}, M_k^{k \in K}, Membr >; \\ &WF(prim) \Leftrightarrow UniqueItfNames(SItf_i^{i \in I} \cup CItf_j^{j \in J}) \wedge WF(Membr) \end{aligned} \tag{4.1}$$

Predicate name	Predicate Definition
UniqueNamesAndRoles	$UniqueNamesAndRoles(Itf_i^{i\in I} : set\ of\ Itf) \Leftrightarrow$ $\forall i, i' \in I.(i \neq i' \wedge Name(Itf_i) =$ $Name(Itf_{i'})) \Rightarrow Role(Itf_i) \neq Role(Itf_{i'})$
BindingNature	$BindingNature(b : Binding) \Leftrightarrow$ $CL(GetSrc(b, Parent(b))) =$ $CL(GetDst(b, Parent(b))) = 1 \vee$ $(CL(GetSrc(b, Parent(b))) > 1 \wedge$ $CL(GetDst(b, Parent(b))) > 1)$

Table 4.5 – Non-functional predicates

A GCM composite component is well-formed if all its external interfaces have distinct names, and its content and its membrane are well-formed.

$$\text{Let } compos : Compos = \mathsf{CName} < SItf_i^{i\in I}, CItf_j^{j\in J}, Membr, Cont >; \tag{4.2}$$
$$WF(compos) \Leftrightarrow UniqueItfNames(SItf_i^{i\in I} \cup CItf_j^{j\in J}) \wedge WF(Membr) \wedge WF(Cont)$$

The content of a GCM component is well-formed if all its interfaces have distinct names, all its sub-components have distinct names, all its bindings have a valid cardinality, all its sub-components are well-formed, the role, type, and nature constraints are respected for all its sub-bindings. The nature constraint for the bindings relies on The $BindingNature$ predicate. It is discussed in Section 4.4.2 because it is related to the non-functional aspects.

$$\text{Let } cont : Cont =< SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_l^{l\in L} >; \tag{4.3}$$
$$WF(cont) \Leftrightarrow \begin{cases} UniqueItfNames(SItf_i^{i\in I} \cup CItf_j^{j\in J}) \wedge \\ UniqueCompNames(Comp_k^{k\in K}) \wedge \\ CardValidity(Binding_l^{l\in L}) \wedge \forall k \in K.WF(Comp_k) \wedge \\ \forall B \in Binding_l^{l\in L}.BindingRoles(B) \wedge \\ BindingTypes(B) \wedge BindingNature(B) \end{cases}$$

The well-formedness of a membrane is only significant for the non-functional aspect, it is defined below.

4.4.2 Non-functional aspects

In this section we define the static semantic constraints ensuring safe composition of the non-functional part of a component-based application. The correctness of a membrane relies on the two predicates defined in Table 4.5:

- Interface naming constraint ($UniqueNamesAndRoles$): if there are two interfaces with the same name in a membrane, then they must have different roles. We recall here that the interfaces of a membrane are not declared explicitly but computed as the symmetry of the interfaces belonging to the parent component and its content. This is as slight relaxation from the *UniqueItfNames* rule of the general case: we generally want to allow corresponding external/internal interfaces pairs of opposite role to have the same name. This also ensures compatibility with the original Fractal model, where internal interfaces were implicitly defined as the symmetric of external ones.

- Binding nature constraint($BindingNature$) is a rule imposing the separation of concerns between functional and non-functional aspects. We want the functional interfaces to be bound together (only functional requests will be going through these), and non-functional interfaces to be connected together as a separate aspect. This is simple to impose in the content of a composite components, but a little trickier in the membrane because of the specific status of interceptors.

  The solution is to qualify as functional all the components in a content and all the interceptors, while all the other components in the membrane are non-functional. As mentioned earlier, the interfaces are declared functional or non-functional. From this we compute for each interface a control level ranging from 1 to 3, where 1 means functional; 2 and 3 mean non-functional. The two levels are needed because a non-functional component inside a membrane can have both functional and non-functional interfaces. However, the computed control level of any of them should indicate that an interface belongs to the non-functional aspect because the interface belongs to a non-functional component. Then, the external functional interfaces of non-functional components will have a control level 2 while their non-functional interfaces will have a control level 3. Then the compatible interfaces are either both "1", or both greater than "1". The $ControlLevel$ function is formally defined as:

$$
\begin{aligned}
&CL(X : Comp) ::= \begin{cases} 2,\ if\ Parent(X, context) : Membr \wedge \\ \quad \neg IsInterc(X, Parent(X))) \\ 1,\ else \end{cases} \\
&CL(X : Cont \mid Membr) ::= 1 \\
&CL(X : Itf) ::= \begin{cases} CL(Parent(X)),\ if\ Nature(X) = \mathsf{F} \\ CL(Parent(X)) + 1,\ if\ Nature(X) = \mathsf{NF} \end{cases}
\end{aligned} \tag{4.4}
$$

This constraint on the nature of bindings was already mentioned in [69], we propose here a formal definition that is simpler and more intuitive.

As the last step, we define the well-formedness predicate for a membrane. A membrane is well-formed if all its sub-components have distinct names, the naming constraint is respected for its interfaces, all its sub-components are well-formed, all its bindings have a valid cardinality, and the role, type, nature constraints are respected for all its sub-bindings,

$$\text{Let } membr : Membr =< Comp_k^{k\in K}, Binding_l^{l\in L} >; \tag{4.5}$$

$$WF(membr) \Leftrightarrow \begin{cases} UniqueCompNames(Comp_k^{k\in K}) \wedge \\ UniqueNamesAndRoles(Itf(membr)) \wedge \\ \forall k \in K.WF(Comp_k) \wedge CardValidity(Binding_l^{l\in L}) \wedge \\ \forall B \in Binding_l^{l\in L}.BindingRoles(B) \wedge \\ BindingTypes(B) \wedge BindingNature(B) \end{cases}$$

4.4.3 Collective communications

One of the crucial features of GCM is to enable one-to-many and many-to-one communications through specific interfaces, namely gathercast and multicast.

In order to specify such communications let us add a Cardinality ($Card$) field in the specification of the GCM interfaces. The cardinality can be *singleton*, *multicast* or *gathercast*. We recall here that a multicast can send requests to several target interfaces at the same time; a gathercast can be plugged to multiple client interfaces.

These new interfaces modify the definition of cardinality validity. In particular, the multicast interface allows two bindings to originate from the same client interface. The *CardValidity* is modified as follows:

$$\begin{aligned} CardValidity(Binding_l^{l\in L} : \text{set of } Binding) \Leftrightarrow \forall itf : Itf. \forall l, l' \in L.l \neq l' \\ (itf = GetSrc(Binding_l, Parent(Binding_l)) = \\ GetSrc(Binding_{l'}, Parent(Binding_{l'}) \Rightarrow Card(itf) = multicast) \end{aligned} \tag{4.6}$$

The intended semantics is that an invocation emitted by a multicast interface is sent to all the server interfaces bound to it. Gathercast interface on the contrary synchronises several calls arriving at the same server interface, they do not entail any structural constraint. Indeed, multicast and gathercast interfaces were designed for sending or synchronising several invocations correspondingly. From an architectural point of view, there is no difference between a gathercast and a singleton server interface because both of them can be plugged to several client interfaces. An interceptor

chain should not contain any multicast functional interface because it should transmit invocations in a one-to-one manner.

4.4.4 Additional rules

In order to facilitate the further analysis and in particular the generation of the behavioural models, we add two additional requirements to the well-formed components. First, we require that no binding has the same component as source and destination: there is no binding looping back directly to the same component. This condition is checked by the predicate below:

$$NoLoopBinding(b : Binding) \Leftrightarrow \tag{4.7}$$
$$Parent(GetSrc(b, Parent(b))) \neq Parent(GetDst(b, Parent(b))$$

The second additional rule states that no two bindings going from the same multicast interface reach the same target component; it is expressed by the predicate below:

$$NoEqualTarget(b,b':Binding) \Leftrightarrow GetSrc(b, Parent(b)) = GetSrc(b', Parent(b')) \wedge \tag{4.8}$$
$$Parent(GetDst(b, Parent(b))) = \mathsf{CName.Name} \Rightarrow$$
$$Parent(GetDst(b, Parent(b))) \neq Parent(GetDst(b', Parent(b)))$$

The two additional predicates should be checked for each binding in a container (in a content or in a membrane). This validation is not required by the GCM model but it simplifies the rules for the construction of the behavioural models presented in Section 6.3.

4.5 Properties

The well-formedness definition of the preceding section guarantees that, from an architectural point of view, the specified component assembly is well-formed. It entails some properties both at deployment time and during execution. More precisely, the constraints specified above ensure the following properties:

Component encapsulation. *Bindings do not cross boundaries.* Indeed, *GetSrc* and *GetDst* predicates are only defined in the context of the parent component, for example, the call to *GetDst* and *GetSrc* inside the definition of the *BindingRoles* predicate ensure that both bound interfaces are either internal in-

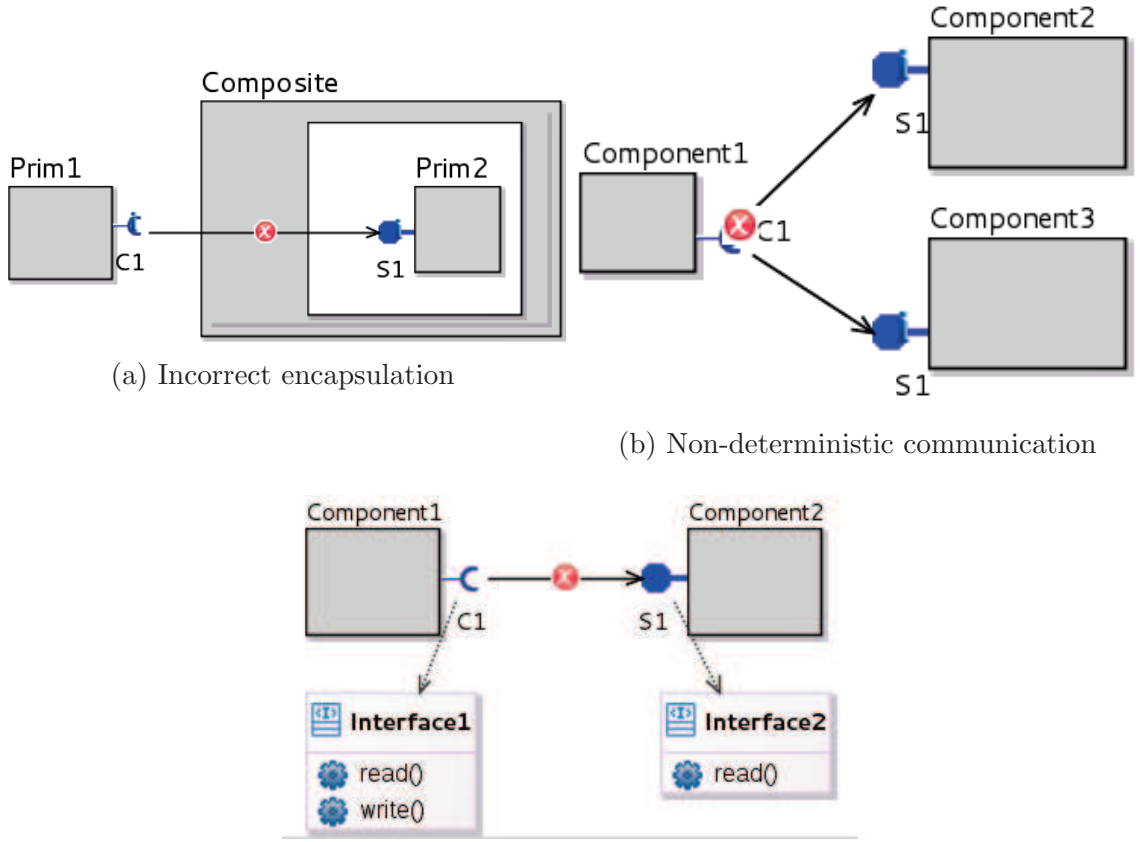
terfaces of the parent component or external interfaces of its sub-components, which guarantees that no binding crosses component boundaries. If the source and the destination interfaces are not have the same context, the predicate returns an error. The property allows preventing issues similar to the one illustrated in Figure 4.3a where a component (`Prim1`) outside a composite communicates directly with a sub-component (`Prim2`).

Deterministic communications. The *CardValidity* predicate guarantees that each singleton client interface is bound to a single server interface, which guarantees that *each communication from a singleton interface is targeted at a single, well-defined, destination*. The predicate helps to avoid a situation illustrated in Figure 4.3b where it is not clear which component (`Component2` or `Component3`) will receive requests from `Component1`. On the other hand, *CardValidity* takes into account collective communications and ensures that in the case of a multicast client, it will be bound to a well-defined set of target server interfaces.

Unique naming. Several predicates ensure the uniqueness of component or interface names in each scope (sub-components of the same component, interfaces of the same component, etc.). This restriction is crucial for *introspection and modification of the component structure*. For example rebinding of interfaces can be easily expressed based on component and interface names.

Separation of concerns. The definition of non-functional aspects ensure that: 1) *each component has a well-defined nature*: functional if it belongs to the content, and non-functional if it belongs to the membrane. 2) *each interface has a well-defined control level*, depending on the component it belongs to and on the nature of the interface. The nature of components and interfaces is defined by the control-level (*CL*) predicate. 3) *Bindings only connect together functional* (resp. non-functional) *interfaces*. 4) We clearly identify *interceptor components* that are the only structural exception to these rules: an interceptor is a component in the membrane that can have a single functional client and a single functional server interface, but as many non-functional interfaces as necessary.

Correct typing. For each request sent by a client interface, there should be the corresponding method on the target server interface which is able to serve it. This is ensured by the predicate *BindingTypes* which checks the compatibility between the method signatures of two interfaces connected by a binding. Figure 4.3c illustrates an example of an architecture where the typing constraint is violated: the issue may occur if at run-time `Component1` invokes method



(a) Incorrect encapsulation

(b) Non-deterministic communication

(c) Incorrect typing

Figure 4.3 – Examples of architecture constraint violations

`write()` on `Component2`, because the former cannot serve the request.

In order to guarantee that the generated ADL deploys correctly, i.e. without runtime error, it is sufficient to ensure that (1) each interface and class the ADL file refers to exists (which is ensured by the generation process presented in Section 5.3.1), that (2) no binding exception will be raised at instantiation time (which is ensured by the well-formedness property and in particular by the determinacy of communications), and that (3) the unique names ensure that the components and interfaces can be manipulated adequately during instantiation. All those arguments ensure that *each ADL file generated by VerCors deploys correctly*, provided the well-formed property is verified by the system.

The properties verified by well-formed components not only guarantee that the ADL generated from a well-formed specification deploys correctly, but also ensure some crucial properties concerning the runtime semantics (deterministic communications, separation of concerns, reconfigurability ...).

4.6 Architecture static analysis in VerCors

All the static semantics constraints defined in the previous sections are encoded in VerCors in order to validate component assemblies. As mentioned in Section 3.3 we rely on Acceleo to implement the check. The technology allows one to define a set of validation rules over a given model in the OCL-based [76] Acceleo language. For complex computations, the language includes a mechanism for external custom Java services invocation which makes the rules specification extremely flexible.

An example of a rule defining the naming constraint for the components is given below:

```
[not self.siblings(Component)->collect(name)->includes(self.name)/]
```

It will be checked for every component in a given conceptual model. Here, `self` is the component on which the constraint is validated; let us denote it C. The function `siblings(Component)` returns the set of all the components in the same container as C. The function `collect(name)` extracts the names of all such components. Finally, `includes(self.name)` checks if there is any component with the same name as C and returns `true` if it finds one.

The given constraint looks very simple as its validation does not involve external Java services invocation. An example of a more complex constraint could be the one ensuring that all bindings connect interfaces of proper nature. Its validation is not trivial as it requires computing the interceptors chains in order to evaluate the control level of each interface.

The graphical editor of VerCors reports about the constraint violations. The elements of the architecture which are not correct are marked with red signs and the error description is given in a standard Eclipse `Problem` panel. Figure 4.4 illustrates an example of a validation result; the architecture contains only one error: there is a binding between two interfaces with incompatible types (`C1` and `S1`). Indeed, if `C1` tries to invoke method `write`, `S1` will not be able to serve it and the application will fail.

4.7 Discussion and Related work

In this chapter we have formalised the architecture of GCM components including the non-functional aspect and interceptors, we have formalised the specification of component well-formedness predicates which ensure that the components are well-encapsulated, the communications are deterministic, and the business-logic of an application is separated from the control part. We fully implement the automatic

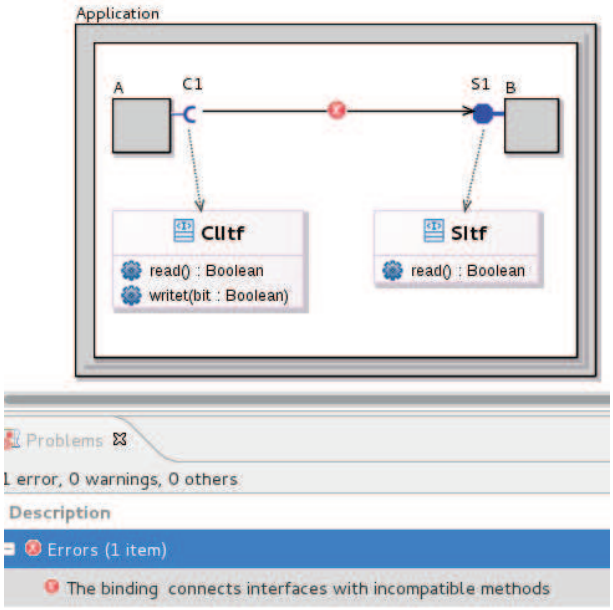


Figure 4.4 – Architecture static correctness validation in VerCors

check of the well-formedness constraints in VerCors.

Even though in this chapter we mainly consider the GCM components, we believe that our formalisation is quite general and the other component models can also benefit from it. For example, SOFA 2.0 [77] which will be discussed in details in Section 7.1 has a structure very similar to the GCM, it features hierarchical composition and componentised component control. Interestingly, similarly to our approach, one of the objectives of SOFA is also to provide formal verification tools for distributed component systems. It would be easy to adapt our formal specification to SOFA 2.0, more precisely: *Micro-controllers* are components dedicated to non-functional concerns, they reside in the membrane and are not hierarchical: to take them into account, we should *restrict the correctness rules for the membrane to only allow primitive components in the membrane*. Also, the SOFA 2 *delegation chains* are chains of interceptors, following exactly the *rules defined in Section 4.3*. Another example could be AOKell [78] – an extension of Fractal specifically for componentising membranes, it is interesting to note that the authors define a notion of control level, which is quite similar to the *ControlLevel* function used in our paper. In this case again, our approach could be used to verify the correct composition of AOKell-based components, and ensure the safe design of AOKell component systems.

The two previous works closest to ours are the formalisation of Fractal in Alloy [79], and the formalisation of GCM in Isabelle/HOL [80]. The first framework focuses on structural aspects in order to prove the realisability of component systems. Except from the core elements (i.e. components, interfaces and bindings), the

authors formalise the standard Fractal controllers (e.g. Lifecycle controller, Binding controller) and the components factory. One of the key goals of the work is to make clear a number of aspects that were left ambiguous in the informal Fractal specification. For example, the authors give a precise definition of the sub-typing relation for the interfaces.

The second work aims at providing lemmas and theorems to reason on component models at a meta-level, and prove generic properties of the component model. In addition to the core elements of a component-based architecture, the authors take into consideration the evolution of a component state at runtime and the semantics of communications based on request/reply by futures mechanism. The authors give an example of how a system reconfiguration can be formally specified in Isabelle. However, none of the two discussed formalisations included the notion of non-functional components, many-to-many interfaces, or interceptors. The formal specification of component correctness defined in this thesis could be used to extend the expressiveness of the component model in the Alloy and the Isabelle frameworks.

Another similar work was provided in [74] where the authors investigated a language for generating correct-by-construction component systems, and reconfiguring them safely. The core contribution is a framework Mefresa which is based on the Coq [81] proof assistant and aims at automatic reasoning on component-based application architecture. The authors show how a GCM-based system structure can be encoded in Coq and how its correctness can be checked with respect to a set of predefined properties. An interesting aspect of the work is an approach to encode and validate scenarios for the dynamic application reconfiguration. Similarly to the cases above, this study does not deal with the structure of the membrane. It could be extended to the enhanced component structure presented here. Also, it could benefit from using the VerCors graphical designer as a front-end, as encoding manually components structure in Coq might be a bit tedious for a non-experienced user.

Finally, we assume the conceptual model to be correct with respect to the static correctness predicates while translating it into an input for the model-checker and generating the implementation code. Both transformation processes will be presented in the following chapter. In particular, we generate one pNet encoding the behaviour of one component and we construct synchronisation vectors to express the communications between components. The unique naming property is crucial for the synchronisation, because the synchronised pNets are identified by the full path to the encoded components, and if two components in a container have the same name, their pNets will also have the same name which might lead to synchronising the wrong entities. The unique naming property is also important for the generated

program execution, because the GCM/ProActive factory does not allow constructing two components with the same name in one container. It is also crucial to make sure that the functional part of a modelled application is properly separated from the non-functional elements before constructing the pNets. The reason is that when building the behaviour model, we pre-process separately the functional and non-functional interfaces and components, hence, any violation of the binding nature constraint can lead to unpredictable results of pNet construction.

Chapter 5

A framework for verifying and running distributed components

Contents

5.1 From application design to pNets 80

5.1.1 Semantics of primitive components 81

5.1.2 Semantics of composite components 92

5.1.3 Implementation 100

5.2 From pNets to CADP 108

5.2.1 Preparing the input: generating Fiacre, EXP and auxiliary scripts 108

5.2.2 Model-checking with CADP 112

5.3 Code generation and execution 115

5.3.1 ADL generation 116

5.3.2 Java generation 118

5.3.3 Code execution 123

5.4 Discussion 124

5.4.1 On the verification 124

5.4.2 On the executable code generation 125

In this chapter we present the core of our framework: the approaches for verification and executable code generation. First, we formally define the construction of pNets encoding the behaviour of GCM components and then we discuss how they are

translated into an input for the CADP model-checker. We present the implementation of the full transformation chain: starting from the user-defined graphical models finishing with the model-checking in CADP. Next, we explain how the implementation code is generated from the VerCors diagrams and discuss the code execution on ProActive.

5.1 From application design to pNets

This section defines formally the behavioural semantics for the GCM/ProActive components. It shows how to build pNets from the specification of a hierarchy of components.

We organise this section as follows. We first give a behavioural semantics for primitive component behaviour and synchronisation of the different elements of the primitive component including queue, body, methods, and future proxies. We then describe the behavioural semantics for composite components, which compose the semantics of their sub-components, synchronising the requests and replies between the sub-components, the composite, and the external components. In particular, we will need to define a new kind of future proxies for handling the delegation mechanism that occurs in the composite components. Then, we discuss how the generation of the formalised models is implemented in VerCors.

Term algebra. The definition of pNets in Section 2.2 relies on a very generic definition of the term algebra. Here we specialise this algebra to take account the specificities of the GCM components.

The term algebra we use is a set of parameterised actions; actions will typically be of the form *Serve_m* for m a method label as defined below. Parameters will be either values (invocation parameters denoted by *arg* or computed results denoted *val*), or future identifiers (denoted by either p, or f, or *fid*). *arg* and *val* range (implicitly) over the set of *values*, this set of values being purposely undefined. In an object-oriented language, those values should be an abstraction of objects. It could be defined depending on the *type* of the value but we will not discuss this aspect here. p, *fid*, and f range over natural numbers.

Labels for identifying methods. Inside the actions, we need identifiers for methods that are more precise than simple method names. We define thus *MethodLabels* as a set of method labels, where a method label encompasses a method name, a signature, and the interface the method belongs to, plus possibly other meta-informations.

Most of the following can be read as if *MethodLabels* were just method names, however at some specific points and to disambiguate different methods, the other informations encoded in *MethodLabels* are also necessary. m_i range over such method labels.

A function MethLabels : $Itf \rightarrow \mathcal{P}(MethodLabels)$ is defined, where MethLabels(Itf_i) returns the set of *MethodLabels* corresponding to the methods of interface Itf_i. MethLabels is also defined for sets of interfaces (union of sets of method labels for each interface). Conversely, for a given method label m, $Itf(m)$ returns the interface of the method. As in Chapter 4, we distinguish between the set of client interfaces (*CItf*) and server interfaces (*SItf*).

Behavioural semantics. The behavioural semantics of components is expressed under the form $[\![Component]\!]$

It relies on the use of several auxiliary functions for expressing the semantics of specific parts of the components: the behaviour of the server methods and local methods (for a primitive component), the behaviour of the *body* of the component serving requests one after the other, a *proxyManager* for managing the available future proxies, the behaviour of each future *proxy*, and finally a delegation behaviour used when a composite component *delegate*s the service of a request to another component. The signature of all these functions is summarised below.

Function	Signature	Description
$[\![\ ]\!]$	$Component \rightarrow pNet$	basic behavioural semantics
$[\![\ ]\!]_{server}$	$MethodLabels \times$ $\mathcal{P}(MSignature \times Impl) \rightarrow pNet$	Server methods
$[\![\ ]\!]_{local}$	$MethodLabels \times$ $\mathcal{P}(MSignature \times Impl) \rightarrow pNet$	Local methods
$[\![\ ]\!]_{body}$	$\mathcal{P}(MethodLabels) \times$ $\mathcal{P}(MethodLabels) \rightarrow pNet$	The body: serves requests in a FIFO order
$[\![]\!]_{proxyManager}$	$MethodLabels \rightarrow pNet$	Manages future proxies
$[\![\ ]\!]_{proxy}$	$MethodLabels \rightarrow pNet$	future proxy
$[\![\ ]\!]_{FutDetect}$	$MethodLabels \rightarrow pNet$	pLTS detecting a future received as request parameter
$[\![\ ]\!]_{delegate}$	$MethodLabels \rightarrow pNet$	Delegation method (in composite components)

5.1.1 Semantics of primitive components

Primitive components are the leaves of the hierarchy; they contain the applicative code from which more complex components, and thus more complex behaviours can

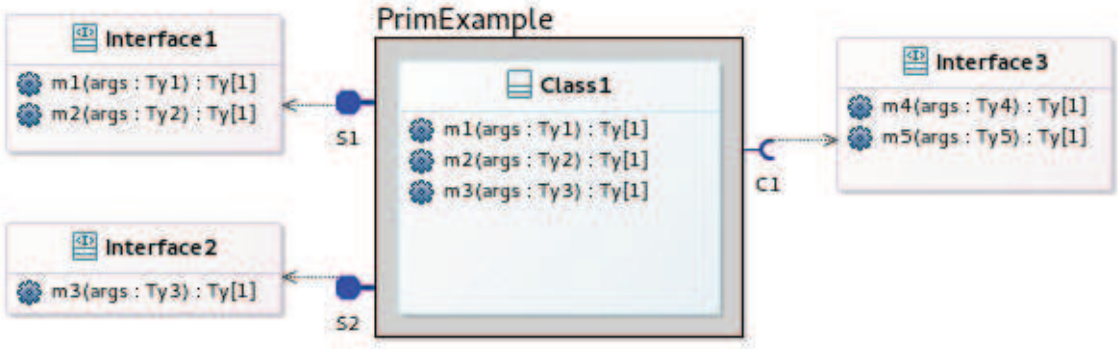


Figure 5.1 – An example of a primitive component

be built. This section gives a behavioural semantics for GCM primitive components, able to receive requests, to serve them in a FIFO order by executing a server method, and to send requests to the external world. Additionally to the global structure of a primitive component and the synchronisation of its sub-entities, this section defines pLTSs describing the behaviour of a FIFO service policy, of proxies for handling futures, of managers for pools of future proxies, and discusses the pLTSs of the server and local methods. More generally, this section shows that pNets provide a convenient abstraction for modelling asynchronous components communicating by asynchronous requests and futures.

Illustrative Example

We first illustrate and explain the structure of the behavioural semantics of primitive components based on the component shown in Figure 5.1. The primitive `PrimExample` has three server methods: `m1`, `m2`, `m3` and two client methods `m4` and `m5`. Each method has input and output parameters. For the sake of simplicity, the given example does not include local methods; their modelling is very close to the server ones and it will be discussed at the end of the section. Figure 5.2 illustrates the structure of the pNet expressing the semantics of the component. It illustrates the global structure, the pNets represented by boxes, and the synchronisation vectors represented by arrows (an ellipse is used when a synchronisation vector involves more than two processes). Note that the direction of an arrow is purely conventional, but goes, as much as possible, from an emission action to a reception action, intuitively following the data flow.

A primitive component can receive incoming requests (iQ_m_i) that are stored in the *Queue* pNet and then served by the *Body* pLTS. The service consists in triggering a $Call_m_i$ to the adequate server method, called $\mathcal{M}i$ in the figure. Once a result is

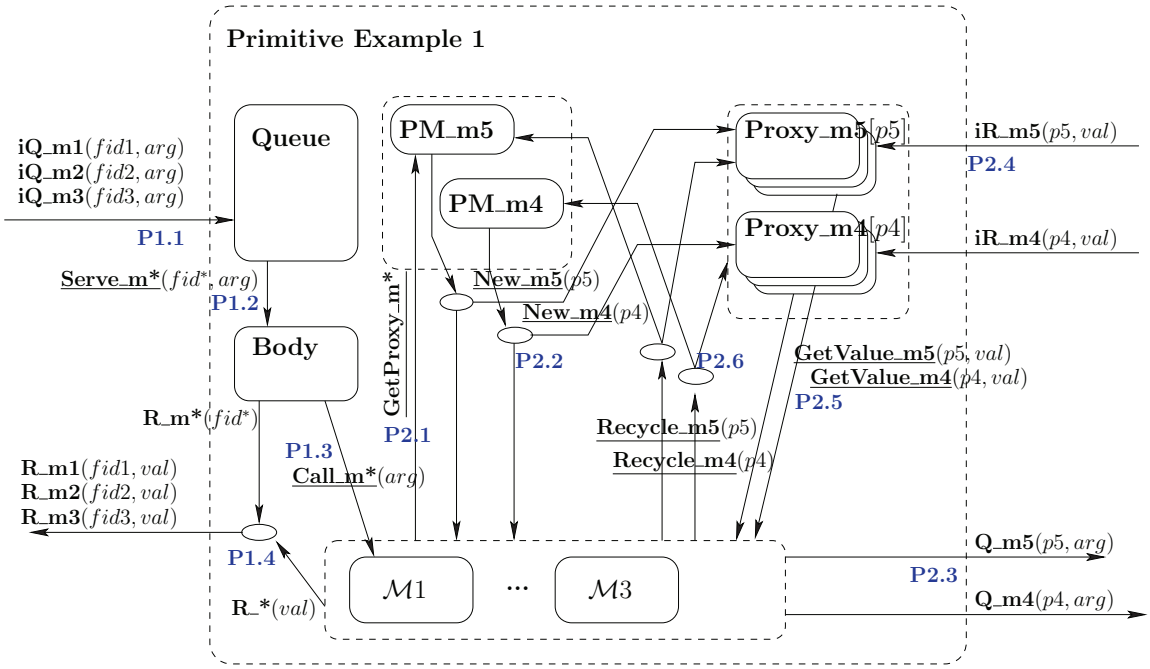


Figure 5.2 – pNet for the PrimExample component from Figure 5.1

computed for the request, a R_m_i action is emitted with the corresponding future identifier *fid* and result value *val*.

The server methods can call external components through client interfaces. There is one proxy manager PM_* for each method of each client interface (proxy managers are both indexed over interfaces and over methods). Then each of those managers manages itself a family of proxies $Proxy_*$. On each proxy manager PM_m_i, the caller can perform a *GetProxy*. Upon request, a fresh future proxy $Proxy_m_i$ is allocated and returned by a New_m_i action that acts as a response to the *GetProxy*; there is a family of future proxies for each method of each client interface. Then the outgoing call is emitted by the caller pLTS with the reference to the corresponding proxy sent as parameter (Q_m_i). Finally when a result is computed the reply iR_m_i is received by the adequate proxy and the result can be accessed by $GetValue_m_i$ actions performed by some server methods. Then, server methods can emit *Recycle* actions that are sent to the adequate proxy and proxy manager. The action notifies the proxy manager that the proxy will not be used by the caller any more, and that it can be given to another process. In fact, there is no notion of proxy recycling in GCM/ProActive because the number of proxies is limited only by the Java heap memory size, however, this is not the case for the model-checking. When generating the behaviour model for the finite state-space model-checking we, obviously, have to limit the number of proxies and in order to be closer to the actual implementation we construct such a model where proxies can be re-used infinitely.

Global structure

This subsection formalises and generalises the principles depicted in the previous section. The behaviour of a primitive component is formalised below and it is computed by the rules shown in this subsection.[1]

$$\frac{\begin{array}{c} m_l^{l\in L} = \text{MethLabels}(\overline{SItf}) \quad \mathcal{Q} = Queue(m_l^{l\in L}) \quad \mathcal{B} = [\![m_l^{l\in L}]\!]_{body} \\ \forall l \in L.\, \mathcal{SM}_l = [\![m_l, Impl]\!]_{server} \qquad \forall j \in J.\mathcal{P}_j = \mathcal{P}(CItf_j) \\ \forall j \in J.\mathcal{PM}_j = \mathcal{PM}(CItf_j) \qquad SV = SV_S(m_l^{l\in L}) \cup SV_C(CItf_j^{j\in J}, L) \end{array}}{[\![CName < \overline{SItf}, CItf_j^{j\in J}, Impl >]\!] = \langle\!\langle \mathcal{Q}, \mathcal{B}, \overleftrightarrow{\langle\!\langle \mathcal{SM}_l^{l\in L} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle \mathcal{PM}_j^{j\in J} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle \mathcal{P}_j^{j\in J} \rangle\!\rangle}, SV \rangle\!\rangle}$$

With the auxiliary rules for building the proxy managers and the proxy families:

$$\frac{m_n^{n\in N} = \text{MethLabels}(CItf) \qquad \forall n \in N.\, \mathcal{F}_n = \overleftrightarrow{\langle\!\langle [\![m_n]\!]^{\mathbb{N}}_{proxy} \rangle\!\rangle}}{\mathcal{P}(CItf) = \overleftrightarrow{\langle\!\langle \mathcal{F}_n^{n\in N} \rangle\!\rangle}}$$

$$\frac{m_n^{n\in N} = \text{MethLabels}(CItf)}{\mathcal{PM}(CItf) = \overleftrightarrow{\langle\!\langle [\![m_n]\!]^{n\in N}_{proxyManager} \rangle\!\rangle}}$$

The pNet corresponding to a primitive component is made of:

- A queue able to receive incoming requests: it can enqueue a request on a method label of one of the server interfaces, we use here a pNet queue constructor.
- A body that will serve all the requests that can reach the queue, it will delegate the treatment of the request to the server methods $\overleftrightarrow{\langle\!\langle \mathcal{SM}_l^{l\in L} \rangle\!\rangle}$.
- Server methods: there is one server method for each method label of a server interface.
- A family $\mathcal{PM}$ of proxy managers indexed both over the set of client interfaces and over the methods of those interfaces: those managers are responsible for allocating a new proxy when requested, and activating those newly created proxies.
- A family $\mathcal{P}$ of future proxies indexed over the set of client interfaces (J), the methods of those interfaces (N), and proxy indices, i.e. integers ($\mathbb{N}$): a proxy

[1] Note the construct $m_l^{l\in L} = \text{MethLabels}(SItf_i^{i\in I})$ that defines both the value of each method label m_l and the set L over which it is indexed. This kind of constructs will be massively used in the rest of this Chapter.

Table 5.2 – Server and client-side synchronisation vectors for primitive components. The synchronised sub-pNets are: $\langle\!\langle \mathit{Queue}, \mathit{Body}, \mathit{ServerMethods}, \mathit{ProxyManagers}, \mathit{Proxies} \rangle\!\rangle$

$$\frac{l \in L \qquad \mathit{fid} \in \mathbb{N}}{\begin{array}{ll} \{\langle \mathit{iQ_m_l}(\mathit{fid}, \mathit{arg}), -, -, -, - \rangle \to \mathit{iQ_m_l}(\mathit{fid}, \mathit{arg}), & [1] \\ \langle \mathit{Serve_m_l}(\mathit{fid}, \mathit{arg}), \mathit{Serve_m_l}(\mathit{fid}, \mathit{arg}), -, -, - \rangle \to \underline{\mathit{Serve_m_l}}(\mathit{fid}, \mathit{arg}), & [2] \\ \langle -, \mathit{Call_m_l}(\mathit{arg}), l \mapsto \mathit{Call_m_l}(\mathit{arg}), -, - \rangle \to \underline{\mathit{Call_m_l}}(\mathit{arg}), & [3] \\ \langle -, \mathit{R_m_l}(\mathit{fid}), l \mapsto \mathit{R_m_l}(\mathit{val}), -, - \rangle \to \mathit{R_m_l}(\mathit{fid}, \mathit{val}) \} & [4] \\ \qquad\qquad \subseteq SV_S(m_l^{l \in L}) & \end{array}} \text{ P1}$$

$$\frac{j \in J \qquad l \in L \qquad m_i \in \text{MethLabels}(\mathit{CItf}_j) \qquad p \in \mathbb{N}}{\begin{array}{ll} \{\langle -, -, l \mapsto \mathit{GetProxy_m_i}, j \mapsto i \mapsto \mathit{GetProxy_m_i}, - \rangle \to \underline{\mathit{GetProxy_m_i}}, & [1] \\ \langle -, -, l \mapsto \mathit{New_m_i}(p), j \mapsto i \mapsto \mathit{New_m_i}(p), j \mapsto i \mapsto p \mapsto \mathit{New_m_i} \rangle \to \underline{\mathit{New_m_i}}(p), & [2] \\ \langle -, -, l \mapsto \mathit{Q_m_i}(p, \mathit{arg}), -, - \rangle \to \mathit{Q_m_i}(p, \mathit{arg}), & [3] \\ \langle -, -, -, -, j \mapsto i \mapsto p \mapsto \mathit{iR_m_i}(\mathit{val}) \rangle \to \mathit{iR_m_i}(p, \mathit{val}), & [4] \\ \langle -, -, l \mapsto \mathit{GetValue_m_i}(p, \mathit{val}), -, j \mapsto i \mapsto p \mapsto \mathit{GetValue_m_i}(\mathit{val}) \rangle \to & \\ \qquad \underline{\mathit{GetValue_m_i}}(p, \mathit{val}), & [5] \\ \langle -, -, l \mapsto \mathit{Recycle_m_i}(p), j \mapsto i \mapsto \mathit{Recycle_m_i}(p), j \mapsto i \mapsto p \mapsto \mathit{Recycle_m_i} \rangle \to & \\ \qquad \underline{\mathit{Recycle_m_i}}(p) \ \} & [6] \\ \qquad\qquad \subseteq SV_C(\mathit{CItf}_j^{j \in J}, L) & \end{array}} \text{ P2}$$

is responsible for receiving the result of a request made towards another component; when the value of the result is needed by a server method, this method asks for the value to the adequate proxy.

Synchronisation vectors

The set of synchronisation vectors for a primitive component is built by two functions: SV_S that provides the set of synchronisation vectors corresponding to the server interfaces, and SV_C for the client interfaces. Each of those sets is defined as the smallest set verifying the rules given in Table 5.2.

Let us explain briefly what are the synchronisation vectors generated by the inference rules, more precisely, we focus on the synchronisation vectors for the $\mathit{GetProxy_m_i}$ actions, in order to explain the rule patters [P2.1]. One synchronisation vector for $\mathit{GetProxy_m_i}$ is generated for each $l \in L$, for each $j \in J$, and for each $i \in I$. Each synchronisation vector synchronises one action[2] $l \mapsto \mathit{GetProxy_m_i}$ of the sub-pNet containing the family of server methods, with one action $j \mapsto i \mapsto \mathit{GetProxy_m_i}$ of the sub-pNet containing the family of proxy managers (for each interface). As each of the synchronisation vectors of families of pNets triggers the action on the indexed

[2] $j \mapsto i \mapsto a$ should be read $j \mapsto (i \mapsto a)$

element of the family, this line allows one action $GetProxy_m_i$ of one server method (indexed by l) to be synchronised with the action $GetProxy_m_i$ of the proxy manager indexed by i of the interface indexed by j.

The set of server synchronisation vectors SV_S defined in rule [P1] encodes the following synchronisations:

- en-queueing an incoming request [P1.1];
- service of a request by the body [P1.2];
- the body calling a server method to serve a request [P1.3];
- the server method providing a result for this served request [P1.4].

In the last case the result both notifies the body process and is returned to the outside of the primitive component. In all the actions, the method argument or the returned value is used as parameter, plus when necessary the identifier of the concerned future (*fid*).

The set of client synchronisation vectors SV_C is defined in rule [P2][3]; it encodes the following synchronisations:

- obtaining a new future proxy which involves a call to the proxy manager [P2.1] and another action [P2.2] for returning a fresh proxy identifier and activating the corresponding future proxy;
- the sending of a request from a server method to an external component [P2.3];
- the reception of a result by the future proxy [P2.4];
- the access to a future value [P2.5] from a server method, the future value is stored in the future proxy; it is interesting to note here that the value of p is provided by the *GetValue* action, it is used to index the right future proxy, and the value of *val* is on the contrary provided by this future proxy and "returned instantly" to the server method;
- the recycling of a future proxy [P2.6].

The function SV_C receives as argument the set L of indices over which server methods range. This argument is necessary because the server methods can perform some of the client-side actions, like $GetValue_m_i$.

[3]Note the indexing of proxy managers (by interfaces and methods) and of proxies (by interfaces, methods, and proxy identifier).

Queue

The *Queue* pLTS simply models a FIFO queue which is able to receive requests from outside of the component and give them to the body. The queue can be constructed from the signatures of the server methods.

Body

The body is a pLTS modelling the service of the different requests: for each server method, the body can dequeue a request corresponding to this method, delegate the service to the appropriate server method pNet, wait until the method finishes its execution, and finally return this result (if there is any) before de-queueing a new request. It can be generated automatically from the set of server methods $m_i^{i\in I}$. $[\![m_i^{i\in I}]\!]_{body} = \langle\!\langle S, s_0, L, \rightarrow \rangle\!\rangle$ where:

- $S = \{s_0\} \cup \bigcup_{i\in I}\{s_i(\mathit{fid}, \mathit{arg})\} \cup \bigcup_{i\in I}\{s'_i(\mathit{fid})\}$
- $L = \bigcup_{i\in I}\{\mathit{Serve_m_i}(?\mathit{fid}, ?\mathit{arg}), \mathit{Call_m_i}(\mathit{arg}), \mathit{R_m_i}(\mathit{fid})\}$
- $\rightarrow = \bigcup_{i\in I}\{s_0 \xrightarrow{\mathit{Serve_m_i}(?\mathit{fid},?\mathit{arg})} s_i(\mathit{fid}, \mathit{arg})\} \cup$
  $\bigcup_{i\in I}\{s_i(\mathit{fid}, \mathit{arg}) \xrightarrow{\mathit{Call_m_i}(\mathit{arg})} s'_i(\mathit{fid})\} \cup \bigcup_{i\in I}\{s'_i(\mathit{fid}) \xrightarrow{\mathit{R_m_i}(\mathit{fid})} s_0\}$

Note, that the *fid* parameter exists only in the labels corresponding to the non-void methods.

For each method m, the body pLTS can always perform the three actions *Serve_m*, then *Call_m*, and then *R_m*. This body encodes a *mono-threaded* component behaviour where no two requests are served at the same time. This corresponds indeed to the behaviour of the GCM/ProActive framework, and more generally to the behaviour of active objects or actors. Allowing the body to serve multiple requests at the same time would be quite easy but the resulting behaviour would be much more complex. Figure 5.3 provides a graphical representation for a pLTS of a body (in the rest of this thesis we will express pLTSs graphically). The figure shows a body able to serve three functional requests m_1, m_2, and m_3 where only the first two of them are supposed to return a result.

The model of the body presented in this section encodes one possible serving policy (the FIFO), but in fact, it would be good to allow the user to choose among several different policies or even define a custom one.

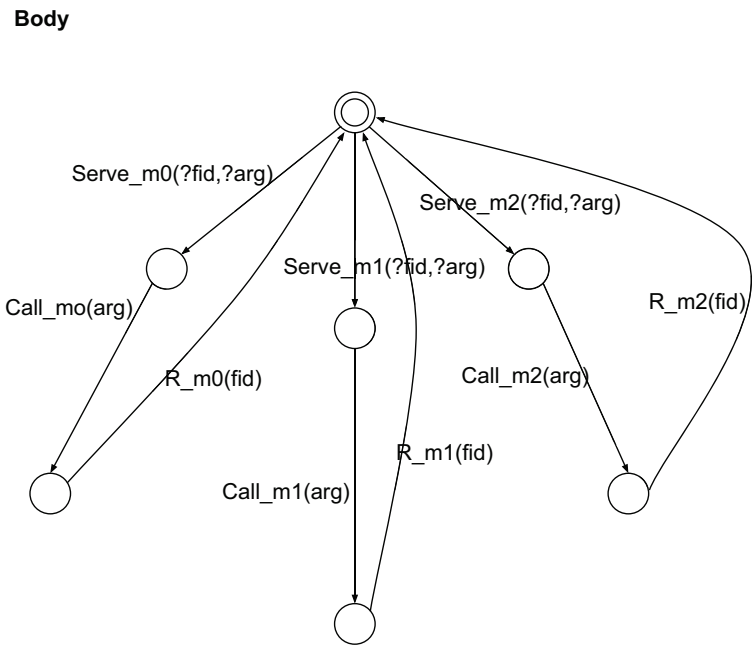


Figure 5.3 – Graphical representation of the behaviour of the Body

Modelling of the Future Proxies

Communicating by asynchronous requests allows each component to execute asynchronously from the others. However it is commonly necessary to obtain a result for some of those asynchronous invocations. A convenient abstraction for dealing with response to asynchronous requests is the notion of futures. Technically, a future is often implemented by a proxy that represents the result and is accessible both locally to know whether the result came back, and remotely by the invoked component that wants to return the result. We represent those notions in our behavioural models. Fresh future proxies are instantiated upon need; this is done by invoking the proxy manager before performing an asynchronous request.

