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

Nuno Gaspar

Mechanized support for the formal specification, verification and execution of component-based applications

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

Soutenue le X Décembre 2014

Jury

Rapporteurs	Xyz Xyz	Affil Xyz
	Xyz Xyz	Affil Xyz
Examinateurs	Xyz Xyz	Affil Xyz
	Xyz Xyz	Affil Xyz
Directeur de thèse	Eric Madelaine	Université de Nice-Sophia Antipolis
Co-encadrant de thèse	Ludovic Henrio	Université de Nice-Sophia Antipolis

Future me, be happy, its done.

Résumé

GCM

Abstract

GCM.. GCM

Resumo

GCM

Acknowledgments

....

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

"Never permit a dichotomy to rule your life, a dichotomy in which you hate what you do so you can have pleasure in your spare time. Look for a situation in which your work will give you as much happiness as your spare time."

Pablo Picasso

Contents

This thesis belongs to the domain of formal methods. In particular, we focus their application on a specific methodology for the development of software: component-based engineering.

Let it be by means of automatic or interactive approaches, the goal of formal methods is to increase the confidence one can place on a software system.

Throughout this thesis, we discuss the application of formal methods techniques in the context of component-based engineering.

1.1 Component-based software engineering

Among all programming paradigms, component-based engineering stands as one of the most followed approaches for real world software development. Its emphasis on a clean separation of concerns makes it appealing for both industrial and research purposes. Furthermore, component-based systems are notorious for their capacity to address the inherent challenges of today's software development. These, promote modular designs, and therefore ease the burden of development and maintenance of applications. Moreover, portability and re-usability of components are further benefits of this paradigm.

There are several component models proposed in the literature [1, 2], each with their own particularities (hierarchical/flat, static/reconfigurable, ...). Yet, their foundation is generally made of three main ingredients: *components*, *interfaces* and *bindings*. A component can be seen as a *building block*, usually a piece of software code. An interface is an access point to/from components. Finally, a binding is a connection established between components, through their interfaces.

Indeed, component-based programming shares some resemblances with object-oriented programming. Their fundamental distinction lies at the fact that in the former paradigm, communications are always explicitly made through the component's interfaces, therefore making all existing dependencies evident.

Another facet of component-based systems concerns software evolution. This methodology of developing software enables *on-the-fly* structural reconfigurations of the architecture of the application. The advantage of modifying the software architecture at runtime comes from the need to cope with the plethora of situations that may arise in a potentially massively distributed and heterogeneous system. Indeed, the ability to restructure is also a key aspect in the field of *autonomic* computing where software is expected to adapt itself. However, this capacity comes with a price: we no longer need only to care about functional concerns, but also about structural ones.

The widespread use of component models together with the interesting chal-

lenges posed by reconfigurable component-based applications make it an exciting research topic for the formal methods community. Within our research group, we focus on the Grid Component Model (GCM) [2] to address the intricacies of grid and cloud computing. Details regarding its specification are discussed in Section 2.1.

1.2 Context — The Spinnaker project

This thesis occurs in the context of the Spinnaker project[1], a collaborative project between INRIA and several industrial partners, where we intend to contribute for the widespread adoption of Radio-Frequency Identification (RFID)-based technology. To this end, our contribution comes with the design and implementation of a non-intrusive, flexible and reliable solution that can integrate itself with other already deployed systems. Specifically, we developed The HyperManager, a general purpose monitoring application with autonomic features. This was built using GCM/ProActive[2] — a Java middleware for parallel and distributed programming that follows the principles of the GCM. For the purposes of this project, it had the goal to monitor the E-Connectware[3] (ECW) framework in a loosely coupled manner.

1.3 Contributions

In this thesis, we contribute to the state of the art in the domain of formal methods and (distributed) component-based systems.

Our first contribution is an industrial case study on the behavioural specification and verification of a reconfigurable distributed application. This promotes the use of formal methods in an industrial context, but also puts in evidence the necessity for alternative and/or complementary approaches in order to better address such undertakings.