Remember future proxies are families indexed by client interface index, method index, and future identifier; proxy managers are indexed by client interface index and method index. We propose a specification of proxy manager and future proxy in Figure 5.4. The behavioural semantics of the proxy manager is defined by the pLTS *ProxyManager_m* shown on the right side of Figure 5.4; in the semantic rules it is denoted by $[\![m]\!]_{proxyManager}$. It maintains a list of available proxies and returns a fresh future (by a *New* action), or if there is no more fresh future, raises an error *NoMoreProxy*. Indeed, in our specification, we let future identifiers be indexed by $\mathbb{N}$ but if one wants to perform finite model-checking, a bound should be chosen on the

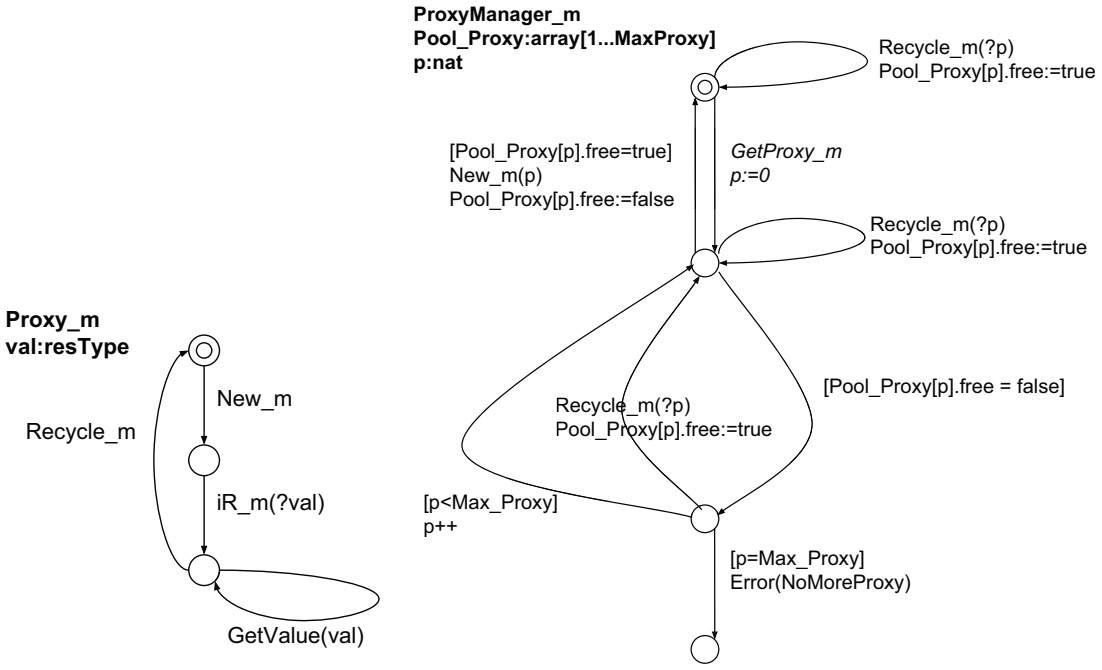


Figure 5.4 – pLTSs for the Future Proxies and Proxy Managers

size of each future proxy family, and in each proxy manager, *Max_Proxy* should be set to the chosen bound. Alternative proxy family specifications, better optimised for some specific usage could of course be designed; instead we propose here a simple specification of those proxies.

Proxies have much simpler behaviour; $[\![m]\!]_{proxy}$ is defined by the pLTS *Proxy_m* shown on the left side of Figure 5.4. Once activated by a *New_m* action, it waits for the corresponding reply ($R_m(?val)$). At this point, the proxy can be accessed to recover the result of the request invocation, it continuously sends the result to the server methods by a $GetValue_m(val)$ action.

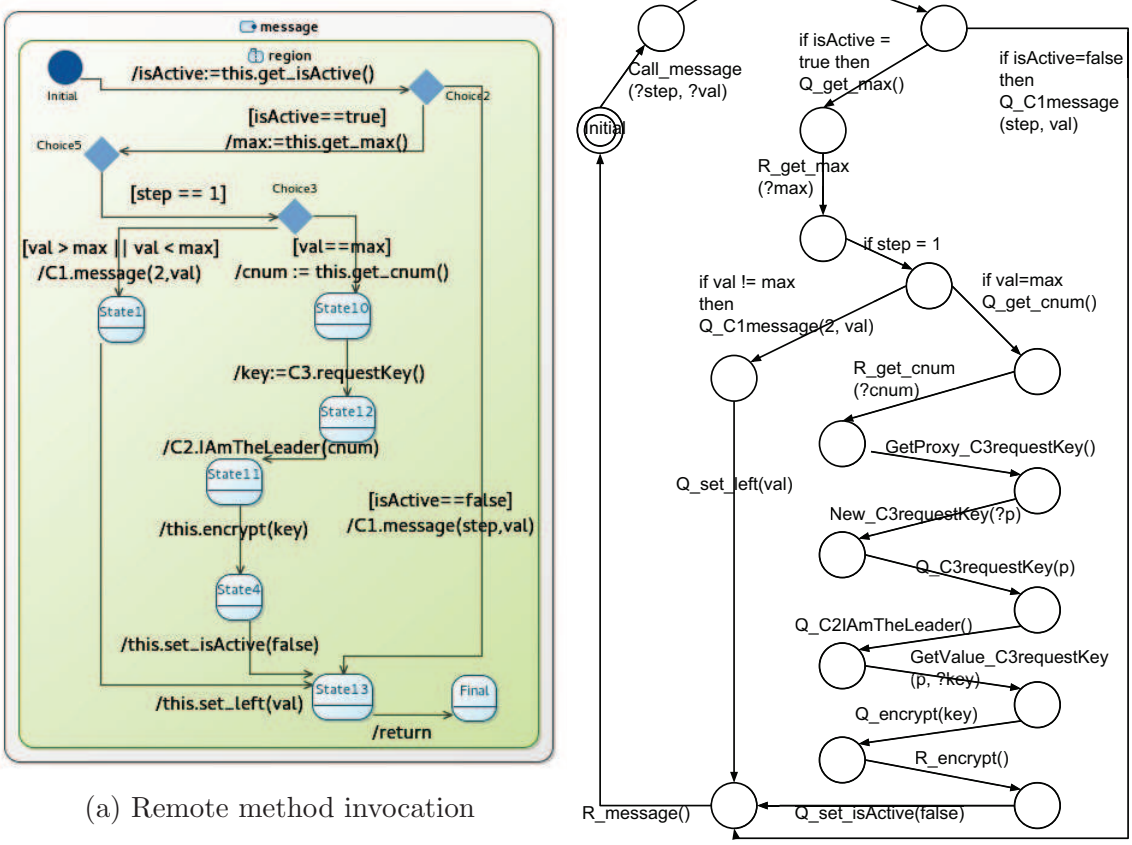
The proxies in Figure 5.4 are endowed with a *Recycle_m* transition, bringing back the proxy in its initial state. This is useful when information can be computed, e.g., by static analysis, that the proxy is not useful anymore, so it can be made available again in the proxy pool of the ProxyManager. The *Recycle_m* event should be sent by the LTS modelling a server method. When such an event is received by the ProxyManager, this sets the corresponding entry in the *Pool_Proxy* to *free*. For generality, the proxy manager accepts *Recycle_m* transitions in every state, even if for mono-threaded GCM components, this action can only be received when the proxy manager is in its initial state.

Server methods

The behaviour for each server method is expressed by a pNet, used when serving the corresponding request. This behaviour is either obtained by source code analysis, or provided by the user. In the context of this thesis, it is obtained by translating each state machine modelling the behaviour of a server method into a pLTS. Figure 5.5 illustrates a part of a `message` method state machine (from Figure 3.6 transformed into a pLTS. The state machine labels prefixed by "//" are not translated because they represent comments. One can notice that the translation is almost straightforward except from the transformation of the method calls.

An invocation of a local method from a state machine of a server method (e.g. `this.encrypt(key)`) is translated into two pLTS actions: one for the method call and one for the local method termination: `Q_encrypt(key)` and `R_encrypt()`. The second action can receive the result of the local call but even if no result is returned, the action is needed in order to ensure that the two methods will not be executed concurrently.

We distinguish two cases of a remote method invocation: when the called method does not return a result and when a result is returned. In the first case, the translation is simple: the call is transformed into a single pLTS action. For example, `C1.message(step, val)` is translated into *Q_C1message(step, val)*. There is no need to synchronise on the method termination for two reasons: first, no result is expected, second, the remote method execution will be done on another component (i.e. by another thread), hence, the current thread does not get blocked. An invocation which is supposed to return a result (e.g. `key:=C3.requestKey()`) triggers the futures mechanism and is encoded in a pLTS with several actions. First, the pLTS asks to allocate a proxy (*GetProxy_requestKey()*), second, if the proxy was successfully found, the pLTS receives its index (*New_requestKey(?p)*). Next, the server method can send the remote method invocation: *Q_requestKey(p)*. The server method can proceed with the execution as long as it does not need the result of the remote call. In our example the server method invokes another remote method in order to report that it is the leader. When the server method tries to use the result of a remote call (for example, in the label `this.encrypt(key)`), it needs to get the value from the proxy that was allocated for the remote method invocation: *GetValue_requestKey(p)*. Then, the result can be used.



(a) Remote method invocation

(b) Asynchronous execution

Figure 5.5 – A state machine and its translation to a pLTS

Treatment of the server and client methods that do not return a result

The treatment of the incoming void method invocations is almost the same as for the non-void ones. The only difference is that if the server method is not supposed to return a result, then there will be no proxy in the caller for receiving the value. In this case, the parameter *fid* is omitted in all the synchronisation vectors of [P1]. The reader should note that since the constructed primitive components are single-threaded, [P1.1] is applied even to the void server requests: the synchronisation tells the body when the server method execution is terminated so that the body can take the next request from the queue. This ensures that only one request is served at each time (following the semantics of GCM/ProActive components).

The proxy manager and proxies are not constructed for the void client methods, because there is no need to receive a result. Moreover, the void client method invocation involves only one synchronisation vector [P2.3] which does not include the proxy index p as a parameter.

Local methods

In fact, a primitive component also includes a set of pLTSs for the local methods $[\![m_l, Impl]\!]_{local}$; their definition was omitted in the rest of the section. A pLTS encoding the behaviour of a local method is obtained by translating the corresponding state-machines. The translation process is the same as for the server methods. Local methods are not involved in any of the rules from [P1] because they cannot be invoked from outside of a component. They introduce two additional sets of synchronisation vectors.

First, similar to the server methods, the local methods can also send client requests. In order to model this, we construct the same the synchronisation vectors as in [P2], where the pNets of the local methods are synchronised with the proxies and proxy managers.

Second, in order to model the communications between the methods inside a primitive component, we synchronise the server methods with the local methods and the local methods with each other. Note, that the server methods cannot be invoked from inside of a component. The synchronisation is done on the local method invocation and its termination in the case of the single-threaded components. The communications are completely synchronous and do not involve futures.

5.1.2 Semantics of composite components

Hierarchical component models, like GCM, allow the specification of new components, based on the composition of others. Such a composition mechanism is very convenient when building large applications. As explained in Section 2.1, we start from a static definition of composition of the system. In the context of this work, the composition is given in a form of VerCors diagrams, but another kind of language can be used for that (for example, GCM ADL). The given model is used to extract component bindings that will define the synchronisation between the emission and the reception of communication actions. A composite also has a request queue for receiving requests coming from the outside or the inside of the component, it treats each of those requests by sending it to the adequate component or emitting a request to the outside world. For this the composite has future proxies but as the requests only transit through the component, we implement special future proxies that perform future redirection: when a future proxy is created, it receives the identifier for another future f' and when the reply will come back, it will be immediately re-sent as a reply for the future f'. Once a request has been delegated to the sub-component, the composite can serve the next request without waiting for the result.

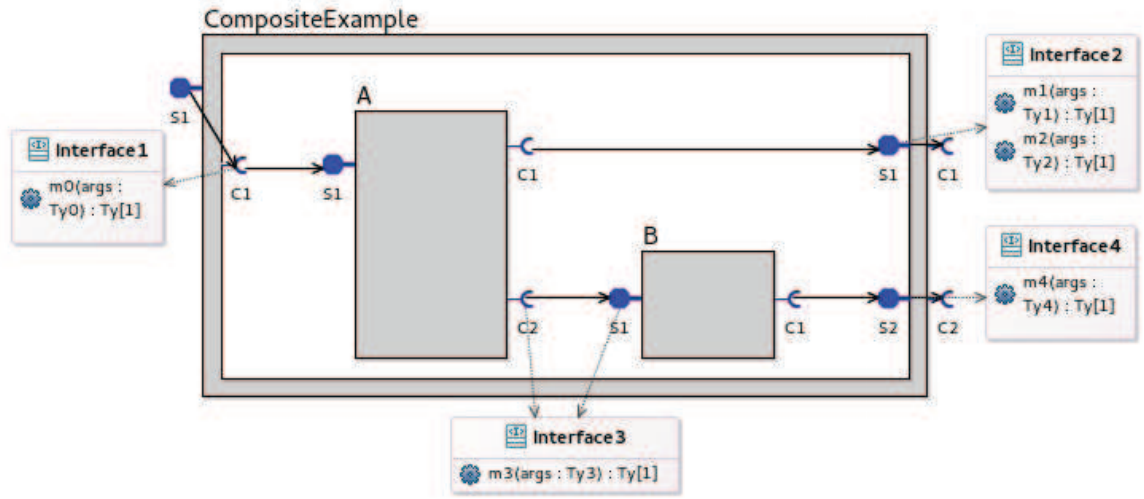


Figure 5.6 – An example of a composite component

Illustrative Example

The example we are going to introduce is based on a composite component illustrated in Figure 5.6. The component has one server method `m0`, three client methods: `m1`, `m2` and `m4`, and two sub-components `A` and `B`. `A` serves the calls to `m0`, invokes `m3` on `B`, `m1` and `m2` on one of the client interfaces of the composite. `B` can invoke the `m4` client method of the composite. All methods involved in the system are supposed to return a result.

Figure 5.7 shows the pNets structure corresponding to the composite component of Figure 5.6. It illustrates the structure of the pNets we generate for specifying the behaviour of a composite component.

Two sub-pNets A and B represent the behaviour of sub-components `A` and `B`. A queue pNet receives *iQ_m0(f,arg)* requests where f is the future corresponding to the request and *arg* is the value passed as argument. *Serve_∗* communications allow the body to retrieve those requests, which will then be treated by the *Deleg_m0* pNet, this pNet receives *Call* communications from the body and delegates the request to an inner component (here, `A`). During this process, a future proxy is created by the proxy manager (process *CPM_m0*), the proxy (process *CProxy_m0[q]*) is responsible for receiving the reply when `A` has finished the request treatment and for forwarding this result to the outside of the composite component: *R_m0(q,val)* that becomes *R_m0(f,val)*. Note that this proxy encodes some basic form of future forwarding: the future q corresponds to the same result as the future f.

Similarly, requests emitted by the inner components arrive in the queue (we draw two *Queue* boxes, but they correspond to the same element), they are then delegated

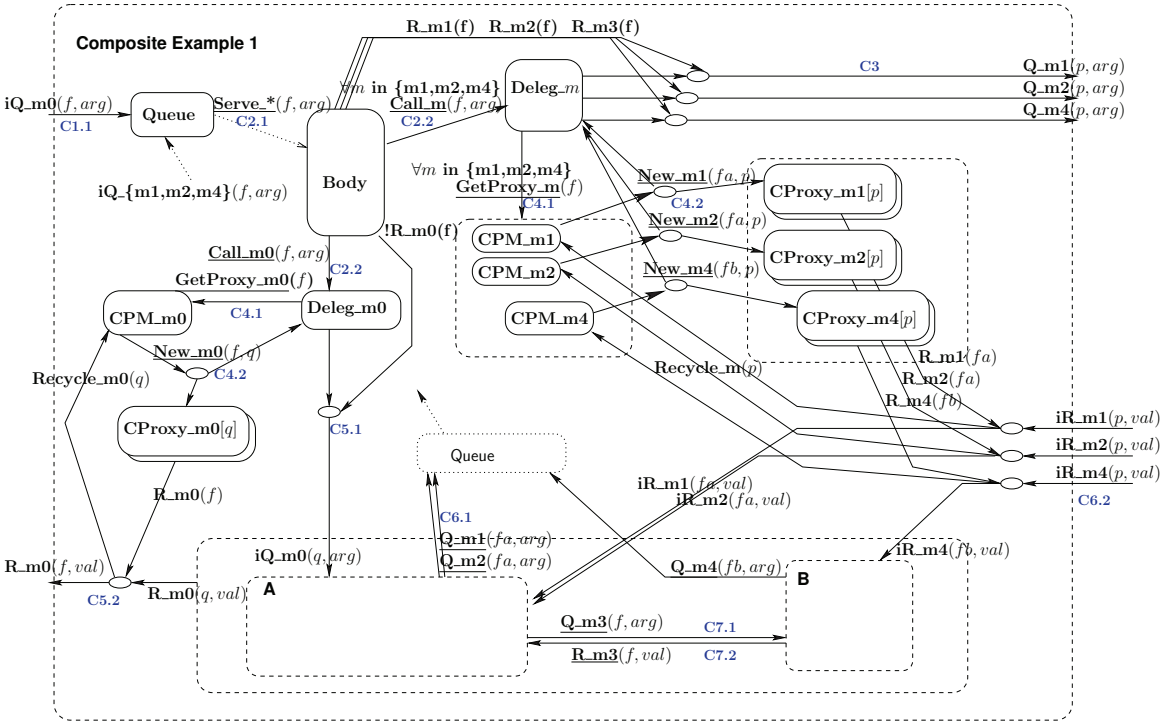


Figure 5.7 – pNet for the composite component from Figure 5.6

to the outside world by a similar mechanism: a *Deleg_m* pNet delegates the call, and creates a future proxy, which will be responsible for sending back the result to the appropriate inner component. Here again the proxy manages the fact that both the future q and the future fa (or fb) represent the same result.

The structure of the proxy-manager (*CPM_**) and of the proxy (*PM_**) is similar to the primitive case: using double indices over interfaces and methods. However, here we have proxies both for client interfaces and for server interfaces (because of their corresponding internal client interfaces).

All the communications expressed above, but also the communication channels between the different inner components – requests *Q_m3* and the corresponding replies *R_m3* – correspond to synchronisation vectors of the pNet of the composite. Each box is a pLTS or a family of pLTSs, except inner components that are more complex pNets.

Global structure

The semantics of a composite component is described below; we assume that all the methods are supposed to return a result and then we discuss a few modifications that have to be done in the case of the void requests. The first difference compared to the semantics of primitive components is that it does not rely on the server method

specification, instead it delegates requests to sub-components, some of the sub-pNets of a composite component's pNet correspond to the behaviour of the sub-components. Like primitive components, the behaviour of composite components includes a proxy manager and proxy families, but in the case of composites, we also need one future proxy family for each method of each *server interface*. Indeed the service of requests received on a server interface will be delegated to a sub-component, and thus a future is necessary to represent the result of such a delegated request.

Note the use of the Symm function from Chapter 4 to take the symmetrical role of an interface and to transform those server interfaces into client ones. For similar reasons, the request queue can receive requests on all methods of all the interfaces of the composite component, both server and client (i.e., internal server) ones.

To delegate a request to an inner component or from an inner component to an external one, "delegation methods" are used, they are denoted $\mathcal{DM}$. Delegation methods transform a request into another and a special proxy for future is used to remember the relationship between the original future and the future of the new delegated request. The building of proxy and proxy manager families are similar to the case of the primitive component (we reuse the same function); however each future proxy is slightly different as shown below. The queue and body are similar to the one of primitive components. The only difference is that the body of a composite additionally includes actions for the treatment of client requests.

$$\overline{m} = \biguplus_{Itf \in \overline{SItf}} \text{MethLabels}(Itf) \uplus \biguplus_{Itf \in \overline{CItf}} \text{MethLabels}(Itf) \qquad \mathcal{Q} = Queue(\overline{m}) \qquad \mathcal{B} = [\![\overline{m}]\!]_{body}$$
$$Itf_h^{h \in H} = \overline{CItf} \uplus \text{Symm}(\overline{SItf}) \qquad \forall h \in H.\, \mathcal{P}_h = \mathcal{P}(Itf_h) \qquad \forall h \in H.\, \mathcal{PM}_h = \mathcal{PM}(Itf_h)$$
$$SV = SV_S(\text{MethLabels}(\overline{SItf}), \text{MethLabels}(\overline{CItf})) \cup SV_C(\overline{CItf}, Itf_h^{h \in H}) \cup SV_B(CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >)$$

$$[\![CName < \overline{SItf}, \overline{CItf}, Comp_k^{k \in K}, \overline{Binding} >]\!] = \langle\!\langle \mathcal{Q}, \mathcal{B}, \mathcal{DMS}(\overline{m}), \overleftrightarrow{\langle\!\langle \mathcal{PM}_h^{h \in H} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle \mathcal{P}_h^{h \in H} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle [\![Comp_k]\!]^{k \in K} \rangle\!\rangle}, SV \rangle\!\rangle$$

With the auxiliary rule for delegation methods:

$$\frac{\forall l \in L.\, \mathcal{DM}_l = [\![m_l]\!]_{delegate}}{\mathcal{DMS}(m_l^{l \in L}) = \overleftrightarrow{\langle\!\langle \mathcal{DM}_l^{l \in L} \rangle\!\rangle}}$$

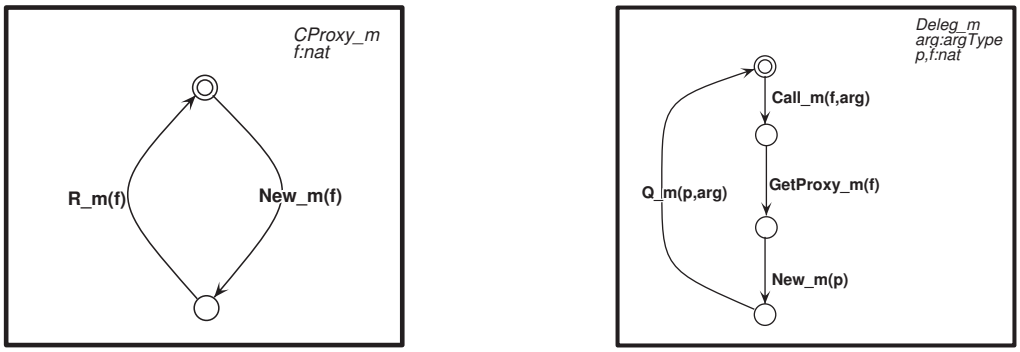


Figure 5.8 – Auxiliary processes proxy and delegate of composite components

Future Proxies

The behaviour of future proxies for a composite is slightly different from the one of a primitive, as illustrated in Figure 5.8: the process *CProxy_m* in the figure gives the new value of the proxy semantics $[\![m]\!]_{proxy}$. The delegation methods create those proxies to remember the identifier of the future that the delegation method should serve. Consequently, the future proxy receives a future identifier and will return it upon need. The future proxy thus first receives a *New* action with a future identifier as parameter and then emits an *R_m(f)*. Such a proxy is automatically recycled as, by construction, we know it is only used once.

Delegation Methods

The *Deleg_m* process, also shown in Figure 5.8 expresses the management of delegate methods: $[\![m]\!]_{delegate}$ is given by the pLTS *Deleg_m*. This delegation process receives a *Call* invocation from the body, creates a future proxy, launches a remote invocation (either to an inner or to an external component) and finishes its execution. This way the composite component can continue its execution and serve another request, but the process of the future proxy is still running in order to redirect the reply towards the right future identifier. The *proxyManager* for composite component is not shown, indeed it is a direct adaptation of the primitive one (Figure 5.4): it behaves exactly the same except that the *GetProxy* action receives a future identifier as parameter, this parameter is then passed as argument in the *New* emission action (it will be used by the future proxy).

Synchronisation Vectors

Synchronisation vectors are now organised into three sets: server-side (SV_S), client-side (SV_C), and binding-related (SV_B) synchronisation vectors. The set of server and

Table 5.3 – Server and client-side synchronisation vectors. The synchronised sub-pNets are:
$\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{m \in \overline{m} \qquad f \in \mathbb{N}}{\langle iQ_m(f, arg), -, -, -, - \rangle \rightarrow iQ_m(f, arg) \in SV_S(\overline{m}, \overline{m'})} \text{ C1}$$

$$\frac{(i \in L \wedge m = m_i) \vee (i \in L' \wedge m = m'_i) \qquad f \in \mathbb{N}}{\begin{array}{l}\{\langle Serve_m(f, arg), Serve_m(f, arg), -, -, - \rangle \rightarrow \underline{Serve_m}(f, arg), \quad [1] \\ \langle -, Call_m(f, arg), i \mapsto Call_m(f, arg), -, - \rangle \rightarrow \underline{Call_m}(f, arg)\ \} \quad [2] \\ \subseteq SV_S(m_l^{l \in L}, m_l'^{l \in L'})\end{array}} \text{ C2}$$

$$\frac{j \in J \qquad m_k^{k \in K} = MethLabel(CItf_j) \qquad k \in K \qquad f, p \in \mathbb{N}}{\langle -, R_m_k(f), k \mapsto Q_m_k(p, arg), -, -, - \rangle \rightarrow Q_m_k(p, arg) \in SV_C(CItf_j^{j \in J}, Itf_h^{h \in H})} \text{ C3}$$

$$\frac{h \in H \qquad m_k^{k \in K} = MethLabel(Itf_h) \qquad k \in K \qquad f, p \in \mathbb{N}}{\begin{array}{l}\{\langle -, -, k \mapsto GetProxy_m_k(f), h \mapsto k \mapsto GetProxy_m_k(f), -, - \rangle \rightarrow \underline{GetProxy_m_k}(f), \quad [1] \\ \langle -, -, k \mapsto New_m_k(p), h \mapsto k \mapsto New_m_k(p, f), h \mapsto k \mapsto p \mapsto New_m_k(f), - \rangle \rightarrow \\ \qquad \underline{New_m_k}(p, f)\} \quad [2] \\ \subseteq SV_C(CItf_j^{j \in J}, Itf_h^{h \in H})\end{array}} \text{ C4}$$

client synchronisation vectors is the smallest set verifying the rules given in Table 5.3.

The server-side synchronisation vectors are defined by rules [C1] and [C2]. [C1] allows external components to enqueue a request in the queue, for each method of a server interface of the composite. Note that replies (R_m) are not part of this rule because they depend on the bindings of the component; consequently they are treated among the binding synchronisation vectors. Rule [C2] uses a bigger set of methods as it takes into account requests of the server interfaces and the client interfaces of the composite; indeed, remember client interfaces have an associated internal server interface accessible by the sub-components of the composite. This second rule uses both arguments of SV_S, i.e., the list of client and server interfaces of the component. It deals with request service [C2.1], and subsequent calls [C2.2] to delegation pNets.

Client-side synchronisation vectors are expressed by rules [C3] and [C4] of Table 5.3. Similarly to the server case, [C3] is specific to external client interfaces (given as first argument of SV_C), whether [C4] is applicable to both external and internal client interfaces (the second argument of SV_C). Remember the internal client interfaces are the symmetric of server interfaces of the composite component. The first rule exports request sending (Q_m) sent by delegate methods to the external components. Note that delegate methods are indexed by the method labels of the interfaces: in the *pNet* definition, $m_l^{l \in L}$ is a disjoint union, and thus each $l \in L$ is considered as equal

Table 5.4 – Binding synchronisation vectors. The synchronised sub-pNets are: $\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{\begin{array}{c}(This.SI, C.SI_2) \in \overline{Binding} \\ SItf' = \mathrm{Get}(This.SI, CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) \\ k \in K \qquad C = \mathrm{Name}(Comp_k) \\ m_n^{n\in N} = \mathrm{MethLabels}(SItf') \qquad n \in N \qquad m'_n = m_n\{\!\!\{SI{\leftarrow}SI_2\}\!\!\} \qquad q,\ f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle -, R_m_n(f), n \mapsto Q_m_n(q, arg), -, -, k \mapsto iQ_m'_n(q, arg)\rangle \to \overline{Q_m_n}(q, arg), & [1] \\ \langle -, -, -, i \mapsto n \mapsto Recycle_m_n(q), i \mapsto n \mapsto q \mapsto R_m_n(f), k \mapsto \overline{R_m'_n}(q, val)\rangle \to & \\ \qquad R_m_n(f, val)\} & [2] \\ \subseteq SV_B(CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) & \end{array}} \text{ C5}$$

$$\frac{\begin{array}{c}(C.CI, This.CI_2) \in \overline{Binding} \qquad k \in K \\ C = \mathrm{Name}(Comp_k) \qquad CItf' = \mathrm{Get}(C.CI, CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) \\ m_n^{n\in N} = \mathrm{MethLabels}(CItf') \qquad n \in N \qquad m'_n = m_n\{\!\!\{CI_2{\leftarrow}CI\}\!\!\} \qquad p,\ f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle iQ_m_n(f, arg), -, -, -, -, k \mapsto Q_m'_n(f, arg)\rangle \to \overline{Q_m_n}(f, arg), & [1] \\ \langle -, -, -, j \mapsto n \mapsto Recycle_m_n(p), j \mapsto n \mapsto p \mapsto \overline{R_m_n}(f), k \mapsto iR_m'_n(f, val)\rangle \to & \\ \qquad iR_m_n(p, val)\ \} & [2] \\ \subseteq SV_B(CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) & \end{array}} \text{ C6}$$

$$\frac{\begin{array}{c}(C.CI, C'.SI) \in \overline{Binding} \qquad k, k' \in K \qquad C = \mathrm{Name}(Comp_k) \\ C' = \mathrm{Name}(Comp_{k'}) \qquad CItf' = \mathrm{Get}(C.CI, CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) \\ m_n^{n\in N} = \mathrm{MethLabels}(CItf') \qquad n \in N \qquad m'_n = m_n\{\!\!\{CI{\leftarrow}SI\}\!\!\} \qquad f \in \mathbb{N}\end{array}}{\begin{array}{ll}\{\langle -, -, -, -, -, (k \mapsto Q_m_n(f, arg), k' \mapsto iQ_m'_n(f, arg))\rangle \to \overline{Q_m_n}(f, arg), & [1] \\ \langle -, -, -, -, -, (k \mapsto iR_m_n(f, val), k' \mapsto R_m'_n(f, val))\rangle \to \overline{R_m_n}(f, val)\ \} & [2] \\ \subseteq SV_B(CName{<}\overline{SItf}, \overline{CItf}, Comp_k^{k\in K}, \overline{Binding}{>}) & \end{array}} \text{ C7}$$

to a method index k of a single interface i. Consequently, the request sending action (Q_m_k) is always issued by the delegate method indexed by k. Rule [C4] allows delegation methods to instantiate new proxies (by calls to the proxy manager and to the future proxies). Compared to the case of the primitive component, note the additional argument f passed to the proxy manager. This future identifier allows the future proxy (indexed p) to remember that the reply it will receive should be forwarded to the caller as the value for the future identifier f (and not p). In other words, the proxy remembers that the future p it will receive is in fact an alias for the future f. Similarly to primitive components, there are two actions for dealing with future proxy creations: *GetProxy* in [C4.1], and *New* in [C4.2].

Finally, the synchronisation vectors for the bindings of the composite component are shown in Table 5.4. There are three rules for building SV_B. Rule [C5] deals with import bindings, i.e. bindings from the composite component's internal client interfaces to inner components. Symmetrically, [C6] concerns export bindings, from

inner components to the composite component's internal server interfaces. The last rule [C7] specifies synchronisations due to bindings between two inner components.

The first rule [C5] deals with import bindings. The first premise of the rule picks an import binding, the next premises find the concerned server interface of the composite component and the destination of the binding, i.e., a sub-component. The only remaining non-trivial premise is $m'_n = m_n\{\!\{SI \leftarrow SI_2\}\!\}$; it replaces in m_n the occurrence of the interface named SI by the interface SI_2. Indeed, remember MethodLabels contain the name of the invoked interface, this name must thus be updated when a request/reply/... is transmitted from an interface to another[4]. Similar premises, renaming an interface name, will also be used in rules [C6] and [C7]. The first item of Rule [C5.1] synchronises the emission of a request by a delegate method with the inner component bound to the concerned internal client interface. This action is also synchronised with the proxy that will receive the result computed by the request. The case [C5.2] concerns the corresponding reply that is issued by the inner component, this reply is sent to the outside of the composite component. Note the particular flow of information here: the inner component emits a value for future q, that is directly synchronised with the future proxy number q of the composite component; the identifier of the future to be sent to the outside becomes f; it is retrieved from the future proxy, and an action $R_m'_n(f, val)$ is emitted. At the same time, a recycling action is triggered in the proxy manager.

The second rule [C6] manages export bindings, it also has one item for request emission [C6.1] and another one for reply reception [C6.2]. A request emitted by the inner component on the first side of the binding is enqueued in the composite (at the other side of the binding). Replies are redirected when received by the composite: when the reply for future p is received, the future proxy at index p is used to retrieve the future identifier f, and finally the result val is transmitted, associated with the future f, to the inner component indexed by k. At the same time, a *Recycle* action is triggered in the adequate proxy manager.

Rule [C7] deals with bindings between two inner components. It considers a binding between an interface of component C and an interface of component C'. It finds k, the index of C, and k', the one of C'; the rule directly transfers requests [C7.1] and replies [C7.2] from one component to the other for all the methods of the client interface bound. Like in the preceding rules, the name of the interface is updated during transmission.

[4]Other meta-informations are encoded in the method label and should also be updated.

Treatment of the methods that do not return any result

Like in the case of primitive components, for the composites, modelling of the requests that do not return any result is slightly different. First, neither server nor client void methods of a composite require a delegate pLTS, a proxy or a proxy manager. Instead, the body of a composite directly synchronises with the serving sub-component in order to forward the incoming requests. The composite component is not notified when a void incoming method invocation has been served while the body of a primitive synchronises with the server method upon its termination. In the case of the void client method call, the body of the composite simply forwards the call outside of the composite. As this is done in a single step, the request ordering is still guaranteed without additional return action in the body. Finally, if a void method invocation occurs between two sub-components, it obviously involves synchronisation upon method call only (Rule [C7.1] and does not require synchronising on the reply (i.e. the Rule [C7.2] is not applied).

In this section we have presented a behavioural semantics for hierarchical components communicating by asynchronous requests. The behaviour of composite components is only to forward requests to the adequate destination. We encode replies by means of futures, and composite components act as reply forwarders. This section generalised the examples of specifications one could find in [30] and formally defines the automatic generation of behavioural models for asynchronous components. In the next section we discuss how these semantic rules are implemented in VerCors.

5.1.3 Implementation

The generation of the behavioural models formalised in the previous section is fully implemented in VerCors. The user can launch the generation on any designed composite or primitive component if the conceptual model is statically correct with respect to the rules discussed in Chapter 4, The selected component will be considered as the *root* of the produced model. If the root is a composite, the generator will construct not only its behavioural model, but also the models for all its sub-components. We recall that as the user-defined input, VerCors takes:

- the graphically designed composition of components with the signatures of server, client and local methods;
- the specification of each method of a primitive component given as a UML state machine;
- the type specification;

- the queue size for each component that will be generated;
- the communications that will be hidden in the root component;
- possibly a scenario specified as a UML state machine.

Then, the generation is split into three steps: pre-processing of the input models, pNets generation, and translating the pNets into an input for CADP. The first two phases are described in this section, and the third one is explained in Section 5.2 where we discuss the model-checking with CADP.

Pre-processing. At the first step the conceptual model (i.e. components, UML interfaces, classes, and state machines) is examined and a so-called *pre-processor* builds an analysed object for each component. The pre-processing is necessary for two reasons: first, it allows analysing the model once so that all the information required for constructing pNets is ready to be used. Second, it serves for the optimisation purpose: the pre-processor extracts the dependencies between components and methods so that only communicating entities are synchronised by the synchronisation vectors. This allows reducing the number of generated communications.

For a primitive, the analysed object includes:

- a list of attributes,
- a list of local methods,
- a mapping from the methods of the server interfaces to the implementing methods of a class,
- a set of analysed state machines that model the behaviour of each method

It is important to store the attributes and local methods in a list but not in a set because their order in the list will reflect the order of the corresponding pNets in the synchronisation vectors. A mapping from the methods of the server interfaces to the implementing methods of a class a necessary because one method can be exposed by several interfaces.

In addition to the parsed transition labels, an analysed state machine contains auxiliary information that will be used during the construction of synchronisation vectors. This includes the accessed primitive's attributes and the invoked local and client methods.

The case of a composite is slightly more complex. Apart from extracting the server and client methods from the interfaces, the pre-processor needs to extract the

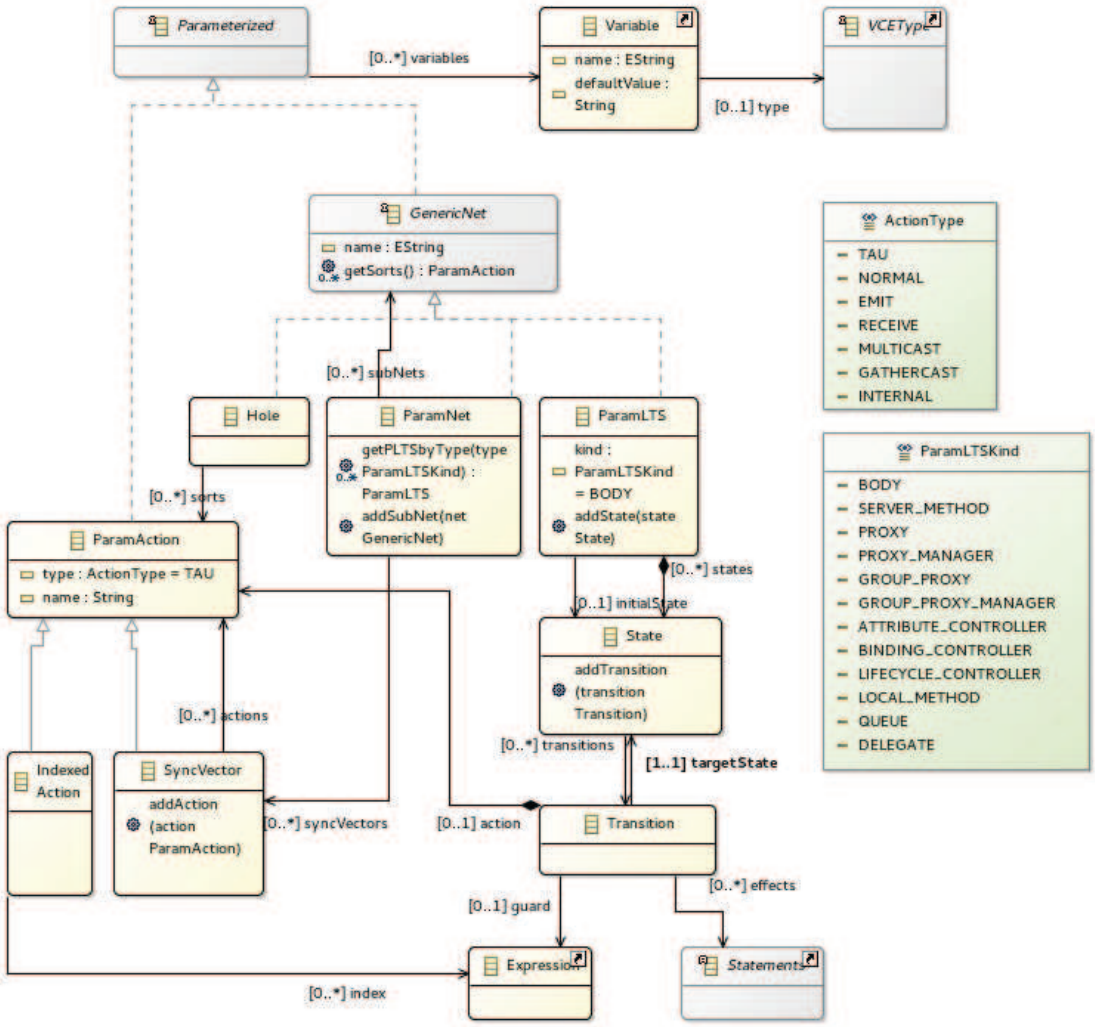


Figure 5.9 – pNets meta-model (simplified)

dependencies between the sub-components, and to map each server method of the composite to the serving component, and the client methods of the sub-components to the client methods of the composite.

Generation. When the auxiliary information has been gathered, VerCors can start building the pNets. The generated structure relies on the pNet classes whose simplified version is illustrated in Figure 5.9. The main difference between the class diagram and the formal definition of pNets is that in the implementation, a `pLTS` is not a subtype of a `pNet`. Instead, we introduce an abstract class `GenericNet` which is extended by a `pLTS`, `pNet` and a `Hole`. A `GenericNet` has an abstract method `getSorts()` which is overridden in each sub-class according to the *Sorts* formalised in Section 2.2.2. Also, a `GenericNet` is an extension of `Parameterized` abstract class which stores a list of parameters as most of our structures are parameterised.

Another class extending the `Parameterized` is a `ParamAction`; as the name indicates, it models a parameterised action. An action has a name (a label), a type: emit, receive or tau (hidden) and a list of parameters. If an action has a type "receive", all its parameters are assumed to be input. This is another deviation from the formalisation where an action can have at the same time input and non-input parameters. Actually, such implementation corresponds to one of the previous version of the formalisation [73] and it is better adapted to the Fiacre language with does not allow mixing input and output parameters in one action (channel). Still, we plan to update the meta-model and to make it coherent with the latest version of pNets formalisation.

A `ParamAction` is extended by the `SyncVector` class which models a synchronisation vector. As a successor of `ParamAction`, `SyncVector` also has a name, a type and a list of parameters. They correspond to the global action expressed by a synchronisation vector. Additionally, `SyncVector` has a list of `ParamActions` which are the parameterised actions involved in the synchronisation.

Another successor of a `ParamAction` is an `IndexedAction` which is used to model indexed actions such as, for instance an action $p \mapsto Recycle_m_i()$ where a proxy at index p is recycled. The indexed actions are mainly used in the synchronisation vectors. The index is implemented as an `Expression` which is not defined here, but it could be a variable, a literal or other types of expressions.

The `ParamNet` class models a pNet and contains a list of `GenericNets` for the sub-nets and a set of synchronisation vectors. The choice of the data structures is important: the order of the sub-nets in the list must correspond to the order of their actions in the synchronisation vectors. This is not guaranteed by the model, but should be taken as a rule during the pNets construction.

The `ParamLTS` class models a pLTS and has a kind, a set of states and a reference to the initial state. The kind indicates which kind of process is expressed by a pLTS: body, server method, proxy, etc. A `State` has a set of outgoing `Transitions` where each transition may have a guard implemented as an `Expression`, a list of effects implemented as `Statement`, a parameterised action, and a reference to the target state.

The last type of a generic net - a `Hole` is not used in this thesis; it is a simple net that does not have any implementation but exposes a set of actions (sort). We use `Holes` for working with so-called open pNets formalised and discussed in [31].

The classes of the presented pNets meta-model can be easily extended. For example, if we decide to model some new kind of a process with pLTSs, we will simply have to add one more literal in the `ParamLTSKind` enumeration. We did this when

we extended the implementation with the pLTSs for the group communications that will be discussed in Chapter 6. If we introduce another type of automata, we will extend the `GenericNet` class; this is what we did when we started working with the open pNets.

Generating primitive components

We implemented the construction of pNets of different entities as independently as possible from each other so that in the future we can easily modify separate elements. This could be useful, for example, when we decide to implement several policies for the body and allow the user to choose among them. Overall, the generation of a primitive component behavioural model involves six builders as illustrated on Figure 5.10: the body builder, the proxy and proxy manager builders, the attribute controller builder, the state machine translator, and the synchronisation vector generator. Each of the constructors is invoked by the main *primitive pNet builder* that assembles the results into a pNet. We include the constructors of the attribute controllers in the figure, we will discuss them in Section 6.2. VerCors does not construct an intermediate representation of the queue but directly generate its Fiacre code. All builders except from the state machine translator construct pLTSs based on some sort of template. For instance, the structure of a proxy is the same for any encoded server method, only the action labels differ depending on the method signature. On the other hand, the structure of a server method pLTS fully depends on the user-defined state-machine and does not follow any pattern.

The case of the local and server method behaviour generation is slightly more complex as it involves the translation of the state machines into pLTSs. The core challenge is to translate the non-void remote method invocations into pLTS actions because we need to know at which location in the pLTS the caller has to obtain the result of the remote method invocation, in other words, where the *GetValue* action has to be inserted. The straightforward solution would be inserting the *GetValue* action just after the remote method invocation but this would force the caller to wait for the result immediately which is too far from what happens in the real implementation and from the futures mechanism. In the current version of VerCors we rely on the notions of *basic blocks*, *future creation points*, and *future use points* in order to provide more parallelism. A basic block is a sequence of states connected by transitions without branching in the intermediate states; incoming branching in the entry state and outgoing branching from the exit state are possible. A future creation point is the statement where a future is created, i.e. a remote method call which result is assigned to a variable. We consider as a future use point an occurrence of the variable in any

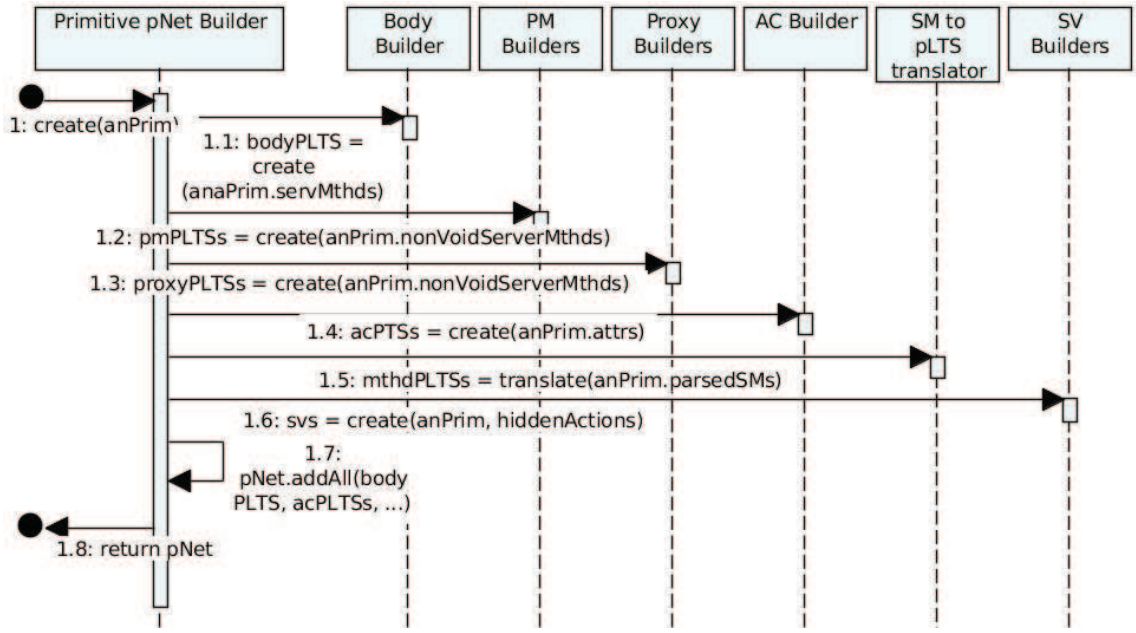


Figure 5.10 – Construction of a pNet of a primitive component

kind of statement or expression except from the case when the variable is used on the left side of an assignment, i.e. its value is "re-assigned". While translating a state machine into a pLTS, we split the state machine into basic blocks, then, we say that the value of each future created within a basic block must be received within the same basic block. This means, that if the future value is used within the the same basic block, we insert the *GetValue* action just before the use point. Otherwise, the value is received at the end of the basic block. We are currently working on a more sophisticated approach for detecting where the future value should be obtained, based on static analysis techniques.

There are a few differences between the formalised structure of the pNet of a primitive component and the one generated by VerCors. First, instead of families of proxies we create only one Java instance of a proxy pLTS for each method. Indeed, at the level of the internal representation of the behavioural graph in Java, we do not unroll the pNets, hence, there is no need to construct several identical proxies. Later, when the pLTSs are translated into Fiacre and EXP, we include as many instances of each proxy in a pNet as there are members in the family.

Next, when generating the internal representation of pNets, we already tend to slightly adapt it to the future translation into Fiacre. Some limitations of the Fiacre language can imply differences between the formal model and the generated pNets. For example, we will translate the pLTS actions into Fiacre channels and the arguments of a Fiacre channel should be either all input arguments or all output arguments. This is not the case for some actions of the pLTSs. For instance, when

a method pLTS gets the future value, it performs *GetValue(p, ?val)* action where p is the proxy index which is known to the pLTS, and *val* is the value that should be received by the pLTS. In this case, we generate two sequential actions in the pLTS of the calling method: *GetValue(p)* and *R_GetValue(?val)* and two corresponding sequential actions in the pLTS of a proxy. The first action defines with which proxy the caller will synchronise, while the second action retrieves the future value. We check carefully that no interleaving of actions can make the semantics different from the original specification.

Then, the constructor of the synchronisation vectors takes as an input the `hiddenActions` structure. It stores the information about communications which should be hidden during model-checking, they can be defined either by the user or by the default policy. Based on this information, the synchronisation vector builder marks certain vectors as "hidden". The instructions for hiding such vectors are then generated in the auxiliary scripts managing the state-space construction.

Finally, the generated synchronisation vectors are slightly different from the formalised ones. First, we implemented construction of several additional vectors that expose actions of pLTSs which could be useful while model-checking. This includes the *ErrorQueue* action which occurs in a pLTS of a queue when it is saturated, *ErrorNoMoreProxy* action which is performed by a proxy manager pLTS when a new proxy cannot be allocated.

Second, in the formalisation we synchronise all server methods of a primitive with all proxies and proxy managers of the client methods of the primitive. This is significantly optimised in the generated vectors: thanks to the pre-processing of the input model, we know exactly with client methods are invoked by the server ones. This allows generating only those communication which actually occur, instead of a full set of all possible interactions.

Generating composite components

Similarly, the generator of the behaviour of a composite invokes several builders constructing the sub-component pNets, the body, the delegate pLTSs, proxies and proxy managers for the client and server methods, the generator of the synchronisation vectors. This is illustrated in Figure 5.11. If the conceptual model includes a scenario state machine, it is also translated into a pLTS and synchronised with the top-level component.

The generation of the pNet of a composite component is very close to the formalisation but still has a few differences. Like for primitives, it constructs only one Java object of a proxy pLTS for each non-void method of an interface, and then a family of

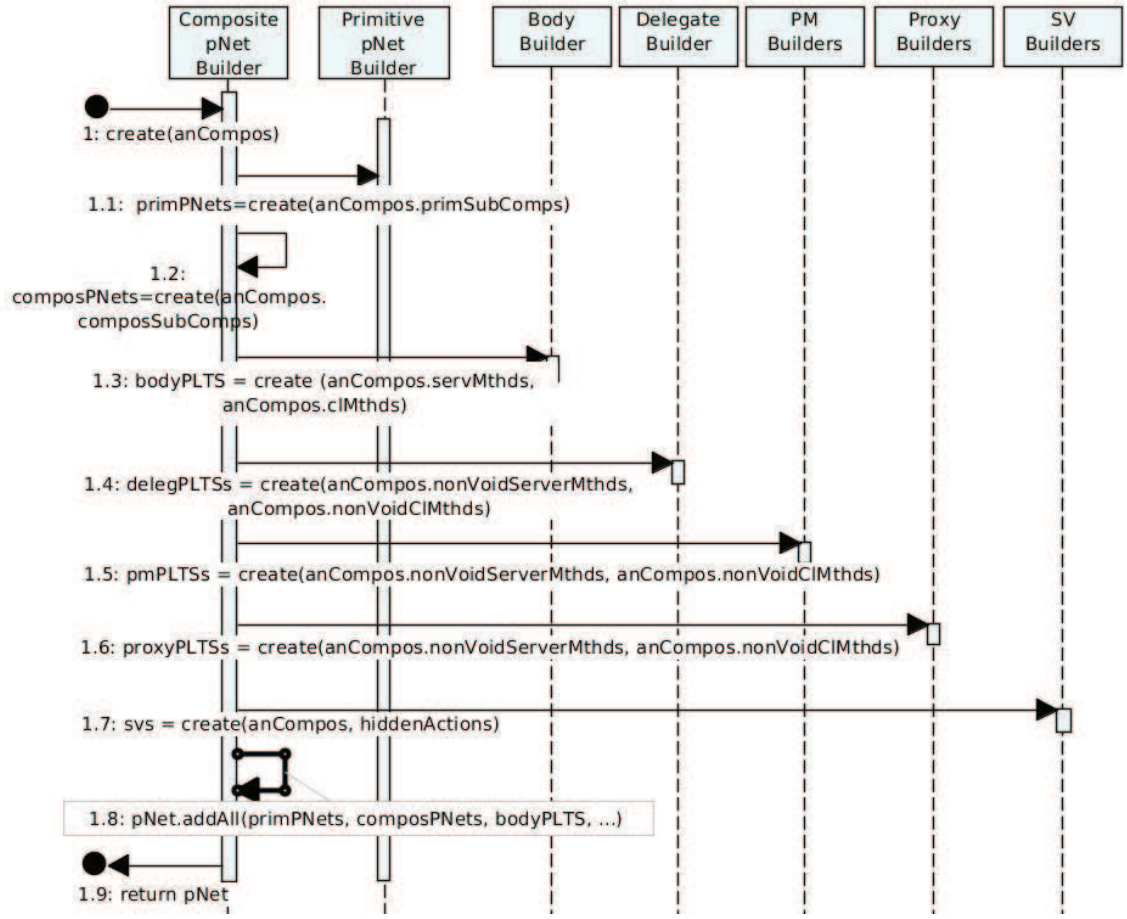


Figure 5.11 – Construction of a pNet of a composite component

instances is included in the model-checked graph. Another difference is that, again, we create additional synchronisation vectors that allow observing the behaviour of the internal pLTSs (the errors from the queue and proxy managers) and pNets of the sub-components. More precisely, the `SV Builder` analyses the synchronisation vectors of the pNets of the subcomponents. For each synchronisation vector that is not marked as "hidden" and that models the internal communication inside a sub-component (for example, a body takes a request from a queue), `SV Builder` takes the global action (the action on the right of a synchronisation vector) and constructs a synchronisation vector that exposes the action. The created synchronisation vector is included in the resulting pNet of the composite under construction.

Summary. There are several advantages in the discussed implementation of the pNets construction. First, the internal behavioural elements (e.g. body, attribute controllers) are generated independently from each other which facilitates modification and substitution of their builders. For instance, this will be useful when we implement several constructors for different body policies. Second, we decouple the

pre-processing phase from the generation phase which allows analysing the conceptual model once and facilitates the pNets construction. Finally, the pre-processor extracts some information which allows optimising the generated model. In the current version we use it for synchronising only the communicating methods and attribute controllers while the straightforward approach would encode the synchronisation between all methods. Still, a lot of optimisation should be done. For example, based on the analysed models we could avoid generating the pLTSs of the local methods that are not used by the other processes.

5.2 From pNets to CADP

The current version of VerCors relies on the model-checker of CADP and in this section we explain how the pNets are translated into the input for CADP and provide the experimental results on the verification of Peterson's leader election algorithm discussed in Section 3.2.

5.2.1 Preparing the input: generating Fiacre, EXP and auxiliary scripts

We recall that in CADP processes are stored as LTSs in BCG format and composed according to rules expressed in EXP format. Additionally, the tool provides techniques for state space minimisation. Instead of generating directly BCG, we rely on a slightly more user-friendly Fiacre format for encoding the pLTSs which can be then translated into BCG by the flac compiler. Overall, in order to prepare the input for CADP, we translate each pLTS involved in a system into Fiacre and the synchronisation vectors of each pNet into EXP. Then, we generate auxiliary scripts that assemble the produced files, build and minimise the state-space in a hierarchical manner.

Fiacre generation. All pLTSs are transformed into Fiacre in the same way regardless of what kind of process they encodes. Figure 5.12 illustrates a pLTS of a body translated into Fiacre. The transformation includes the following steps:

1. Transition analysis: the translator analyses the pLTS and constructs three sets: *Vars* stores the variables of the pLTS, *Types* stores the data types involved in the actions, and *Actions* keeps the actions.

2. Each state of the pLTS is mapped to a Fiacre state characterised by a name.

3. *Types*, *Actions*, state names and *Vars* are translated into Fiacre.

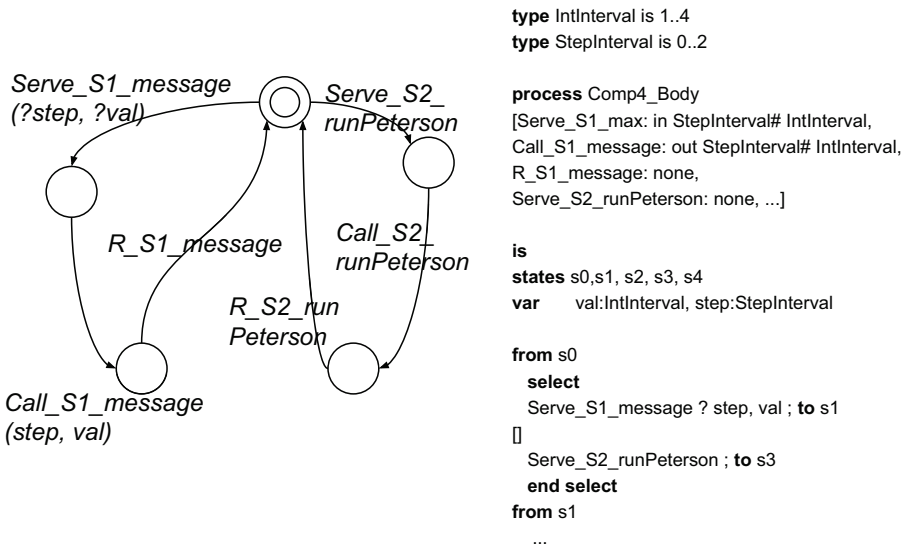


Figure 5.12 – Fiacre code of a body

4. For each pLTS state the translator transforms its outgoing transitions into Fiacre.

EXP generation. The generated processes are then translated into the BCG format and composed by EXP. The translation of the synchronisation vectors into EXP is quite straightforward if all synchronised actions have either no parameters or parameters of the same type. In this case, we translate only the action names into EXP gates and omit the offers.

When the parameters of the synchronised actions have different types, we cannot synchronise actions only by gates but we have to include offers. For this, we explicitly generate each value of a parameter in the synchronisation vector. Let us consider a simple example. Assume, we have two LTSs A and B that are synchronised by a vector $\langle a(x), b(y)\rangle \rightarrow c(x, y)$ where x is of type boolean and y can have a value from an integer interval from 2 to 3. If we try to express such synchronisation in EXP using only gates (i.e. $\langle a, b\rangle \rightarrow c$), CADP will not accept the vector because the synchronised actions have different offers (i.e. different types of parameters). As an overcome, we instantiate with all possible values all parameters whose types do not match and generate a synchronisation vector for each possible instantiation:

$$< \mathsf{a\ !0,\ b\ !2} > \rightarrow \mathsf{c\ !0\ !2}$$
$$< \mathsf{a\ !1,\ b\ !2} > \rightarrow \mathsf{c\ !1\ !2}$$
$$< \mathsf{a\ !0,\ b\ !3} > \rightarrow \mathsf{c\ !0\ !3}$$
$$< \mathsf{a\ !1,\ b\ !3} > \rightarrow \mathsf{c\ !1\ !3}$$

Now, the synchronised actions are specified as a gate followed by offers and such synchronisation can be done.

Another limitation of EXP is that it does not support indexed actions which are used in several synchronisation vectors of the GCM pNets. For instance, an indexed action is needed when a server method of a primitive component accesses a proxy at index p in order to retrieve the result of a remote method invocation: $\langle GetValue_m(p, val), p \rightarrow GetValue_m(val) \rangle \rightarrow GetValue_m(p, val)$. While constructing such kind of communications in EXP, we rely on the fact that the number of proxies is fixed at the time when .exp files are created, and that the generated EXP synchronisation vector includes as many proxies as there are in the family. Hence, we can explicitly instantiate the value of p and synchronise the server method action with a proxy at each possible index depending on the value of p. For instance, if the family size is equal to two, our example synchronisation vector will be translated into two EXP vectors as follows:

$$< \mathsf{GetValue_m!0}, \mathsf{GetValue}, - > \rightarrow \mathsf{GetValue_m!0}$$
$$< \mathsf{GetValue_m!1}, -, \mathsf{GetValue} > \rightarrow \mathsf{GetValue_m!1}$$

Scripts generation. When all pieces of the behavioural model have been produced, VerCors generates scripts for the automata construction and state space reduction. We first discuss in general the algorithm encoded by the scripts, and then we give the technical details on what kind of scripts are generated.

```
procedure BUILDPNET(net)
    for each subnet in net do
        if subnet is not pLTS then
            subnet.bcg ← buildPNet(subnet)
        else
            subnet.bcg ← flac(subnet.fiacre)
            subnet.bcg ← minimise(subnet.bcg)
        end if
    end for
    net.bcg ← generate_and_minimise(net.exp)
    if net ≠ root then
        net_complete.bcg ← net.bcg
        net.bcg ← hide_and_minimise(net.bcg)
    end if
     return net.bcg
end procedure
```

Algorithm 1 Constructing CADP input

The scripts encode the Algorithm 1 which is based on the bottom-up approach for behavioural model construction. The `buildPNet` procedure described by the algorithm constructs the `.bcg` file for a pNet given as an input and the `.bcg` files for all its sub-nodes as follows. First, it iterates over all sub-nets and constructs their `.bcg` files. For this, the procedure invokes `buildPNet` for each the sub-pNet which is not a pLTS; for each sub-pLTS it invokes the Flac compiler which translates the pLTS into BCG format and the bcg_min tool that minimises the state-space of the constructed `.bcg` file. After all pieces of the pNet have been constructed, `buildPNet` invokes EXP.OPEN which assembles the sub-nets with respect to the synchronisation vectors and creates a minimised `.bcg` file of the resulting automaton. If the generated pNet does not model the root component, the scripts make a copy of its `.bcg` file, hide some of its labels, and minimise the state space. Copying the initial file could be optional, it is needed in order to allow the user to model-check the original behaviour of a sub-component that is not affected by its container.

```
1 "Application_Comp1_complete.bcg"=  branching reduction of "Application_Comp1_SV.exp";
2 "Application_Comp1.bcg" = branching reduction of gate hide
3   "CALL_SET_MAX", "R_GET_MAX", ...
4 in "Application_Comp1_complete.bcg"
```

Listing 5.1 – An example of SVL script

More technically, for each component being translated into input for the model-checker (i.e. the root component and its sub-components at all levels of hierarchy), VerCors generates a `.svl` file. Listing 5.1 presents an example of the script generated for one of the participants of the leader election use-case from Figure 3.4. The SVL script constructs the `.bcg` file modelling the component behaviour (line 1). Next, if the component is not the root (which is the case for our example) the script hides some of the internal communications (for instance, the access to attribute controllers), minimises the state space and produces the final model (lines 2-4). As a result, two files are generated: `Application_Comp1_complete.bcg` which represents the full behavioural graph of the component, and `Application_Comp1.bcg` in which the internal communications are hidden.

Additionally, VerCors generates one `.sh` file for the whole model being constructed. The script invokes the Flac compiler on a `.fiacre` file of each pLTS included in the tree of the generated system. Then, it triggers execution of all produced SVL scripts starting from the components at the lowest levels of hierarchy and finishing by the root.

Table 5.5 – Behaviour graph files (all with Queue size of 3)

Graph	States	Transitions	Computation time
Behaviour of `Comp4`	3.217.983	45.055.266	2m48.520s
Comp4 (internal communication hidden, minimized by branching simulation)	90.821	1.306.138	5m23.030s
full application	296	661	47m1.673s

5.2.2 Model-checking with CADP

Following the procedure described in the previous section, VerCors produces an input for the CADP model-checker. We experimented with generating and model-checking the behaviour of our use-case example of the leader election algorithm illustrated in Figure 3.4. Table 5.5 presents size information for some of the intermediate behaviour graphs. The last line is for the hierarchical construction of the full model of the application (including the scenario which triggers the election process once), and the time includes the whole model-generation workflow. The time needed to generate .fiacre, .exp files and scripts from VerCors is negligible.

We use the Model Checking Language (MCL) [44] to express the behavioural properties we want to prove on our system. MCL is the input language of CADP allowing one to express various temporal logic formulas (CTL, ACTL, PDL) together with a predicate logic on data values, which will be verified on an input LTS. We make a short overview of the part of MCL syntax used in this thesis and then give examples of properties that we have checked on our use-case example.

MCL. Basic MCL allows specifying expressions, action formulas, regular formulas, and state formulas. An action in an MCL formula is a transition label of the model-checked LTS which represents a gate (action name) and possibly a list of offers (action parameters). An *expression* can be constructed from literals, data variables, unary and binary operators. *Action formulas* are logical formulas built from action predicates, regular expressions, and boolean operators. *Action predicates* provide a mechanism for specifying transition labels whose offers match a certain specification.

Action formulas can be assembled into *Regular formulas* which rely on the following elements:

- A single action formula `A` is a regular formula which is satisfied by a single transition whose label satisfies `A`. For example, a regular formula `"Call_setValue.*"` is satisfied by all transitions whose labels start by `"Call_setValue"`.

- Regular formulas can be concatenated using `"."` operator: a formula `R1.R2` is satisfied by a sequence of transitions which consists of two concatenated sub-sequences where the first sub-sequence satisfies `R1` and the second one satisfies `R2`.

- The operators `"*", "+", "?"` are used to express a repetition (a sequence satisfying the formula is repeated zero or more times), a strict repetition (a sequence satisfying the formula is repeated one or more times), and an option formula (a sequence satisfying the formula occurs once or does not occur at all) correspondingly.

Finally, expressions, action formulas, and regular formulas can be assembled into *state formulas* which rely on various operators and modalities and allow one to define predicates over the set of states of an LTS. In this thesis we will rely on the formulas using the following modalities and operators:

- the possibility modality `"<" R ">" F` which is satisfied iff there exists a transition sequence (a path) in the input LTS which goes from a state satisfying a regular formula `R` to the state satisfying a state formula `F`;

- the necessity modality `"[" R "]" F` which is satisfied iff all paths going from a state satisfying a regular formula `R` lead to a state satisfying `F`;

- classical logical operators like ”or”, ”xor”, ”end”;

- `"true"` formula which is satisfied by any state and `"false"` formula which cannot be satisfied by any state.

MCL allows one to define custom macros and libraries of operators parameterised by action and state formulas. Moreover, CADP includes a set of libraries with MCL operators which can be used from the user-defined formulas. Examples of such operators are given below:

- `Absence_Before(R1, R2)` is satisfied iff a sequence of transitions satisfying `R1` never occurs before a sequence of transitions satisfying `R2`;

- `Inev(R)` is true iff all transition sequences lead to a transition sequence satisfying `R`;

Examples of properties. We used MCL in order to specify properties of our use-case example in the context of a scenario where the election algorithm is triggered only once.

An important parameter of the state-space generation is the size of the request queues of our components. Indeed, the size of an LTS encoding explicitly the states of a queue of max length L, receiving N possible different values is in the order of $O(N^L)$, so it is important to use small sizes both for the domain of request parameters, and for the length of the queue. However, if we use a length too small, then it is possible that the queue saturates during the normal activity of the application. We encode a formula checking that the queue of a component cannot be saturated. Remark that this check involves the exhaustive analysis of the behaviour and interactions of the components, and this can be done only by the model-checker. To be able to detect saturation in the model-checker, our Queue model includes a specific event 'Comp_ErrorQueue'.

An example of such a formula for `Comp1` is given below:

<true* . 'Comp1_ErrorQueue.*' >true

This formula means that the `Comp1_ErrorQueue` action is reachable in the behaviour graph. If we set the length of `Comp1` queue to 2, the model-checker answers `true`: the queue can saturate. If we set it to 3, the result is `false`, we are safe.

Now we do a similar query for the whole application. If the queue size of all the components is set to 3 the formula evaluates to `true`. The model-checker gives us a diagnostic in the form of a path in the global graph: each primitive component can drop a request in the application queue, corresponding to calls on the client interface `LeaderItf`. If 3 requests are not served before the 4^{th} arrives, then the queue saturates. If we set the composite queue length to 4, the queue never saturates.

Now that we have proved that our model is not limited by the size of queues, we can prove some of its functional properties. We check that after a call to `runPeterson()`, it is inevitable (under fairness hypothesis) that either the leader is elected or one of the queues is saturated. The model-checker answers `true`: the election terminates. We also proved that with adequate queue size, they never saturate.

['Call_RunPeterson'] Inev ('Q_IamTheLeader.*' or 'ErrorQueue.*')

Then, we prove that the event `Q_IamTheLeader` is emitted only once (i.e. only one leader is elected):

Absence_Before ('Q_IamTheLeader.*', 'Q_IamTheLeader.*')"

Recall that in order to illustrate the future-based communications, we extended our use-case with the methods `requestKey` and `encrypt`. The former is a client

method invoked by the leader component in order to obtain an encryption key. Since it is a client method, the variable which should store the received result in the leader component is a future. Hence, the leader component should not get blocked after the method invocation but continue performing computations which do not involve the encryption key. In particular, just after asking for the encryption key, the leader component invokes the `IAmTheLeader` method on its client interface in order to report that it is the leader. In order to check that the communications in the generated graph are indeed implementing futures properly, we verify the following formula which states that a key must be always received before IamTheLeader() is invoked:

Existence_Between('R_RequestKey.*', 'Q_requestKey.*', 'Q_IamTheLeader.*')

The model-checker answers `false` and provides an example of system behaviour where `IamTheLeader()` method is invoked before the key is received. This proves that a component is not blocked if the key is not needed.

5.3 Code generation and execution

When the user has checked that the conceptual model of the designed application is statically correct with respect to the rules formalised in Chapter 4 and if the model-checker has proven the desired functional properties of the modelled system as discussed in the previous section, then VerCors can generate the implementation code of the designed application. The generated code is ready to run in the GCM/ProActive environment.

Figure 5.13 illustrates the general workflow of the process. The generator takes as an input a component diagram with components and connections between them, a UML class diagram with UML classes and interfaces, UML state machine diagrams specifying the behaviour of the server methods and a type diagram. Additionally, the platform asks the user to specify the name of the package in which the generated files will be placed. Before starting the generation, VerCors invokes the state machine parser which parses the UML state machines in the same way as for pNets construction (this step is omitted in the figure). Then, two generators are applied to transform the user-defined conceptual model into an input for ProActive: an ADL generator and a Java code generator. As explained in Section 2.1.3, ProActive takes as an input the component architecture specified in a GCM/ADL XML-based file, a set of Java classes implementing the behaviour of the primitive components and a set of Java interfaces with the signatures of GCM interfaces.

In this section we first discuss the ADL and Java code generators and then the execution of the produced code.

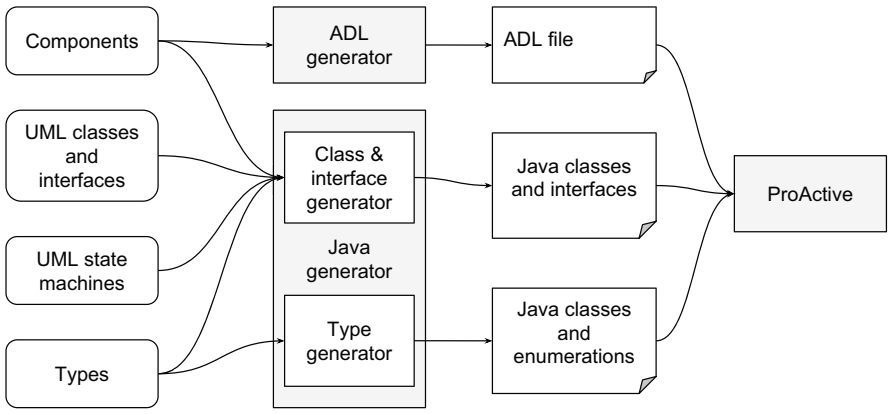


Figure 5.13 – The workflow of implementation code generation

5.3.1 ADL generation

We start by an overview of the ADL generator which transforms a component architecture model into an XML-based file. The kernel of its current version is mainly based on the code taken from the previous versions of VerCors and adapted by an engineer who worked on the project, hence, the content of this sub-section should not be considered as an exclusive contribution of this thesis. Still, we extended the generator with the non-functional part construction as will be explained in 6.

```
1 @XmlRootElement(name = "component")
2 public class JAXBComponent {
3     private String name;
4     private List<JAXBInterface> interface;
5     @XmlAttribute(name = "content class})
6     private String clazz;
7     ...
8 }
9
10 public static void main() {
11   JAXBComponent c = new JAXBComponent();
12   JAXBInterface itf = new JAXBInterface();
13   itf.setName("myItf");
14   itf.setType("server");
15   c.setName("myComponent");
16   c.getInterfaces().add(itf);
17   c.setClazz("ImplClass");
18 }
```

Listing 5.2 – An example of a JAXB-based class

```
1 <component name="myComponent">
2   <interface name="myItf" role="server"/>
3   <content class="ImplClass"/>
4 </component>
```

Listing 5.3 – An XML file generated by JAXB

The JAXB framework. Converting Java objects into XML is a very classical task which is supported by multiple libraries. The ADL generator of VerCors is implemented using the JAXB (Java Architecture for XML Binding) technology [82] which is a Java library for converting Java objects to XML representations (the process is also known as "marshalling"). Listing 5.2 illustrates the usage of JAXB. Lines 1-8 present an example of a class which stores a model of a simple component characterised by a name, a list of exposed interfaces and an implementation class. We omit the definitions of the companion `JAXBInterface` class. Lines 10-17 demonstrate an instantiation of the given class, and Listing 5.3 shows how the constructed object is marshalled. JAXB requires a Java class being translated into XML to be annotated with *@XmlRootElement* (line 1). This and the other annotations used for the mapping can be followed by additional parameters. For example, the programmer can specify the name of the generated XML element as an attribute of *@XmlRootElement* annotation. By default, the class name is used for this purpose. The technology allows for straightforward mapping of class fields, i.e. each field (of a generated class) of type String is mapped to an XML element with the corresponding name and value without any additional effort from the programmer. For instance, the field `name` from line 3is translated this way. The fields whose types are Java classes annotated with *@XmlRootElement* (like the interface at line 4) are also mapped into XML elements. The programmer can use annotations in order to customise the mapping. For instance *@XmlAttribute* with additional parameters preceding a field of a class allows specifying properties of the generated XML attribute. By default, he order of the generated entities reflects the order of the corresponding fields in the Java classes.

The generator. The ADL generator takes as an input Java objects computed from the GCM components which were defined by the user in the front-end graphical editor. We recall that those objects are based on the EMF technology and their generated classes do not include the annotations required for using JAXB. Hence, VerCors needs an intermediate set of classes that are able to store the component model and could be translated into XML. We will call them JAXB classes. VerCors has such classes for the GCM components (a more complex version of the class shown in Listing 5.2), interfaces, bindings and the rest of the GCM elements but not for the UML elements because they are not translated into XML.

Overall, the ADL construction procedure has the following workflow. The generator takes as an input the user-defined components and starting from the root component it converts them into JAXB objects. Then, it invokes a dedicated factory from the JAXB library which prints JAXB objects in an XML-based file.

We plan to reuse the JAXB objects for the plug-in dedicated to reverse engineering ADL files into the models of VerCors graphical front-end. Indeed, the JAXB technology also supports "unmarshalling" XML files, i.e. translating an XML file into Java objects. We could rely on this mechanism in order to allow the user to import an existing ADL file into VerCors. This could be particularly useful for analysing GCM-based projects which were not initially designed in VerCors.

5.3.2 Java generation

The Java code generation includes three main steps: translating types into Java classes, converting UML interfaces into Java interfaces and translating UML classes with state machines defining method behaviour into Java classes. All three steps are based on the Acceleo [58] technology dedicated to model-to-text transformation.

```
1 [template public generateRecord(rtype : RecordType)]
2 [file (rtype.getTypeName().concat('.java'), false, 'UTF−8')]
3 [if not(getPackage().toString().equalsIgnoreCase(''))]
4 package [getPackage(true) /].types;
5 [/if]
6
7 import java.io.Serializable;
8
9 public class [rtype.getTypeName()/] implements Serializable {
10 [for (field : Field | rtype.fields)]
11   [getTypeName(field.type)/] [field.name/];
12 [/for]
13 }
14 [/file]
15 [/template]
```

Listing 5.4 – An Acceleo template translating VerCors record type into a Java class

Acceleo and type generator. Acceleo follows a template-based approach for model-to-text transformation. In a template-based approach, a programmer should define a so-called *template* in a dedicated language (in our case it is the Acceleo language) that would take an object being translated and use a sequence of the language statements in order to construct the corresponding text. An Acceleo template has access to the fields of the object, allows for classical control-flow statements including loops and if-else conditions, and is able to invoke external Java services. From the programmer-defined templates, Acceleo automatically produces Java code of the corresponding generator (model-to-text translator) which can be invoked from elsewhere.

Listing 5.4 demonstrates the simplest Acceleo template implemented in VerCors. It translates a record type into a Java class. The template signature is given at line 1:

it specifies the template name and an input parameter `rtype` of type `RecordType`. Line 2 defines the name of the file where the text will be printed and lines 3-13 define the content of the file. As for any other Java class, the generated code should include the name of the package in which the class is included. For this, Acceleo invokes an external Java service at line 3 in order to get the user-defined package name and if it is not empty, concatenates the name with *".type"* which means that the generated type will be included in the "package_name.types" package. Line 7 prints an imported class. Line 9 specifies the signature of the class being printed: for this, it gets the name of the record type. Finally, the loop at lines 10-12 iterates over all fields of a record and for each field it prints the type and the name. In order to get the name of a field type, the template also invokes an external Java service which maps a VerCors type into a Java type.

Listing 5.5 provides an example of a record type `RecordExample` translated into a Java class. The type has two fields: a boolean field `b`, and a field `e` of a user-defined enumeration type `EnumType` which definition is omitted here.

```
1 package example.types;
2
3 import java.io.Serializable;
4
5 public class RecordExample implements Serializable {
6   Boolean b;
7   EnumType e;
8 }
```

Listing 5.5 – Java code of a record type generated by VerCors

We defined templates for the record and enumeration VerCors types. Then we used Acceleo in order to produce the corresponding code generator. While constructing the implementation code, VerCors invokes the generator for each record and enumeration type of the user-defined type diagrams; the order in which the Java classes are produced is not important.

Translating UML interfaces and classes. Similarly, we defined templates translating UML interfaces and classes into Java code. The case of interfaces is quite trivial: the template iterates over the UML methods of an interface and for each method it generates the signature in Java. Converting UML classes is, however, more complex. We recall that the UML classes are used in VerCors in order to specify the implementation of primitive components business logic. We say that a class is "attached" to a primitive when the behaviour and the attributes of the component are defined by the class.

```
1 public class Class0 implements Serializable, BindingController, ElectionItf, Class0AC{
2   //local variables
3   public boolean isActive = false;
4   public int left = 1;
5   public int cnum = 1;
6   public int max = 1;
7
8   //client interfaces
9   protected ElectionItf C1;
10  protected MonitorItf C2;
11  protected KeyStorageItf C3;
12
13  //Binding controller's methods
14  public void bindFc(...) {...}
15  public String[] listFc() {...}
16  ...
17
18  //server methods
19  public  void message(int step, int val) {...}
20
21  //local methods
22  public  void encrypt(int key) {...}
23
24  //Attribute controller methods
25  public  void set_max(int value) {...}
26  ...}
```

Listing 5.6 – Generated Java code of a primitive component

While translating a class attached to a primitive, we have to take into account the component's server and client interfaces, the attributes and operations of the class. Listing 5.6 demonstrates a simplified version of the `Class0` attached to one of the primitive components participating in the Peterson's leader election algorithm modelled in Chapter 3; Figure 5.14 illustrates once again this component. The Acceleo template dedicated to a UML class translation invokes an external Java service in order to obtain the set of server and client interfaces of a primitive. The server interfaces are included in the list of interfaces implemented by the class (`ElectionItf` at line 1), the client interfaces are converted into the fields of the class (lines 8-11) which can be then accessed by the class methods in order to perform a remote method invocation. Additionally, we generate several auxiliary methods which will be used by the ProActive factory in order to bind or access the interfaces (lines 13-15). The user-defined class attributes are translated into class fields (lines 2-6). Finally, VerCors generates the signatures and the bodies of the user-defined server and local methods (line 19). We distinguish three kinds of methods: methods with a state machine specifying the behaviour, methods for which the user did not provide behaviour description, and attribute set/get methods. The first case involves translation of a state machine into Java code which will be explained in the next paragraph. If the

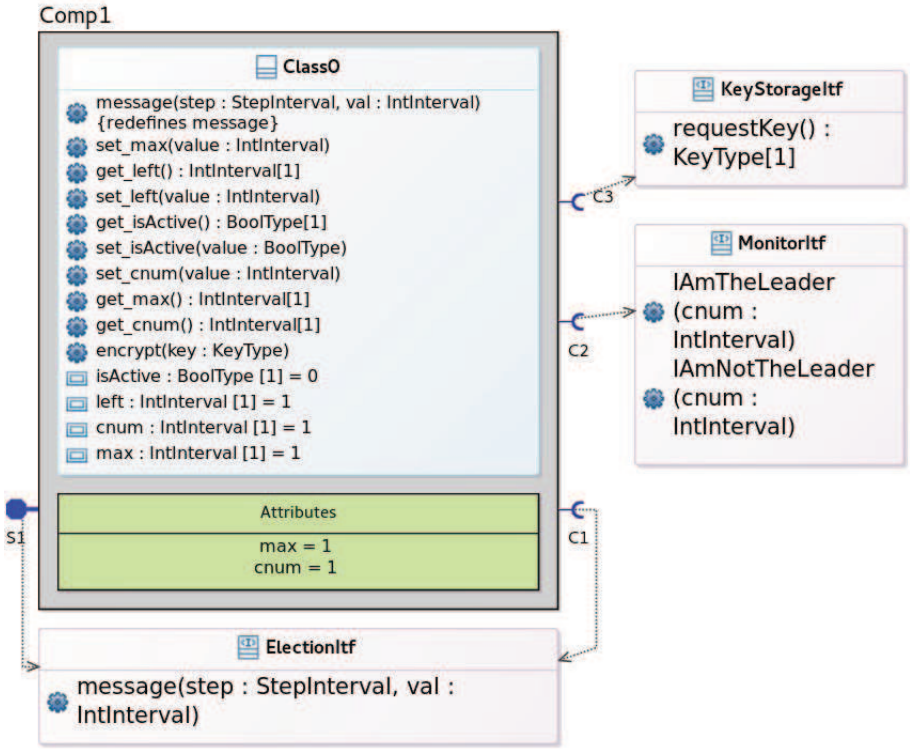


Figure 5.14 – A primitive with an attached UML class

user has not specified a method behaviour, an empty body is generated. Finally, VerCors produces standard code for the get- and set-methods which will be discussed in Section 6.2

Translating UML state machines into Java code. VerCors allows specifying one state diagram for each method of a primitive component which is then translated into Java code. Multiple studies [83, 84, 85] are dedicated to the translation of various types of state machine diagrams into executable code; most of them try to deal with the state machine hierarchy. However, the translation in our case is much simpler because a state machine in VerCors has only one region, only one level of hierarchy and no actions assigned to the states.

As a part of the transformation process, VerCors constructs an enumeration `State` which stores the names of all states of the state machines being translated. Note that here we generate one enumeration type for all state machines modelling the behaviour of a given application. Each generated method has a local variable `curState` of type `State` which holds the current state of the state machine and actions are taken in a switch-case statement depending on the value of the variable. Listing 5.7 demonstrates the Java code constructed by the platform for the state machine from Figure 5.15. In addition to the `curState` variable, VerCors translates the user-defined

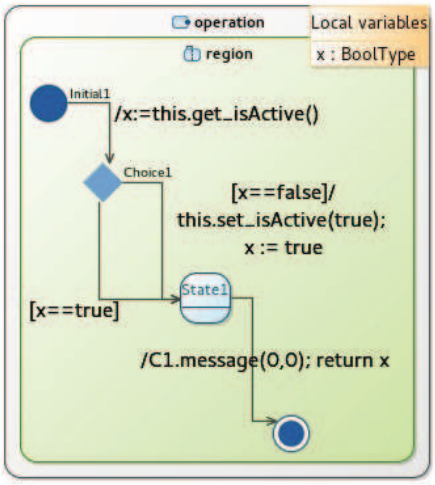


Figure 5.15 – A simple state machine

local variables of a state machine (line 2). Then, the platform generates an infinite loop comprising a switch-case statement which checks the value of `curState`. For each state of a state machine VerCors creates a `case` statement that includes the Java code corresponding to the label of the outgoing transitions. Note, that an `if-else` statement is included in the case of multiple outgoing transitions (lines 11-20). In the current version, the translation of UML state machines is fully implemented in Java, but we believe, we could benefit from porting it to Acceleo. The reason is that the Acceleo code generators are easier to maintain and the code of the model-to-text transformation templates looks cleaner than a Java code generator with printing instructions.

```
1  State curState = State.Initial;
2  boolean x;
3  while(true) {
4    switch (curState) {
5    case Initial:
6      x = this.get_isActive();
7      curState = State.Choice1; break;
8    case Choice1:
9      if(x == true) {
10       curState = State.State1; break;
11     }
12     else if(x == false) {
13       this.set_isActive(true);
14       x = true;
15       curState = State.State1; break;
16     }
17   case State1:
18     C1.message(0, 0); return x;
19 }}}
```

Listing 5.7 – Generated Java code of a state machine

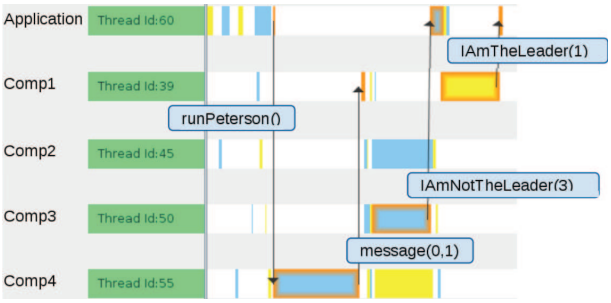


Figure 5.16 – Code execution

The advantage of such an approach is that the generated code mirrors the state machine structure. However, a significant drawback is that the code is long and not very convenient for the programmer since *do-while, for, while* constructs cannot be written as such in the state machine, but will rather be encoded within the state structure, separated by case instructions. We still discuss the possibility to enhance our framework by implementing a plug-in which would be able to recognise the classical control-flow patterns in a state machine and to generate more user-friendly code. On one hand, we would like to improve the quality and the readability of the produced code. On the other hand, this would require significant effort on the static analysis and restrict the user-defined input. This could be useful if we expected the user to modify the generated code which is not the case in our framework, because the constructed implementation has been model-checked by VerCors, and if the user tries to modify the logic of the generated methods, the verified functional properties cannot be guaranteed any more.

5.3.3 Code execution

The code generated by VerCors can be executed on top of the ProActive platform. In order to check that the produced application, indeed, behaves as we expect, we use a dedicated visualiser [86] which analyses and shows the request exchange between ProActive active objects at run-time. For instance, we generated ProActive/Java code of our use-case example of Peterson's leader election algorithm from Figure 3.4; the resulting execution is shown in Figure 5.16. Black arrows represent request emissions (the figure only shows some of them). Yellow and blue rectangles show request processing. For example, we can see how the call to *runPeterson* of Application is transmitted to Comp4 and at the end of the *runPeterson* request processing Comp4 triggers the elections on Comp1 by calling *message(0,1)*. At the end of the algorithm execution we can see how Comp3 reports to the Application that it is not the leader and Comp1 claims to be the leader.

5.4 Discussion

5.4.1 On the verification

In this chapter we presented how conceptual models graphically specified in VerCors can be automatically translated into an input for the CADP model-checker. The transformation includes several steps: first, we generate an internal representation of system behaviour encoded in pNets, then, the constructed model is translated into Fiacre and EXP formats, finally, the created files are translated by Flac and CADP processes into BCG format which can be given as an input to the Evaluator model-checker. Here we discuss some choices that we had to make while constructing the verification part of the platform, some ideas for the future work, and we highlight several advantages of the presented approach.

First, we should mention the choice of the underlying technologies and formats. As it was discussed in Section 3.4, we believe that our framework benefits from using pNets as an intermediate format for two main reasons. First, as a parameterised structure, pNets can be potentially transformed into an input for an infinite state-space verifier, and, second, they bridge the gap between the graphically specified components and the hierarchical composition of automata taken as an input by CADP. Using Fiacre as an intermediate format could be, in fact, avoided. We could try to generate LotosNT files directly from pNets, but we believe that as the Fiacre format is very close to the pLTSs constructed by VerCors, it facilitated the implementation and debugging of the generator. The usage of the CADP verification platform will be justified in Section 7.5 where we make an overview of several other verification platforms.