[1] Project OSEO ISIS. `http://www.spinnaker-rfid.com/`

[2] `http://proactive.activeeon.com/index.php`

[3] `http://www.tagsysrfid.com/Products-Services/RFID-Middleware`

Our second contribution is a framework, developed with the Coq proof assistant, for reasoning on software architectures: Mefresa. This encompasses the mechanization of the GCM specification, and the means to reason about reconfigurable GCM architectures. Further, we address behavioural concerns by formalizing a semantics based on execution traces of synchronized transition systems. Overall, it provides the first steps towards a complete specification and verification platform addressing both architectural and behavioural properties.

Finally, our third contribution is a new Architecture Description Language (ADL), denominated Painless, and its proof-of-concept integration with the ProActive middleware. Painless allows to specify parametrized GCM architectures, along with their structural reconfigurations, in a declarative-like language. Compliance with the GCM specification is evaluated by certified functional code extracted from Mefresa. This permits the safe deployment and reconfiguration of GCM applications.

The following publications resulted from this thesis.

- Nuno Gaspar and Eric Madelaine. *Fractal á la Coq.* Conférence en Ingénierie du Logiciel, Rennes, France, June 2012.
- Nuno Gaspar, Ludovic Henrio and Eric Madelaine. *Bringing Coq into the World of GCM Distributed Applications.* International Journal of Parallel Programming, pp. 1-20, 2013.
- Nuno Gaspar, Ludovic Henrio and Eric Madelaine. *Formally Reasoning on a Reconfigurable Component-Based System — A Case Study for the Industrial World.* International Symposium on Formal Aspects of Component Software (FACS'2013), October 2013.

Moreover, a talk entitled *Formal Reasoning on Component-Based Reconfigurable Applications*, was given at the student session of the 40^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'2013).

1.4 Organisation of this thesis

The remaining of this thesis is organized as follows.

- Chapter 2 gives an overview of the **technical background** required for the understanding of this thesis. Pointers for relevant literature are indicated for the reader desiring further details.

- Chapter 3 discusses an **industrial case study** on the formal specification and verification of a reconfigurable GCM/ProActive application. This is achieved by formally defining its behavioural semantics and check it against the desired properties by means of model-checking techniques.

- Chapter 4 presents MEFRESA, a **framework for the reasoning on software architectures**. It is tailored for the GCM, and developed using the Coq proof assistant.

- Chapter 5 shows how we leverage the MEFRESA framework and Coq's certified extraction mechanism to provide an **extension to the GCM/ProActive middleware**. We propose a new approach for the specification of reconfigurable GCM architectures.

- Chapter 6 addresses the mechanization of a **behavioural semantics using the Coq proof assistant**. We show how we can interactively reason about transition systems. Further, we illustrate its use in the context of the GCM.

- Chapter 7 discusses relevant **related work** w.r.t. to the main contributions of this thesis.

- For last, chapter 8 discusses the **final conclusions** about this thesis, and indicates perspectives for future work and improvements.

Chapter 2

Preliminaries

"We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology."

Carl Sagan

This chapter discusses the required background for the reading of this thesis. Each of its constituents Sections are self-contained presentations to relevant material for its understanding. The reader already familiar with some of the Sections may opt to skip them.

Section 2.1 introduces the GCM component model. Its reference implementation is discussed in Section 2.2. Then, Section 2.3 presents a formalism for specifying behavioural semantics. Section 2.4 discusses the Fiacre specification language. For last, Section 2.5 provides an overview of the Coq proof assistant.

Contents

2.1 The GCM Component Model

Proposed by the CoreGrid European network of Excellence, the GCM [2] is based on the Fractal Component Model [1], with extensions addressing the issues of grid computing: deployment, scalability and asynchronous communications. Essentially, it benefits from Fractal's hierarchical structure, introspection capabilities, extensibility, and the separation between interfaces and implementation. In the following, we do not discriminate what is inherited from the Fractal component model and what is an extension. We discuss the whole features defining the GCM.

The main objectives of the GCM are the implementation, deployment, and management of complex and distributed systems. Indeed, there are several proposals for component models in the literature, yet, most suffer from limited support for extension, adaptation, and distribution. To this end, the GCM offers customizable communication patterns, transparent remote component access, component composition, introspection (i.e. monitoring of the running system), and (autonomic) (re)configuration capabilities. Moreover, the GCM intends to be suited for a wide range of application scenarios.