Second, we have several ideas of how the pNets encoding the behaviour of various processes of GCM components could be enhanced. For instance, the state machine translation process could be improved by applying static analysis techniques in order to detect more precisely where a server or a local method should retrieve the result of a remote method invocation. This would make the behavioural model closer to the GCM/ProActive implementation and provide maximum parallelism. Furthermore, the fact that we model processes which get blocked quite early increases the probability of a deadlock in the behaviour graph which would not occur in the real implementation.

Regarding the state-space reduction, we believe, that our framework benefits from several applied techniques. First, we rely on the possibility of hiding the internal actions of sub-components that should not be observed during model-checking, which can be followed by significant state-space reduction based on bisimulation minimisa-

tion techniques. Second, modelling the environment scenario has significant impact on the generated system as it reduces the combinations of input requests which can be received by the designed application. In fact, the application of state-space reduction techniques to the behavioural models of GCM components has been studied in [24] .

Additionally, we already do some optimisation of the generated pNet model: we analyse the behaviour of primitive components and the structure of composite ones, and we construct only those communications that actually occur. In addition, we do not generate the proxy, proxy manager and delegate pLTSs for the methods which do not return any result. Still, we believe that there are more possibilities to optimise the produced structure. For example, we could statically detect that a particular client method of a primitive is invoked only once; in this case, the generation of the proxy manager would not be needed.

5.4.2 On the executable code generation

In this chapter we also presented the generation of the executable code from VerCors. We would like to highlight that this is the first version of the platform which fully automatically produces the business logic code for the modelled components. In fact, the generation process is quite straightforward, but still its implementation involved a few technical choices.

First, we would like to discuss the choice of the plug-in architecture. We separated the ADL generator from the Java code generator for two main reasosns. First, we allow the user to generate only an ADL file without the Java classes. This can be useful for the users who do not model and verify component business logic but provide it separately from the model of component architecture specified in VerCors. Another reason is that we would not like the changes in one of the generators to affect another one and to keep both generators independent. For instance, it is very likely that in the future we will have to implement an alternative ADL generator that will support parameterised topologies as discussed in [87], and we would like to be able to reuse the same Java code generator.

Second, there are several strategies for generating the ADL files: in the current version of VerCors we construct one ADL file containing the description of the root component and its sub-components. Another possibility would be producing one ADL file for each generated component and including references from the ADL of a composite to the files containing its sub-components. The first strategy is convenient for debugging the generated application because it does not require switching between multiple files. On the other hand, producing one file per component allows for more re-usability: the programmer can use the sub-components' ADL independently from

their containers. However, in this case VerCors cannot guarantee those properties which were proven by the model-checker for a sub-component in the context of its container. We believe that both strategies should be implemented in VerCors so that the user can choose between them. The reason why we opted for a single file construction is related only to the ease of debugging. The good point is that the intermediate JAXB objects are already ready to carry the information necessary for implementing the second strategy. We only need to take care of the way the JAXB factory is invoked to construct the ADL files, and of the paths to the generated files.

Next, we did not actually choose the underlying technology for the ADL generator implementation, because we reused the code from the previous version of VerCors. We believe that there was no need for changing the technology, because JAXB is still maintained, it was relatively easy to adapt the code to the ecore models of the new VerCors version. The choice of the approach for Java code generation was also quite obvious, because Acceleo is a state-of-the art model-to-text transformation technique which is well-integrated with Obeo Designer.

Finally, the current version of the Java code generator relies on a very simple algorithm for translating state machines into executable code. It traverses the states of a state machine in a random order and for each state it constructs a `case` statement which encodes the outgoing transitions. The advantage of such approach is that the produced code mirrors the modelled state machine. On the other hand, the generated code is not very "user-friendly", and enhancing is in the scope of the future work as it will be discussed in Section 8.2.

In this chapter we discussed an approach for translating the graphical model of a GCM-based application architecture and behaviour into an input for the model-checker and executable Java code. In the next chapter we provide the details on modelling, verification and executable code generation of the advanced component features such attribute controllers, group communications and non-functional aspects.

Chapter 6

Modelling and verifying advanced GCM features with VerCors

Contents

6.1 Non-functional components and interceptors 129

6.1.1 From application design to pNets 129

6.1.2 Implementing pNet generation and integration with CADP 134

6.1.3 Code generation 134

6.2 Component attributes and attribute controllers 135

6.2.1 Graphical specification 136

6.2.2 From application design to pNets 137

6.2.3 Implementing pNet generation and integration with CADP 138

6.2.4 Code generation 139

6.3 Reconfigurable multicast interfaces 140

6.3.1 Graphical specification 141

6.3.2 From application design to pNets 142

6.3.3 Implementing pNet generation and integration with CADP 155

6.3.4 Code generation 157

6.4 Reconfiguring multicasts from NF components 157

6.4.1 Graphical specification 157

6.4.2 From application design to pNets 158

6.4.3 Implementing pNet generation and integration with CADP 158

6.4.4 Code generation 159

6.5 Examples 161
6.5.1 Composite pattern 161
6.5.2 Springoo 171
6.6 Discussion 173

In this chapter we explain how the advanced features of GCM components can be modelled, verified and generated in our framework. We start by presenting the design and verification of the non-functional part of GCM components. Second, we discuss the attribute controllers and access to the attributes of primitive components. Then, we present our approach to modelling and verification of applications with multicast interfaces which enable N-to-one communications. In such communications, a request from one component can be sent to several targets simultaneously. Moreover, the presented multicast interfaces can be reconfigured at run-time, i.e. the group of the target components can be modified during program execution. Next, we integrate the definition of the non-functional part of a component and the reconfigurable multicast interfaces. More precisely, we explain how the multicast interfaces can be reconfigured by component-controllers located in the membrane. We present an illustrative example where we model and verify an application including all the discussed advanced features. Finally, we summarise the contribution of the chapter and discuss shortly the future work.

Auxiliary operators. In this chapter we will often modify the structure of the pNets of primitive and composite components: we will extend them with other subnets and synchronisation vectors. For this, we will rely on the following operators:

- $\oslash$ - "restriction" : Let $pNet$ be a pNet and $\mathcal{A}$ be an indexed set of labels (strings); $pNet \oslash \mathcal{A}$ returns a pNet similar to $pNet$ but with restricted synchronisation vectors. The synchronisation vectors of $pNet \oslash \mathcal{A}$ are the ones of of $pNet$ except all the synchronisation vectors containing an element of $\mathcal{A}$ as part of their global synchronisation label. For example, if m belongs to $\mathcal{A}$ then all the vectors containing m in their global label will be removed, in that case the global labels concerned could be: Q_m, $!Q_m$, $?Q_m$, R_m, $Serve_m$, ... Remember that method labels contain the name of the interface that contains the method and consequently, removing a method label cannot remove an action concerning another method with the same name in another interface.

- $\oplus$ - "extension" : For $I \in \mathcal{I}_P$ and $I' \in \mathcal{I}_P$ disjoint, let $pNet = \langle\!\langle P, pNet_i^{i \in I}, SV_k^{k \in K} \rangle\!\rangle$ be a pNet and $pNet_i'^{i \in I'}$ be a pNet family (possibly empty).

Let $SV'_k{}^{k\in K'}$ be synchronisation vectors over $I \uplus I'$, i.e., $\forall k \in K'.SV'_k = \alpha'^{j\in J_k}_j \rightarrow \alpha'_k$ where $\alpha'_k \in \mathrm{Sort}(pNet)$, $J_k \in \mathcal{I}_P$, $J_k \subseteq I \uplus I'$, and $\forall j \in J_k \cap I.\, \alpha_j \in \mathrm{Sort}(pNet_j)$, and $\forall j \in J_k \cap I'.\, \alpha_j \in \mathrm{Sort}(pNet'_j)$. $pNet \oplus \langle\!\langle pNet'^{i\in I'}_i, SV'_k{}^{k\in K'} \rangle\!\rangle$ extends $pNet$ with the new sub-pNets $pNet'_i$. The original synchronisation vectors are kept (they do not synchronise the new sub-pNets); and the new synchronisation vectors: $SV'_k{}^{k\in K'}$ are added to the ones of $pNet$:

$$pNet \oplus \langle\!\langle pNet'^{i\in I'}_i, SV'_k{}^{k\in K'} \rangle\!\rangle = \langle\!\langle P, pNet^{i\in I}_i \uplus pNet'^{i\in I'}_i, SV^{k\in K}_k \uplus SV'_k{}^{k\in K'} \rangle\!\rangle$$

6.1 Non-functional components and interceptors

In this section we discuss how the membrane of a composite component including components can be analysed and translated into implementation code. We do not explain here the graphical specification of a non-functional part as it was presented in details in Section 3.2. In Chapter 4 we also formalised the well-formedness conditions for the componentised membrane of primitives. In the current version of our framework we verify only the membrane of composite components, the aspect dealing with the componentised membrane of the primitives is in the scope of the future work. It should be mentioned that a membrane can include sub-components of two kinds: component-controllers and interceptors. We currently do not construct the pNets for the latter.

6.1.1 From application design to pNets

As for the functional part, in order to generate input for the model-checker, we, first translate the specification of a non-functional part of an application into an intermediate format, i.e. into pNets. More presidency, we extend the pNets presented earlier with the sub-nets and synchronisation vectors encoding the non-functional elements. The advantage of using pNets is that they are convenient for encoding the behaviour of GCM/ProActive components, and at the same time they allow for the finite abstractions useful for model-checking.

Illustrative example

Figure 6.1 illustrates an example of a composite component `Composite` with a componentised membrane. Its content includes one sub-component `Prim1`, and its membrane includes two component-controllers (`Contr1, Contr2`) and one interceptor `Monitor`. The composite serves calls to two non-functional methods: `m1` and `m2`,

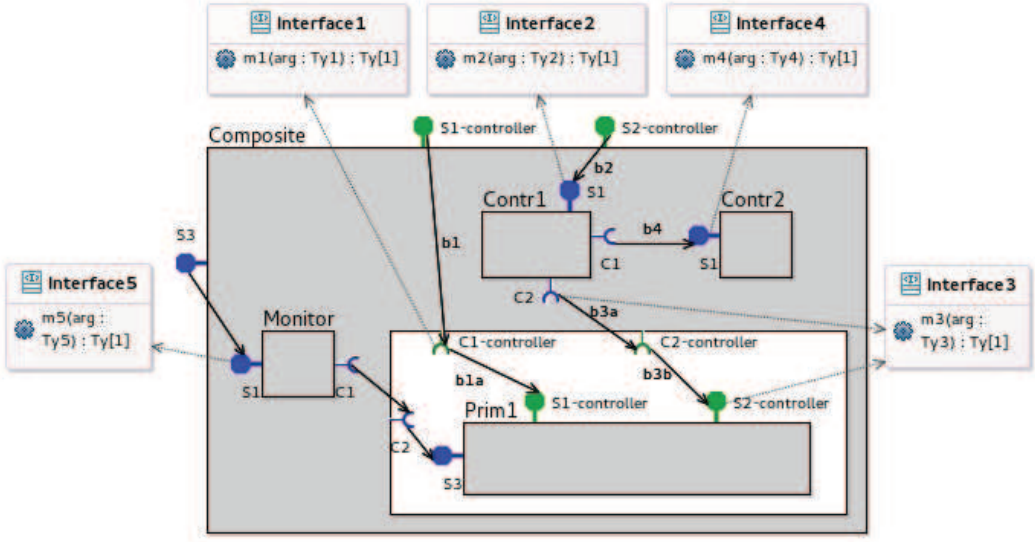


Figure 6.1 – Bindings in a membrane

and to one functional method `m5`. We include the functional method in the example in order compare the treatment of the functional and non-functional requests. The `m1` and `m5` method calls are processed by `Prim1` (in the content) while the invocations of `m2` are served by `Contr1` (in the membrane). The component-controllers communicate with each other: `Contr1` can invoke method `m4` on `Contr2`. Finally, the composite includes a non-functional internal interface `C2-controller` which is not visible from outside of the composite. It is used by `Contr1` which can invoke the method `m3` on it, and the invocation will be forwarded to `Prim1`.

In order to include the specification of the non-functional elements in the model of a composite component behaviour, we extend the set of sub-nets encoding the behaviour of a composite with several other pNets and synchronisation vectors. In fact, as we will demonstrate in this section, the pNets modelling the non-functional elements of a composite are very similar to the ones for the functional part. In particular, we generate proxy, proxy manager, and delegate pLTSs processing the non-functional requests, and the pNets for the components in the membrane.

Figure 6.2 illustrates the pNet of the composite depicted in Figure 6.1; for the sake of simplicity, we omit the parameters of the actions in the figure. The pNet includes three sub-pNets for the sub-components: *Contr1*, *Contr2*, and *Prim1*. The `Monitor` component is not included in the structure because we have not formalised the pNets for the interceptors in the current version of the framework. Instead, we assume that the functional call goes directly to the plugged component. This assumption is coherent with the role of interceptors. One can notice that from the figure of the pNet it is not possible to understand which sub-net models a functional

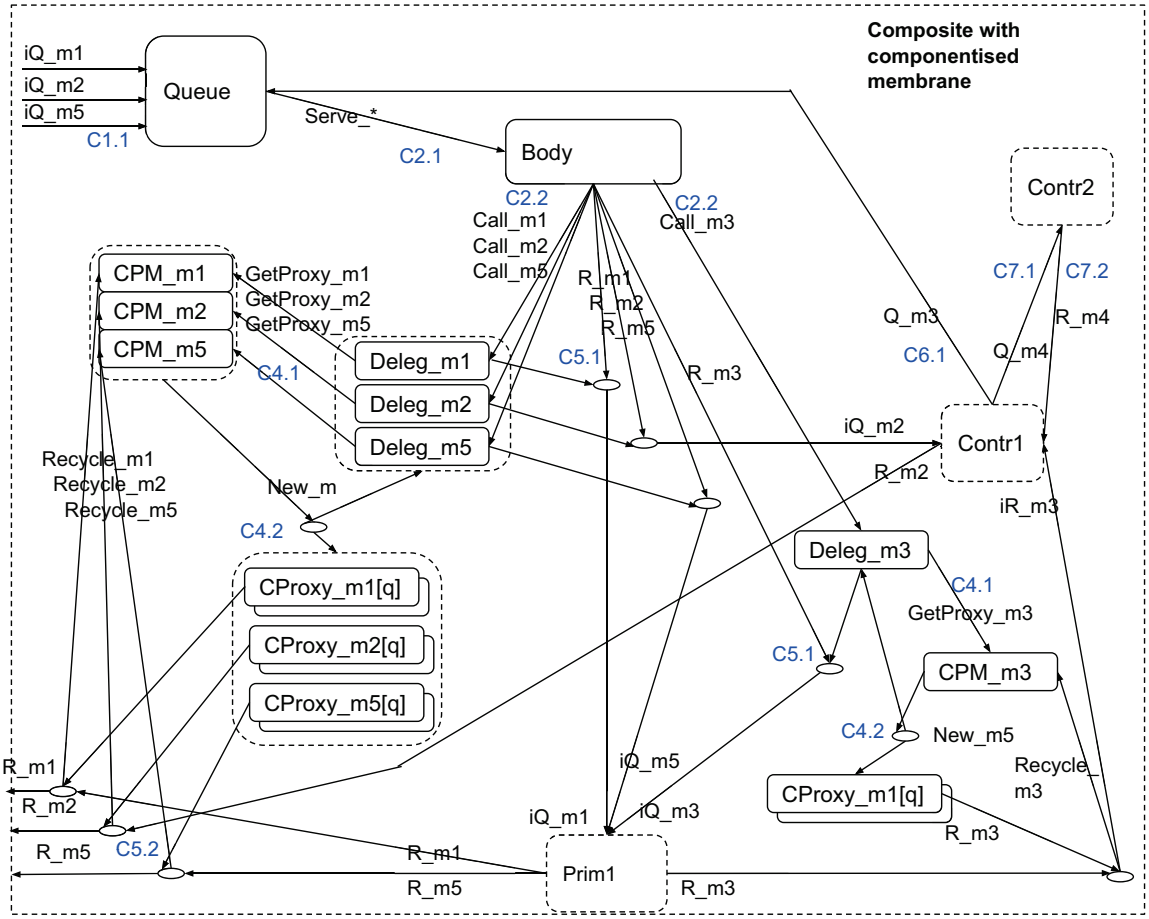


Figure 6.2 – pNet of a component with a componentised membrane

or a non-functional sub-component. Indeed, at the level of pNets we do not separate elements of the two concerns as their separation has already been carefully checked during the static validation. In addition, the pNet of a composite includes the pLTSs that process the non-functional requests (*CPM_m1*, *CProxy_m1*, *Deleg_m1*, etc) and the synchronisation vectors for the treatment of requests. The treatment of the non-functional method calls is very similar to the treatment of the functional ones. Hence, the synchronisation vectors modelling the communications among the non-functional elements are based on the rules given in Tables 5.3 and 5.4.

The queue of the composite can receive external requests to the methods exposed on its server interfaces (*iQ_m1*, *iQ_m2*, and *iQ_m5*) and the invocations to `m3` (*iQ_m3*) can be received from *Contr1*. Then, those requests are treated by the proxy, proxy manager, and delegate pLTSs similar to the functional calls discussed in Section 5.1. The results of the calls to the external methods are returned to outside of the composite: *R_m1*, *R_m2*, *R_m5*, and the result of the invocation of `m3` is returned to the caller, i.e. to `Contr1`. The communication between the two component-controllers

in the membrane is modelled in exactly the same way as if it occurred between two functional sub-components.

Below we provide a detailed description of the sub-nets and synchronisation vectors generated for the non-functional part of a composite.

Sub-nets

First, we generate the proxy, proxy manager, and delegate pLTSs for each method of the external non-functional client and server interfaces of a composite in the same way as for the functional methods.

The construction of functional and non-functional internal interfaces is different. For each internal functional interface of a composite, there exists a symmetric external one, hence, while building the pLTSs of the functional methods, we discussed only external interfaces. However, an internal non-functional interface may not have the corresponding external one, when it is connected to a component inside a membrane. Such interfaces enable the communications between sub-components in the membrane and in the content. Figure 6.1 illustrates an example of such an interface: the `C2-controller` internal interface receives method calls from the `Contr1` component in the membrane and forwards them to `Prim1` in the content. The example illustrates a client interface, but an internal non-functional server interface could be also modelled in a similar way. In this case, a method invocation would be triggered by a component in the content and served by a component in the membrane. According to the semantics of GCM/ProActive components, the invocations to such kind of interfaces also go through the queue and the body of the composite, hence, the proxy, proxy manager, and delegate pLTSs should be generated for the methods of such interfaces (*CProxy_m2*, *CPM_m2*, and *Deleg_m2* in the example).

Second, we include the pNets encoding the behaviour of the non-functional sub-components in the pNet of a composite (*Contr1* and *Contr2* in the example). In fact, the pNets encoding the behaviour of the components inside the membrane have exactly the same generation procedure as the ones of the functional components discussed in Section 5.1.

Synchronisation vectors

As it was discussed in Section 5.1.2 the synchronisation vectors of a composite are organised into three sets: server-side (SV_S), client-side (SV_C), and binding-related (SV_B) synchronisation vectors. We start by explaining how the first two sets should be extended with the synchronisation vectors for the non-functional interfaces. Then, we discuss the binding-related communications.

Requests to the non-functional server interfaces of a composite are treated in exactly the same way as the functional calls, hence the proxy, proxy manager, and delegate pLTSs generated for the methods of the non-functional interfaces should be synchronised following the rules from Table 5.3. A non-functional server request is, first dropped in the queue (Rule [C1]), then taken by the body (Rule [C2.1]) and forwarded to the delegate method (Rule [C2.2]), the new proxy is allocated by the proxy manager (Rule [C4]), and then the proxy can wait for the reply. A non-functional client request is treated in a similar manner except that it cannot be en-queued by an external component (i.e. Rule [C1] is not applied), and it is forwarded to outside of the composite (Rule [C3]).

The only exception is the internal non-functional interfaces which are connected to the components in the membrane but not to the external interfaces (e.g. `C2-controller` in Figure 6.1). Their methods are not exposed to outside of the composite, and hence the synchronisation vectors [C1], [C3] which express the communications with the external environment are not generated for them. Still, all the interactions between the queue, the body, proxies, proxy managers, and delegate methods are involved in the treatment of the requests, and, hence the synchronisation vectors based on Rules [C2], [C4] are constructed.

The communications which occur on the bindings in the membrane should be also encoded with synchronisation vectors which extend the (SV_B) set. We distinguish four cases of possible types of interactions, and each of them is illustrated in Figure 6.1 (see `b1`, `b2`, combination of `b3a` and `b3b`, and `b4`).

First, a binding can connect directly an external and an internal non-functional interfaces of a composite (see `b1`). In this case, the treatment of request is exactly the same as for the functional calls, because the requests are served (or called) by the sub-components in the content. Such communications rely on several rules from Table 5.4. A server method call should be forwarded by the delegate structure to the plugged sub-component inside the content [C5.1], and the invocation result should be returned to the caller [C5.2]. A client method invocation should be forwarded from the caller located in the content to the queue [C6.1], and its result should be returned to the caller [C6.2].

Second, a component in the membrane can be connected to a non-functional external interface of the composite (e.g. `b2`). In fact, this is very similar to the previous case, and it involves the same synchronisation vectors (i.e. the vectors expressed by the Rules [C6], [C7)] but the synchronised sub-components are located in the membrane, not in the content.

Third, component-controllers can use non-functional internal interfaces of the

composite in order to communicate with the sub-components inside the content (`b3a`). Invocations going through such interfaces are also processed by the queue and the body of the composite. Hence, all components plugged to an internal non-functional interface should synchronise with the pLTSs of the composite but not with each other. In the given example, `Contr1` should be synchronised with the queue of the composite on a method invocation and with the corresponding proxy on the result reception (Rule [C6]). `Prim1` should be synchronised with the corresponding delegate pLTS on a method invocation, and when it sends the computed result, it synchronises with the proxy pLTS ([C5]).

Finally, sub-components inside a membrane can communicate with each other (e.g. `b4`). They are synchronised on a method call and on a reply following the Rules [C7], i.e. in the same way as the communicating sub-components in a content.

6.1.2 Implementing pNet generation and integration with CADP

As it was discussed in the previous section, all the constructs encoding the non-functional part of a component are based on the ones for the functional elements. Hence, in order to generate a pNet of a composite with a componentised membrane from VerCors, we did not have to implement any additional builder. One additional analysis step that we have to do while pre-processing the input models is extracting a set of internal non-functional interfaces of composite components which are connected to the sub-components in a membrane. Then, in addition to the pNet generation process discussed in Section 5.1.3, we invoke the proxy, proxy manager, and delegate builders for the methods of the non-functional interfaces, and the pNet builders for the sub-components in a membrane. The synchronisation vector generator constructs additionally the synchronisation vectors encoding the communications related to the non-functional aspect as it was discussed in the previous section. While producing .fiacre and .exp files, VerCors does not distinguish the constructs encoding functional and non-functional features. The internal communications in the sub-components of a membrane are by default hidden in the behaviour graph of the enclosing component.

6.1.3 Code generation

VerCors includes the specification of a component membrane in the generated ADL file and Java code. This requires an additional pre-processing step where the input architecture is analysed in order to extract interceptors and to map functional interfaces of a composite to a chain of interceptors.

```
1 <definition name="Composite">
2  <interface name="S3" role="server" signature="interfaces.FItf" interceptors="Monitor.S1"/>
3  <component name="Prim1">
4   ...
5  </component>
6  <binding client="this.S3" server="Prim1.S3"/>
7  <controller desc="composite">
8   <interface name="S1−controller" role="server" signature="p.interfaces.NFItf"/>
9   <interface name="S2−controller" role="server" signature="p.interfaces.NFItf"/>
10  <interface name="C1−controller" role="internal−client" signature="p.interfaces.NFItf"/>
11  <interface name="C2−controller" role="internal−client" signature="p.interfaces.NFItf"/>
12
13  <component name="Contr1"> ... </component>
14  <component name="Contr2"> ... </component>
15  <component name="Monitor"> ... </component>
16  <binding client="this.S1−controller" server="this.C1−controller"/>
17  <binding client="Contr1.C2" server="this.C2−controller"/>
18  <binding client="Contr1.C1" server="Contr2.S1"/>
19  <binding client="this.S2−controller" server="Contr1.S1"/>
20  <binding client="this.C1−controller" server="Prim1.S1−controller"/>
21  <binding client="this.C2−controller" server="Prim1.S2−controller"/>
22  </controller>
23 </definition>
```

Listing 6.1 – Generated ADL file of a composite with a componentised membrane

Listing 6.1 illustrates a simplified version of an ADL file generated for `Composite` from Figure 6.1. The specification of its membrane is given in lines 7-22. It includes the non-functional internal and external interfaces (lines 8-11), both component-controllers (lines 13-14) and an interceptor (line 15), bindings in the membrane (lines 16-19), and bindings connecting the non-functional internal interfaces and the components in the content (lines 20-21). One can notice that all the bindings in the membrane are included in the ADL specification except from the ones connecting an interceptor to the interfaces which calls it intercepts. Instead, the interceptor is referenced by the corresponding interface in line 2.

The Java classes of component-controllers are constructed in the same way as the classes implementing functional components. For the interceptors we generate two additional methods which are invoked when a request is intercepted and after it has been served.

6.2 Component attributes and attribute controllers

A GCM primitive component can have attributes which are used by its methods. In fact, we should distinguish between the two aspects: the statefull components (i.e. the attributes of a component which can be accessed and modified by its methods)

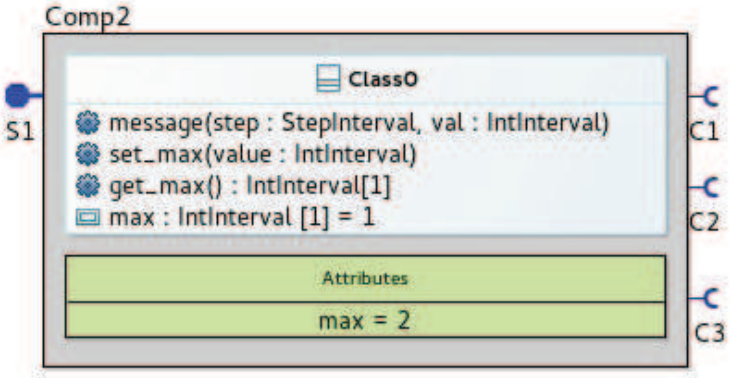


Figure 6.3 – Graphical specification of a component attribute

and the get/set access to the attributes of a component from outside. For the first aspect, we discuss in this section how the attributes of a primitive can be designed graphically, accessed from its methods, translated into pNets, and generated in the implementation code. However, we have not yet decided which graphical notations should be used for the second aspect. Hence, if the user would like to make an attribute accessible from outside of a component, he must explicitly define a server method which implements the necessary access. In the generated code we make the attributes accessible from outside of a component because this is needed for the GCM/ProActive factory during the deployment.

6.2.1 Graphical specification

As it was discussed in Section 3.2 the attributes of a primitive component should be declared in the UML class attached to the component. Figure 6.3 illustrates a simplified version of a leader election algorithm participant from our use-case discussed in Chapter 3. The component has only one attribute - `max`. In addition, for each attribute of a class, the user should declare so-called set- and get-methods which are used to modify and access the value (`set_max()` and `get_max` in our example). The user should not define any state machine for the behaviour of these methods. The default initial value of an attribute can be provided in the class specification (it is equal to 1 in our case). If the user wants to define a specific value for a particular component, then it should be given in a dedicated green area (the initial value is equal to 2 in our example).

6.2.2 From application design to pNets

Global structure. For each pair of a get- and a set-methods defined for an attribute, we construct a pLTS called *attribute controller* which stores the value of the attribute and provides actions to modify and to access it. The pLTSs should be included in the pNet of the primitive together with the synchronisation vectors modelling access to the attributes.

The behaviour of a primitive component with attribute controllers is formalised below. We extend the definition of a primitive with a set *Attributes* which includes its attributes.

$$[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K}, Attributes >]\!]^{AC} = \\ [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K}, Attributes >]\!] \oplus \langle\!\langle \mathcal{AC}, SV_{AC} \rangle\!\rangle$$

Where $\mathcal{AC}$ is computed by the following rule:

$$\frac{a_i^{i\in AI} = Attributes(Comp)}{\mathcal{AC} = \overleftrightarrow{\langle\!\langle [\![a_i]\!]_{ac}^{n\in AI} \rangle\!\rangle}}$$

Here the function $[\![\ ]\!]_{ac}$ returns the pLTS of an attribute controller.

Attribute controller pLTS. The behaviour of an attribute controller is encoded as a pLTS as illustrated in Figure 6.4. This pLTS stores a variable `max` which represents the modelled attribute, and provides actions to access its value (*Call_Get_max()*, *R_Get_max(max)*) and to modify it (*Set_max(?max)*).

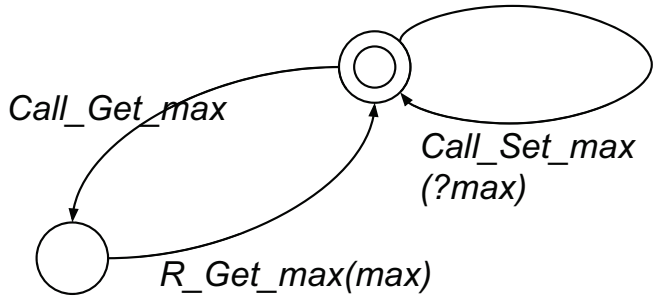


Figure 6.4 – An attribute controller pLTS

Table 6.1 – Attribute controller synchronisation vectors for primitive components. The synchronised sub-pNets are: $\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies, AttributeControllers \rangle\!\rangle$

$$\frac{m_l^{l\in L} = \text{MethLabels}(\overline{SItf}) \qquad a_i^{i\in AI} = Attributes(Comp)}{\begin{array}{ll} \{\langle -,-,l \mapsto Call_Get_ac_i,-,-,i \mapsto Call_Get_a_i\rangle \rightarrow \underline{Call_Get_a_i}, & [1] \\ \ \langle -,-,l \mapsto R_Call_Get_a_i(val),-,-,i \mapsto R_Call_Get_a_i(val)\rangle \rightarrow \underline{R_Call_Get_a_i(val)}), & [2] \\ \ \langle -,-,l \mapsto Call_Set_a_i(val),-,-,i \mapsto Call_Set_a_i(val)\rangle \rightarrow \underline{Call_Set_a_i(val)})\} & [3] \\ \qquad\qquad \subseteq SV_{AC}(m_l^{l\in L}) & \end{array}}\ \text{AC}$$

Synchronisation vectors. The pNet of a primitive component should include one attribute controller pLTS for each attribute. The pLTSs can be accessed from the server and local methods of the component; in order to enable such access, the pNet of the primitive is extended with the synchronisation vectors given in Table 6.1. The vectors rely on an additional construct Attributes(*Comp*) which provides a set of attribute names of the generated primitive where each attribute has a unique name. Rules [AC.1] and [AC.2] synchronise the server methods and the attribute controllers when a server method modifies the value of an attribute. Rule [AC.3] allows server methods to access the attribute values. As in the previous definitions, the rules include only pLTSs of the server methods, but the local methods are synchronised with the attribute controllers in the same way.

In order to make an attribute accessible from outside of a primitive, we would need to extend the queue and the body with the actions for the treatment of the requests to the attribute controller, and to synchronise them as for any other server method.

6.2.3 Implementing pNet generation and integration with CADP

VerCors generates one attribute controller pLTS for each attribute of a primitive component and synchronises it with the pLTSs encoding the server and local methods of the component. The construction requires several steps in addition to the pNet generation process discussed in Section 5.1.3. First, at the pre-processing phase, VerCors extracts a set of attributes for each generated primitive; and for each state machine encoding the behaviour of a method, the pre-processor extracts the set of attributes accessed by the state machine. Second, while constructing a pNet encoding the behaviour of a primitive, for each attribute belonging to the component, the platform additionally invokes an attribute controller builder which creates the corresponding

pLTS. Finally, VerCors generates the synchronisation vectors following the rules from Table 6.1. Thanks to the analysis done at the pre-processing phase, the synchronisation between methods and attribute controllers which do not communicate is not generated.

6.2.4 Code generation

Listing 6.2 provides an example of the generated Java class for `Class0` from Figure 6.3. In order to include attributes in the generated code of a GCM/ProActive primitive component, we have to produce several additional constructs.

```
1 public class Class0 implements Class0AC ... {
2   public int max = 1;
3
4   public  void set_max(int value) {
5     this.max = value;
6   }
7   public  int get_max() {
8     return this.max;
9   }
10 }
```

Listing 6.2 – A Java class implementing the behaviour of a primitive component with an attribute

First, we include a field corresponding to each attribute in the Java class (line 2). If its default value is specified in the UML class, it is also translated into Java code.

GCM/ProActive provides a mechanism to set a value of an attribute which is specific to a particular component (the value that is graphically defined in the green area included in the specification of a primitive). For this, we need to include the value in the ADL description of the component as shown in Listing 6.3. The GCM/ProActive factory reads the value of each attribute in the ADL file and assigns it during the component construction.

```
1   <component name="Comp2">
2    ...
3      <attributes signature="example.interfaces.Class0AC">
4         <attribute name="_max" value="2"/>
5         ... other attributes
6      </attributes>
7      <controller desc="primitive"/>
8   </component>
```

Listing 6.3 – An ADL specification of a component attribute

Line 3 in in this ADL description refers to the Java interface `Class0AC` which methods will be invoked by the factory in order to set the values of the component attributes. The interface includes the signatures of the set- and get-methods for the

attributes included in the ADL description of the component. The interface should be implemented by the class modelling the behaviour of the component (`Class0` in our example). Such kind of interfaces are also generated by VerCors.

Finally, VerCors automatically produces the code of the set- and get-methods for each attribute (see lines 4-9 in Listing 6.2).

6.3 Reconfigurable multicast interfaces

One of the core features of the GCM component model is collective communications which occur through the multicast and gathercast interfaces.

A multicast interface (or a *multicast* for short) is a client interface which can send several requests to different targets simultaneously, and then gather the results. Moreover, the group of target interfaces can be reconfigured at run-time, i.e. the bindings going from a multicast interface can be dynamically added and removed. According to the specification of GCM, the policy of a multicast (i.e. how one request is transformed into several requests, and how multiple results are assembled in a single result) is customizable. GCM/ProActive defines several default policies for the multicast interfaces. The unicast policy is used to send one argument to one destination interface. The broadcast policy copies the list of the arguments and sends the request to all the plugged target interfaces. The scatter policy splits the list of arguments and sends a part of it to each target interface. The round robin policy distributes each element of the list parameter in a random manner to the connected server interface.

A gathercast interface can receive several method invocations at the same time and transform them into one request. The gathercast interfaces were not studied in this work mainly due to the lack of time. For their modelling and verification we plan to rely on the same techniques as for the multicasts.

In this section we show how a multicast with a broadcast policy can be graphically modelled, we formalise the generation of pNets encoding one-to-N communications and explain their construction in VerCors. Finally, we briefly discuss the executable code generation for the multicast interfaces. This section describes the structure of a reconfigurable multicast interface and the way the reconfiguration instructions are processed, and in Section 6.4 we will explain how the reconfiguration of a multicast can be triggered from a non-functional component. The idea of how the pNets can be used to encode multicast interfaces was presented in [63]. However, this thesis is the first work presenting the chain from the graphical design to the generation of the input for the model-checker and the executable code of the applications with

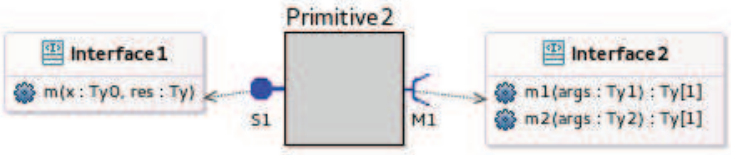


Figure 6.5 – A primitive component with a multicast Interface

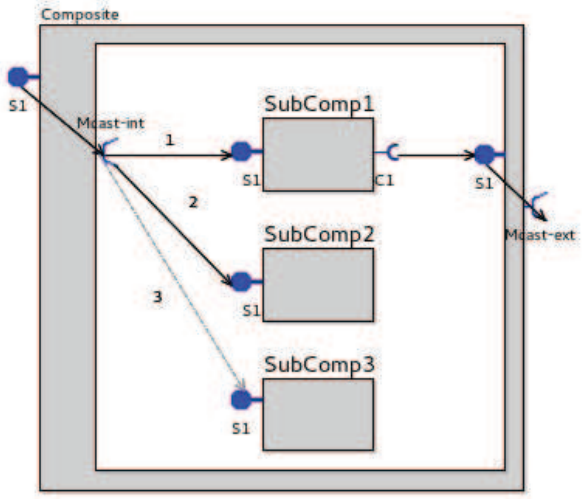


Figure 6.6 – A composite component with multicast internal and external interfaces

multicast interfaces.

6.3.1 Graphical specification

As it was discussed in Section 3.2, the representation of a GCM interface modelled in VerCors changes depending on the values of various properties. In particular, the user can set the cardinality of an interface either to `singleton` or to `collective`. A collective client interface is a multicast interface. It can belong either to a component (an external interface) or to a content of a composite (an internal interface).

An example of a multicast interface attached to a primitive component is illustrated in Figure 6.5 (see `M1`). Figure 6.6 shows an example of a composite with an external multicast `Mcast-ext` and an internal multicast `MCast-int`. The user can assign indices to the bindings going from a multicast interface in order to refer to them while modelling system reconfiguration (this will be explained in Section 6.4). In order to analyse application reconfiguration, the user should also model bindings which do not exist when the modelled application is launched but can appear during system execution, e.g. the dashed binding `3` going from `Mcast-int` to `S1` of `SubComp3`.

6.3.2 From application design to pNets

Components with multicast interfaces graphically specified in VerCors can be translated into pNets for analysis. This section provides a model for multicast interfaces where the capabilities of the pNets' synchronisation vectors are fully used and allow one component to broadcast a request to several others, or one component to provide a reply that would reach the right index in a group of futures. For this, we define richer future proxies that can handle a list of results, and provide a result as soon as enough results are available. This section defines $[\![\ ]\!]^{MC}$, the behavioural semantics for components equipped with reconfigurable multicast interfaces.

Principle of the approach

We first explain the principle of the approach focusing on the multicast interfaces of the primitive components. When a client interface is of type multicast, it may have a variable number of outgoing bindings (it is bound to the server interfaces of several components). Invocations emitted by a multicast interface are broadcasted to all the server interfaces bound to it. In GCM, depending on the interface policy, arguments of requests emitted by the interface can be dispatched in a parameterisable manner. Here, we suppose that the argument is broadcasted to all the destination components. Then results will come back in an asynchronous way from the elements of the group. The encoding of multicast interfaces relies on the two pLTSs of Figures 6.8 and 6.7 for dealing with the specific future proxies:

- one Group Proxy Manager for each method of each multicast interface: Figure 6.7 shows the process *GrPM_m* that defines the value of $[\![m]\!]_{proxyManager}$ in case the method m belongs to a multicast interface. Compared to a classical proxy manager, each group proxy manager is also in charge of managing changes in the group content (bindings and unbindings). Each binding/unbinding operation targeted at a client multicast interface is thus broadcasted to all the group proxy managers of all the methods of this interface.

- for each method in the multicast interface, an indexed family of Group Proxies: Figure 6.8 defines a process *GrProxyPrim_m* that overrides the value of $[\![m]\!]_{proxy}$ in case m belongs to a multicast interface. Upon each request invocation, a corresponding proxy is activated and initialised. Each incoming reply to this request will update a result vector (additional conditions check that a given component does not reply twice). Results can be accessed by the server and local methods that can query either the result vector totally filled (*GetValue_m*), a partially filled vector (*WaitNth_m*), or an element from the

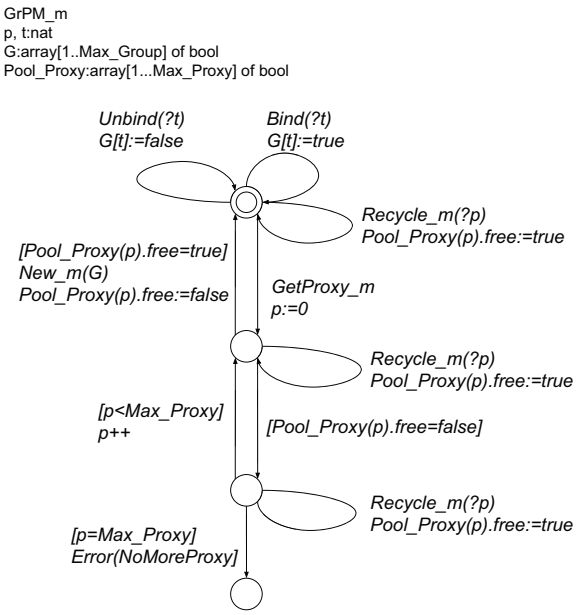


Figure 6.7 – A Group Manager

vector (*GettNth_m*) at a given index. As in the case of the standard future proxies, a group proxy can be recycled, whenever it can be decided that it will never be used again.

Note that in the Group Manager pLTS, the group variable G is modified by *Bind* and *Unbind* actions. The values of the elements in G reflect to the status of the bindings at the corresponding indices in the target group. If $G[t]$ is equal to `true`, then the binding at index t is bound, and the corresponding target interface will be requested during the method invocation. Each time the Group Manager activates a new Group proxy (*New_m* action), it sends a copy of the value of G, so that even if reconfigurations occur, each proxy keeps its own copy of Group, on which the invocation has been performed.

The pNet of a primitive component with a multicast interface is shown in Figure 6.9, it corresponds to the primitive component that was shown in Figure 6.5. The figure shows the parts of the pNet that are specific to the handling of multicast interfaces. It illustrates that binding operations received in the queue are broadcasted to each group proxy manager of the targeted multicast interface. Then proxy creation (*New_m*) is synchronised similarly to the case of usual interfaces except that the current status of the multicast interface (G) is transmitted to the created proxy. The main specificity of the synchronisation vectors for multicast interfaces is the fact that a request emission is also synchronised with the corresponding proxy that provides the correct value of G targeted by the invocation (*MC*(G)), and the outgoing request

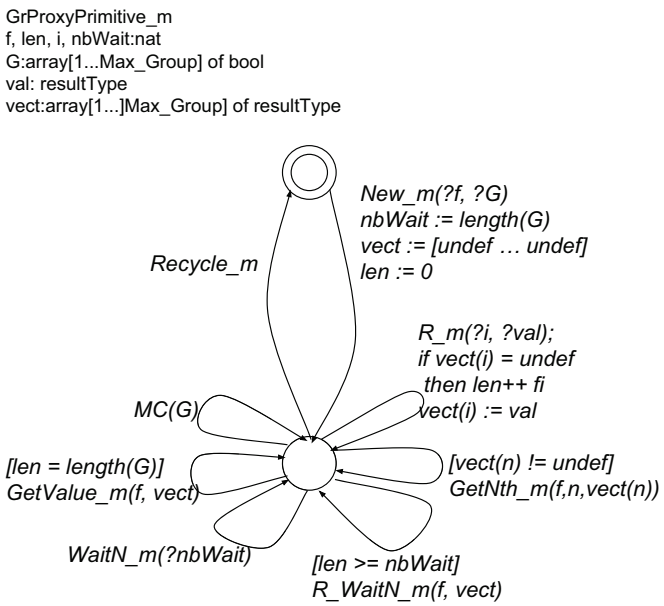


Figure 6.8 – A Group Proxy for a method of a primitive component

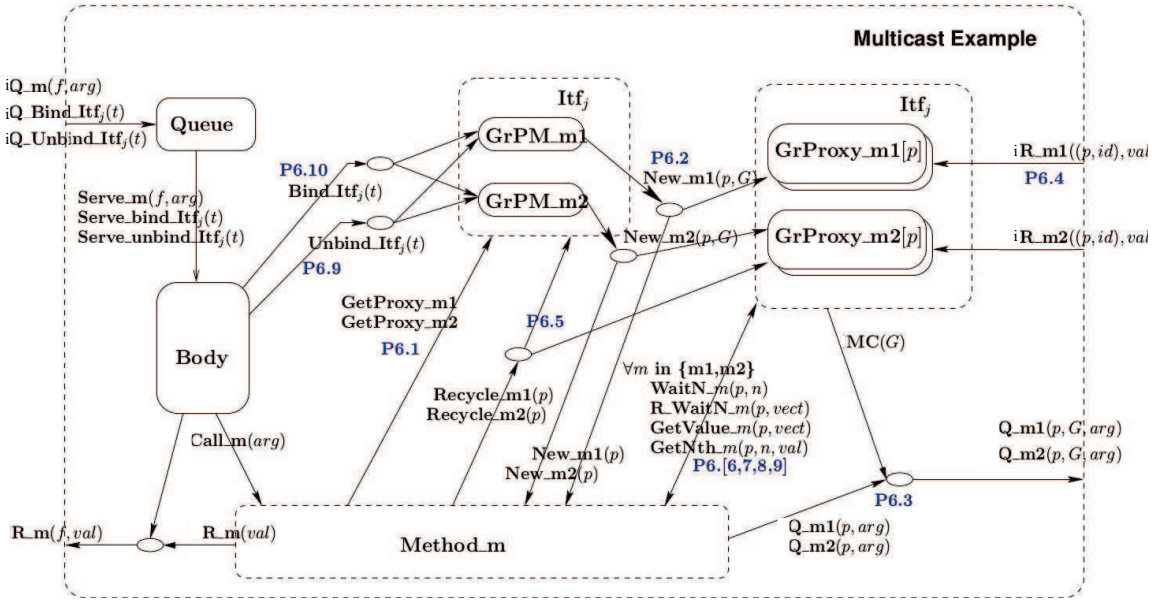


Figure 6.9 – pNet model for Figure 6.5

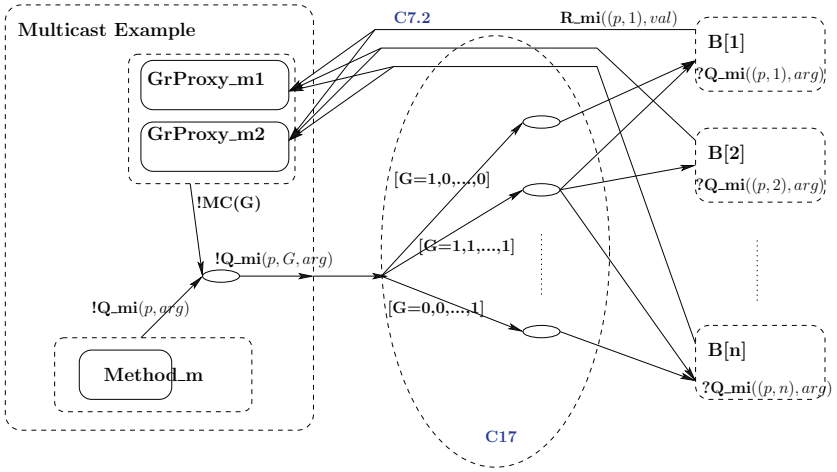


Figure 6.10 – Dynamic Connector for a Multicast Interface

also sends G as argument. This argument containing the invoked group will be used at the higher level in the hierarchy by the encompassing composite component. We will also use variable g for the target interfaces which status is included in the group G: g ranges over qualified names. Finally, actions for accessing the group proxy (*WaitN_m*, *GetValue_m*, *GetNth_m*) can be invoked by the server methods.

Figure 6.10 illustrates the principle of the synchronisation occurring at the higher hierarchical level, in the composite that contains the component with the multicast interface. Remember that we use the maximal value of the group size (the number of all bindings going from a multicast including normal and dashed bindings). This size is used to create an array that the connector uses to deliver requests to the adequate server interfaces depending on the group value at the time of invocation.

In the case of multicast interfaces, the proxy has to know the group addressed by the invocation.eTo simplify notations, we rather create a bigger array of results but only the ones corresponding to bound elements of G will be used. Knowing G at invocation time is also useful to implement a *GetValue* primitive returning a result if all replies came back. It is important to note that the group G known by the proxy is the one that was active *at invocation time* regardless of bind/unbind operations that occurred after the invocation.

As for reconfigurable interfaces, the queue and the body of the components that contain multicast interfaces must be able to accept and handle bind and unbind requests. In order to handle reconfiguration, the queue and the body are extended so that they can handle the *Bind_Itf* and *Unbind_Itf* requests. Each action dedicated to the reconfiguration has a parameter t which represents the index of the target interface which is bound or undound in the group G. When a request to bind or unbind an interface arrives, it is first dropped in the queue. Then, the body takes

it following the FIFO policy. The request is then forwarded to the proxy manager. The proxy managers receives *Bind*/*Unbind* actions and stores the current value of the group (in the variable G), whereas the group proxies are responsible for emitting the current value of the group at invocation.

Finally, the structure of future identifiers has to be enriched for dealing with multicast interfaces: in the following a future identifier, like f, p, can be either a classical future identifier, or a couple made of a classical future identifier and an index, the index being used to identify uniquely the destination of the invocation among G, the group of invoked components. This way we will be able to distinguish replies originating from two different members of the group, and concerning the same invocation. Consequently when constructing the synchronisation vectors (more precisely, in Table 6.4), we rely on a function $\text{Index}_G(g)$ that returns an integer for each $g \in G$. This index will be attached to the future identifier to uniquely identify the destination of the multicast invocation. The function should be injective so that two targets cannot receive the same index; we let *id* range over those indices.

Multicast Interfaces for Primitive Components

The pNet of a primitive component with multicast interfaces is similar to the pNet without multicast interfaces, except that the proxy managers and the proxies for methods of multicast interfaces are replaced by the pLTSs defined in Figures 6.7 and 6.8: $[\![\ ]\!]_{proxyManager}$ is overloaded for multicast interfaces, it returns the classical proxy manager for a singleton interface, and the new one of Figure 6.7 for a multicast interface, and similarly for $[\![\ ]\!]_{proxy}$. In addition, to these changes in the behavioural semantics of future proxies and their managers, the synchronisation vectors are modified as shown in the next rule: synchronisation vectors containing the name of a multicast interface are replaced by a new one. Note that, as method labels contain the name of the interface, $\oslash(CItf_j^{j\in J'})$ removes all the synchronisation vectors containing the name of a method m_i of an interface $CItf_j$ among its action labels.

$$\frac{m_l^{l\in L} = \text{MethLabels}(SItf_i^{i\in I}) \qquad CItf_j^{j\in J'} = \{CItf_j | j \in J \wedge CItf_j \text{ is multicast}\}}{[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!]^{MC} = [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!] \ \oslash(CItf_j^{j\in J'}) \oplus \langle\!\langle SV_C^{MC}(CItf_j^{j\in J'}, L)\rangle\!\rangle}$$

The synchronisation vectors SV_C^{MC} for the multicast interfaces of a primitive component are defined in Table 6.2, they correspond to Figure 6.9. We introduce a new variable *vect*, similarly to *arg* and *val*, *vect* ranges over arrays of values. In the

Table 6.2 – Synchronisation vectors for multicast client interfaces in primitive components. The synchronised sub-pNets occur in the following order: $\langle\!\langle Queue, Body, ServerMethods, ProxyManagers, Proxies \rangle\!\rangle$

$$\frac{j \in J \qquad l \in L \qquad m_i^{i \in I} = \text{MethLabels}(CItf_j) \qquad i \in I \qquad p \in \mathbb{N} \qquad CItf_j \text{ is multicast}}{\begin{array}{ll}
\{\langle -, -, l \mapsto GetProxy_m_i, j \mapsto i \mapsto GetProxy_m_i, - \rangle \rightarrow GetProxy_m_i, & [1] \\
\langle -, -, l \mapsto New_m_i(p), j \mapsto i \mapsto New_m_i(p, G), j \mapsto i \mapsto p \mapsto New_m_i(G) \rangle \rightarrow & \\
\qquad New_m_i(p, G), & [2] \\
\langle -, -, l \mapsto Q_m_i(p, arg), -, j \mapsto i \mapsto p \mapsto MC(G) \rangle \rightarrow Q_m_i(p, G, arg), & [3] \\
\langle -, -, -, -, j \mapsto i \mapsto p \mapsto R_m_i(id, val) \rangle \rightarrow iR_m_i((p, id), val), & [4] \\
\langle -, -, l \mapsto Recycle_m_i(p), j \mapsto i \mapsto Recycle_m_i(p), j \mapsto i \mapsto p \mapsto Recycle_m_i \rangle \rightarrow & \\
\qquad Recycle_m_i(p), & [5] \\
\langle -, -, l \mapsto WaitN_m_i(p, nb), -, j \mapsto i \mapsto p \mapsto WaitN_m_i(nb) \rangle \rightarrow & \\
\qquad WaitN_m_i(p, nb), & [6] \\
\langle -, -, l \mapsto R_WaitN_m_i(vect), -, j \mapsto i \mapsto p \mapsto R_WaitN_m_i(vect) \rangle \rightarrow & \\
\qquad R_WaitN_m_i(p, vect), & [7] \\
\langle -, -, l \mapsto GetNth_m_i(p, nb, val), -, j \mapsto i \mapsto p \mapsto GetNth_m_i(nb, val) \rangle \rightarrow & \\
\qquad GetNth_m_i(p, nb, val), & [8] \\
\langle -, -, l \mapsto GetValue_m_i(p, vect), -, j \mapsto i \mapsto p \mapsto GetValue_m_i(vect) \rangle \rightarrow & \\
\qquad GetValue_m_i(p, vect), & [9] \\
\langle -, Bind_CItf_j(t), -, j \mapsto (i' \in I \mapsto Bind(t)), - \rangle \rightarrow Bind_CItf_j(t), & [10] \\
\langle -, Unbind_CItf_j(t), -, j \mapsto (i' \in I \mapsto Unbind(t)), - \rangle \rightarrow Unbind_CItf_j(t), & [11] \\
\langle Serve_Bind_CItf_j(t), Serve_Bind_CItf_j(t), -, -, -, - \rangle \rightarrow Serve_Bind_CItf_j(t), & [12] \\
\langle Serve_Unbind_CItf_j(t), Serve_Unbind_CItf_j(t), -, -, -, - \rangle \rightarrow & \\
\qquad Serve_Unbind_CItf_j(t)\} & [13] \\
\qquad \subseteq SV_C^{MC}(CItf_j^{j \in J}, L) &
\end{array}} \text{ P6}$$

rules, we write $(i' \in I \mapsto (Bind(t)))$ to represent the family $Bind(t)^{i' \in I}$; this represents a synchronisation vector that broadcasts the action $Bind(t)$ to all the elements inside I, here all the proxy managers of the reconfigured interface.

Cases [P6.1] and [P6.2] are used to create a proxy: compared to Section 5.1.1, the content of the group targeted by the invocation (G) is transmitted to the proxy. The element [P6.3] expresses request emission, with the proxy emitting the adequate value of G. Compared to singleton interfaces, reply reception uses the fact that an index id is attached the future, and transmits this index to the future proxy for multicast interface. Recycle [P6.5] is similar to the non-multicast case. Elements [P6.6] and [P6.7] are used for waiting for a given number, nb, of responses: first, nb is sent to the proxy ($WaitN_m_i$ action), and then a reply is sent back to the server method by $R_WaitN_m_i$. The two next rules [P6.8] and [P6.9] do not require to work in a request/reply manner, the proxy can directly emit $GetNth_m_i$ and $GetValue_m_i$ actions when they are enabled, i.e. when the necessary replies have arrived. The next

two elements ([P6.10] and [P6.11]) deal with the reconfiguration of the multicast interface: they transmit bind and unbind orders from the body to all the group proxy managers corresponding to the reconfigured interface. The two last items [P6.12] and [P6.13] synchronise the request queue with the body in order to serve bind and unbind requests. One can notice that the *iQ_Bind* and *iQ_Unbind* actions of a queue are not included in any synchronisation vectors. Indeed, the en-queueing of bind and unbind requests is not exposed to outside of a component. Instead, it is done by the component-controllers inside the membrane. Hence, the component-controllers should synchronise with the queue in order to modify interfaces as it will be discussed in Section 6.4.

Similarly to the client methods of singleton interfaces, the methods of multicasts can be invoked not only by server methods but also by local ones. In this case, a pLTS of a local method should synchronise with the corresponding proxy and proxy manager.

Multicast Interfaces for Composite Components

Concerning composite components, two aspects have to be added. First, composite components can also have multicast client interfaces which can be either external or internal. Second, composite components have to encode the synchronisation between multicast interfaces and the plugged components inside the composite.

Similarly to primitive components, the proxy managers and the future proxies are different for multicast interfaces compared to normal interfaces. The proxy manager for a method of a multicast interface is the same as the one for primitive components (see Figure 6.7). Since a composite itself does not encapsulate any application logic, a group proxy is quite different and quite simpler than the primitive component case, it is shown in Figure 6.11. The process *GrProxyComposite_m* defines the behavioural semantics of the proxy for a multicast interface of a composite component: $[\![m]\!]_{proxy}$. After creation and emission of a $MC(G)$ action, this future proxy accumulates replies and when all futures have been received, a $R_m(f, vect)$ action is emitted. Note that, as there is no application logic encapsulated in the composite component, a given policy must be chosen to know when a reply is issued from a multicast interface belonging to a composite. Here we choose to reply the whole vector of replies when it is completely filled. It would also be possible to implement a different policy, for example return the most frequent result or wait until at least half of the replies have been filled.

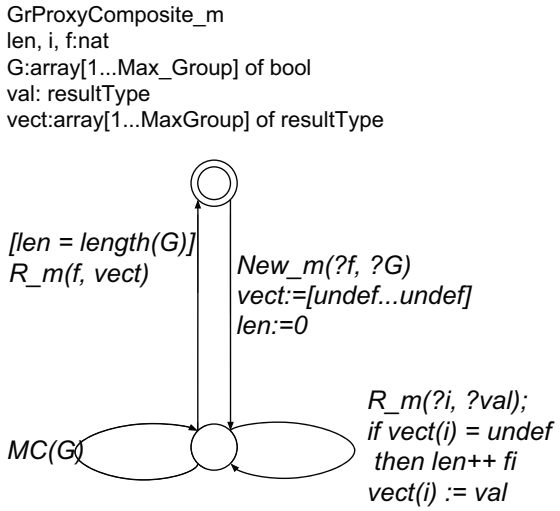


Figure 6.11 – The Proxy of a multicast interface inside a composite component

The behavioural semantics of proxies and their managers is different for the multicast and singleton interfaces. Then, the behavioural semantics of a composite component supporting multicast interfaces is the following: it redefines the synchronisation vectors for transmitting request and replies concerning multicast interfaces, and the ones concerning the group proxies and their management.

$m_i^{i\in L}$ are the methods that belong to multicast interfaces of the composite or its sub-components

$m_i^{i\in L'}$ are the methods that belong to multicast interfaces of the composite component

$$Itf_h{}^{h\in H} = (CItf_j^{j\in J}) \uplus (\mathrm{Symm}(SItf_i)^{\,i\in I})$$

$$SV^{MC} = SV_C^{MC}(CItf_j^{j\in J}, Itf_h^{h\in H}) \cup SV_C(CItf_j^{j\in J}, Itf_h^{h\in H}) \cup SV_B^{MC}(TopBinding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, CName)$$

$$[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, TopBinding_b^{b\in B} >]\!]^{MC} = [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, TopBinding_b^{b\in B} >]\!] \oslash Q_m_l^{l\in L} \oslash GetProxy_m_l^{l\in L'} \oslash New_m_l^{l\in L'} \oslash R_m_l^{l\in L'} \oplus \langle\!\langle SV^{MC} \rangle\!\rangle$$

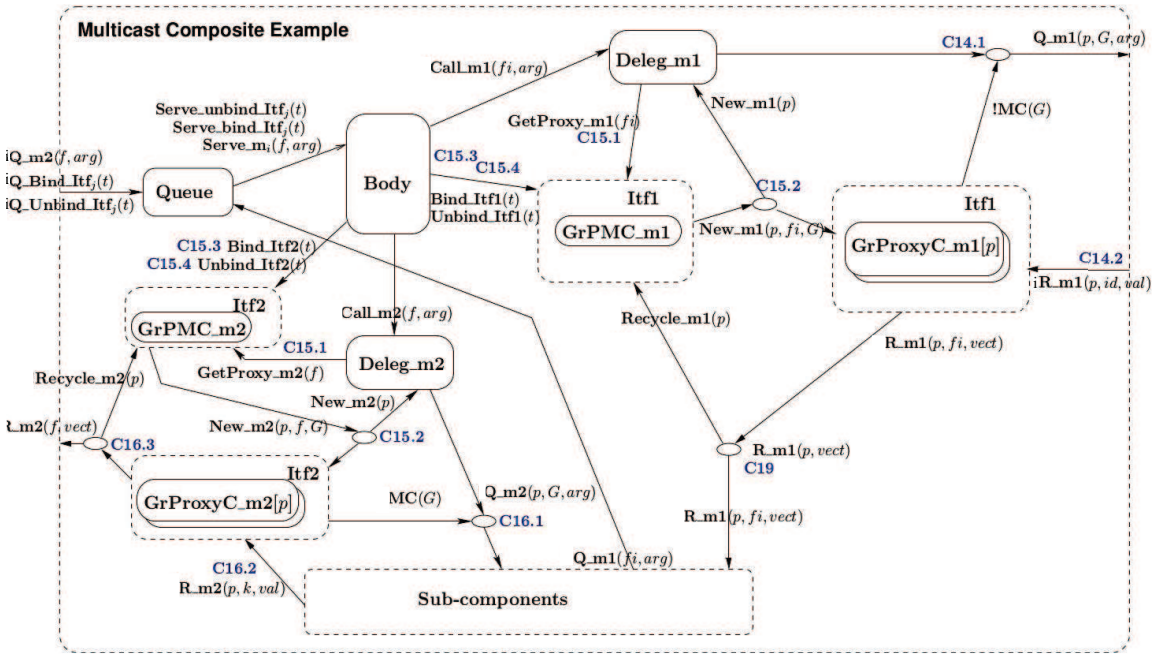


Figure 6.12 – pNets for the Composite component with multicast internal and external interfaces

Figure 6.12 illustrates the construction of synchronisation vectors for a composite component having one internal client multicast interface, and one external client multicast interface. The rules for generating the synchronisation vectors dealing with the multicast server and client interfaces of a composite component are shown in Table 6.3.

The first rule [C14] defines the emission [C14.1] of a request on a multicast external client interface and the reception of a reply by this interface [C14.2]. The request is emitted by the delegation method indexed k, the proxy provides the target group G. The emission of request by *internal* multicast client interfaces will be described below as it depends on the bindings inside the composite component. The reply reception [C14.2] is similar to the reply reception in a primitive component [P6.4].

The second rule [C15] expresses the creation of a new future proxy and the binding/unbinding of interfaces, it is very similar to the primitive component case, except that, as it is the case for a normal interface of a composite component, the future corresponding to the request served by the composite component is transmitted to the proxy.

We now describe how we generate synchronisation vectors for the bindings involving a multicast interface. As it was defined in Section 4.4.4, we require that there is no looping binding and two bindings from the same MC interface do not reach the same component; this allows us here to write synchronisation vectors for expressing

Table 6.3 – Synchronisation vectors for multicast interfaces in composite components. The synchronised sub-pNets occur in the following order:
$\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{j \in J \qquad m_k^{k\in K} = MethLabel(CItf_j) \qquad k \in K \qquad p \in \mathbb{N} \qquad CItf_j \text{ is multicast}}{\begin{array}{ll}\{\langle -, -, k \mapsto Q_m_k(p, arg), -, j \mapsto k \mapsto p \mapsto MC(G), -\rangle \to Q_m_k(p, G, arg), & [1]\\ \langle -, -, -, -, j \mapsto k \mapsto p \mapsto R_m_k(id, val), -\rangle \to iR_m_k((p, id), val)\} & [2]\\ \subseteq SV_C^{MC}(CItf_j^{j\in J}, Itf_h^{h\in H}) & \end{array}} \text{ C14}$$

$$\frac{h \in H \qquad m_k^{k\in K} = MethLabel(Itf_h) \qquad k \in K \qquad f, p \in \mathbb{N} \qquad Itf_h \text{ is multicast}}{\begin{array}{ll}\{\langle -, -, k \mapsto GetProxy_m_k(f), h \mapsto k \mapsto GetProxy_m_k(f), -, -\rangle \to GetProxy_m_k(f), & [1]\\ \langle -, -, k \mapsto New_m_k(p), h \mapsto k \mapsto New_m_k(p, f, G), h \mapsto k \mapsto p \mapsto New_m_k(f, G), -\rangle \to & \\ \qquad New_m_k(p, f, G)\} & [2]\\ \langle -, Bind_Itf_h(t), -, h \mapsto (k' \in K \mapsto Bind(t)), -\rangle \to Bind_Itf_h(t), & [3]\\ \langle -, Unbind_Itf_h(t), -, h \mapsto (k' \in K \mapsto Unbind(t)), -\rangle \to Unbind_Itf_h(t), & [4]\\ \langle Serve_Bind_Itf_h(t), Serve_Bind_Itf_h(t), -, -, -\rangle \to Serve_Bind_Itf_h(t), & [5]\\ \langle Serve_Unbind_Itf_h(t), Serve_Unbind_Itf_h(t), -, -, -\rangle \to Serve_Unbind_Itf_j(t)\} & [6]\\ \subseteq SV_C^{MC}(CItf_j^{j\in J}, Itf_h^{h\in H}) & \end{array}} \text{ C15}$$

multicast interfaces in a simpler way. Indeed, those restrictions ensure that in each of the synchronisation vectors expressed below, each sub-pNet of the composite pNet performs a single action in a given synchronisation vector.

In order to define reconfigurable bindings for multicast interfaces, we rely on *TopBinding*, the maximal set of bindings that can exist. In practice, it is specified by the application architect: in VerCors is it the combination of the normal and dashed bindings going from the multicast interfaces. Then we define four rules for building synchronisation vectors from *TopBinding*. For each of the three first rules, we build G_{max} the maximal set of qualified names that can be bound to the considered multicast interface. Then we consider all the possible subsets G of G_{max}; these are the possible sets on which the request invocations originating from the multicast interface can arrive. Note that SV_B has now *CName* as additional parameter, it is the name of the composite component that contains the bindings. We use two auxiliary functions for computing G_{max}, and for obtaining the index of the component inside a qualified name:

$$\begin{array}{l}\text{Gmax}(TopBinding_b^{b\in B}, QName, CName) = \\ \qquad \{C.Itf | (QName, C.Itf) \in TopBinding_b^{b\in B} \wedge C \neq This\}\{\!\{This \leftarrow CName\}\!\}\end{array}$$

$$Target(C.Itf, Comp_k^{k\in K}) = k \in K \text{ such that } C = \text{Name}(Comp_k)\}$$

Note that Gmax renames the occurrences of *This* into *CName* because when the encompassing composite is bound, it is referred by its name, not *This*.

Table 6.4 shows rules for building synchronisation vectors related to bindings involving a multicast interface. Rule [C16] deals with the case when a server interface is multicast, or more precisely an internal client interface is multicast. The item [C16.1] in the synchronisation vector expresses request emission. Each request emitted by a delegation method is broadcasted to the bound interfaces, where the destination set G is taken from the adequate group proxy. For each destination of the invocation $g \in G$, the index of the target component is obtained thanks to the *Target* function; then $\text{Index}_G(g)$ is attached to the future identifier. Replies can originate from each g member of G independently (asynchronously); overall, we build one reply vector for each element of G_{max}. The synchronisation vectors for replies are split into two vectors compared to singleton interfaces: the return of results from sub-components [C16.2] is done independently from the reply of the overall result [C16.3], the second only occurs when the vector of replies is filled. The last item of the first rule is in fact unrelated to bindings, it is however more natural to mention it here; it sends a reply out of the composite component when the vector of replies of a future proxy f of a multicast server interface has been completely filled. Method renaming (computation of m'_g from m_j) relies on a function *Itf* that returns the interface of a method label.

The second rule [C17] deals with the case when a sub-component has a client multicast interface that sends request to other sub-components. The rules are quite similar to the previous case except that the emitter component has to be found (it is indexed by k). The synchronisation between sub-components for the transmission of a request synchronises the emitter k with the elements of G, or more precisely with the sub-components indexed by $\mathit{Target}(g)$ for $g \in G$. Again, indices of the destination components are attached to the future identifier. The set of synchronised sub-components is a family of $\mathit{card}(G) + 1$ elements (remember that the definition of well-formed components ensures that k cannot be among the indices in G, i.e. that there is no loop binding). There is no need to specify a rule for replies here because the case of singleton interfaces still applies (except that it is instantiated for the maximal binding set, *TopBinding*, and that it returns a future identifier that contains an index $\text{Index}_G(g)$).

Rule [C18] deals with the case when a sub-component has a client multicast interface that sends a request to other sub-components, but also to the encompassing component (e.g. `M1` in Figure 6.13). This rule applies when the invocation is performed on a target group G that contains *CName.Itf* where *CName* is the name of

Table 6.4 – Binding synchronisation vectors for multicast interfaces. The synchronised sub-pNets occur in the following order: $\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{\begin{array}{c} i \in I \qquad SI = \mathrm{Name}(SItf_i) \qquad SI \text{ is multicast} \\ m_j^{j \in J'} = \mathrm{MethLabels}(SItf_i) \\ j \in J' \qquad p, f \in \mathbb{N} \qquad G_{\max} = \mathrm{Gmax}(TopBinding_b^{b \in B}, This.SI, CName) \qquad G \subseteq G_{max} \\ g \in G_{max} \qquad k = Target(g, Comp_k^{k \in K}) \qquad \text{for all } C.Itf \in G.\ m'_{(C.Itf)} = m_j\{\!\{Itf(m_j) \leftarrow Itf\}\!\} \end{array}}{\begin{array}{l} \{\langle -, -, j \mapsto Q_m_j(p,arg), -, i \mapsto j \mapsto p \mapsto MC(G), \\ \qquad \left(Target(g, Comp_k^{k \in K}) \mapsto iQ_m'_g((p, \mathrm{Index}_G(g)), arg)\right)^{g \in G} \rangle \to Q_m_j(p,arg), \quad [1] \\ \langle -, -, -, -, i \mapsto j \mapsto p \mapsto R_m_j(id, val), k \mapsto R_m'_k((p, id), val) \rangle \to R_m_j(p, val) \quad [2] \\ \langle -, -, -, i \mapsto j \mapsto Recycle_m_j(p), i \mapsto j \mapsto p \mapsto R_m_j(f, vect), - \rangle \to R_m_j(f, vect)\} \quad [3] \\ \qquad \subseteq SV_B^{MC}(TopBinding_b^{b \in B}, SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, CName) \end{array}} \text{ C16}$$

$$\frac{\begin{array}{c} k \in K \qquad C = \mathrm{Name}(Comp_k) \\ CItf'^{i \in I'}_i = CItfs(Comp_k) \qquad i \in I' \qquad CI = \mathrm{Name}(CItf'_i) \qquad m_j \in \mathrm{MethLabels}(CItf'_i) \\ CI \text{ is multicast} \qquad G_{\max} = \mathrm{Gmax}(TopBinding_b^{b \in B}, C.CI, CName) \qquad G \subseteq G_{max} \\ \nexists Itf.CName.Itf \in G \qquad f \in \mathbb{N} \qquad \text{for all } C.Itf \in G.\ m'_{(C.Itf)} = m_j\{\!\{Itf(m_j) \leftarrow Itf\}\!\} \end{array}}{\begin{array}{l} \langle -, -, -, -, - , \\ \left(k \mapsto Q_m_j(f, G, arg), \left(Target(g, Comp_k^{k \in K}) \mapsto iQ_m'_g((f, \mathrm{Index}_G(g)), arg)\right)^{g \in G}\right) \rangle \to \\ \qquad Q_m_j(f, arg) \\ \qquad \in SV_B^{MC}(TopBinding_b^{b \in B}, SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, CName) \end{array}} \text{ C17}$$

$$\frac{\begin{array}{c} k \in K \qquad C = \mathrm{Name}(Comp_k) \\ CItf'^{i \in I'}_i = CItfs(Comp_k) \qquad i \in I' \qquad CI = \mathrm{Name}(CItf'_i) \qquad CI \text{ is multicast} \\ m_j \in \mathrm{MethLabels}(CItf'_i) \qquad G_{\max} = \mathrm{Gmax}(TopBinding_b^{b \in B}, C.CI, CName) \qquad G \subseteq G_{max} \\ G = \{CName.Itf\} \uplus G' \qquad f \in \mathbb{N} \qquad \text{for all } C.Itf \in G.\ m'_{(C.Itf)} = m_j\{\!\{Itf(m_j) \leftarrow Itf\}\!\} \end{array}}{\begin{array}{l} \langle iQ_m_{CName.Itf}(f, arg), -, -, -, - , \\ \left(k \mapsto Q_m_j(f, G, arg), \left(Target(g, Comp_k^{k \in K}) \mapsto iQ_m'_g((f, \mathrm{Index}_G(g)), arg)\right)^{g \in G'}\right) \rangle \to \\ \qquad Q_m_j(f, arg) \\ \qquad \in SV_B^{MC}(TopBinding_b^{b \in B}, SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, CName) \end{array}} \text{ C18}$$

$$\frac{\begin{array}{c} k \in K \qquad C = \mathrm{Name}(Comp_k) \\ (C.CI, This.CI_2) \in TopBinding_b^{b \in B} \qquad CI_2 \text{ is multicast} \qquad j \in J \qquad \mathrm{Name}(CItf_j) = CI_2 \\ f, q \in \mathbb{N} \qquad m_n^{n \in N} = \mathrm{MethLabels}(CItf_j) \qquad n \in N \qquad m'_n = m_n\{\!\{CI_2 \leftarrow CI\}\!\} \end{array}}{\begin{array}{l} \langle -, -, -, j \mapsto n \mapsto Recycle_m_n(q), j \mapsto n \mapsto q \mapsto R_m_n(f, vect), k \mapsto iR_m'_n(f, vect) \rangle \to \\ \qquad R_m_n(q, vect) \\ \qquad \in SV_B^{MC}(TopBinding_b^{b \in B}, SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, CName) \end{array}} \text{ C19}$$

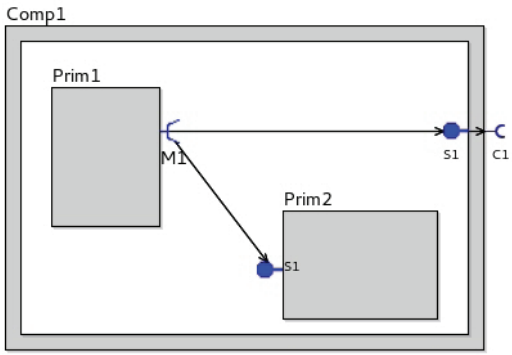


Figure 6.13 – A multicast of a sub-component sends an external request

the composite component. Note that as bindings are reconfigurable, the composite component can be bound or not, and depending on whether it is bound, [C17] or [C18] applies. This rule only applies for request transmission, it is similar to the preceding rule except that the composite component also receives a request and that the set of destination sub-components is obtained from G', the elements of G that are not the composite component.

The last rule [C19] deals with the sending of replies from a multicast client interface of the composite component to a sub-component. As replies for multicast are bound similarly to replies for singleton interfaces, the only difference is the time when the client interface of the composite sends the reply. Indeed, if the interface is singleton the reply occurs as soon as one reply is received, and the proxy for future is used to rename the future identifier (see Section 5.1.2). In the case of a multicast interface, the reply occurs independently from external communications *when the vector of replies is entirely filled*. This is visible in the rule because the global action is just an observable action of the form R_m instead of a communication reception of the form iR_m. Rule [C14.2] that specifies the reception of the reply iR_m by the composite still applies for receiving replies from other components.

Methods that do not return any value. We do not need to construct a group proxy for a void method of a multicast because there is no need to wait for the result. However, we should still create the group manager in order to store the configuration of the current group. Figure 6.14 illustrates the structure of a group manager for a void method. It has only three actions: one to bind an interface, one to unbind an interface, and an action to invoke the method on the current group.

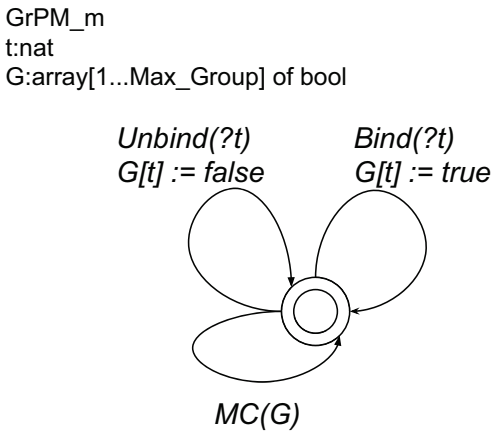


Figure 6.14 – A Group Manager for a void method

6.3.3 Implementing pNet generation and integration with CADP

At the pre-processing phase, VerCors analyses all user-defined multicast interfaces and gathers the information necessary for the construction of the pNets. For each multicast we extract:

- the list of all possible outgoing bindings. Recall that the user can define bindings of two types: the ones bound during application construction and the ones bound at the execution time (the dashed bindings); moreover, the user can associate indices to the bindings. The order in the list of the possible outgoing bindings must correspond to the user-defined indices associated to the bindings if there are any;
- the initial group size: the number of bindings bound at application construction time;
- the maximum group size: the number of bindings which can be potentially bound to the interface;

Then, for each method of an internal and external multicast interface, VerCors generates a group proxy and a group manager pLTSs and synchronises them. The information gathered during pre-processing is used in order to construct the type and the initial value of the variable G which stores the group of target interfaces in a group manager. Several constructs included in the pLTSs illustrated in Figures 6.7 and 6.11 are not supported by Fiacre and by the meta-model of VerCors pNets. Hence, in order to prepare the generated pLTSs to their translation into Fiacre, VerCors produces a

group proxy and a group manager which are structurally slightly different from the ones presented in the previous section.

In particular, an expression `[len=length(G)]` which compares the number of received replies and the number of bound interfaces in a group proxy cannot be directly included in Fiacre. The reason is that the construct `length(G)` is not defined anywhere. Instead, in the generated group manager we store a variable `gr_length` representing the size of the current group; its value is modified upon *Bind* and *Unbind* actions, and it is sent to the newly allocated proxy so that the proxy can compare it to the number of obtained results.

Another difference is that the `undef` expression which corresponds to an undefined reply from a target interface in a group is not supported by VerCors types. In order to store the information regarding received replies, we add an additional array variable in a group proxy.

Finally, the synchronisation vectors in EXP and in the meta-model of pNets in VerCors cannot be associated with guards like the multicast method invocation vectors illustrated in Figure 6.10. In order to enable reconfigurable connectors, we rely on the renaming capability of CADP. More precisely, in the pLTS of a group proxy we rename the action *MC(G)* so that the value of the parameter G is included in the guard, and we generate separately an invocation synchronisation vector for each possible group combination. For example, an invocation of a method `m` of the internal multicast `Mcast-int` in Figure 6.6 can be encoded with the following synchronisation vectors:

$$< \mathsf{MC_0_0_0}, -, -, - > \rightarrow \mathsf{MC_0_0_0}$$
$$< \mathsf{MC_1_0_0}, \mathsf{iQ_m}, -, - > \rightarrow \mathsf{MC_1_0_0}$$
$$< \mathsf{MC_1_1_0}, \mathsf{iQ_m}, \mathsf{iQ_m}, - > \rightarrow \mathsf{MC_1_1_0}$$
$$< \mathsf{MC_1_1_1}, \mathsf{iQ_m}, \mathsf{iQ_m}, \mathsf{iQ_m} > \rightarrow \mathsf{MC_1_1_1}$$
$$< \mathsf{MC_0_1_0}, -, \mathsf{iQ_m}, - > \rightarrow \mathsf{MC_0_1_0}$$
$$< \mathsf{MC_0_1_1}, -, \mathsf{iQ_m}, \mathsf{iQ_m} > \rightarrow \mathsf{MC_0_1_1}$$
$$< \mathsf{MC_0_0_1}, -, -, \mathsf{iQ_m} > \rightarrow \mathsf{MC_0_0_1}$$
$$< \mathsf{MC_1_0_1}, \mathsf{iQ_m}, -, \mathsf{iQ_m} > \rightarrow \mathsf{MC_1_0_1}$$

where the participating sub-nets are organised in the following order: $<$ GrProxy_m, SubComp1, SubComp2, SubComp3 $>$ (for the sake of simplicity we omit the synchronised pNets which are not involved in the method invocation). Depending on the value of the group, the pLTS of the group proxy emits *MC_0_0_0(args)* or *MC_1_0_0(args)*, etc).

6.3.4 Code generation

If the designed application with multicast interfaces has been proved correct with respect to the user-defined requirements, VerCors can generate its GCM/ProActive code. In the ADL file, multicast client interfaces should be tagged as `collective`. The bindings which should not be initially constructed by the factory (e.g. the dashed binding in Figure 6.6) are not included in the ADL description. Finally, the generated Java code does not distinguish invocation of methods which belong to singleton and multicast interfaces.

6.4 Reconfiguring multicast interfaces from component-controllers

The internal and external interfaces of a composite component can be reconfigured by the non-functional components located in its membrane. In this section we explain how such reconfiguration can be modelled graphically, how it is encoded in pNets and translated into Java code.

6.4.1 Graphical specification

While executing a server or a local method, a component-controller can trigger reconfiguration of an interface of its container. As for any primitive in VerCors, the behaviour of non-functional primitives is defined using UML state machines (see Section 3.2). Hence, the requests that trigger the reconfiguration should be also specified in the state machines. For this, we introduce two specific state machine instructions: `unbind` and `bind`, which represent method invocations on the `parent` component. They take two input parameters: the name of the interface to be modified and the index of the binding to be unbound or bound (remember the binding indices discussed in Section 6.3.1).

Figure 6.15 illustrates an example of an application where a component-controller unbinds an internal interface of its container. More precisely, `Controller` has one server method `unbindM1` which has only one instruction: `parent.unbind(Interfaces.M1, 1)`. Here, `M1` is the name of the internal interface which will be modified, and `1` is an index of the binding which should be removed.

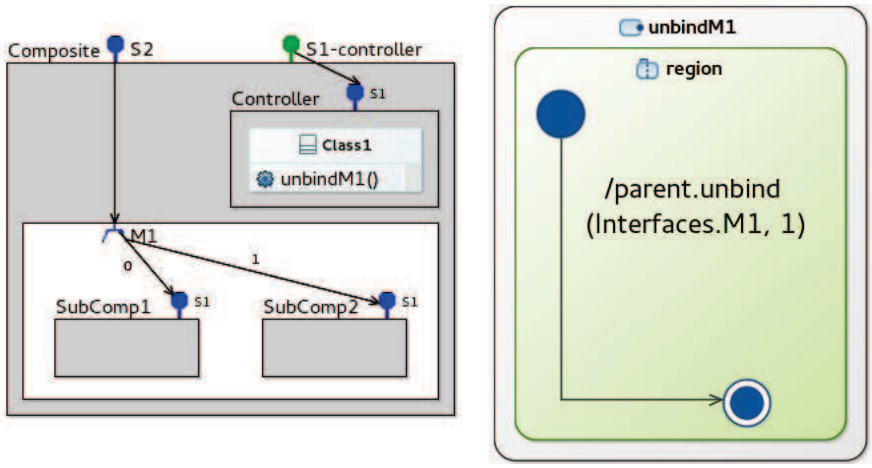


Figure 6.15 – Modelling binding reconfiguration

6.4.2 From application design to pNets

When generating the pNet of a composite, for each method of its multicast interfaces we construct the pLTSs of a group proxy and a group manager which encode possible reconfiguration. The only additional thing needed in order to trigger the reconfiguration is to synchronise the pNet of a component-controller with the queue of the composite when the reconfiguration request is en-queued. More precisely, we generate a synchronisation vector propagating each reconfiguration action of a component-controller so that it is visible from outside. Then, we synchronise it with the queue of the composite.

In our example, we extend the pNet of `Controller` with the following synchronisation vector:

$$\langle -, -, \mathit{Unbind_Parent_M1}(t) \rangle \rightarrow \mathit{Unbind_Parent_M1}(t)$$

where the sub-nets are organised as follows: $\langle \mathit{Queue}, \mathit{Body}, \mathit{unbindM1} \rangle$. Next, when generating the pNet of `Composite`, we synchronise the reconfiguration action of `Controller` with the queue of the composite:

$$\langle \mathit{iQ_Unbind_M1}(t), -, -, -, \mathit{Unbind_Parent_M1}(t), - \rangle \rightarrow \mathit{iQ_Unbind_M1}(t)$$

where the order of the synchronised sub-nets is:

$\langle \mathit{Queue}, \mathit{Body}, \mathit{ProxyManagers}, \mathit{Proxies}, \mathit{MembraneSubcomps}, \mathit{ContentSubcomps} \rangle$

6.4.3 Implementing pNet generation and integration with CADP

In order to generate pNets of applications with component-controllers which reconfigure interfaces of their containers, we include several additional steps in the pNet construction process discussed in Section 5.3.

First, pre-processing the state machines which model the behaviour of component-controllers becomes slightly more complex. In addition to the usual analysis, we extract all reconfiguration instructions. For each reconfiguration instruction, we create a structure which stores a reference to the modified interface, a reference to the modified binding, and the instruction type (bind or unbind). This information is stored in the parsed state machine and used when the transition label with the corresponding instruction is translated into a pLTS action.

When generating the synchronisation vectors for a pNet of a component-controller, we check whether the parsed state machines modelling its behaviour include reconfiguration instructions. If so, each analysed instruction is translated into a pLTS action and included in a synchronisation vector in order to be visible from outside of the component-controller. This allows us to synchronise it later with the queue of the composite-container.

Finally, when constructing the pNet of a composite, we check if the state-machines modelling the behaviour of its component-controllers include reconfiguration instructions. If so, each reconfiguration instruction is translated into a pLTS action and synchronised with the corresponding action in the queue of the composite.

6.4.4 Code generation

From the model of an application with reconfiguration of multicast interfaces, VerCors generates executable ProActive/Java code. However, the way reconfiguration instructions are specified in GCM/ProActive and in VerCors is quite different, thus the straightforward translation of reconfiguration statements from state machines to Java code is not possible.

For a given composite component, GCM/ProActive allows the programmer to get an instance of the `PAMulticastController` class which is able to reconfigure the multicast interfaces. For this, it has two methods: `bindGCMMulticast` and `unbindGCMMulticast`; both of them take two input parameters: the name of the multicast source interface and a reference to the target interface of the modified binding. On the other hand, in VerCors the reconfigured binding is referenced through the name of its source interface and a binding index (which does not exist in GCM/ProActive). When translating state machine instructions into Java code, we can compute statically the target interface from the binding index if the index is a constant value like in our example in Figure 6.15. However, it is not always the case: the index of the reconfigured binding in a state machine instruction can be specified as a variable.

The solution we offer is to generate in the Java code a specific map. The map stores the relations between the indices and the interfaces targeted by the corre-

sponding bindings as it is specified in VerCors. Then, the map can be used during the program execution in order to retrieve the desired target interface based on its index. More precisely, when generating the ADL file of a component-controller, we include an additional attribute `hostReconfBindings` which encodes a mapping from the reconfigurable multicast interfaces to the list of target interfaces. The order of the elements in the lists corresponds to the indices of the plugged bindings. Listing 6.4 illustrates the attribute included in the ADL file of `Controller` from Figure 6.15.

```
1 <attribute name="hostReconfBindings" value="M1:[SubComp1.S1, SubComp2.S1]"/>
```

Listing 6.4 – An ADL attribute which stores reconfigurable interfaces

By default, when the GCM/ProActive factory constructs a component, it invokes a user-defined set-method for each of its attributes specified in the ADL file. We generate a specific implementation of the `set_hostReconfBindings` method which parses the value of `hostReconfBindings` and constructs two maps. `itfsMap` maps the name of a reconfigurable multicast interface to the list of names of the target interfaces, `compsMap` maps the name of a multicast interface to the list of names of the target components. The elements in the lists must be ordered according to the attribute value in the ADL file. These names are used later in order to retrieve the target interface by its index.

```
1 public class Controller extends AbstractPAComponentController implements ... {
2   //a map from the source reconfigurable interfaces to the target interfaces
3   private Map<String, List<String>> itfsMap;
4   //a map from the source reconfigurable interfaces to the target components
5   private Map<String, List<String>> compsMap;
6   //the method computes and returns a reference to an interface based on its name,
7   // the name of its host component and the container of its host component
8   public Object getInterface(Container compContainer, String compName, String itfName) {...}
9   //the method parses the input string and fills itfsMap and compsMap
10  public void setHostReconfBindings(String val) {...}
11  public void unbindM1() {  ...
12    String tgtCompName = this.compsMap.get("M1").get(1);
13    String tgtItfName = this.itfsMap.get("M1").get(1);
14    ISingle tgtItf = (ISingle)this.getInterface(Container.CONTENT, tgtCompName, tgtItfName );
15    Utils.getPAGCMLifeCycleController(this.hostComponent).stopFc();
16    Utils.getPAMulticastController( this.hostComponent).bindGCMMulticast("M1", tgtItf);
17    Utils.getPAGCMLifeCycleController(this.hostComponent).startFc();
18    ... }}
```

Listing 6.5 – Java code of a component-controller

Listing 6.5 provides a simplified snippet of the Java code generated for the class implementing the behaviour of `Controller`. The maps storing the information about the reconfigurable interfaces are declared in lines 2-5. Line 9 defines a method which uses ProActive API in order to retrieve the reference to an interface by its name, the name of its component, and the container of its component. This method will be used

in order to get the target interface of the reconfigured binding. The information about the container of the component to which the interface is attached is needed because the target interface can be attached to a component in the content (when reconfiguring an internal interface) but also to a component outside the composite (while the reconfigured interface is external). Line 12 defines `setHostReconfigBindings` method which is used by the factory to set the values of `itfsMap` and `compsMap`. Finally, the translation of `parent.unbind(Interfaces.M1, 1)` instruction is given in the lines 16-21. It, first gets the names of the target interface and of the target component (lines 16-17) based on the source interface and the binding index. Here, the binding index is equal to 1 but it could also be a variable or an expression. Then, we use the retrieved names for the invocation of the `getInterface(...)` method in line 18 which returns a reference to the target interface. Now, all the information needed for the reconfiguration is gathered, but before modifying an interface of a composite component, we have to stop its functional part (line 19). Note, that here we stop `this.hostComponent`, i.e. the composite containing our component-controller. Once the functional part is stopped, the reconfiguration can be performed (line 20). Finally, the composite can be started again (line 21).