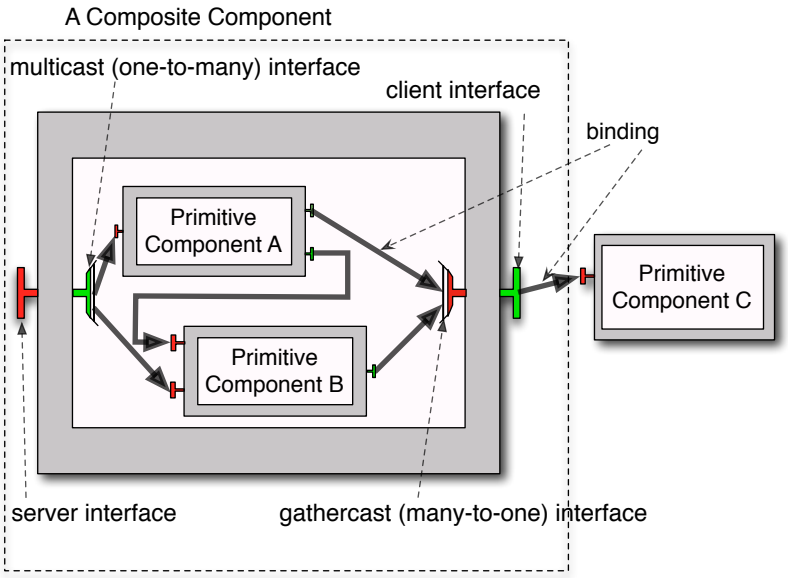

Figure 2.1: A simple GCM architecture

An example of a GCM architecture is depicted by Figure 2.1. Basically, the core elements of a GCM application are (1) *components*, which can be *composite* or *primitive* — depending on whether they have inner components or not —, (2) *interfaces*, and (3) *bindings*. As expected, interfaces act as access points, and bindings establish communications between components.

Both synchronous and asynchronous communications are supported. These, can be *one-to-one*, but also of collective nature: *one-to-many* (*multicast*), *many-to-one* (*gathercast*), and even *many-to-many*. Moreover, support for autonomic aspects are provided by means of non-functional interfaces.

In the remainder of this section we delve into the GCM intricacies and further discuss its characteristics that are relevant for the understanding of this thesis. For more details the interested reader is pointed to the GCM's technical specification [3].

2.1.1 Overview of the GCM specification

Component introspection In the GCM, components are endowed with introspection capabilities of its external features. One of the advantages of being a

hierarchical component model is the possibility to adjust the granularity of abstraction, i.e. components can be seen as white or black boxes. As a black box, one cannot see its internal structure, solely its external interfaces.

Each interface has a name. All external interfaces of one component must have distinct names, but no such restriction is imposed on interfaces belonging to different components. Moreover, an interface is also characterized by its *role*: it can be a *client* or *server* interface. The former emits operation invocations, while the latter receives them. Intuitively, one should see the client interfaces as the means for service methods to interact with other service methods.

Component's interfaces can be introspected through two different Application Programming Interface (API)s: the Component interface, and the Interface interface. The former specifies the methods getFcInterfaces() and getFcInterface(string iftName), that return an array of references to the component's interfaces, and a reference to the interface whose name matches the value itfName passed as argument, respectively. The latter, focuses on the introspection from an interface perspective. In particular, it specifies a getFcItfName() method, that returns a string with the name of the interface, and getFcItfOwner(), that returns the Component reference of the component to which it belongs.

Component configuration A GCM component, primitive or composite, is arranged by two parts: a *membrane* and a *content*. The former — see the grey parts of each component of Figure 2.1 — contains a set of controllers that expose non-functional interfaces. These allow to dynamically reconfigure the structure of a GCM application. The latter, is either an object — for primitive components — or a finite set of subcomponents — for composite components —, which are under control of the enclosing component's controllers.

A membrane can have interfaces of *external* and *internal visibility*. External interfaces are available from outside the component, while internal interfaces are accessible to the component's direct subcomponents. A component's interfaces can share the same name provided they have a different visibility. Moreover, interfaces are also classified regarding their *functionality*: they can be *functional* or *control*. The former corresponds to an interface providing or requiring a functionality to the component. Intuitively, it means it is part of the application logic. The

latter provides "non-functional" aspects, such as reconfiguration capabilities. This capacity to reconfigure the application's architecture at runtime is of paramount importance, and plays a key role in the realm of autonomic computing.

A *binding* is a communication path from a client interface to a server interface. There are three types of bindings. A *normal* binding (1) is established between two external interfaces from components at the same hierarchical level, i.e. possessing the same enclosing component. An *import* binding (2) is made from a (sub)component to its enclosing component. Last, an *export* binding (3) binds a component to one of its subcomponents. For the sake of clarity, these are depicted by Figure 2.2, Figure 2.3, and Figure 2.4, respectively.

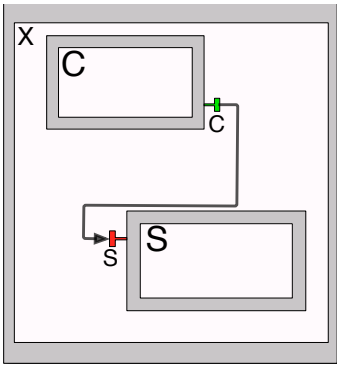

Figure 2.2: Normal Binding

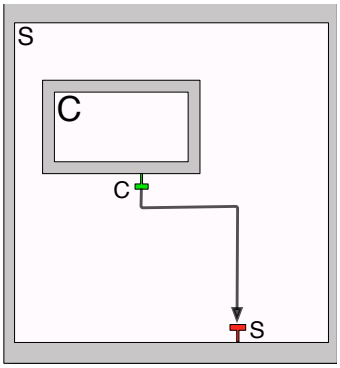

Figure 2.3: Import Binding

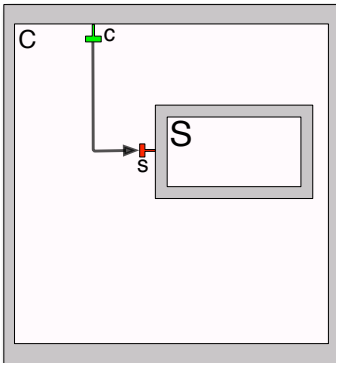

Figure 2.4: Export Binding

This can be seen as a structural constraint ensuring that no binding can "cross" a component boundary except through its interfaces. Further, a client interface can be bound to at most one server interface, while several client interfaces can be bound to the same server interface. This constraint is relaxed if the client interface is of *multicast cardinality*. Other cardinality values include *singleton*, for simple one-to-one communications, and *gathercast* that act as a *rendez-vous* for gathering data.

Component runtime instantiation In the GCM, component creation is achieved through *factories*. Basically, components are created by other special type of components called component factories.

The GCM offers a generic component factory and a standard component factory. The former allows the creation of several kinds of components, while the latter is more specific, it can only create components of the same type. For both factories, one can wonder: *if components are created from component factories, how are component factories created? From other component factories?* Indeed, this would lead to an infinite recursion. This is solved by the inclusion of a *bootstrap* component factory that needs not to be created explicitly. Naturally, it is able to create several kinds of components, namely component factories.

Component typing A simple type system is defined for composing components. It mainly reflects the characteristics of the component's interfaces. In particular it focuses on the *cardinality* and *contingency* attributes of an interface.

There are four possible values for an interface cardinality. Considering an interface of type **T**, it can be *singleton* (1), entailing that the component owning it must have exactly one interface of type **T**. *Collection* (2), indicating that an arbitrary number of interfaces of type **T** can coexist on a given component. However, their name must begin with the same name specified in **T**. Since there is an infinite number of such interfaces, they cannot be created at the same time, they must be created lazily, i.e., at invocation time. An interface may also be *multicast* (3): it is like a singleton, but transforms each invocation into a set of invocations. At last, an interface of *gathercast* (4) cardinality also behaves as a singleton, but transforms a list of invocations into a single invocation.