To sum-up, we have developed a framework for modelling, verification, and generation of hierarchical component-based applications with functional and non-functional aspects, reconfigurable multicast interfaces, attributes, and component-controllers that can launch the reconfiguration. In the following section we will demonstrate how our techniques can be applied in practice.

6.5 Examples

In this section we introduce the examples of two projects created in the VerCors platform. In the first example we present a hierarchical software system with component-controllers that reconfigure multicast interfaces and with attribute controllers. The second example illustrates the usage of interceptors.

6.5.1 Composite pattern

Problem statement. In order to test our approach, we designed, model-checked, and generated the code of a refined version of an application which was proposed as a challenge problem at the SAVCBS[1] workshop on specification and verification of component-based systems in 2008. Later, the problem statement was published

[1]http://www.eecs.ucf.edu/ leavens/SAVCBS/2008/challenge.shtml

in [88] as a part of a set of benchmarks for verification tools. The problem is dedicated to the *composite pattern* which is very common in component- and object-oriented programming. According to the challenge, there exists a tree of components (in GCM terms we call it a "hierarchy of components"), and a client has a uniformed interface to access any sub-tree. Each composite in the tree has a counter `childrenNum` which stores the number of its sub-components at all levels of hierarchy. The client can add a sub-component anywhere in the system, and this should increment the value of `childrenNum` for all its ancestors. The challenge is to keep the value of `childrenNum` up-to-date.

We modelled the discussed application in VerCors. Since we plan to apply a finite state-space model-checker, we cannot allow the client to add an infinite number of sub-components. Instead, we model a system with three levels of hierarchy where each composite (except from the leaves) can have three functional sub-components and one non-functional sub-component responsible for the reconfiguration and for storing the `childrenNum` variable.

In order to keep the proper encapsulation, we allow the client to have access only to the interfaces of the root component of our hierarchy. More precisely, the client can use a server interface of the root component and invoke the method `addSubcomp(parentId)` on it. Here, `parentId` is the identifier of the component inside which a new sub-component should be added. This parent component can be located anywhere in the hierarchy. It should be also mentioned that each composite stores an attribute `myId` which has a unique value and which is compared to `parentId`. The `addSubcomp(parentId)` can return `true` or `false` depending on whether a sub-component has been successfully added inside the component with the given `parentId`. The result is negative in two cases: either the component with the `parentId` is not accessible (it has not been added to the system yet or it does not exist at all) or if such a component has already added the maximum number of sub-components. Finally, each composite has an attribute `childrenNum` that stores the total number of sub-components added to its sub-tree.

Graphical design. Figure 6.16 illustrates the component diagram of our use-case example; for the sake of simplicity we hide the internal structure of most of the components. The root component `Comp1` has three composite sub-components with an identical structure: `Comp11`, `Comp12`, and `Comp13`. Each of them also has three sub-components with an identical structure. Each composite component (except from the ones at the lowest level of hierarchy) has an internal multicast interface `C1` which can be bound to the three functional sub-components. All the bindings

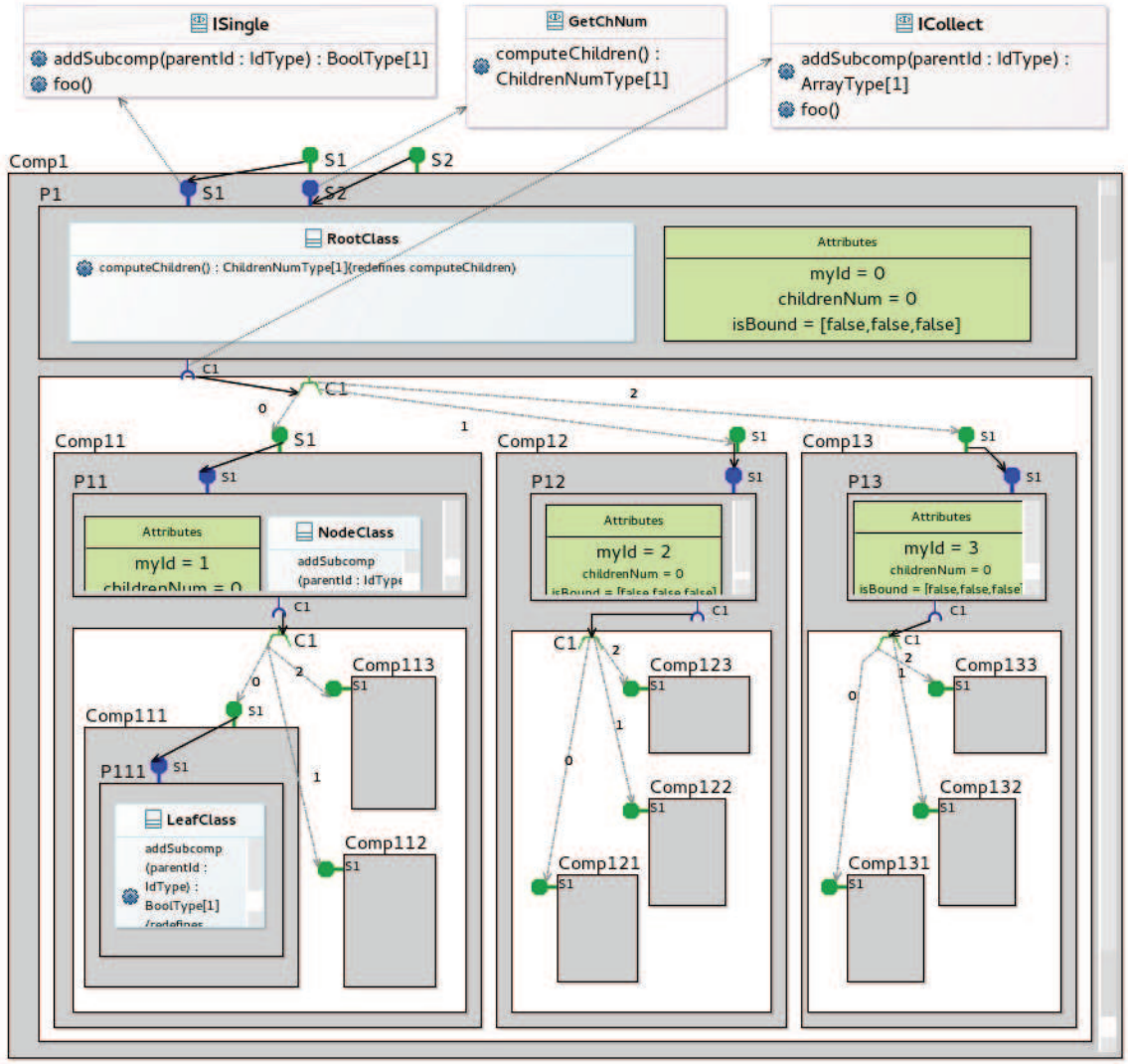


Figure 6.16 – VerCors model of the composite pattern

going from the multicast interfaces are indexed. One can notice that the bindings are dashed which means that they do not exist when the application is launched but they can be added at run-time. In fact, the current version of VerCors does not allow modelling applications with dynamically created components. Instead, we consider a component to be "added" in the tree when it is bound to the multicast interface `C1` of its container. Sub-components cannot be added to the composites at the lowest levels of hierarchy (i.e. to `Comp111`, `Comp112`, etc.).

Each composite has a component-controller responsible for adding sub-components. The behaviour of the component-controllers at the lowest levels of hierarchy is modelled by the class `LeafClass`. The behaviour of the component-controllers of Comp11, Comp12, and comp13 is modelled by `NodeClass`. The root controller of the root component is implemented by `RootClass` which extends

Figure 6.17 – The class diagram of the composite pattern

`NodeClass`; all the classes are illustrated in Figure 6.17. `NodeClass` has three attributes: `myId` is the unique identifier, `childrenNum` is the number of all functional sub-components at the lower levels of hierarchy, and `isBound` represents the current status of the internal multicast interface `C1` of the encompassing composite. The latter attribute is an array of three boolean values. The value at a given index is `true` if the outgoing binding with the corresponding index is bound. Otherwise, it is `false`. Since no outgoing binding is initially bound to the multicast interfaces, `isBound` is initially equal to `[false, false, false]` for all the components.

Each component-controller has a server method `addSubcomp` accessible from outside of its composite. The method is responsible for adding a sub-component and it takes one input parameter `parentId` which represents the identifier of the component where a sub-component should be added. If, for instance, the client would like to bind a functional sub-component inside `Comp12`, he would invoke the method with the argument equal to one. `addSubcomp` returns `true` if a sub-component has been successfully added in the component on which the method was invoked or in any of its sub-components. The method is implemented differently for `NodeClass` and for `LeafClass` because the former cannot add any sub-component to the leaves of the hierarchy.

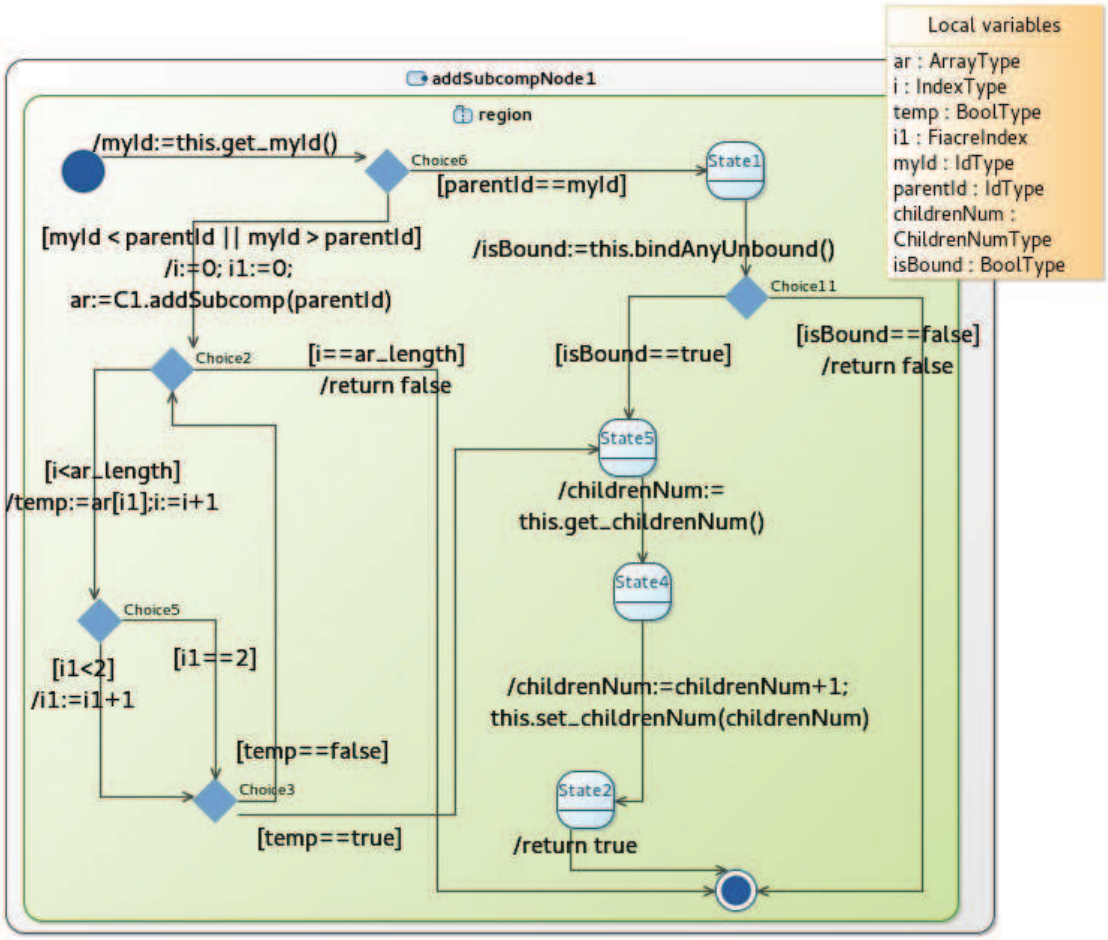


Figure 6.18 – addSubcomp method

Figure 6.18 illustrates a state machine modelling `addSubcomp` of `NodeClass`. First, in state `Choice6` it checks if the received `parentId` equals its own identifier. Then, two options are possible:

- If the two values differ, `addSubcomp` request is sent to the sub-components through the multicast interface `C1`: `ar:=addSubComp(parentId)`. This invokes `addSubcomp` on all the sub-components bound to `C1`. Then, we iterate over the array of received responses in order to check whether one of them is equal to `true`. If such a response is found, the `childrenNum` value of the current component is incremented, and the method returns a positive reply. Otherwise, it returns `false`.

- If the input parameter is equal to the identifier of the current component, it means that a new sub-component should be added inside the encompassing composite. However, before doing the reconfiguration, we have to check that adding a sub-component is still possible. Recall that our application can have

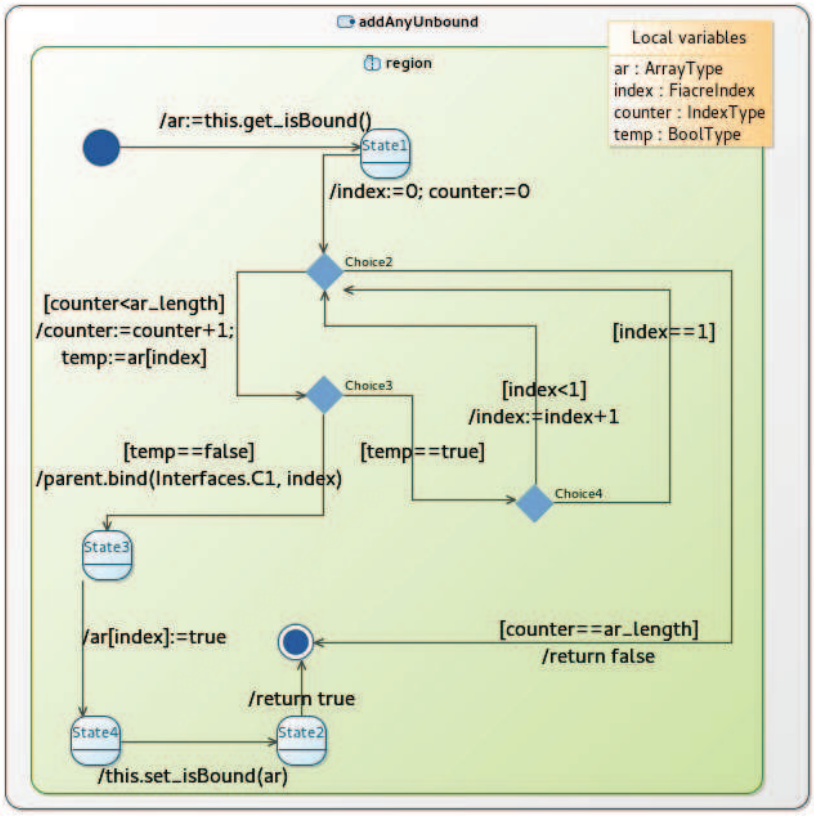


Figure 6.19 – addAnyUnbound method

only finite number of components, and we modelled our system so that not more than three sub-components can be added inside a composite. The check is done by `addAnyUnbound` local method which is invoked by the transition `/isBound:=this.addAnyUnbound()` and illustrated in Figure 6.19. The method iterates over the `isBound` array in order to find the index of a binding which has not been bound yet. If such an index exists, the method binds the binding at the corresponding index (`/parent.bind(Interfaces.C1, index)`, sets the value of `isBound` at the index to `true` in order to remember that the binding has been added, and returns `true` to `addSubcomp` in order to report that a sub-component has been successfully bound. `addSubcomp`, in its turn, increments the value of `childrenNum` and returns `true`. If all three sub-components have been bound before, the method returns `false` and does not modify the counter of the number of children.

A sub-component cannot be added at the lowest levels of hierarchy. Hence, `addSubcomp` of `LeadClass` always returns `false` and does not perform any additional actions.

The `addSubcomp` method of the server interface `S1` of `Comp1` is the entry point of

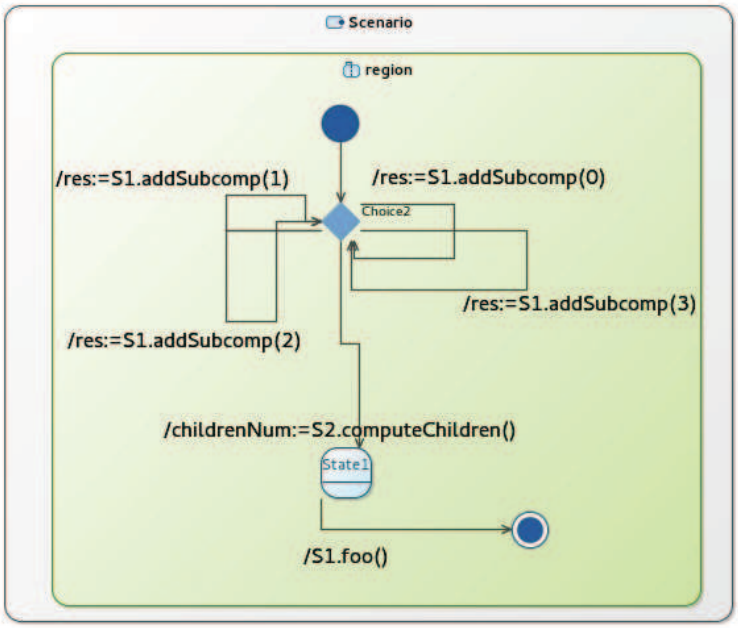


Figure 6.20 – Scenario for the composite pattern application

the modelled application. If the user wants to add a sub-component somewhere in the hierarchy, he should invoke this method with the identifier of the target component. In addition, each component-controller has a method `foo`. The method does not encapsulate any logics, and we will use it later for the debugging purposes. Whenever it is possible, the `foo` method invokes the `foo` method on the internal multicast interface of the composite-container, thus propagating the call to the sub-components.

Finally, the root composite has an interface `S2` with the `computeChildren` method which returns the value of `childrenNum` stored by its component-controller.

Model-checking and executable code generation. We used VerCors in order to generate the pNets of the modelled application and to translate them into the input for CADP. To reduce the global state-space of the system, we synchronised it with the scenario illustrated in Figure 6.20. It adds random number of sub-components to random composites in the application: there are no guards on the transitions going from the choice state, hence, each time the next transition is chosen non-deterministically. Then, the scenario requests the value of `childrenNum` of the root component and invokes the `foo` method on the system.

We started building the model following the bottom-up approach: first, we constructed the components at the lower levels of hierarchy. From the very beginning we realised that considering the reconfiguration, the variety of possible requests and the values of the parameters, the state-space was going to be large. Hence, we de-

cided to start by generating the necessary files and by building the state-space for `Comp11`, `Comp12`, and `Comp13`. Then, we generated the remaining processes for `Comp1` and constructed the final state-space using the obtained models for the three sub-components.

Building the state-space for Comp11, Comp12, and Comp13 Recall that when the user launches the state-space generation, he has to choose the root component of the application and optionally the state machine modelling the behaviour of the environment (i.e. the scenario). By default, VerCors constructs the state-space starting from the processes at the lowest levels of hierarchy like if they are not affected by the scenario, and then synchronises the root component with the scenario. The reason is that in the current version we do not have tools to compute automatically the impact of the scenario on the sub-components while building their state-space. However, the scenario of our use-case example is quite generic and it is not difficult to infer from the business logic of the application which requests from the environment modelled in Figure 6.20 can eventually reach `Comp11`, `Comp12`, and `Comp13`. In particular, we know that the method call `S1.addSubcomp(0)` is not forwarded to these three components because `0` is the identifier of the root component, hence, `Comp1` will try to add a sub-component inside its content instead of forwarding the call. Also, `S2.computeChildren()` cannot be invoked on the sub-components simply because they do not serve this method. Hence, we were able to create a state machine that models the scenario for `Comp11`, `Comp12`, and `Comp13`: it is similar to the scenario for the root component but it does not have the two transitions with the instructions `S1.addSubcomp(0)` and `S2.computeChildren()`. Then, we automatically generated from VerCors the state-space for `Comp11`, `Comp12`, and `Comp13` synchronised with their scenario.

The time to generate .fiacre, .exp files and the auxiliary scripts from VerCors is negligible. The overall time to construct `Comp11` was `30 minutes`. This includes the time to run the Flac compiler, to build the sub-components of `Comp11`, and to construct the final product synchronised with the scenario. Eventually the model of `Comp11` was only `994` states. The reason why constructing so few states took 30 minutes is that, again, when the sub-processes of `Comp11` are being constructed, they are not synchronised with the scenario. It means that building the queue of the component and its sub-component `P11` takes a lot of time, because all possible interleaving of the incoming requests have to be taken into consideration.

Then, in a similar way we generated the state-space for `Comp12` and `Comp13`. Since they have exactly the same behaviour and structure as `Comp11`, constructing

their state-spaces took the same time. At this point we were wondering how much time we would be able to gain, if we could analyse automatically the modelled system in VerCors and predict that the behaviour of the three components is similar. We modified manually several files generated for `Comp11` and generated .bcg for some of them in order to obtain the state-space for `Comp12`. It took us approximately two minutes which means that detecting statically the similarities between components and optimising the generated scripts accordingly would save us 28 minutes for building the state-space of `Comp12`.

Building the final product We used VerCors to generate automatically all the files necessary to build the state-space of our use-case example synchronised with the scenario illustrated in Figure 3.7. We modified the generated `.sh` script so that it does not lunch the constructions of `Comp11`, `Comp12` and `Comp13` because we had already built them.

The final non-reduced state-space of `Comp1` synchronised with the scenario was `143.689.330` states, and the minimiser of CADP managed to reduce it to `31.699.470` states. Its construction and minimisation took us almost `11 hours`. This includes running Flac compiler for all the pLTSs of `Comp1` and for the scenario, building the state-space of `P1`, and synchronising all this with `Comp11`, `Comp12`, and `Comp13`. Most of the time was used for constructing the queue and synchronising the final product. Running the Flac compiler for the queue of the `Comp1` took us almost `6 hours`. The reason is that, at this point, the queue is not affected by the scenario and it should be able to handle all possible interleaving of the requests both from outside of the composite and from its internal components.

To sum-up, building the state-space of the whole system took us almost `13 hours` and the obtained state-space is `31.699.470` states. We also modelled in VerCors and generated the state-space for a similar scenario and a similar system but with only two functional sub-components at each level of hierarchy and `myId` ranging from `0` to `2`. Then, we model-checked several properties on both examples.

Checking properties. When specifying the properties, we have to remember the relations between the events in the graphical model and the corresponding actions in the BCG graph. For instance, the action `Scenario_S1_addSubcomp` in the behaviour graph corresponds to the event where the scenario (the environment) invokes the `addSubcomp` method on the `S1` interface of the root component. Such translation should be automatised in the wizard for the specification of the model-checked properties, but in the current version we specify the properties manually.

First, we checked that it is possible to add a sub-component at the two higher levels of hierarchy, i.e. that there exist paths in the behavioural graph where a call to `addSubcomp` with `parentId` equal to 0 (the id of `Comp1`) or to 1 (the id of `Comp11`) returns `true`. The reply from the model-checker after the verification of the following formula was TRUE.

<true* . 'Scenario_S1_addSubcomp !POS (0)' . (not 'Scenario_S1_addSubcomp.*')*
. 'R_S1_addSubcomp !POS(1)'>true

and

'<true* . 'Scenario_S1_addSubcomp !POS (1)' . (not 'Scenario_S1_addSubcomp.*')*
. 'R_S1_addSubcomp !POS(1)'>true

Then, we checked that if a component has not been bound in the system, it cannot be accessed. More precisely, we checked that `Comp11`, `Comp12`, and `Comp13` cannot be accessed unless a sub-component is added to `Comp1`.

Absence_Before ('Comp1[1| 2| 3].*', 'Scenario_S1_addSubcomp !POS (0)')

Next, we checked that sub-components cannot be added inside a composite more than twice. For this we used an MCL pattern `Bounded_Existence_Globally` which verifies that a given action predicate is satisfied in the model exactly two times. The following formula checks this property for `Comp12`:

Bounded_Existence_Globally ('Comp12_P12_Bind_C1.*')

For the use-case with two sub-components at each level the model-checker answers TRUE. However, in the case of three sub-components it answers FALSE and provides an example of a path where three sub-components are added in the content of `Comp12`. We checked a similar formula for all the composites in both examples.

Finally, we checked the main property of the system stating that the value of `childrenNum` variable of the root component is equal to the number of sub-components added at all levels of hierarchy. In order to obtain the value of `childrenNum`, the scenario invokes the `computeChildren` method on the `S2` interface of the root component. In order to count the number of components which are active in the application, we invoke the `foo` function on the root composite. The call is propagated to the sub-components at all levels of hierarchy, and we count how many components have processed the method invocation. If the value returned by `computeChildren` is equal to x, we expect the number of components that received the `foo` request to be $x+1$ because the root component should also receive it. The following MCL formula corresponds to the property:

[true* . R_S2_ComputeChildren ?x:Nat .
(not 'R_S1_addSubcomp.*')* . "Scenario__S1_foo"]
<('.*Serve_S1_foo.*' . (not ('R_S1_addSubcomp.*' or "Scenario_S1_foo"))*){x+1}>true

The first line of the formula explores all paths in the behaviour graph where the number of children `x` has been returned by the R_S2_ComputeChildren action, a new component has not been added to the system ((not 'R_S1_addSubcomp.*')*), and after the `foo` method invocation was sent (`Scenario_S1_foo`). Then the formula states that all such paths are followed by a path where the `foo` request has been served `x+1` times, no component has been added in the meanwhile and the root component has not received any other `foo` method invocation. The model-checker answers TRUE. If in the second line we replace `x+1` by `x+2`, which means that we expect the `foo` method invocation to be served at least `x+2` times, the model-checker answers FALSE. The reason is that as there are `x` counted sub-components in the system, the overall number of composite components is `x+1` (the total number of added children plus the root composite). Hence, the `foo` method will be served by at most `x+1` components.

We have also generated and executed the code of the modelled application on GCM/ProActive.

The modelled application involves all the advanced features discussed in this chapter except from the interceptors. It has a non-functional part specified for the composite components, the attribute controllers providing access to the attributes (`myId`, `isBound`, etc.), and multicast interfaces in the composite components which are reconfigured by the component-controllers. The properties we proved by model-checking demonstrate that the modelled application performs the reconfiguration correctly: a new component can be added to different parent composites, after a new component has been added, it can be accessed. In addition, we have proven that the attribute of the root component storing the total number of components added in the system, indeed, has the correct value. Generating the behavioural model from VerCors did not require any additional effort except from the specification of the scenario for the sub-components, i.e. modelling of one additional state machine.

6.5.2 Springoo

In this section we present an example of an application using interceptors which were not included in the previous use-case. We assisted our colleagues from Telecom ParisTech in modelling and generating of their application called Springoo in VerCors. The details of the use-case example were published in [89]. Its component diagram is illustrated in Figure 6.21.

Springoo is a web application that conforms to the three-tier Java Enterprise Edition (JEE) platform architecture, providing typical commercial web services through

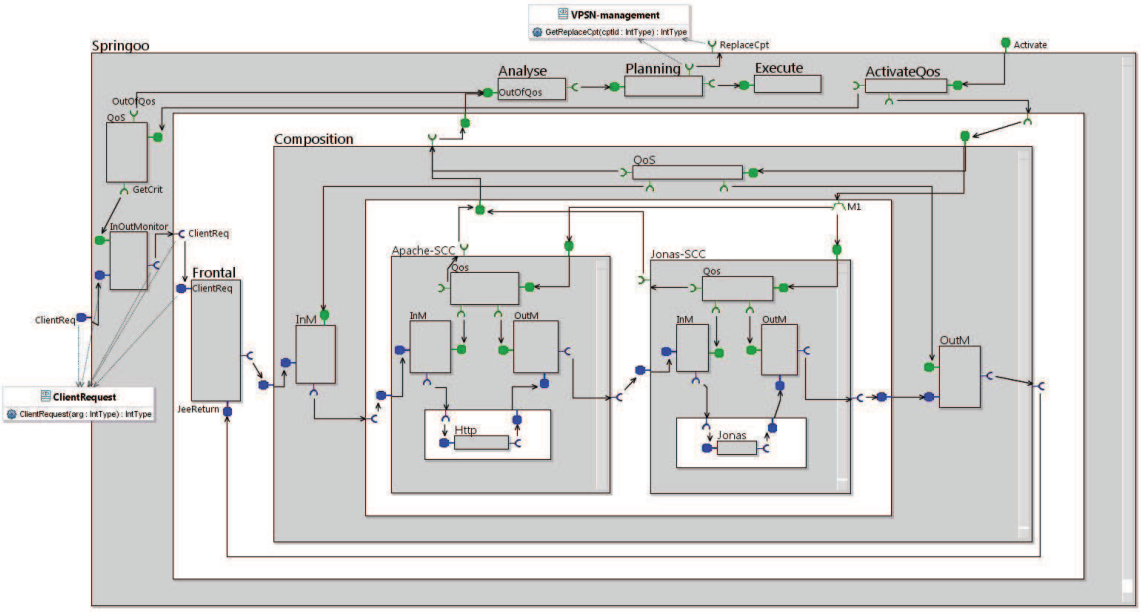


Figure 6.21 – Springoo application modelled in VerCors

an Apache/Jonas/MySQL architecture. The business logic of the application is performed by the `Apache-SCC` and `Jonas-SCC` components. Both components are "self-controlled" in the sense that all functional requests going to and from these components are monitored by the interceptors `InM` and `OutM` located in their membranes. The information gathered by the monitors is then forwarded to the `Qos` component which analyses it and provides it to an external component through its server interface. The behaviour of the `Apache-SCC` and `Jonas-SCC` is additionally controlled by the membrane of the encompassing composite `Composite`. Its `Qos` component-controller gathers and analyses the information from the `Qos` components of `Apache-SCC` and `Jonas-SCC`, and from the interceptors in the membrane of `Composite`. Finally, a so-called MAPE sequence of components in the membrane of the root composite `Springoo` uses the information gathered by the `Qos` components in order to the reconfigure system so that it ensures the desired properties. More precisely, the `InOutMonitor` intercepts the incoming requests to the application. The `Qos` component, as usual, takes the gathered metrics, processes them, and provides them to the `Analyser`. The latter additionally requests the information from the `Composite` and analyses it. The analysis result is used by the `Planner` in order to plan system reconfiguration if it is necessary. Finally, the reconfiguration is performed by the `Executor` component. The reconfiguration was not modelled in VerCors because for this, the authors used GCM-script [90] instead of the GCM/ProActive API. GCM-script is a high-level scripting language for the reconfiguration, it is not yet supported by VerCors.

The behaviour of the system was not model-checked in VerCors mainly because Springoo takes the reconfiguration decisions based on the metrics related to the time, and the timed systems cannot be analysed with our platform. Still, we validated its static correctness with respect to the properties formalised in Chapter 4 and generated the ADL file and the skeletons of the Java classes and interfaces including the component-controllers and intercepts. The authors integrated their implementation of the business logic in the generated code and ran it on GCM/ProActive.

Even though we did not use VerCors to model-check functional properties of the Sprinoo use-case, modelling this example was useful both for us and for our colleagues from Paris for several reasons. First, it allowed us to evaluate the plug-in performing the static correctness validation discussed in Chapter 4. Second, we received a feedback from the developers from outside of our laboratory on the usage of the platform: they explained us what was easy and what was difficult to model in their application. Based on their feedback ,we significantly enhanced the graphical user interface. Finally, thanks to using VerCors, our colleagues managed to model their application and ensure that the design is statically correct (the components are properly encapsulated, the separation of concerns is respected, etc.), they obtained a set of diagrams illustrating the modelled architecture, and they generated automatically the skeleton of the executable code which they used after in order to implement their application.

6.6 Discussion

In this chapter we presented how component-based applications with advanced features can be graphically modelled, verified, and generated in VerCors. Such features include non-functional components, attribute controllers, and multicast interfaces reconfigurable by the non-functional components.

As it was discussed in Section 3.2, we believe that the advantage of our approach to modelling of the non-functional aspect of an application is the strong separation of concern: the user can define separately the business logic and the control part of a system. The designed membrane of a composite component can be transformed into a set of pNets which are then included in the automaton encoding the behaviour of the encompassing component. The rules for generating and synchronising pNets of the non-functional elements are very similar to the ones dealing with the functional aspect. Still, we have to take into account a new type of communications - the interactions between the components in the membrane and in the content of a composite component. In this thesis we consider verification of the non-functional part of the

composite components, and at the next steps our framework should be extended to deal with the membrane of the primitives.

Second, we allow the user to model attributes of primitive components and we assist in implementing the behaviour of the attribute controllers to access and to modify their values. A small technical detail which could be improved in the way the attributes are specified in the current version of VerCors, is that the user has to declare manually the set- and get-methods. Their signatures could be generated automatically. Still, we automatise all the procedures at all other steps: we construct a pLTS encoding a controller for each attribute and generate the Java code of the set- and get-methods based on a standard template.

We also presented an approach to model, verify, and generate reconfigurable multicast interfaces. While defining the corresponding graphical formalism, we had several ideas on how the bindings outgoing from the multicast interfaces should be identified and referenced from the reconfiguration instruction. Our first idea was to keep the graphical formalism unchanged and to refer to a binding by its source and target interfaces. Soon, we realised that the drawback of such approach is that each time we model a reconfiguration, the identifier of the modified binding must be hard-coded. This is why we decided to introduce binding indices so that the reference to the reconfigured binding can be parameterised as we did for our composite pattern example. We have also formalised and partially implemented the generation of pNets encoding multicast interfaces. As the next step, we plan to finish implementing the construction of pNets for primitive components with multicast interfaces in VerCors. We believe that we already have most of the necessary structures, we only need to encode the generation of group proxies for primitives and the synchronisation vectors which are already formalised.

The multicast interfaces which can be modelled and verified in the current version of VerCors implement the broadcast policy. We plan to encode the remaining three policies provided by GCM/ProActive (i.e. unicast, round-robin, and scatter) in pNets and to generate the corresponding executable code so that the user can choose the policy for each interface separately. The idea for the pNet generation is to construct an additional pLTS which would intercept a call going from a multicast interface a modify it with respect to the chosen policy. The pLTS should be included in the pNets that owns the multicast. Recall that for the broadcast policy a call going from a multicast interface includes the current target group G and the list of the arguments for the method invocation *arg*. The additional pLTS should analyse G and arg and use them to transform the request according to the chosen policy. For instance, for the unicast policy it should randomly select one member of G and one member of arg,

and then send the method invocation with the computed arguments. Moreover, we would like to allow the user to define graphically a custom policy in a form of a state machine, and then to translate it automatically into a pLTS and into implementation code.

Concerning the gathercast interfaces, they can be modelled in VerCors. As the next step, we should also formalise and generate the pNets for the components with gathercast interfaces and produce their executable code. We believe that in order to encode gathercast interfaces with pNets we should follow the same approach as for the multicast interfaces. The proxy and proxy manager should maintain a group variable which can be modified depending on the reconfiguration requests. While the proxy of a multicast emits one request, gathers several results, and forwards them to the caller, the proxy of a gathercast should gather several requests, transform them into one request, forward it to the serving method or component, and then return the result to several callers.

Finally, we used the VerCors platform for two complex use-cases which involved the advanced features of GCM components discussed in this chapter. This experience allowed us to evaluate the generator of the input for the model-checker, the plug-in for the verification of the static correctness of the architecture, and the generation or the executable code. We observed that our platform could benefit from optimising the generated input for the model-checker based on the analysis of the behaviour of the input model and of the designed scenario.

This chapter presented the last contribution of this thesis; in the following chapter we discuss to related approaches and position our work with respect to the similar studies.

Chapter 7

Related work

Contents

7.1 The SOFA 2 project 178

7.2 The BIP Component Framework 181

7.3 Rebeca formal modelling language and development tools 183

7.4 ABS 188

7.5 Other frameworks 191

7.5.1 Component models and tools 192

7.5.2 Verification platforms 196

7.6 Summary 200

7.6.1 On the verification tools 200

7.6.2 On the component development frameworks 200

This chapter reviews the state-of-the-art frameworks for modelling and verification of distributed component-based systems. We start by detailed overview of the four specification formalisms and dedicated tools that are the closest to GCM and VerCors. Each of the four frameworks has strong results on the verification aspect and features some of the elements this work focuses on (e.g. components, futures, reconfiguration). For each framework we discuss its advantages and drawbacks compared to VerCors. Then, we briefly review the other development platforms. Finally, we present the verification tools that could be potentially used by VerCors as an alternative to CADP.

7.1 The SOFA 2 project

Among the existing component models and development tools, the approach closest to VerCors/GCM is presented in the SOFA 2 [77] project. It comprises a component model supported by a development framework and a runtime environment. We start this section by describing the SOFA 2 component model and comparing it to GCM. Then, we present the dedicated tools. Finally, we elaborate on the positioning of our work with respect to the SOFA 2 project.

The SOFA 2 component model. The SOFA 2 component model has the same advanced features as the GCM including hierarchical components, the support of dynamic reconfiguration, separation between functional and non-functional concerns. A SOFA-based application represents an assembly of components with their interfaces and a set of connectors used for communication between components.

A SOFA component is modelled by its *frame* and *architecture*. A frame is a black-box view and is associated with a set of provided and required interfaces. An interface is characterised by a signature and a number of properties. The first property, *isCollection* is similar to the cardinality of GCM interfaces and defines how many bindings can be attached to the interface. The second property, *connectionType* defines the reconfiguration pattern [91] applicable to the interface. An architecture is a grey-box view on a SOFA component. An architecture can implement several frames which is similar to a GCM component implementing several interfaces. An architecture can be associated with a set of subcomponents and connections between them. If the set is empty, the architecture refers to a primitive component; otherwise, to a composite one.

The interfaces are connected by *connectors* which are similar to the GCM bindings at the design stage. During program execution the connectors may be implemented by various interaction types depending on the underlying infrastructure [92]. The interaction types include a classic synchronous client-server call, asynchronous message passing and uni- or bidirectional streaming of data. A connector in SOFA has much more features than a GCM binding. For example, it can have a monitoring interceptor, while in GCM interceptors are modelled as components inside a membrane. A connector can slightly change a request in order to solve minor incompatibilities between components which is not possible in GCM. In theory, a GCM interceptor can impact request arguments, but such behaviour cannot be modelled and analysed in the current version of VerCors.

The control part of a SOFA component consists of so-called *microcomponents*. As opposed to the GCM controllers, the controllers in SOFA are flat, they do not

feature connectors, control part or distribution. A SOFA controller could be seen as a simple class implementing an interface. The way to assemble microcomponents is by constructing so-called *delegation chains* which connect several controllers. A microcomponent can also be an interceptor which resembles the way interceptors are implemented in GCM.

On the dynamic reconfiguration side, SOFA 2 supports multiple patterns allowing one to add and remove components and connectors at run-time [1].

The tools. The SOFA 2 component model is supported by a modelling platform SOFA IDE, a set of tools for verification and an execution framework.

SOFA IDE is a model-driven environment for graphical design of the SOFA 2 components. The tool supports UML-based modelling of the components architecture. The executable code for connectors can be automatically generated as described in [93]. The behaviour of the component interfaces can be textually expressed with Behavior Protocols [94]. Then, the behaviour should be manually associated to a component in an XML-based file.

The correctness check for the SOFA applications can be done only at the level of the Behavior Protocols. A Protocol Checker presented in [95] takes a set of protocols as an input, creates a parse tree for each protocol, combines the trees and creates a state space reflecting the parallel composition of the protocols. Based on the obtained model, the Protocol Checker is able to verify the compliance between two protocols and translate the Behavior protocols into an input for the CADP Caesar tool.

In [96] the authors present an approach for checking whether the Java implementation of a component behaviour obeys the specified protocol. For this, the authors combine a slightly modified versions of Java PathFinder (JPF) [97] and the Protocol Checker. JPF is a model-checker of Java byte code implemented as an extension of the Java Virtual Machine (JVM). Unlike the usual JVM, JPF performs all possible executions of a program and builds its state space as a tree-like structure where the branches reflect the interleaving of thread instructions. As in the classical model-checking, the tool traverses the constructed state space to check built-in properties like assertion violation, and deadlocks. Except from that, JPF includes several extensions allowing more complex analysis. Among them, we are interested in the mechanism of *Listeners* which allows the programmer to define an observer that will register particular type of events that occurred in the byte code e.g. thread start, object creation, byte code instruction execution. The result of the monitoring can be used to check custom properties. In [96] the authors configure JPF listeners so that they record all invoke and return instructions for the interface methods of SOFA

component byte code and then notify the Protocol Checker. The Protocol Checker, in its turn, checks whether the instruction is possible with respect to the state space of the specified behaviour protocols. If the execution is not possible, the protocol violation is reported.

The behaviour description can be also traslated into Promela and checked by the Spin [98] model-checker as discussed in [99]. Three types of errors can be detected by the described approach: bad activity (a component emits an event but there is no other component to accept the event), no activity (a deadlock) and divergence (a component behaviour contains a cycle from which there is no way to reach an accepting state).

SOFAnode is a distributed execution environment for the SOFA 2 components implemented as Java classes. The environment consists of a set of component containers called *deployment docks* which can be located on different machines. A deployment dock can execute a component which was assigned to the dock during the deployment. SOFAnode can be managed from an Eclipse-based tool MConsole where the user can start and stop docks as well as visualize system execution.

Positioning. The SOFA 2 framework for modelling and verification of distributed component-based systems is very close to VerCors/GCM. However, there is a number of advantages of the work presented in this thesis. First, in order to master SOFA 2, the user will have to additionally learn the Behavioral Protocols and the XML specification linking a component and its behaviour. In VerCors, both components architecture and behaviour are modelled in integrated graphical editors; the user does not need to know any formalism for behaviour specification except from the UML state machines that are well-known among programmers. Moreover, the behaviour is associated to the components within the graphical designer and the user does not need to be aware of the structure of the generated ADL description.

Second, GCM and VerCors provide more expressiveness for modelling and verification of the non-functional part of an application than SOFA which, to the best of our knowledge, allows neither structured controllers nor specifying relations between controllers.

Finally, we believe that the SOFA framework could benefit from validation of the architecture static constraints formalised in Chapter 4. To the best of our knowledge, the current version of SOFA only supports verification at the level of the Behavior Protocols. Implementation of the architecture static validation could be useful for the programmers who would like to make sure that the components assembly is statically correct before dealing with the Behavioral Protocols.

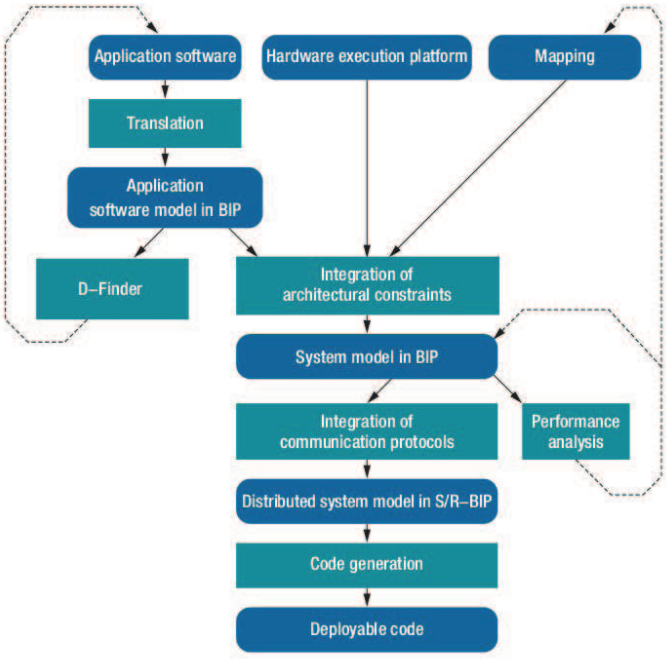


Figure 7.1 – The BIP design flow [2]

At the same time, we should note that SOFA 2 project comprises a number of interesting features which are not included in VerCors. First, it would be useful to allow the user to model and verify various communication styles in VerCors. Another interesting task would be checking whether the generated components implementation obeys the modelled behaviour by adapting the approach presented in [96].

7.2 The BIP Component Framework

BIP (Behavior Interaction Priority) [2] is a framework allowing rigorous model-based design and programming of complex hierarchical component-based systems. BIP is supported by a set of tools for reverse engineering various programming languages into BIP language, a dedicated model-checker and an executable code generator. In this section we first make an overview of the BIP formalism and then present the tools for BIP systems development.

The BIP component model. In BIP [100], a software system is designed as a composition of components connected by possible interactions. The interactions are characterised by priorities which are used for resolving conflicts and defining the scheduling policies. The components expose ports which are used for the communication with the other components.

The interactions in BIP are structured in *connectors* characterised by a set of connected port types and the data variables that can be transferred between the ports. As opposed to the GCM bindings, connectors may be hierarchical, i.e. a connector itself can expose a single port that may be bound to another connector. All the communications done through the connectors are synchronous. An interaction that can be performed via a connector is described by a subset of the connector's ports and can be additionally restricted by a guard expression.

There are two types of components in BIP: *atomic* and *compound* ones. The atomic components are simple entities similar to the GCM primitive components. They expose a set of ports and perform some behaviour modelled as a Petri net or a labelled transition system. A compound component is characterised by a set of contained subcomponents, connectors, priorities and a set of exposed ports that define an interface of the component. The ports exposed by a compound can belong to the connectors or components inside the compound.