Regarding the interface's contingency attribute, an interface can be *optional* or *mandatory*. The operations of an optional interface are not guarantee to be available at runtime, while a mandatory interface provides such guarantee. Basically, mandatory interfaces are made for components absolutely requiring other components to work, whereas optional interfaces are useful for components that may use others, if present. For instance, a parser component absolutely needs a lexer component, but can work with or without a logger component.

2.1.2 The GCM ADL

One may need to define arbitrarily complex architectures. With the increase of complexity in the system to be described, it is important to have a precise way to describe such architectures. In the realm of component-based engineering, this is usually achieved by means of an ADL. The GCM follows this approach by supporting its own GCM ADL [4].

The GCM ADL is a XML-based language. As a root element it contains the *definition* element. Basically, it describes the structure of the application via *component*, *interface* and *binding* elements. The component element possesses a *name* and *definition* attribute. These identify the component, and its owner, respectively. Moreover, it may contain the following child elements:

- *comment*: a free form of text documenting the component. (0-unlimited);
- *interface*: description of the interface provided by the component. (0-unlimited);
- *component*: reference to a subcomponent. (0-unlimited);
- *binding*: description of the binding hold by the component. (0-unlimited);
- *content*: class which represents the components. (0-1);
- *attributes*: list of attributes of the component. (0-1);
- *controller*: controller of the component. (0-1);
- *virtualNodes*: list of virtual nodes the component should be deployed on. (0-1);
- *exportVirtualNodes*: list of export virtual nodes. (0-1) .

A *comment* element is a simple text element, adding some contextual information. An *interface* element possesses the following attributes:

- *name*: identifier of the interface. (required);
- *role*: 'client' or 'server'. (required);

- *signature*: signature of the interface. (required);
- *contingency*: 'mandatory' or 'optional'. (optional);
- *cardinality*: 'singleton', 'collection', 'gathercast' or 'multicast'. (optional);
- *comment*: a free form of text documenting the interface. (0-unlimited).

Its constituting attributes should pose no doubt. However, the careful reader may notice that there is no attribute specifying whether an interface is of internal or external visibility. This must be handled in an automated manner. For instance, a multicast server interface of a composite component is in fact the composition between a server interface and a multicast internal client interface.

Next, the component child element of the component element portrays GCM's hierarchical nature. The binding element is solely composed by two attributes, *client* and *server*, which hold the name of the client and server interfaces involved in the binding, respectively. The content element features solely the *class* attribute that specifies the class implementing the component.

The *attributes* element are key/value pairs that can be used to parametrize the component. The *controller* element indicates the component controller that manages the component w.r.t. non-functional aspects. Finally, the remaining elements *virtualNodes* and *exportVirtualNodes* serve the purpose of facilitating the deployment task, i.e., one may want to distribute its application to a set of predefined nodes.

This Subsection discussed the main ingredients of the GCM ADL. For a more detailed description of its intricacies, namely its XML schema, the interested reader is pointed to its technical specification [4].

2.2 ProActive — A middleware for distributed programming

ProActive[1] is an open source Java middleware for the programming of multi-threaded, parallel, and distributed applications. An application adopting ProAc-

[1] Available online: `http://proactive.activeeon.com/index.php`

tive is composed by entities denominated *active objects* [5]. Each active object contains a distinguished element, the *root*, that is its only entry point. Moreover, it possesses its own thread of control, every incoming method call is automatically placed in a queue of pending requests, and is able to decide by which order it serves them.

Further, incoming method calls are asynchronous with transparent *future* objects. Basically, a short *rendezvous* occurs at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the callee. Then, if the invoked method returns a result, a future object is created on the callee side and returned back as a result to the caller. Thus, a future can be seen as a placeholder for the return value of an active object method invocation. The object that issued the invocation can therefore proceed its execution. At a later stage, if it needs to read the actual returned value, it blocks until the value is available — i.e. until the request is processed and returned back from the callee — or continues transparently if meanwhile the value has already been obtained. This mechanism is called *wait-by-necessity*.

Another common nomenclature found in the literature for the future concept is *promise* [5]. Both terms are often used interchangeably.

2.2.1 GCM/ProActive — a reference implementation for GCM

As mentioned earlier, ProActive is the *de facto* reference implementation for GCM. For this reason, whenever focusing on its component model part, it is often referred to by GCM/ProActive.

In GCM/ProActive a primitive component is implemented through an active object, and thus includes its standard features. Components communicate asynchronously through the established bindings between their interfaces.

For instance, let us consider two components of some architecture as depicted by Figure 2.5. Components **A** and **B** both possess a server interface (in red), and the former is also equipped with a client interface (in green). A binding between **A**'s client interface and **B**'s server interface connects them together. Moreover, they both include a request queue.

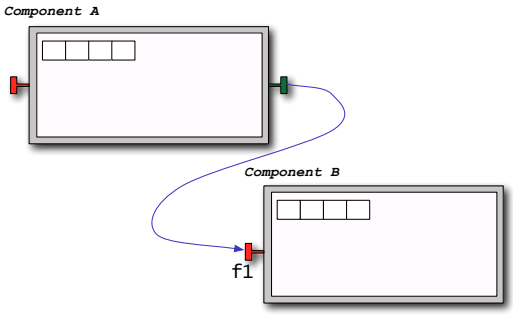

Figure 2.5: Components **A** and **B**

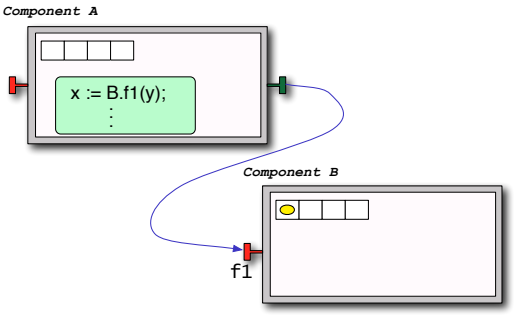

Figure 2.6: **A** invokes $\mathbf{f}_1$ from **B**

Whenever **A** invokes the method $\mathbf{f}_1$ from **B**'s server interface, a request is inserted into **B**'s queue. At this point, the variable **x** holds a future for the reply of $\mathbf{f}_1$, and **A** proceeds its execution (Figure 2.6).

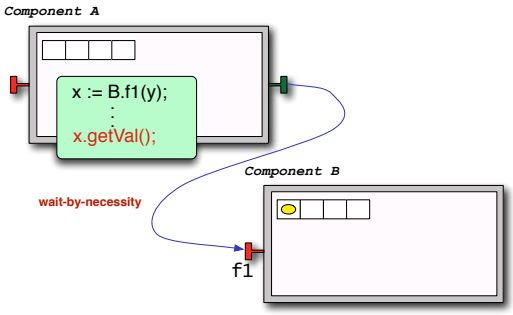

Figure 2.7: **A** blocks! *wait-by-necessity*

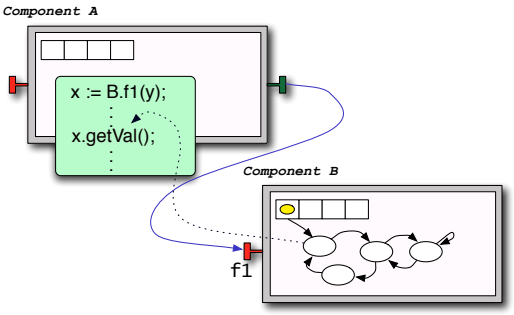

Figure 2.8: **B** replies, and **A** is happy

If **A** needs to access the value of **x**, then it may be the case that **B** has not yet treated the request, and thus **A** blocks until it gets the reply from $\mathbf{f}_1$'s invocation (Figure 2.7). Alternatively, it may be the case that **B** already replied, and therefore **A** can proceed its execution transparently (Figure 2.8).

As expected, GCM components are mono-threaded, i.e. only one request is treated at a time. It may seem restrictive, but it has the benefit of ensuring thread-safety. Nevertheless, we may refer that recent work aims at improving performance by allowing multi-threading for active objects [6].

2.2.2 Specifying architectures with the ADL

As discussed in Subsection 2.1.2, the GCM defines its ADL as a means to specify component architectures. GCM/ProActive includes support for such feature. For instance, let us consider the simple GCM application depicted by Figure 2.9.