The tools. Figure 7.1 illustrates the workflow of a BIP-based application development. First, a software application is translated from the actual code into a BIP model. The framework supports full or partial source to source translation from various input languages such as C+XML, MATLAB/Simulink [101], Lustre and model-based formalisms such as AADL [102]. Alternatively. the user can specify the system directly in BIP textual language or using the EMF framework. In addition, the user should provide the specification of the hardware target platform and a mapping of the atomic components to the processing units. A BIP model of the system is derived based on this information and it takes into account the hardware architecture constraints. Then, the obtained models can be used for verification, simulation and generation of the software code.

The verification of deadlock-freefom and safety properties of BIP-based applications is done by the D-Finder tool [103]. The verification method of D-Finder takes a BIP program constructs a predicate characterising the deadlock states. Then, it computes a local invariant for each atomic component in the system and an invariant on the interactions of components. Finally, the tool checks satisfiability of the conjunction of the obtained invariants in order to prove the deadlock-freedom.

The drawback of D-Finder is that it does not take into account the data transfer on the interactions. A solution to this issue was proposed in [104] where the authors investigate an approach for the verification of the safety property of BIP systems with infinite state-space and data transfer. The authors encode a BIP program into as a symbolic transition system and give it as an input to the nuXmv [105] model-checker.

A completely different verification technique applied to a BIP system in [106] is based on the runtime verification approach. For a given component-based system and a property to be verified, the authors synthesize a monitoring BIP component M and modify the given atomic components so that they communicate with M. M observes the information provided by the other components and emits verdicts regarding the property violation. The authors provide not only formalisation of the approach, but also prove the equivalence of the behaviour of the initial and monitored systems.

Finally, from a BIP model the user can generate C++ code for several execution platforms. There are two main compilation flows offered by the framework: engine-based compilation for non-distributed systems (single-, multi-threaded and real-time implementations) and generation of so-called Send/Receive BIP models for distributed implementation. In the second case, the synchronous communications are transformed into asynchronous message-passing as explained in [107].

Positioning. BIP is a powerful framework offering variety of tools for distributed systems modelling, verification and executable code generation. The clear advantages of the BIP framework compared to VerCors is the ability to model real-time systems, the notion of priorities, the translation from the source code to the BIP models. Moreover, BIP allows expressing complex synchronisation patterns thanks to the hierarchical connectors and to the interactions which involve several ports. In this sense BIP is closer to pNets which describe a system on much lower level than BIP but also allow interactions (i.e. synchronisation vectors) to involve several entities. However, when specifying the interactions between GCM components in VerCors, the user can only define one-to-one, n-to-one, and one-to-n communications. However, to the best of our knowledge, BIP does not allow modelling and verification of reconfigurable systems, does not support the separation of functional and non-functional concerns, and the parameter-parameter passing is not very well integrated with the analysis of component hierarchy.

7.3 Rebeca formal modelling language and development tools

In the domain of actor-based frameworks, the work closest to ours is Rebeca [18] which provides an interpretation of the actor model equipped with a formal semantics[108] and a set of tools for modelling and verification. In this section we, first describe the Rebeca model and compare it to GCM. Then, we discuss the development tools. Finally, we introduce one of the key research directions presented by the authors of

Rebeca - the techniques which allow avoiding state explosion when model-checking actor-based applications; we also discuss here how those techniques could be applied to VerCors/GCM.

Rebeca modelling language. A Rebeca-based application is a composition of actors communicating by asynchronous message-passing. An actor in Rebeca is called *rebec*. A rebec is executed in its own thread and possesses a FIFO queue which stores the incoming messages. A rebec implements a *reactive class* which comprises three types of definitions: known rebecs, state variables and message servers. A rebec can communicate only with the known rebecs which implies that in order to send a message, a rebec should know the receiver. Message servers are used to process messages. A message server has a name, a set of input parameters (possibly empty), and a body implementing the behaviour. The behaviour is composed of a sequence of statements representing assignments, sending messages, choices and creation of new rebecs. Every rebec has a special message server dedicated to its initialisation. A rebec manipulates variables of two kinds: the data variables which model data, and the rebec variables which correspond to the indices in the list of known rebecs. The former can be used for specifying the receiver of a sent message. This allows one to change dynamically the topology of a model.

In terms of the GCM model, a rebec can be seen as a primitive component which also has a FIFO queue and is also executed in its own thread. A reactive class could be seen as a class implementing a GCM component with attributes modelling state variables. The server methods of GCM components are similar to the message servers in Rebeca. Several key differences between Rebeca and GCM should be highlighted here.

First, thanks to the notion of interfaces, a GCM component does not need to store any information about the other components in order to communicate with them. As a result, it becomes easier to plug a GCM component in different contexts, because for this purpose the programmer only needs to change a binding but not the implementation of the component.

Second, the GCM components are hierarchical while Rebeca actors are flat. The studies on assembling rebecas in composite components were presented in [109] and [110]. The authors introduced one-level hierarchy aiming at facilitating verification and avoiding state explosion during model-checking. On the contrary, a GCM-based application is not limited in the levels of hierarchy. This provides the users with an expressive mechanism for describing components structure at different levels of abstraction. Additionally, thanks to the structured components, the GCM model

imposes separation between functional and non-functional parts of the application.

Rebeca and GCM rely on different communication paradigms. As opposed to GCM, Rebeca actors do not implement futures mechanism. Instead, rebecs can communicate either by asynchronous message-passing, or via completely synchronous rendez-vous mechanism. This makes the generation of Rebeca behavioural model easier.

The tools. Rebeca is supported by a set of tools for modelling, simulation and verification. The modelling environment closest to VerCors is ReUML-Designer [111] with the following workflow. First, the developer models an actor-based application in a graphical editor based on UML profiles. Rebecs are modelled as UML classes which is similar to the way GCM primitive components are specified in VerCors. For the behaviour specification, ReUML-Designer relies on the UML sequence diagrams providing better global view on the messages exchange between actors than the UML state machines. Rebeca code can be generated from the graphical model.

Rebeca is supported by a set of formal verification frameworks. Modere is a tool for model-checking Rebeca programs. Since it was designed specially for Rebeca, it includes state-space reduction techniques which are based on the Rebeca computation model. Modere is integrated in the Afra [112] platform which takes as an input SystemC code, translates it to Rebeca, and verifies LTL and CTL properties of the obtained application. Additionally a set of tools were developed for the translation of Rebeca programs into input languages for various model-checkers such as Promela, the input language of Spin and SMV - the input language of NuSMV model-checker. An approach and a tool for distributed model-checking of Rebeca actors is presented in [113].

The timed Rebeca actors can be translate to the Erlang executable code and Maude.

The techniques to avoid state explosion. Depending on the target model-checker, a number of techniques can be applied to a Rebeca program in order to avoid the state explosion issue. First, since Rebeca relies on finite-space model-checkers, the queues of the actors should have a user-defined finite size. Additionally, the user should abstract the data domains. This aspect is similar in VerCors.

Second, compositional [114] and modular[109] verification techniques can be applied to the Rebeca programs. In compositional verification, an initially designed system is decomposed and local properties are checked on the obtained subsystems. Then, it should be proved that the conjunction of the local properties implies the

global property. In order to apply compositional verification to a Rebeca program, the user should assemble rebecs in components. Every component is assumed to be executed in an environment which sends arbitrary messages. The properties proved for the components by model-checking are then used to ensure the global properties of the system.

As opposed to Rebeca, VerCors does not do compositional verification. Doing the compositional verification of the models produced by VerCors would require significant effort due to the hierarchical structure which makes it difficult to decompose the global property into properties of components located at different levels of hierarchy.

The modular verification technique is slightly different from the compositional approach. It is applied to verify the re-usable subsystems and to prove their specifications. The application of the modular verification to the models produced by VerCors is straightforward because VerCors relies on the bottom-up approach for building system state-space. More precisely, the tool recursively constructs the behaviour model of the components starting from the primitives at the lowest levels of hierarchy and then assembles them into the behaviour of the encompassing containers. As a result, the user has a behaviour graph for every component in the system not impacted by the behaviour of its container and is able to verify properties for any component in the hierarchy as for an open system.

Third, the behaviour of the environment can be modelled for the Rebeca applications. For this purpose, the user should specify a set of requests (possibly unbounded) to be processed by rebecs at any time, in an interleaving with processing requests in the queues. Such approach is not applicable to VerCors, because only the root component can get requests directly from the environment. Instead, we offer the user to design the global behaviour of the environment as any kind of possibly unbounded and non-deterministic scenario modelled as a state-machine.

Finally, the symmetry and partial order reduction techniques can be applied to Rebeca as discussed in [115]. The symmetry reduction technique consists in identifying classes of sub-graphs with identical structure in the state-space and constructing only one sub-graph for each class. The state-space of a Rebeca program can be seen as a graph where a state is a combination of the local states of the rebecs involved in the system, and a transition is labelled by an enabled action of one of the rebecs. A `local state` of a rebec is defined by the values of its local variables and the status of its queue. An `enabled action` of a rebec is a service of the first message in its queue by its reactive class. The service of a message is considered to be atomic. A transition $s \xrightarrow{\alpha_i} t$ occurs if an action $alpha_i$ enabled in a rebec r_i and it changes the state of the system from s to t. Thanks to the fact that several rebecs can be instantiated from

the same reactive class (hence, they have the same internal behaviour), and that the list of the known rebecs is defined, the symmetry among the rebecs can be detected automatically, and the state-space can be reduced.

In fact, the state-space of a Rebeca program is significantly reduced thanks to the fact that service of a message is considered as an atomic action. Hence, there is no need to consider the interleaving of particular statements executed by different rebecs in parallel. In VerCors we partially do such kind of optimisation when hiding the internal computations inside sub-components but we cannot say it s as efficient as the reduction applied to Rebeca programs.

The concurrency in the Rebeca applications is modelled by the interleaving of actions of different rebecs. According to the partial order reduction approach, the execution of some actions can be postponed to the following states, thus, there is no need to explore all possible interleaving of actions. The partial order reduction techniques applied to Rebeca programs rely on the notion of safe actions. An action is safe if it does not influence satisfiability of the verified property and if it is independent from all actions of the other processes. Two actions are independent if (1) the execution of one of them does not prevent from executing the other one, and (2) no matter in which order the actions are executed, their processing will result in the same state. Since the execution of a safe action does not disable the other actions enabled in a given state, those actions can be postponed to the next state, and the state-space can be reduced. Safe actions in Rebeca programs can be detected statically.

For this kind of approaches VerCors relies on the techniques embedded in the CADP toolbox [22].

Positioning. To conclude, even though Rebeca supported by the dedicated development tools and VerCors/GCM share a lot of common features, the frameworks should be applied to to different domains of applications. Rebeca is targeted to the flat asynchronous actor-based systems where new participants can be added easily. VerCors would be better applicable to software with complex hierarchical structure that relies on request/reply with futures mechanism. It should be mentioned that the current research on Rebeca covers timed and probabilistic systems which are not considered by the work presented in this thesis. Another advantage of Rebeca is the strong mechanism for the state-space reduction based on the computational model.

7.4 The Abstract Behavior Specification language and tools

The abstract behavior specification language (ABS) [116] is a rich Java-like specification language for modelling concurrent and distributed object-oriented programs. ABS features formal semantics and is supported by a deductive verification toolset KeY-ABS [117] which allows verification of unbounded systems. A verified ABS specification can be then translated into several executable back-end languages. In this section, we first provide an overview of the core ABS language and its extension. Then, we compare ABS to GCM and discuss the possibility of using VerCors for modelling and verification of the ABS programs. Finally, we make an overview of the development tools dedicated to ABS.

The core ABS language. An ABS program represents a set of objects which can store data, serve methods and implement interfaces. Objects can invoke methods on each other but the data of another object cannot be accessed directly. The ABS objects are assembled into COGs (Concurrent Object Groups) which represent the units of distribution. A COG can have several threads, but only one thread (and one object) within a COG can be active at each time while all processes executed in the other objects are suspended. The scheduling is non-deterministic. The ABS objects within one COG can communicate either synchronously or asynchronously, while objects from two different COGs must communicate asynchronously. The asynchronous communications in ABS rely on request/reply with futures paradigm.

In [118] the authors demonstrated and formally proved that a COG can be translated into a multi-threaded active object. We could try to adapt the result of their work and to model ABS COGs in VerCors with primitive components as the relation between actiove objects and GCM components is discussed in details in [26]. We could also partially re-use the results of [119] where the authors explain how the ABS objects can be represented as components. However, making primitives multi-threaded would require effort on the modification of the body policy.

The ABS extensions. The ABS language was extended with a set of features providing the developers with additional modelling elements including component-based modelling, I/O specification, and design and analysis of real-time systems.

In [119] the authors explore the similarities between the ABS objects and components and extend the ABS language with the minimum elements required for the component-based modelling, trying to keep the extended semantics as close as pos-

sible to the core ABS. An ABS object can be seen as a component and its methods represent input ports. The following elements were introduced additionally in ABS. First, every object that represents a component can be either in a safe or not in a safe state. Second, an object stores the information about its output ports (similar to the GCM client interfaces) which can be modified only if the object is in a safe state. Third, a method executed by an object can be annotated as critical. An object executing a critical method is not in a safe state. The next introduced element is a primitive which allows waiting for an object to return to a safe state in order to modify a port. Finally, the notion of locations was introduced in order to enable components hierarchy; the location of a component can be changed at runtime.

Another interesting extension of the ABS language is the mechanism for I/O explained in [120]. In general, modelling and verification of I/O is not supported by ABS because it relies on the underlying platform. The developer can use a so-called foreign-language interface in order to print something from an ABS program or to connect an ABS program to the legacy code in another language. The mechanism is illustrated in [120] by connecting Java code and ABS. For a Java class used by an ABS application, the programmer should additionally define a default ABS implementation of all interface methods which will be used for simulation of the ABS code without Java. It would be interesting to investigate whether a similar approach could be applied in VerCors. For example, a UML state machine specifying a server method could include calls to Java classes defined aside.

Modelling of real-time systems is possible with the Real-Time version of ABS. In one of the latest works [121], the authors formally model the Hadoop YARN clusters [122] and validate their approach through comparison of the model-based analysis and the actual performance of the cluster.

The tools. An ABS program can be created in an Eclipse plugin text editor [123]. Mastering the tool and the ABS language should be easy for the developer who is familiar with object-oriented programming, because the syntax and structure of ABS resemble Java. The execution of an ABS program can be visualized with an ABS debugger based on the Eclipse debugger. Additionally, UML sequence diagrams illustrating the lifeline of each COG can be automatically generated. The diagrams show a program execution and are updated after each debugger step.

An ABS program can be verified by the deductive verification tool KeY-ABS which is built on top of the KeY interactive theorem prover [124]. Figure 7.2 illustrates the workflow of KeY-ABS. The tool takes as an input an ABS program and a .key file containing invariants, functions, predicates and specific rules required for verification.

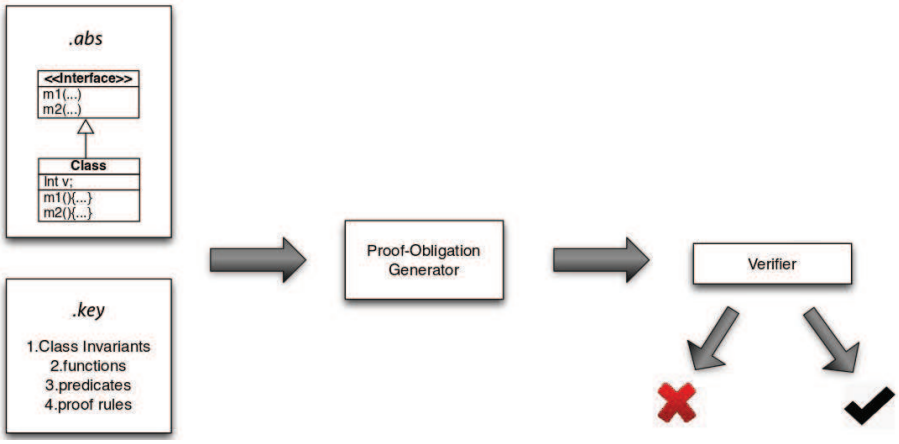


Figure 7.2 – Verification workflow of KeY-ABS[117]

Then, the user is asked which method should be verified. The input is translated into a specific type of formula which is then checked by the theorem-prover. A positive answer from the prover means that the method satisfies the invariant. If the formula cannot be proved, KeY-ABS asks the programmer for additional instructions. 90% of the proof is done automatically.

VerCors and KeY-ABS follow different verification approaches. The current version of VerCors is sufficient for model-checking functional properties of an abstracted model with finite state-space. In contrast, KeY-ABS proves preservation of invariants on unbounded systems. On the other hand, the advantage of model-checking is that it avoid writing invariants: instead, only program properties should be specified by the user.

A deadlock in the ABS programs can be automatically detected by the DF4ABS [125] framework. It extracts abstract behavioural descriptions of the methods of the input program. Such a description is called a *contract*, and it includes the method invocations and the dependencies between various statements. Then, the contracts are analysed in order to detect deadlocks.

COSTABS [126] is a tool able to predict the cost of ABS programs. The analysis is organised in three steps. At the first step, the *cost model* should be chosen; it defines the consumption of which resources should be analysed. Five options are offered by the framework: the *steps cost model* allows predicting the number of instructions executed by the program; the *memory cost model* predicts the amount of memory required for various instructions; the *objects cost model* counts the number of objects involved in the system; the *task-level cost model* estimates the number of tasks processed during the execution. The last cost model - *termination model* - does not count any resources but checks that all loops in the program terminate. The second step

of the analysis is a so-called "size analysis" which estimates how the size of the data changes along the program. The analysis works well for sequential fragments of code, but the precision can be lost significantly for the programs involving concurrency. At the third step, the results of the size analysis and the chosen cost model are used in order to estimate the cost of the program.

Several back-ends exist for translating ABS models into executable languages. In [118] the authors present a translator of ABS programs into Java code. The resulting program is based on multithreaded active objects and can be executed in a distributed manner on the ProActive platform. The authors formally prove the correctness of the translation. A back-end for Haskell code generation is presented in [127].

Positioning. Both ABS and VerCors/GCM target analysis of distributed object-oriented applications which rely on request/reply by futures. Both frameworks feature specification formalisms which can be easily learned by non-experts in formal methods, modules generating executable code and verification capabilities. We can, nevertheless, highlight several cases in which VerCors/GCM would be better applicable than ABS. First, VerCors/GCM provides better modelling capabilities for the systems with complex hierarchical structure because it allows modelling and visualizing components hierarchy. Second, for those systems where the separation between business logic and control part is crucial, VerCors is able to ensure this property. Third, to the best of our knowledge, modelling group communications is not possible in ABS. Finally, VerCors should be used by the programmers who opt for model-checking but not invariant specification. One of the advantage of model-checking used in VerCors is that it does not require as much guidelines from the user as KeY-ABS needs for the verification of invariants. On the other hand, one could notice that compared to DF4ABS, our tool requires manual specification of the deadlock-freedom formula while DF4ABS can check everything fully automatically. In fact, constructing the deadlock-freedom formula in MCL is quite simple, and we plan to assist in the specification of this and the other formulas in VerCors as it will be discussed in the next chapter.

7.5 Other frameworks

In this section we discuss several other frameworks for modelling and verification of component-based systems. We start by the component models and dedicated development tools. Then, we make an overview of the verification tools that could be potentially used by VerCors as an alternative to CADP.

7.5.1 Component models and tools

Palladio. Another framework for component-based systems modelling is provided in the Palladio [128] project which presents the Palladio Component Model (PCM) and a development environment. PCM allows specification of primitive and composite components, their interfaces with method signatures and connectors. In addition, the information about resource consumption should be specified for the served methods. The PCM is supported by an Eclipse-based development environment where the user can model a system and simulate its execution. Alternatively, the Palladio models can be automatically extracted from the system implementation provided in Java, C or C++. Based on the input model and the results of simulation, the tool is able to predict a number of system performance metrics such as the response time, throughput and resource consumption.

While both Palladio and VerCors are used for system analysis at the early design stage, the two frameworks have different objectives. Palladio targets mainly performance analysis while VerCors is able to verify functional properties of a system by applying formal verification techniques. We should also highlight here several limitations of the Palladio framework. First, the results of performance prediction for concurrent systems significantly differ from the actual performance as mentioned in [128]. Another limitation is that Palladio does not support modelling of reconfigurable systems. While we did not address the performance here, we showed that we are able to handle certain forms of reconfiguration in VerCors.

DEECo. Another framework developed by the authors of the SOFA component model is the DEECo (for Dependable Ensembles of Emerging Components) component model [129]. It is used for the implementation of large-scale distributed ensembles of components which cooperate together in order to achieve a common objective. An application in DEECo is constructed from components assembled into ensembles. A component features a hierarchical data structure called *local knowledge* which is a mapping from the data variables to their values. The knowledge of a component can be exposed to the other components through its interfaces. A component has a set of *processes* which are basically tasks manipulating the knowledge. A process can be scheduled so that it is either executed periodically or triggered each time a particular condition is met. The DEECo components are assembled into flat ensembles where one component plays the role of a *coordinator* and the others are the *members*. When a programmer describes an ensemble, he should specify the membership conditions, i.e. the interfaces that should be exposed by the coordinator and the members of the ensemble. The ensembles are formed dynamically from the set of components that

satisfy the defined conditions. The components do not explicitly send messages to each other, instead they exchange their knowledge. The ensemble specification defines how the coordinator exchanges the knowledge with the members, thus defining an abstraction for the one-to-many communications. The DEECo component model features the formal semantic including the time aspect and presented in [130].

The DEEco component model is implemented in the jDEECo framework [131] which provides the Java libraries necessary for the development, deployment, and execution of the DEECo ensembles. The framework is integrated with the Java PathFinder model-checker for the verification of the designed applications.

One of the advantages of the DEECo framework is that thanks to the notion of ensembles and membership conditions it allows implementing much more dynamic applications than GCM/ProActive systems. On the other hand, the dynamically evolving structure of such applications makes it more challenging to reason about their properties formally. To the best of our knowledge, neither modelling of future-based communications, nor design of hierarchical applications is possible with DEECo as opposed to the GCM/VerCors framework.

Helena. Helena[132] is another framework for modelling highly dynamic ensembles of autonomic distributed components. In addition to the components and ensembles, the Helena approach relies on the notions of *roles*. A Helena *component* is characterised by a set of attributes and operations that can be served. It can be said that a component "fulfils" particular *roles* which describe the functionalities of a component in terms of attributes and operations. There can be several components able to fulfil the same role. A set of roles connected by so-called *role connectors* form a Helena *ensemble*. The components filling the roles of an ensemble collaborate in order to achieve a particular goal. The ensemble structure specifies the size of the queue for the input messages of each role. The behaviour of a role can be modelled as an LTS whose transitions are labelled by either message sending or message reception over a role connector. The LTSs of roles behaviour are assembled into an ensemble behaviour automaton where each state represents an ensemble state and transitions are labelled by message labels. The Helena roles can communicate both by synchronous and asynchronous message-passing. The communication style depends on the role queue size specified by the user. Thanks to the role-based modelling, the Helena framework allows one to design highly dynamic heterogeneous systems with rich communication patterns. On the other hand, the variety of components that which can be dynamically included in the ensembles, makes it more challenging to reason about the designed applications.

HelenaText[133] is an Eclipse-based text editor for Helena ensembles specification. The editor features syntax highlighting and content assistance. A HelenaText file can be automatically translated into the implementation code that can be executed on the jHelena[134] platform. A model specified in HelenaText can be transformed into Promela [135] and model-checked with Spin as demonstrated in [136]. At this point, the programmer chooses the communication paradigm for the translated processes (either asynchronous or synchronous message-passing) by specifying the input queue size for the roles. The typical property verified by the authors states that in a modelled peer-to-peer system supporting distributed storage of files the requester will always receive the requested file. The peers are connected in a non-parameterised ring topology.

While both Helena and VerCors/GCM are used for modelling and implementation of distributed systems, they should be applied for different types of applications. To the best of our knowledge, neither hierarchical components, nor futures mechanism can be modelled, verified or executed with Helena. On the contrary, more custom patterns like publish-subscribe are easy to express in Helena and more difficult to encode in VerCors.

Credo, Reo, and Creol. Credo[137] is a toolsuite for modelling and analysis of highly reconfigurable asynchronous distributed systems. The design process comprises two main steps. First, the high-level application dataflow is designed with Reo[138] which allows specifying a reconfigurable network of components where only a *facade* of a component is visible. A facade consists of a set of communication points called *ports*, event declarations, and an abstract behaviour modelled by constraint automata [139] specifying the order of raised events and port operations. The ports of components are connected into a *network*. A *networkmanager* defines how the events raised by the components are handled by the network. The ports are synchronised with an automaton modelling the communication inside the network. Overall, this specifies the inter-component communications.

At the second step, the intra-component behaviour is designed with an object-oriented executable modelling language Creol [140]. In fact, Creol programming model is very close to ABS. Creol objects rely on asynchronous method calls and processor release points, feature an execution thread, a set of attributes and methods to be served. The communication paradigm in Creol is based on request/reply by futures mechanism

Credo, is able to check the conformance of the models specified in Reo and Creol. The tool transforms a facade with its behaviour into an intermediate abstract be-

haviour specification [141] from which a Creol model is derived. The transformation result is executed together with the original component specification in Creol in a special version of Maude[142] configured for the testing purpose. In addition, Credo can check conformance and between the Creol model and the actual implementation in C by testing. Furthermore, a Creol skeleton of the application can be automatically derived from the network specification. The network specification can be checked for absence of deadlocks.

Apart from the tools provided in Credo, the Creol language is supported by an Eclipse-based modelling environment which encompasses a type-checker and a simulation environment. Creol programs can be executed using Maude which additionally provides means for model-checking of infinite-state models.

Credo is a powerful framework for modelling and analysis of distributed systems where a reconfigurable network of processes is modelled independently from the actual processes implementation. However, in order to master Credo, the user will have to learn Reo, Maude and Creol while in VerCors the system specification is based on the UML models well-known among the programmers. Additionally, VerCors is able to generate executable Java code of a system.

CORBA Component Model and Cadena. The CORBA Component Model (CCM) [3] was designed by the Object Management Group for modelling and implementation of distributed components. Components in CORBA have provided and required ports, publish events on the ports, store attributes. CCM allows components to be dynamically created, connected, and disconnected.

CCM is supported by a variety of development frameworks. The one closest to VerCors is presented in the Cadena [143] project. Cadena is implemented as an Eclipse plugin and supports the following development workflow. First, the user should load a library of predefined domain-specific components and define his own project-specific components with their dependencies. Second, the user can specify the non-functional information such as the distribution related data. Finally, Cadena uses Bogor [144] to generate system state-space and model-check the global system properties. Bogor is able to check deadlock-freedom and safety properties expressed as assertions and invariants. Alternatively, the conceptual model can be transformed into an input for the dSpin[145] model-checker able to verify LTL formulas.

Overall, Cadena includes a number of features that are not supported by VerCors such as real-time systems analysis, modelling of sensor networks, integration of domain-specific libraries of components. It would be very useful to implement the latter functionality in VerCors. On the negative side, Cadena does not support

modelling hierarchical systems and request/reply by futures.

Omega2. A tool for modelling and verification of timed component-based systems which completely relies on the UML formalism is presented in the OMEGA project. An application architecture can be modelled with hierarchical UML composite structures and classes, the behaviour on the components' ports can be designed as UML state machines. An OMEGA component can be either executed in its own thread or share a thread with the other components. The components can communicate either synchronously or asynchronously. The specified models can be translated into an input for the IF [146] validation environment. From the given model, IF generates Promela code, on which one can model-check LTL formulas in Spin. The tool is also linked to CADP Evaluator for checking MCL formulas and Kronos[147] for verifying TCTL formulas on timed automata.

Again, VerCors does not target timed systems. Instead, as opposed to OMEGA, VerCors provides capabilities for design and verification of system reconfiguration and ensures separation between the business logic and the control part of an application.

A graphical designer and a code generator for CADP. Except from the model-checking part, CADP is also equipped with a front-end designer ELOTON[148] and a code generation module [149], i.e. CADP could be used for modelling, verification and executable code generation. ELOTON aims at helping the users write LOTOS formal specification. It comprises a text editor featuring text highlighting, auto-completion, error marking, and a graphical visualizer of the specified graphs. The code generator translates a distributed system specification from LotosNT to the executable C code. The generated program can be connected to the external code via user-modifiable C-functions. We believe that the programmers who are not familiar with LOTOS should benefit from using VerCors because it allows specifying software on higher level than ELOTON and is better adapted to the user who is not an expert in formal methods. For example VerCors allows one to define system behaviour as a set of UML state-machines while in ELOTON it is specified textually in LOTOS. Regarding the code generator, it would be interesting to investigate whether we could use the approach presented in [149] to connect the executable code produced by VerCors to the external functions.

7.5.2 Verification platforms

In this section we discuss some of the verification platforms that could be potentially used by VerCors as an alternative to CADP. We analyse here three verification toolsets

that are used by many component development frameworks in the literature, namely: Spin, NuSMV, and Maude.

Spin. Spin[98] (for Simple Promela INterpreter) is an open-source model-checker for multi-threaded software systems specified in the high-level language called Promela (a Process Meta Language)[150]. It is linked to a tool which is able to extract the Promela models from the implementation level C code based on some guidelines from the user[151]. Spin can benefit from exploiting several cores for model-checking as presented in [152]. The tool supports verification of systems with dynamically growing number of processes (a Promela process can instantiate other processes at run-time). Besides model-checking, the tool provides simulation capabilities which allow for the early system prototyping.

Spin supports verification of liveness and safety properties as well as deadlock-freedom, the logical consistency of the specification and absence of unexecutable code for the finite systems. The verified property can be expressed as an LTL formula, as a process invariant (based on assertions) or a Büchi automaton.

Given system specification in Promela and a property to be checked, Spin generates the C code of the verifier which implies that a new verifyer will be constructed for each property. In order to tackle the state explosion, Spin performs on-the-fly verification, which avoids construction of the global state graph. Additionally, Spin relies on the partial order reduction techniques [153] in order to reduce the number of states considered during model-checking depending on the verified formula.

Spin is a popular model-checker which has been proven to be efficient by multiple examples from industry and academia [154, 155, 156]. Seeing, the successful results of applying Spin to large systems, it would be interesting to investigate whether the GCM components could be translated into Promela and model-checked with Spin. The main challenge here would be translating the components hierarchy, because Promela supports only specification of flat systems. Such issue does not appear while using CADP because CADP includes the mechanisms for modelling hierarchical structures. This is one of the reasons why we prefer using CADP but not Spin for our first verification experiments in VerCors. Another reason is that unlike Spin, CADP uses branching bisimulation for state-space reduction which are efficient for the GCM systems because many processes in various components have bisimular behaviour (e.g. the attribute controllers, proxy managers).

NuSMV. NuSMV[157] is a model-checker supporting symbolic verification of synchronous and asynchronous systems. The tool takes a set of hierarchical finite state

machines expressed in the SMV language and checks system properties written in CTL or LTL.

Each state machine has local variables and could be seen as a reusable module which can be instantiated multiple times. The modules are composed together (either synchronously or asynchronously) into a so-called parent module. The latter has access to the local variables of its sub-modules and can pass any of them by reference to another sub-module as well as its own variables. Thus, the state machines can share variables. The composition style - synchronous or asynchronous - is defined by the parent module. In the first case all sub-modules move at each step simultaneously. In the case of asynchronous composition, only one randomly chosen sub-module proceeds at each step. In order to ensure fair interleaving, the programmer can define a fairness constraint saying that each asynchronously composed module will execute infinitely often. Every SMV program should have the root module `main`. A state machine can express deterministic or non-deterministic behaviour.

Similar to CADP, for each verified property, NuSMV answers whether it holds on the given system and provides a counterexample if the property is not satisfied. The verification engine is an implementation of the symbolic model-checking [158] which combines BDD-based (Binary Decision Diagrams) and SAT-based techniques [159] for bounded model-checking. Before the verification is started, the input system is preprocessed in several steps including flattening and applying reduction techniques [160]. Then, the user chooses which verification mechanism should be triggered: either the symbolic model-checking or SAT-based verification. In the first case, the model-checker builds a BDD-based representation of the input model and checks LTL or CTL formulas on it. In the second case, NuSMV needs to be linked to an external SAT-solver that could be either MiniSat[161] or Zchaff[162]. The user should provide the length of a counterexample, and NuSMV translates the given LTL model-checking formula into a SAT problem.

The latest version of NuSMV is distributed as an Eclipse plug-in and is an open-source project. It is used as a back-end model-checker by multiple projects aiming at rigorous development of embedded systems [163, 164, 165]. It would be interesting to apply the symbolic model-checking techniques provided by NuSMV to the GCM components modelled in VerCors. For this, we would first have to translate the pLTSs into the finite state machines. The main challenge here would be to deal with the synchronisation mechanism of SMV which is different from the one of LOTOS and EXP as discussed in [166]. In fact, some studies about the symbolic reasoning on pNets have already been started in [31], but they are still at a theoretical stage.

Maude. Maude [142] is a rich framework based on the rewriting logic for rigorous development of various application types including concurrent and distributed object-oriented systems. The platform comprises an executable modelling language and a set of analysis tools.

The Maude language is based on the rewriting logic which has an underlying equational logic as a parameter. The basic elements of a Maude program are *equations* and *rules* that have rewriting semantics meaning that during the program execution the left-hand side pattern will be replaced by an instance matching the right-hand side pattern. Equations are used as simplification patterns while rules can be seen as transition rules for a possibly concurrent system and a way to express the interaction between different processes. Maude supports user-defined syntax of operators, objects and types with inheritance. A Maude program containing rules and possibly equations is called a system module. The modules can form a hierarchy.

A Maude program can be both executed, formally analysed, and verified. The verification toolsuite includes an inductive theorem prover, a tool able to prove that a Maude program terminates, a model-checker for the temporal logic formulas (LTL, CTL, CTL* and others), tools for specification and verification of real-time and probabilistic systems.

Systems from different domains can be specified and analyzed with Maude. This includes semantics of programming languages, distributed algorithms, biological applications. We believe that VerCors could benefit from translating a GCM system specification into Maude mainly because Maude supports parametrised modules which could be potentially used for analyzing GCM applications with parameterised topologies. Another reason is that we could try to apply the Maude LTL bounded model-checker [167] which able to verify infinite state-space systems. The tool checks LTL formulas on symbolic representation of an application state-space and either finds a finite state-space for which the formula is fully verified, or performs the verification up to a given bound and does not succeed to produce the result, or provides a counterexample.

Moreover, VerCors could benefit from the rich and expressive type system supported by Maude. However, translation of GCM components specification into equations and rules of rewriting logic would require significant effort. Still, we believe that it should be possible because a translation to Maude has already been done for the ABS language which shares a lot of common features with GCM/ProActive. Moreover, Maude provides a mechanism for modelling object-oriented systems.

7.6 Summary

7.6.1 On the verification tools

Table 7.1 summarises the core elements of the three discussed model-checkers and CADP: the logics for the verified properties, the way processes communicate or synchronise and the techniques to deal with the state-space explosion. The table does not take into account real-time and probabilistic systems.

We believe that CADP was a perfect choice for the first experiments with model-checking in VerCors for several reasons. First, the way processes synchronise in CADP mirrors the synchronisation vectors of pNets which significantly simplifies the input model generation. Second, the state-space reduction techniques provided by CADP are efficient for working with the model of GCM components. We could benefit from applying the reduction techniques based on the tau-actions which are very useful for hiding the internal logics of the sub-components. Another advantage is that CADP implements state-space reduction by branching bisimulation which allows merging processes with similar behaviour.

Among the other verification tools, the most interesting one would be probably Maude as it includes techniques for bounded model-checking LTL formulas of systems with infinite state-space. However, translating the pNets into the rewriting logics of Maude would require significant effort. In fact, it might be even easier not to use pNets as an intermediate format but generate the Maude code directly from the user-defined design. In this sense, it should be easier to experiment with generating input for the NuSMV model-checker where the processes are encoded as finite state-machines. On the other hand, the communication paradigm of the NuSMV modules is quite different from the one of pNets: pNets have no shared variables. In would be interesting to see whether we could benefit from storing a model of GCM/ProActive components in BDD and from symbolic model-checking. Regarding the generation of Promela code, it is difficult to estimate whether applying Spin would provide us with any advantage because we would have to investigate a technique for pNets flattening and its impact on the state-space. Still, we consider the possibility of experimenting with Spin as it has demonstrated good results on handling large state-spaces.

7.6.2 On the component development frameworks

From the analysis of the existing frameworks for rigorous development of component-based systems, we can highlight the combination of several features that give VerCors an advantage and make it different from the other platforms. First, its graphical

	Spin	NuSMV	Maude	CADP
Input logics	LTL	LTL, CTL	LTL, CTL, CTL*	MCL (LTL, CTL, PDL)
Synchronisation/ communication mechanism	Message channels: synchronous or asynchronous (buffered)	Shared variables	Asynchronous message-passing and complex patterns (Maude rules) for synchronous interactions	Synchronisation vectors
Methods to fight state-space explosion	Spin applies the partial order reduction[153] and for each LTL formula it builds a partial state-space.	Cone of influence reduction [160]: NuSMV constructs a partial model that does not involve variables which are not affected by the checked property; NuSMV uses BDDs to store large state-space.	Partial order reduction [168]	Reduction based on tau actions: replacing a strongly connected component of tau-actions by a tau action, partial order reduction by analysing tau-actions and others; Reduction by strong and branching bisimulation;

Table 7.1 – Verification tools

designer mainly relies on the UML formalism which is well-known among the programmers and makes it easier to be mastered by the non-experts in formal methods. Second, we managed to automatise completely both implementation code production and the generation of the input for the formal verifier so that the user does not need to be aware of the GCM ADL syntax and does not need to write manually the LTSs that will be model-checked. Third, the graphical editor is supported by a static validation engine which ensures a range of properties for a component assembly. Finally, the users of VerCors can benefit from all core features of the GCM/ProActive semantics such as group communications, hierarchical components, run-time reconfiguration, request/reply by futures, flexible construction of non-functional part, separation between application control and business logic.

At the same time, VerCors could benefit from a wide range of features that we observed in the related approaches. In particular, implementing various communication

styles similar to the connectors from SOFA 2 would provide more flexibility and allow the user to model and verify the communication paradigm he prefers. We could try the technique implemented by the SOFA 2 authors to check that the implementation code produced by VerCors, indeed, obeys the designed behaviour. Similarly to BIP, we could create a plug-in for reverse engineering to convert ADL specification and implementation code into the conceptual models analysed by VerCors. This would allow one to verify already existing systems. Next, we should investigate how we can connect the analysed conceptual models to a legacy code like the authors of ABS did. Another useful work would be implementing predefined libraries of domain-specific re-usable components similar to the CORBA ones. All these are just examples of enhancements that we could inherit from the related frameworks and there is much more research and implementation to be done on the VerCors platform.

Chapter 8

Conclusion

Contents

8.1 Summary 203

8.2 Perspectives 206

8.2.1 Modelling and analysis of parameterised architectures . . . 206

8.2.2 Modelling and analysis of multi-threaded components . . . 207

8.2.3 Modelling and analysis of reconfigurable systems 208

8.2.4 Extending the pNet generator 210

8.2.5 Properties specification and visualising the results of model-checking 211

8.2.6 Static analysis and type-checking of state machines 211

8.2.7 Other ideas of the future work 213

8.1 Summary

Asynchronous software components provide a convenient programming abstraction for design and implementation of large-scale distributed systems, where each component acts as an autonomous entity which communicates with the other components by asynchronous request/reply. Development of such systems is is a challenging task including a huge number of issues ranging from the safe composition of components to the applications performance and reliability. This thesis aims at facilitating the development of safe component-based distributed systems by integrating the techniques for their modelling, verification, and generation in a single framework. We highlight below the main contributions of this dissertation.

First, we have designed a graphical language for the specification of architecture and behaviour of component-based systems. Our formalism allows modelling applications with hierarchical structure, separation between functional and non-functional concerns, reconfiguration capabilities, and group communications. Our graphical language reuses a lot of UML elements (i.e. state machines, classes, and interfaces) which are well-known among the software developers. This makes our formalism easy-to-learn. Another advantage is that the specifications of component architecture and behaviour are integrated together. As a result, after translating the graphical models into input for the model-checker and into executable code, the programmer does not need to provide any additional information in order to relate the structural and behavioural descriptions. In fact, this is the first concrete graphical language for the specification of architecture and behaviour of GCM components. We have implemented a front-end graphical designer for modelling distributed systems in our language.

Second, we have formalised the architecture of component-based applications and its static correctness rules. The validity of an architecture with respect to the formalised constraints guarantees that the component assembly possesses a number of properties such as correct encapsulation of components, separation between functional and non-functional concerns, deterministic communications. We have also formalised the notion of interceptors which are special components used for the communication between the functional and non-functional components. Our formalisation targets first of all GCM components but it is general enough to be extensible in order to be applied to several other component models. We have implemented validation of the formalised rules on the graphically designed models. This allows the user to check the static correctness of the designed architecture. Static correctness of a component architecture is a necessary prerequisite for the future analysis and generation of the executable code.

Third, we have formalised a set of semantic rules for the translation of components graphically specified in our language into a model of their detailed behaviour expressed in terms of pNets. The generated behavioural model encodes the core and the advanced features of GCM/ProActive components including communications based on futures, reconfigurable multicast interfaces, and hierarchical functional and non-functional components. We have implemented the formalised translation of the graphical models into pNets and the translation of the pNets into an input for CADP model-checker. This allows the user to translate the designed model into an input for the model-checker fully automatically in order to check its functional properties. Then, the programmer can verify the properties related to the safety and liveness

of the modelled application, reachability of a particular action, and inevitability of a given event. In addition to the files that encode the behaviour of the modelled system, we also generate a set of auxiliary scripts managing the state-space. They hide actions of the sub-components that should not be observed during model-checking and launch the tools from the CADP platform that perform minimisation. We also allow the user to reduce the state-space by modelling the behaviour of the environment in the form of a state-machine. While in the current version we rely on finite-state model-checking, the pNets encoding the behaviour of GCM components are parameterised, hence, they can be potentially transformed into input for a infinite-state verification tool.

Finally, we have designed and implemented a plug-in translating the conceptual model of a component-based application into executable code. The generated code includes an XML-based file with the architecture of a designed system and a set of Java interfaces and classes with the code implementing the designed business logic. The generated code can be executed on the GCM/ProActive middleware. This allows the programmer to obtain automatically the code of an application which conceptual model has been validated statically and model-checked.

Overall, the core challenge addressed by this thesis is the construction of safe distributed applications. We have integrated the techniques for model-driven design and formal verification in a single framework for modelling, verification, and generation of component-based hierarchical distributed systems. In this thesis, we have demonstrating the usage of our platform by modelling, verifying, and generating several applications. Our platform can be installed on top of Eclipse IDE.

The strong point of our approach lies in the combination of the model-driven techniques and formal methods and in full automation of the generation processes. On one hand, we allow the programmer to specify the model of his application on a high level of abstraction using well-known and popular UML notations. On the other hand, we rely on the powerful and exhaustive formal analysis techniques in order to ensure the quality of the designed models. In order to assist the programmers in the development of complex applications, we try to automatise as many steps as possible. We fully automatically transform the design of a system into an executable application. We request very few input data from the user in order to translate the graphical models into an input for the model-checker.

The core challenge in the development of our framework was to bridge the gap between the three models: the graphical specification, the input for the model-checker, and the executable code. While translating the graphical models into the model-checked graphs, we took advantage of using an additional step where we translate

the graphical specification into intermediate structures. The generated structures are based on the behavioural semantics for hierarchical asynchronous components which we have defined and formalised in this thesis in terms of pNets. Our intermediate model at the same encodes the structure and the behaviour of a user-designed application and encodes all the details of the behaviour and communications which can be model-checked.

The gap between the graphical models and executable code is not so big. While implementing this translation we took advantage of using the model-to-text transformation techniques which allowed us to keep our code clean and helped to maintain it. The combination of the features that can be modelled, verified, and generated with our platform make our framework original and expressive. It includes the hierarchical components, the future-based communications, the reconfigurable multicast interfaces, and the functional and non-functional components.

8.2 Perspectives

In this section we present the ideas for the future work on our framework for modelling, analysis, and generation of component-based systems presented in this thesis. We discuss the enhancements necessary to allow our framework to handle wider range of applications. We also explain how our platform can be made more user-friendly.

8.2.1 Modelling and analysis of parameterised architectures

The large-scale real-world applications are often composed of multiple components which implement similar behaviour, and the number of such components in the system is often unknown at the design stage. Moreover, it can vary during the program execution. An example of such application can be the implementation of map-reduce programming model for parallel and distributed processing of big data. According to it, the input data is split into independent parts which are processed by several nodes executing the map() method in parallel. The result of the execution is given as an input to the reduce() tasks which are also processed in parallel. The number of the workers executing the tasks can depend on various factors such as, for instance the size of the input data.

As one of the directions for future work, we would like to extend our framework with techniques for modelling and verification of systems with variable number of components, i.e. systems with parameterised topologies. This requires, first of all, extending the GCM ADL. In [87] the authors introduced examples of component-

based architectures with parameterised topologies following several patterns (a ring, a matrix, a pipeline, etc). The authors presented how such applications can be encoded in ProActive/GCM ADL. We can use the results of their work as a starting point for the future research and extend their ADL (and ProActive/GCM ADL) for dealing with more generic parameterised architectures.

In order to model such systems in VerCors, we need to extend our graphical formalism. Regarding the verification, the good point is that the pNets encoding the behaviour of GCM components are already parameterised, hence modelling systems with parameterised topologies in pNets should not require much additional effort. However, we should still find a way to translate the graphically specified parameters to the pNet counterpart. Finally, we should extend the executable code generator of VerCors so that it can construct parameterised ADL files. Moreover, we should enhance the ProActive platform so that it can build applications defined in a parameterisd ADL with the value of parameters given during deployment. There is no notion of parameterised ADL in the current version of ProActive.

8.2.2 Modelling and analysis of multi-threaded components

Currently, the VerCors platform can be used to model and to verify only single-threaded primitive components. GCM/ProActive already supports programming of multi-threaded primitive components, and we would like to extend our framework with techniques for modelling, verification and generation of multi-threaded primitive components.

Similar to a single-threaded primitive, a multi-threaded one is implemented in GCM/ProActive as an active object, but in the second case it can have several threads. Such active objects are called "multiactive objects". For a multiactive object, the programmer has to specify the number of threads (it is called the "thread limit") which can run in parallel and the requests that can be executed in parallel (we also call them "compatible requests"). The thread limit allows avoiding a thread explosion. In order to define which requests can be executed in parallel, the programmer should split the server methods into groups and specify the compatibilities between the groups. Two requests to the methods from compatible groups can run in parallel. Moreover, the FIFO policy of the queue of a multiactive object is adapted so that a request can overtake the other requests and be executed if two conditions are satisfied. First, it should be compatible with the currently running requests, and, second, it should be compatible with the requests that are located before it in the queue.

What makes the implementation of GCM/ProActive multiactive objects espe-

cially interesting for the developer is the priority specification mechanism [169] which allows the programmer to control the scheduling of requests. The programmer can use annotations to specify the priorities between the groups. Then, instead of inserting a new request at the end of the queue, its position is computed depending on the specified priorities. More precisely, a request that belongs to a group G is inserted in the queue before the first request which group has lower priority than G. The advantage of such approach is that it allows the programmer to customise the request scheduling.

Modelling and analysis of components based on GCM/ProActive multiactive objects in VerCors requires, first of all, additional graphical notations for the specification of the thread limit and the priorities. We suppose that they should be defined for a given UML class implementing the behaviour of a component. The generated pNets should be also modified: we should encode two additional processes and modify the body. The first process should control the number of threads that are currently running. When the body takes a new request from the queue, it should check with this process whether there is an available thread. Whenever a method is served, it should notify this process that its thread is released. In addition, we should encode the process inserting the next request in the queue with respect to the user-defined priorities. The body should be modified in such a way that it can take from the queue a request that should be executing according to the compatibility relation, i.e. we need to encode the overtaking of requests. Finally, we should annotate the generated GCM/ProActive code with respect to the specified groups, compatibilities, priorities, and thread limits.

8.2.3 Modelling and analysis of reconfigurable systems

In this thesis we discussed how a GCM application with reconfigurable multicast interfaces can be designed, verified, and generated in our framework. This is not the only reconfiguration that can be performed for the real GCM systems: the singleton interfaces can be also bound and unbound, the existing components can be started and stopped, and new components can be added to the application at run-time. Extending our framework with tools for modelling, analysis, and generation of such kind of reconfigurations is in the scope of the future work.

Graphical specification of reconfigurable singleton interfaces should follow the same principle as for the multicasts. Recall that the latter can have several outgoing bindings, some of them are created during the construction of the application, others are bound at run-time. A singleton can also have several outgoing bindings but at most one of them can be bound at each time. As for the multicast interfaces,

the bind and unbind operations should be triggered from the state machines. The construction of pNets for reconfigurable singletons was formalised in [73]. While the reconfiguration of a multicast interface is encoded in the proxy and proxy manager of its methods, a singleton interface requires an additional binding controller pLTS. It stores the identifier of the currently bound target interface, provides actions for its modification, and synchronises with the rest of the component on the method invocations. Implementing construction of such a pLTS in VerCors and synchronising it with the other processes is not more complex than generating reconfigurable multicast interfaces.

Another type of reconfiguration that we would like to model and to verify in VerCors is starting and stopping a component; this is done by a so-called lifecycle-controller in ProActive/GCM. There are two key points that make the reconfiguration challenging. First, we should be able to start and stop separately the membrane and the content (or the implementation class) of a composite (or of a primitive). When the content of a composite is stopped but its membrane is running, the component can serve only the non-functional requests while the functional ones are accumulated in the queue. This changes the policy of the body: instead of serving calls in a FIFO order, it checks with the lifecycle-controller the current status of the component, and depending on the current status it takes the first request of a functional or of a non-functional type from the queue. The second challenge is imposed by the hierarchy: normally, when the content of a composite component is stopped, its sub-components at all levels of hierarchy should be also stopped as it is explained in [170].

The instructions triggering component start and stop should be graphically specified in the state machines. Encoding such kind of reconfiguration in pNets requires extending each component with a pLTS modelling the lifecycle-controller, modifying the policy of the body, and extending the pLTS of the queue and the body with the actions treating the start and stop requests.

Finally, a sub-component can be added in a membrane or in a content of a component at run-time. For the graphical specification of such reconfiguration we can follow the same approach as for the bind/unbind operations: the representation of a component can be different depending on whether the component is added to the system during its construction or at run-time. In addition, we should extend our set of the architecture static validation rules to check that no binding is plugged to a non-existing component during the construction of the application. When constructing the pNets, we should generate all the components specified in the system.

8.2.4 Extending the pNet generator

We discuss below the features of the component-based applications which we would like to be able to verify with our framework in addition to the ones fully implemented in this thesis. Since we rely on the pNets as an intermediate format, we first of all generate the pNets encoding those features. For some of them, the construction of pNets is already formalised (e.g. the multicast interfaces for primitive components) but not yet implemented in VerCors. For the others, we still need to provide the necessary formalisation and to encode their construction in our platform.

In addition to the future objects discussed in this work, components can also rely on so-called *first-class futures* - the futures that can be transmitted between components before their value is known. Recall that with the usual futures, a component has to wait for the result of the remote computation before sending it to another component. In the case of the first-class futures, a component can send a reference to a future object (for instance, as an argument of a request) even if its value has not been received yet. Encoding in pNets GCM applications that communicate by first-class futures was presented in [73]. The authors introduce the notion of a *generalised reference to the future* which is a special variable encoding the name of the component and the index of the proxy holding the future; in some sense it is a unique identifier of the future. The reference can be then transmitted between components as an argument of a request. When the component holding the future receives its value, it forwards the value to all other components holding the generalised reference. This requires small modifications in the structure of proxy and proxy manager pLTSs, an additional pLTS in the component which waits for the future value to be forwarded, and several additional synchronisation vectors. The main challenge in constructing the additional elements will be to identify statically which arguments of which requests can be a references to a future.

In this work we discussed how the interceptors can be graphically modelled and recognised among the sub-components in a membrane. The next step is to extend the pNets encoding the behaviour of GCM components with interceptors in order to model-check their properties.

Another enhancement of the pNet generator that should be carried in a short-term period is the construction of primitive components with a componentised membrane. More precisely, the pNet of a primitive should be extended with an additional set of sub-nets modelling the behaviour of the sub-components. The set of synchronisation vectors should be extended with the vectors for the bindings inside the membrane. Finally, we should generate and synchronise the proxy, proxy manager, and delegate pLTSs for the methods of the non-functional interfaces.

8.2.5 Properties specification and visualising the results of model-checking

This work targets the software developers who are not experts in formal methods, and we believe that considering the complexity of the underlying formal techniques, we achieved significant progress in automatising the construction of the input for the model-checker. However, a necessary prerequisite for making our tool accessible for the non-experts in formal techniques is assisting our users in the specification of the model-checked properties and translating the diagnostics provided by the model-checker to the graphical language of our front-end editor.

The core challenge in the high-level specification of the model-checked properties is the difference between our graphical specification language and MCL with the language of the actions of pNets. Our idea is to create a wizard where the user can select the actions performed by the designed model and the MCL pattern corresponding to the verified formula. Then, we can translate the chosen actions into the labels of the model-checked behavioural graph and use them as the arguments of the chosen MCL pattern. For example, the user can state that a given request cannot be emitted by a component twice. Then, the pNet action encoding the corresponding request emission should be used as an argument of the *AbsenceBefore* MCL pattern.

Another approach to the high-level specification of the expected behaviour is to model it as a so-called observer automaton. In the case of VerCors, it can be designed as a state machine. This would require developing another language for the labels of the transitions for expressing the communications between components. Then, we can translate a state machine into an MCL formula. For instance, a state machine with a simple sequence of transitions without branching can be converted to a formula stating that the corresponding sequence of actions exists in the behaviour graph.

Regarding the diagnostics provided by the model-checker, CADP can return as a result of verification an example (or a counterexample) of a path in the input behavioural graph that satisfies (or does not satisfy) the checked property. In our case, the path is constructed from the labels of the actions of the pNets modelling the behaviour of the input system. We should interpret each such label and highlight the graphical elements that are involved in the actions. We should not highlight them all at once, but do it step-by-step following their order given by the model-checker.

8.2.6 Static analysis and type-checking of state machines

In the current version of VerCors we statically validate the architecture of the designed system but not its behaviour. Recall that the behaviour of each server and local

method is defined as a state machine and we would like to extend our framework with tools for their static analysis and type-checking. It is required for three main reasons.

First, analysing a state machine before translating it into a pLTS is needed in order to identify better where the system should block and wait for the value of a future variable. This will make the generated model closer to the real implementation. We are currently working on such an analysis.

Second, analysing a state machine statically allows identifying various errors early at the design stage. We can already detect variables that are used before being initialised and verify that a method invoked by a state machine exists. We should also check that a guard condition is indeed a boolean expression, that the operands in the expressions have compatible types, that the arguments of a method call have correct types, etc. Such an analysis will allow us to generate correct executable code and to warn the user about mistakes before using the model-checker.

Finally, the static analysis of state machines can be used to generate more user-friendly implementation code. Recall that the current version of VerCors produces a switch-case statement in the body of a server (or of a local) method and translates each state of the corresponding state machine into a case-statement. Such code mirrors the structure of a state machine but it is not very easy to follow. On one hand, this is not an issue because the logic of the generated code is not supposed to be modified as it has been model-checked. On the other hand, the programmer who maintains the code is not always the designer of the state machine, and he needs to understand clearly what the code does in order to maintain the development of the software. In order to generate more user-friendly code, we can recognise in the structure of a state machine patterns that correspond to various control statements like for-loop, do-while loop, etc and to translate them in the corresponding code. The simplest example is a sequence of transitions without branching or guard conditions; it can be translated into a sequence of instructions in the executable code. On the other hand, such an approach can lead to some restrictions on the structure of the state machines because the programmer will be allowed to use only those constructs which can be recognised by our analysis plug-in. Hence, we would like to allow the user to choose: either to compose his state machine only from the structures that can be recognised by our translator and converted into a user-friendly code, or to define any combination of states and transitions and to translate them into the switch-case statements.

8.2.7 Other ideas of the future work

In addition to the future work that we discussed in details in the previous sections, there is a number of other enhancements from which our framework can benefit.

Several aspects that were already mentioned in this dissertation are related only to the implementation of the VerCors platform and can be solved in a short-term period. In particular, in the current version of the platform, the size of the families of proxies has the default value for any request. We should develop a wizard where the user can define the size of the families of proxies for each kind of request separately (or for the groups of requests). This will make our tool more flexible and optimise the state-space because we will not generate more proxies than needed according to the user input.

Second, we should improve the way the attributes of primitive components are handled as it was discussed in Section 6.2. Remember that in the current version of the VerCors platform, the programmer has to explicitly define state machines that provide access to the attributes to outside of the primitive. The state machines are then translated into pNets as any other server method. Instead of asking the user to specify a state machine, we can offer him to select the attributes that are accessible through the server interfaces. We already generate the pLTSs of the attribute controllers which store the values of the attributes and provide actions to access them from the component. For the chosen attributes, we should extend the queue and the body pLTSs with the actions for handling requests to their attribute controllers and synchronise them with the rest of the system. On the contrary, in the generated executable code all attributes are accessible from outside of a primitive. This should be also modified depending on the input from the user.

As the longer-term tasks, we plan to experiment with integrating our platform with several other verification tools. We discussed in Section 7.6.1 the possibility of generating input for the Maude and NuSMV frameworks but we are open to experimenting with other tools. In order to enhance the integration of our platform with CADP we would like to investigate whether the pNets can be converted into LotosNT instead of Fiacre. Another idea for the future research is providing techniques for modelling and verification of different communication styles as it is done by the authors of SOFA for their connectors.

Bibliography

[1] Petr Hnetrynka, František Plášil, Vladimir Mencl, and Lucia Kapova. *SOFA 2.0 metamodel*. technical report. Department of Software Engineering Faculty of Mathematics and Physics, Charles University, 2005 (cit. on pp. 2, 7, 179).

[2] 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". In: *IEEE Software* 28.3 (2011), pp. 41–48 (cit. on pp. 2, 7, 181).

[3] Object Management Group. *CORBA Component Model 4.0 Specification*. Specification Version 4.0. Object Management Group, 2006 (cit. on pp. 2, 195).

[4] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. "GCM: a grid extension to Fractal for autonomous distributed components". In: *Annales des Télécommunications* 64.1-2 (2009), pp. 5–24 (cit. on pp. 3, 14).

[5] Activeeon. *ProActive Parallel Suite*. 2016. URL: `http://proactive.activeeon.com/` (cit. on pp. 3, 18).

[6] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. 2013 (cit. on p. 4).

[7] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Tech. rep. Object Management Group, 2011 (cit. on pp. 4, 8, 34).

[8] Obeo. *Uml to Java Generator*. last visited: 08.2016. URL: `http://marketplace.obeonetwork.com/module/uml2java-generator` (cit. on p. 4).

[9] *ArgoUML - an open source UML modeling tool*. last visited: 08.2016. URL: `http://argouml.tigris.org/` (cit. on p. 4).

[10] Bruno Pagès. *BOUML - a UML 2 tool box*. last visited: 08.2016. URL: `http://www.bouml.fr/index.html` (cit. on p. 4).

[11] Leonor M. Barroca and John A. Mcdermid. “Formal Methods: Use and Relevance for the Development of Safety Critical Systems”. In: *The Computer Journal* 35 (1992), pp. 579–599 (cit. on p. 5).

[12] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon Web Services Uses Formal Methods”. In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73 (cit. on p. 5).

[13] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. “Model Checking Programs”. In: *Automated Software Engg.* 10.2 (Apr. 2003), pp. 203–232 (cit. on p. 5).

[14] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. “Model checking flight control systems: The Airbus experience”. In: *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on.* 2009, pp. 18–27 (cit. on p. 5).

[15] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* 1995 (cit. on p. 6).

[16] E. Allen Emerson and A. Prasad Sistla. “Symmetry and model checking”. In: *Formal Methods in System Design* 9.1 (1996), pp. 105–131 (cit. on p. 6).

[17] Ludovic Henrio, Florian Kammüller, and Marcela Rivera. “An Asynchronous Distributed Component Model and Its Semantics”. In: *Formal Methods for Components and Objects: 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures.* Ed. by Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 159–179 (cit. on p. 6).

[18] Marjan Sirjani. “Rebeca: Theory, Applications, and Tools”. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures.* 2006, pp. 102–126 (cit. on pp. 7, 183).

[19] Antonio Cansado and Eric Madelaine. “Formal Methods for Components and Objects: 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures”. In: ed. by Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Specification and Verification for Grid Component-Based Applications: From Models to Tools, pp. 180–203 (cit. on pp. 8, 38, 55, 57).

[20] Solange Ahumada, Ludovic Apvrille, Tomás Barros, Antonio Cansado, Eric Madelaine, and Emil Salageanu. "Specifying Fractal and GCM Components with UML." In: *SCCC*. IEEE Computer Society, 2007, pp. 53–62 (cit. on pp. 8, 37).

[21] Ludovic Henrio, Eric Madelaine, and Min Zhang. "pNets: an Expressive Model for Parameterised Networks of Processes". In: *Formal Approaches to Parallel and Distributed Systems (4PAD)-Special Session of Parallel, Distributed and network-based Processing (PDP)*. Turku, Finland, 2015 (cit. on pp. 8, 25, 27).

[22] Nicolas Coste, Hubert Garavel, Holger Hermanns, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. "Ten Years of Performance Evaluation for Concurrent Systems Using CADP". In: *4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA 2010*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6416. Amirandes, Heraclion, Greece, Oct. 2010, pp. 128–142 (cit. on pp. 8, 187).

[23] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. "Formally Reasoning on a Reconfigurable Component-Based System - A Case Study for the Industrial World". In: *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*. 2013, pp. 137–156 (cit. on p. 8).

[24] Rabéa Ameur-Boulifa, Raluca Halalai, Ludovic Henrio, and Eric Madelaine. "Verifying Safety of Fault-Tolerant Distributed Components". In: *Formal Aspects of Component Software: 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*. Ed. by Farhad Arbab and Peter Csaba Ölveczky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 278–295 (cit. on pp. 8, 125).

[25] 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". In: *Softw. Pract. Exper.* 36.11-12 (Sept. 2006), pp. 1257–1284 (cit. on p. 14).

[26] Françoise Baude, Ludovic Henrio, and Cristian Ruz. "Programming distributed and adaptable autonomous components - the GCM/ProActive framework". In: *Softw., Pract. Exper.* 45.9 (2015), pp. 1189–1227 (cit. on pp. 17, 18, 188).

[27] R. Greg Lavender and Douglas C. Schmidt. “Pattern Languages of Program Design 2”. In: ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Active Object: An Object Behavioral Pattern for Concurrent Programming, pp. 483–499 (cit. on p. 18).

[28] Ludovic Henrio, Fabrice Huet, and Zsolt István. “Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Florence, Italy, June 3-5, 2013. Proceedings”. In: ed. by Rocco Nicola and Christine Julien. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. Multithreaded Active Objects, pp. 90–104 (cit. on p. 21).

[29] Ludovic Henrio and Justine Rochas. “Declarative Scheduling for Active Objects”. In: *SAC 2014 - 29th Symposium On Applied Computing*. Ed. by Sung Y. Shin. ACM Special Interest Group on Applied Computing. Gyeongju, South Korea: ACM, Mar. 2014, pp. 1–6 (cit. on p. 21).

[30] Tomàs Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. “Behavioural models for distributed Fractal components”. In: *Annals of Télécommunications* 64.1-2 (2009), pp. 25–43 (cit. on pp. 25, 100).

[31] Ludovic Henrio, Eric Madelaine, and Min Zhang. “A Theory for the Composition of Concurrent Processes”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*. 2016, pp. 175–194 (cit. on pp. 25, 103, 198).

[32] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. “CADP 2011: a toolbox for the construction and analysis of distributed processes”. In: *STTT* 15.2 (2013), pp. 89–107 (cit. on p. 29).

[33] Rim Abid, Gwen Salaün, Francesco Bongiovanni, and Noël De Palma. “Verification of a Dynamic Management Protocol for Cloud Applications”. In: *11th International Symposium, ATVA 2013*. Vol. 8172. Dang Van Hung and Mizuhito Ogawa. Hanoi, Vietnam, Oct. 2013, pp. 178–192 (cit. on p. 29).

[34] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. “Automated Reverse Engineering using Lego®”. In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014 (cit. on p. 29).

[35] Bart Theelen, Joost-Pieter Katoen, and Hao Wu. “Model checking of Scenario-Aware Dataflow with CADP”. In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 653–658 (cit. on p. 29).

[36] Tommaso Bolognesi and Ed Brinksma. “Introduction to the ISO Specification Language LOTOS”. In: *Comput. Netw. ISDN Syst.* 14.1 (Mar. 1987), pp. 25–59 (cit. on p. 30).

[37] Mihaela Sighireanu. *LOTOS NT User’s Manual*. 2000 (cit. on p. 30).

[38] Frédéric Lang. “Formal Techniques for Networked and Distributed Systems - FORTE 2006: 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006. Proceedings”. In: ed. by Elie Najm, Jean-François Pradat-Peyre, and Véronique Viguié Donzeau-Gouge. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Chap. Refined Interfaces for Compositional Verification, pp. 159–174 (cit. on p. 30).

[39] Rocco De Nicola and Frits Vaandrager. “Semantics of Systems of Concurrent Processes: LITP Spring School on Theoretical Computer Science La Roche Posay, France, April 23–27, 1990 Proceedings”. In: ed. by Irène Guessarian. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990. Chap. Action versus state based logics for transition systems, pp. 407–419 (cit. on p. 31).

[40] E. M. Clarke, E. A. Emerson, and A. P. Sistla. “Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263 (cit. on p. 31).

[41] Michael J. Fischer and Richard E. Ladner. “Propositional dynamic logic of regular programs”. In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 194 –211 (cit. on p. 31).

[42] Radu Mateescu and Mihaela Sighireanu. “Efficient on-the-fly model-checking for regular alternation-free mu-calculus”. In: *Science of Computer Programming* 46.3 (2003). Special issue on Formal Methods for Industrial Critical Systems, pp. 255 –281 (cit. on p. 31).

[43] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Patterns in Property Specifications for Finite-state Verification”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. Los Angeles, California, USA: ACM, 1999, pp. 411–420 (cit. on p. 31).

[44] Radu Mateescu and Damien Thivolle. “A Model Checking Language for Concurrent Value-Passing Systems”. In: *FM 2008*. Ed. by Jorge Cuellar and Tom Maibaum. Vol. 5014. Turku, Finland: Springer Verlag, May 2008, pp. 148–164 (cit. on pp. 31, 112).

[45] Henrik Reif Andersen. “Model checking and boolean graphs”. In: *Theoretical Computer Science* 126.1 (1994), pp. 3 –30 (cit. on p. 31).

[46] Hubert Garavel and Frédéric Lang. “Formal Techniques for Networked and Distributed Systems: FORTE 2001 IFIP TC6/WG6.1 — 21st International Conference on Formal Techniques for Networked and Distributed Systems August 28–31, 2001, Cheju Island, Korea”. In: ed. by Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee. Boston, MA: Springer US, 2002. Chap. SVL: A Scripting Language for Compositional Verification, pp. 377–392 (cit. on p. 31).

[47] Hubert Garavel, Radu Mateescu, and Wendelin Serwe. “Large-scale Distributed Verification Using CADP: Beyond Clusters to Grids”. In: *Electronic Notes in Theoretical Computer Science* 296 (2013). Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC)., pp. 145 –161 (cit. on p. 32).

[48] B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. “The syntax and semantics of Fiacre”. In: March 2009 (cit. on p. 32).

[49] *flac compiler*. URL: `http://gforge.enseeiht.fr.` (cit. on p. 33).

[50] Alberto Rodrigues da Silva. “Model-driven engineering: A survey supported by the unified conceptual model”. In: *Computer Languages, Systems and Structures* 43 (2015), pp. 139 –155 (cit. on p. 33).

[51] *Eclipse IDE*. URL: `https://eclipse.org/` (cit. on p. 35).

[52] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. 2008 (cit. on p. 35).

[53] *Graphical Modeling Framework*. URL: `http://www.eclipse.org/gmf-tooling/` (cit. on p. 36).

[54] *Eclipse UML meta-mode*. URL: `https://wiki.eclipse.org/MDT/UML2/Getting_Started_with_UML2` (cit. on pp. 36, 52).

[55] *Obeo Designer*. URL: `http://www.obeodesigner.com/` (cit. on p. 36).

[56] Etienne Juliot and Jérôme Benois. *Viewpoints creation using Obeo Designer or how to build Eclipse DSM without being an expert developer*. Obeo (cit. on p. 36).

[57] *Obeo UML Designer*. URL: `http://www.umldesigner.org/overview/` (cit. on p. 37).

[58] *Acceleo code generator*. URL: `http://wiki.eclipse.org/Acceleo` (cit. on pp. 37, 118).

[59] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu. "A Specification Language for Distributed Components Implemented in GCM/ProActive". In: *The Common Component Modeling Example: Comparing Software Component Models*. Ed. by Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 418–448 (cit. on p. 37).

[60] Annie Ressouche, Robert de Simone, A Bouali, and V. Roy. *The FC2Tool user manuel*. 1994. URL: `http://www-sop.inria.fr/meije/verification/` (cit. on p. 38).

[61] Tomas Barros. "Formal specification and verification of distributed component systems". PhD Thesis. University of Nice - Sophia Antipolis, 2005 (cit. on p. 38).

[62] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. "Transparent First-class Futures and Distributed Components". In: *Electronic Notes in Theoretical Computer Science* 260 (2010). Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008), pp. 155 –171 (cit. on p. 38).

[63] Rabéa Ameur Boulifa, Ludovic Henrio, and Eric Madelaine. "Behavioural Models for Group Communications". In: *WCSI-10: International Workshop on Component and Service Interoperability*. 2010 (cit. on pp. 38, 140).

[64] Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, and Eric Madelaine. *Integrated environment for verifying and running distributed components - Extended version*. Research Report RR-8841. INRIA Sophia-Antipolis, Dec. 2015, p. 24 (cit. on p. 41).

[65] Gary L. Peterson. "An O(Nlog N) Unidirectional Algorithm for the Circular Extrema Problem". In: *ACM Trans. Program. Lang. Syst.* 4.4 (Oct. 1982), pp. 758–762 (cit. on p. 44).

[66] Danny Dolev, Maria M. Klawe, and Michael Rodeh. "An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle". In: *J. Algorithms* 3.3 (1982), pp. 245–260 (cit. on p. 45).

[67] *CUP - LALR Parser for Java*. URL: `http://www2.cs.tum.edu/projects/cup/index.php` (cit. on p. 53).

[68] *JFlex - a lexical analyzer generator*. URL: `http://jflex.de/` (cit. on p. 53).

[69] Paul Naoumenko. "Designing Non-functional Aspects With Components". PhD Thesis. Univ. of Nice-Sophia Antipolis, 2010 (cit. on pp. 56, 60, 70).

[70] C. Canal, C.S. Pasareanu, Antonio Cansado, Ludovic Henrio, Eric Madelaine, and Pablo Valenzuela. "Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008) Unifying Architectural and Behavioural Specifications of Distributed Components". In: *Electronic Notes in Theoretical Computer Science* 260 (2010), pp. 25 –45 (cit. on p. 56).

[71] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. "Painless support for static and runtime verification of component-based applications". In: *Fundamentals of Software Engineering (FSEN'2015)*. Teheran, Iran, Apr. 2015, p. 15 (cit. on p. 56).

[72] *Papyrus modeling environment*. URL: `https://eclipse.org/papyrus/` (cit. on p. 57).

[73] Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine, and Alexandra Savu. *Behavioural Semantics for Asynchronous Components*. Research Report RR-8167. INRIA, Dec. 2012, p. 58 (cit. on pp. 60, 103, 209, 210).

[74] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. "Bringing Coq Into the World of GCM Distributed Applications". In: *International Symposium on High-level Parallel Programming and Applications&, HLPP*. Paris, France, 2013 (cit. on pp. 60, 76).

[75] Ludovic Henrio, Oleksandra Kulankhina, Dongqian Liu, and Eric Madelaine. "Verifying the correct composition of distributed components: Formalisation and Tool". In: *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014.* 2014, pp. 69–85 (cit. on p. 60).

[76] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on p. 74).

[77] Michal Malohlava, Petr Hnetynka, and Tomas Bureš. "{SOFA} 2 Component Framework and Its Ecosystem". In: *Electronic Notes in Theoretical Computer Science* 295 (2013). Proceedings the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), pp. 101 –106 (cit. on pp. 75, 178).

[78] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. "A Component Model Engineered with Components and Aspects". In: *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*. 2006 (cit. on p. 75).

[79] Philippe Merle and Jean-Bernard Stefani. *A formal specification of the Fractal component model in Alloy*. Research Report RR-6721. INRIA, 2008 (cit. on p. 75).

[80] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. "A Framework for Reasoning on Component Composition". In: *FMCO 2009*. Lecture Notes in Computer Science. Springer, 2010 (cit. on p. 75).

[81] Yves Bertot. "Coq in a Hurry". In: *CoRR* abs/cs/0603118 (2006) (cit. on p. 76).

[82] *Java Architecture for XML Binding (JAXB)*. URL: `https://jaxb.java.net/` (cit. on p. 117).

[83] Hans J. Köhler, Ulrich Nickel, Jörg Niere, and Albert Zündorf. "Integrating UML Diagrams for Production Control Systems". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 241–251 (cit. on p. 121).

[84] Iftikhar Azim Niaz and Jiro Tanaka. "Code Generation From Uml Statecharts". In: *in Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*. 2003, pp. 315–321 (cit. on p. 121).

[85] Emil Sekerinski and Rafik Zurob. "iState: A Statechart Translator". In: *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings*. Ed. by Martin Gogolla and Cris Kobryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 376–390 (cit. on p. 121).

[86] *A viewer for ProActive Active objects*. URL: `https://github.com/scale-proactive/A-viewer-tool-for-multiactive-objects.git` (cit. on p. 123).

[87] Amine Rouini. “Parametric Component Topologies: language extension and implementation”. Master thesis. Univ. of Nice-Sophia Antipolis, 2010 (cit. on pp. 125, 206).

[88] K. Rustan M. Leino and Michal Moskal. “VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0”. In: *Proceedings of Tools and Experiments Workshop at VSTTE*. 2010 (cit. on p. 162).

[89] Tatiana Aubonnet, Ludovic Henrio, Soumia Kessal, Oleksandra Kulankhina, Frédéric Lemoine, Eric Madelaine, Cristian Ruz, and Noëmie Simoni. “Management of service composition based on self-controlled components”. In: *Journal of Internet Services and Applications* 6.15 (2015), p. 17 (cit. on p. 171).

[90] Matías Ibañez, Cristian Ruz, Ludovic Henrio, and Javier Bustos-Jiménez. “Reconfigurable Applications Using GCMScript”. Accepted at IEEE Cloud computing. Special issue: Autonomic clouds. Apr. 2016 (cit. on p. 172).

[91] Petr Hnětynka and František Plášil. “Dynamic Reconfiguration and Access to Services in Hierarchical Component Models”. In: *Proceedings of the 9th International Conference on Component-Based Software Engineering*. CBSE’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 352–359 (cit. on p. 178).

[92] Tomas Bureš and František Plášil. “Communication Style Driven Connector Configurations”. In: *LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743*. Springer-Verlag, 2004, pp. 102–116 (cit. on p. 178).

[93] Tomas Bureš and František Plášil. “Generating Connectors for Homogeneous and Heterogeneous Deployment”. In: 2006 (cit. on p. 179).

[94] František Plášil and Stanislav Visnovsky. “Behavior Protocols for Software Components”. In: *IEEE Trans. Softw. Eng.* 28.11 (Nov. 2002), pp. 1056–1076 (cit. on p. 179).

[95] Martin Mach, František Plášil, and Jan Kofron. “Behavior Protocol Verification: Fighting State Explosion”. In: *International Journal of Computer and Information Science* 6.1 (2005), pp. 22–30 (cit. on p. 179).

[96] Pavel Parizek, František Plášil, and Jan Kofron. “Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker”. In: *30th Annual IEEE / NASA Software Engineering Workshop , 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*. 2006, pp. 133–141 (cit. on pp. 179, 181).

[97] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. "Model Checking Programs". In: *Automated Software Engg.* 10.2 (Apr. 2003), pp. 203–232 (cit. on p. 179).

[98] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Softw. Eng.* 23.5 (May 1997), pp. 279–295 (cit. on pp. 180, 197).

[99] Jan Kofron. "Checking Software Component Behavior Using Behavior Protocols and Spin". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. Seoul, Korea: ACM, 2007, pp. 1513–1517 (cit. on p. 180).

[100] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*. SEFM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12 (cit. on p. 181).

[101] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. "Compositional Translation of Simulink Models into Synchronous BIP". In: *IEEE Fifth International Symposium on Industrial Embedded Systems - SIES 2010, University of Trento, Italy, July 7-9, 2010*. IEEE, 2010, pp. 217–220 (cit. on p. 182).

[102] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. "Translating AADL into BIP - Application to the Verification of Real-Time Systems". In: *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*. Vol. 5421. Lecture Notes in Computer Science. Springer, 2008, pp. 5–19 (cit. on p. 182).

[103] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. "D-Finder: A Tool for Compositional Deadlock Detection and Verification". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 614–619 (cit. on p. 182).

[104] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. "Formal Verification of Infinite-State BIP Models". In: *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*. Ed. by Bernd Finkbeiner, Geguang Pu, and Lijun Zhang. Cham: Springer International Publishing, 2015, pp. 326–343 (cit. on p. 182).

[105] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 334–342 (cit. on p. 182).

[106] Ylies Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. "Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation". In: *Software and System Modeling* 14.1 (2015), pp. 173–199 (cit. on p. 183).

[107] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. "Automated distributed implementation of component-based models with priorities". In: *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*. ACM, 2011, pp. 59–68 (cit. on p. 183).

[108] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Modeling and Verification of Reactive Systems using Rebeca". In: *Fundam. Inform.* 63.4 (2004), pp. 385–410 (cit. on p. 183).

[109] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. "Modular Verification of a Component-Based Actor Language". In: *J. UCS* 11.10 (2005), pp. 1695–1717 (cit. on pp. 184, 185).

[110] *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*. IEEE Computer Society, 2005 (cit. on p. 184).

[111] Fatemeh Alavizadeh and Marjan Sirjani. "Using UML to Develop Verifiable Reactive Systems". In: *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*. 2006, pp. 554–561 (cit. on p. 185).

[112] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. "Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system". In: *Science of Computer Programming* 98, Part 2 (2015). Special Issue on Programming Based on Actors, Agents and Decentralized Control, pp. 184 –204 (cit. on p. 185).

[113] Ehsan Khamespanah, Marjan Sirjani, Mohammad Reza Mousavi, Zeynab Sabahi-Kaviani, and Mohamadreza Razzazi. "State Distribution Policy for Distributed Model Checking of Actor Models". In: *ECEASST* 72 (2015) (cit. on p. 185).

[114] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models". In: *J. UCS* 11.6 (2005), pp. 1054–1082 (cit. on p. 185).

[115] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. "Symmetry and partial order reduction techniques in model checking Rebeca". In: *Acta Inf.* 47.1 (2010), pp. 33–66 (cit. on p. 186).

[116] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. "ABS: A Core Language for Abstract Behavioral Specification". In: *Formal Methods for Components and Objects (FMCO)*. Ed. by Marcello M. Bonsangue Bernhard K. Aichernig Frank S. de Boer. Vol. 6957. Graz, Austria, 2010, pp. 142–164 (cit. on p. 188).

[117] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. "KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS." In: *CADE*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 517–526 (cit. on pp. 188, 190).

[118] Ludovic Henrio and Justine Rochas. "From Modelling to Systematic Deployment of Distributed Active Objects". In: *11th International Federated Conference on Distributed Computing Techniques (DisCoTec 2016)*. Heraklion, Greece, June 2016 (cit. on pp. 188, 191).

[119] Reiner Hähnle, Michiel Helvenstijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Wong. "HATS Abstract Behavioral Specification: The Architectural View". In: *FMCO - Formal Methods for Components and Objects - 2011*. Ed. by Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 7542. Turin, Italy: Springer Berlin / Heidelberg, Oct. 2011, pp. 165–185 (cit. on p. 188).

[120] Reiner Hähnle. "The Abstract Behavioral Specification Language: A Tutorial Introduction." In: *FMCO*. Ed. by Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 7866. Lecture Notes in Computer Science. Springer, 2012, pp. 1–37 (cit. on p. 189).

[121] Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. "ABS-YARN: A Formal Framework for Modeling Hadoop YARN Clusters". In: *Proc. 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*. Ed. by Perdita Stevens and Andrzej Wasowski. Vol. 9633. Lecture Notes in Computer Science. Springer, 2016 (cit. on p. 189).

[122] *Yarn Scheduler Load Simulator*. URL: `https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html` (cit. on p. 189).

[123] HATS project. *ABS Eclipse Plug-in*. URL: `http://tools.hats-project.eu/eclipseplugin/installation.html` (cit. on p. 189).

[124] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Berlin, Heidelberg: Springer-Verlag, 2007 (cit. on p. 189).

[125] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. "A framework for deadlock detection in core ABS". In: *CoRR* abs/1511.04926 (2015) (cit. on p. 190).

[126] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. "COSTABS: A Cost and Termination Analyzer for ABS". In: *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. PEPM '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 151–154 (cit. on p. 190).

[127] Nikolaos Bezirgiannis and Frank Boer. "SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings". In: ed. by Rūsiņš Freivalds Mārtiņš, Gregor Engels, and Barbara Catania. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. Chap. ABS: A High-Level Modeling Language for Cloud-Aware Programming, pp. 433–444 (cit. on p. 191).

[128] 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. Buenes Aires, Argentina: ACM, 2007, pp. 54–65 (cit. on p. 192).

[129] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. "DEECO: An Ensemble-based Component System". In: *Proceedings of the 16th International ACM Sigsoft Symposium*

on Component-based Software Engineering. CBSE '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 81–90 (cit. on p. 192).

[130] Bureš T. Al Ali R., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., and Plášil F. *DEECo Computational Model - I*. Technical report. Charles University in Prague, 2013 (cit. on p. 193).

[131] *JDEECo - a Java implementation of the DEECo component system*. URL: `https://github.com/d3scomp/JDEECo` (cit. on p. 193).

[132] Rolf Hennicker and Annabelle Klarl. "Foundations for Ensemble Modeling - The Helena Approach". In: *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi (SAS 2014)*. 2014 (cit. on p. 193).

[133] Annabelle Klarl, Lucia Cichella, and Rolf Hennicker. "From Helena Ensemble Specifications to Executable Code". In: *Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers*. 2014, pp. 183–190 (cit. on p. 194).

[134] Annabelle Klarl and Rolf Hennicker. "Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework". In: *23rd Australian Software Engineering Conference, ASWEC 2014, Milsons Point, Sydney, NSW, Australia, April 7-10, 2014*. 2014, pp. 15–24 (cit. on p. 194).

[135] Annabelle Klarl. "From Helena Ensemble Specifications to Promela Verification Models". In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015, pp. 39–45 (cit. on p. 194).

[136] Rolf Hennicker, Annabelle Klarl, and Martin Wirsing. "Model-Checking Helena Ensembles with Spin". In: *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*. 2015, pp. 331–360 (cit. on p. 194).

[137] Immo Grabe, Mohammad Mahdi Jaghoori, Joachim Klein, Sascha Klüppelholz, Andries Stam, Christel Baier, Tobias Blechmann, Bernhard K. Aichernig, Frank S. de Boer, and Andreas Griesmayer. "The Credo Methodology - (Extended Version)". In: *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*. 2009, pp. 41–69 (cit. on p. 194).

[138] Farhad Arbab. "Reo: A Channel-based Coordination Model for Component Composition". In: *Mathematical. Structures in Comp. Sci.* 14.3 (June 2004), pp. 329–366 (cit. on p. 194).

[139] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. "Modeling component connectors in Reo by constraint automata". In: *Science of Computer Programming* 61.2 (2006). Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03), pp. 75 –113 (cit. on p. 194).

[140] Einar Broch Johnsen and Olaf Owe. "An Asynchronous Communication Model for Distributed Concurrent Objects". In: *Software and Systems Modeling* 6.1 (Mar. 2007), pp. 35–58 (cit. on p. 194).

[141] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild B. Torjusen. "Fundamentals of Software Engineering: Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers". In: ed. by Farhad Arbab and Marjan Sirjani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Executable Interface Specifications for Testing Asynchronous Creol Components, pp. 324–339 (cit. on p. 195).

[142] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J.F. Quesada. "Maude: specification and programming in rewriting logic". In: *Theoretical Computer Science* 285.2 (2002). Rewriting Logic and its Applications, pp. 187 –243 (cit. on pp. 195, 199).

[143] John Hatcliff, Xianghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems". In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. 2003, pp. 160–173 (cit. on p. 195).

[144] Robby, Matthew B. Dwyer, and John Hatcliff. "Bogor: An Extensible and Highly-modular Software Model Checking Framework". In: *SIGSOFT Softw. Eng. Notes* 28.5 (Sept. 2003), pp. 267–276 (cit. on p. 195).

[145] Claudio Demartini, Radu Iosif, and Riccardo Sisto. "dSPIN: A Dynamic Extension of SPIN". In: *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*. Vol. 1680. Lecture Notes in Computer Science. Springer, 1999, pp. 261–276 (cit. on p. 195).

[146] Marius Bozga, Susanne Graf, and Laurent Mounier. "Automated validation of distributed software using the IF environment". In: *Workshop on Software Model Checking (in connection with CAV '01)*. Ed. by Willem Visser Scott D. Stoller. Vol. 55. Electronic Notes in Theoretical Computer Science 3. Paris, France: Elsevier, July 2001, pp. 370–381 (cit. on p. 196).

[147] Sergio Yovine. "KRONOS: a verification tool for real-time systems". In: *International Journal on Software Tools for Technology Transfer* 1.1 (1997), pp. 123–133 (cit. on p. 196).

[148] Giuseppe De Ruvo and Antonella Santone. "An Eclipse-based Editor to Support LOTOS Newcomers". In: *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2014 IEEE 23rd International Conference on*. 2014, pp. 372–377 (cit. on p. 196).

[149] Hugues Evrard and Frédéric Lang. "Automatic Distributed Code Generation from Formal Models of Asynchronous Concurrent Processes". In: *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2015)*. Turku, Finland, Mar. 2015 (cit. on p. 196).

[150] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991 (cit. on p. 197).

[151] G.J. Holzmann and M.H. Smith. "An Automated Verification Method for Distributed Systems Software Based on Model Extraction". In: *IEEE Transactions on Software Engineering* 28.4 (2002), pp. 364–377 (cit. on p. 197).

[152] Gerard J. Holzmann and Dragan Bosnacki. "The Design of a Multicore Extension of the SPIN Model Checker". In: *IEEE Trans. Softw. Eng.* 33.10 (Oct. 2007), pp. 659–674 (cit. on p. 197).

[153] Doron Peled. "Combining Partial Order Reductions with On-the-fly Model-Checking". In: *Proceedings of the 6th International Conference on Computer Aided Verification*. CAV '94. London, UK, UK: Springer-Verlag, 1994, pp. 377–390 (cit. on pp. 197, 201).

[154] Moatz Kamel and Stefan Leue. "Validation of a Remote Object Invocation and Object Migration in CORBA GIOP using Promela/Spin". In: *International SPIN Workshop* (cit. on p. 197).

[155] M. Lowry K. Havelund and J. Penix. "Formal Analysis of a Space Craft Controller using Spin". In: *International SPIN Workshop* (cit. on p. 197).

[156] Gerald Lüttgen and Victor Carreño. "Analyzing Mode Confusion via Model Checking". In: *International SPIN Workshop* (cit. on p. 197).

[157] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking". In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, 2002 (cit. on p. 197).

[158] Kenneth L. McMillan. *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993 (cit. on p. 198).

[159] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "Integrating BDD-based and SAT-based Symbolic Model Checking". In: *Proc. "Frontiers of Combining Systems, FROCOS'02"*. Vol. 2309. LNAI. Santa Margherita, Italy: Springer, 2002 (cit. on p. 198).

[160] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. *Compositional Reasoning in Model Checking*. 1998 (cit. on pp. 198, 201).

[161] Enrico Giunchiglia and Armando Tacchella, eds. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Vol. 2919. Lecture Notes in Computer Science. Springer, 2004 (cit. on p. 198).

[162] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535 (cit. on p. 198).

[163] Bernd Kolb, Markus Völter, Daniel Ratiu, Domenik Pavletic, Kolja Dummann, and Tamás Szabó. *MBEDDR*. URL: `http://mbeddr.com/` (cit. on p. 198).

[164] *Two Towers*. URL: `http://www.sti.uniurb.it/bernardo/twotowers/` (cit. on p. 198).

[165] *Goanna*. URL: `http://redlizards.com/` (cit. on p. 198).

[166] José Vander Meulen and Charles Pecheur. "Efficient Symbolic Model Checking for Process Algebras". In: vol. 5596. LNCS. Springer, 2008 (cit. on p. 198).

[167] Santiago Escobar and José Meseguer. "Symbolic Model Checking of Infinite-state Systems Using Narrowing". In: *Proceedings of the 18th International Conference on Term Rewriting and Applications*. RTA'07. Paris, France: Springer-Verlag, 2007, pp. 153–168 (cit. on p. 199).

[168] Azadeh Farzan and José Meseguer. "Partial Order Reduction for Rewriting Semantics of Programming Languages". In: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA)*. Ed. by G. Denker and C. Talcott. 2006 (cit. on p. 201).

[169] Ludovic Henrio and Justine Rochas. “Declarative Scheduling for Active Objects”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1339–1344 (cit. on p. 208).

[170] Ludovic Henrio and Marcela Rivera. “Stopping safely hierarchical distributed components: application to GCM”. In: *CBHPC ’08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*. Karlsruhe, Germany: ACM, 2008, pp. 1–11 (cit. on p. 209).