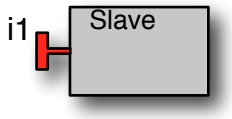

Figure 2.9: Simple GCM application example

It is composed solely by one primitive component, named *Slave*, possessing one server interface named i_1. Listing 2.1 illustrates the specification of this architecture.

```
1 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
    EN" "classpath://org/objectweb/proactive/ core/component/adl/xml/
    proactive.dtd">
2 <definition name="org.objectweb.proactive.examples.userguide.
    components.adl.starter.adl.Slave">
3 <interface signature="org.objectweb.proactive.examples.userguide.
    components.adl.starter.Itf1" role="server" name="i1">
4 <content class="org.objectweb.proactive.examples.userguide.components
    .adl.starter.SlaveImpl"/>
5 <controller desc="primitive"/>
6 </definition>
```

Listing 2.1: GCM/ProActive *slave* component ADL

The contents of the ADL should pose no doubt. Line 1 concerns *XML* validation aspects, basically it allows standard validators to check if the *XML* file is well formed. The component architecture aspects *per se* begin at line 2. Its *definition* possesses the *name* attribute indicating its classpath, and contains three child elements: *interface*, *content* and *controller*. The interface's signature points to the *java* interface file holding signature for the supported service methods. Next, it is specified as having a server *role* and with name i_1 (line 3). The *content* holds the *class* attribute that indicates the actual *java* implementation for this component.

i.e. it implements the service methods specified in the interface's *signature* (line 4). At last, the *controller* specifies the default primitive component controller (line 5). Moreover, no information is provided regarding *virtual nodes*. This is usually the case when deploying the application locally.

Let us now look at a more elaborated example. Figure 2.10 depicts a slightly more complex architecture featuring three components: *Composite*, *Master* and the previously discussed *Slave*.

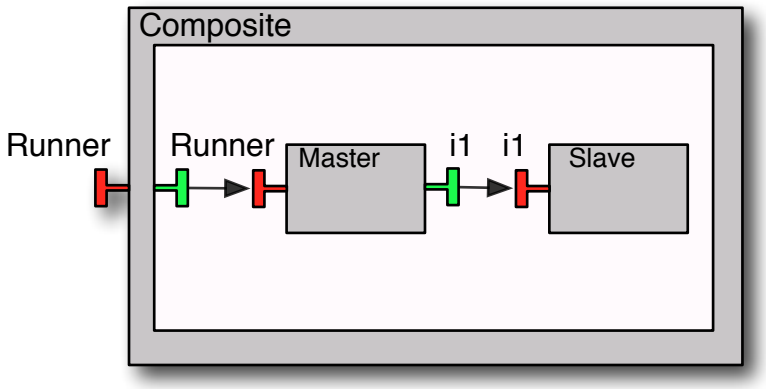

Figure 2.10: GCM *composite* application example

The *Composite* encapsulates the two remaining components and exposes one external server named *Runner*. Its symmetric counterpart, i.e. internal and client, is bound to the server interface of the *Master* component, also named *Runner*. Further, the *Master* component is also equipped with a client interface named i_1 that is bound to the *Slave*'s server interface. Listing 2.2 shows its ADL.

```
1 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
    EN" "classpath://org/objectweb/proactive/core/component/adl/xml/
    proactive.dtd">
2
3 <definition name="org.objectweb.proactive.examples.userguide.
    components.adl.composite.adl.Composite">
4   <interface signature="org.objectweb.proactive.examples.userguide.
      components.adl.composite.Runner" role="server" name="runner"/>
5
6   <component name="Master" definition="org.objectweb.proactive.
      examples.userguide.components.adl.composite.adl.Master"/>
```

```
7   <component name="Slave" definition="org.objectweb.proactive.
        examples.userguide.components.adl.composite.adl.Slave"/>
8
9   <binding client="this.runner" server="Master.runner"/>
10  <binding client="Master.i1" server="Slave.i1"/>
11
12  <controller desc="composite"/>
13 </definition>
```

Listing 2.2: GCM/ProActive *Composite* component ADL

Line 3 starts the *definition* of this *Composite* by indicating its classpath, and including the child elements relevant for this specification. As mentioned above, the GCM ADL does not include an attribute for specifying whether an interface is of external or internal visibility. Indeed, there is only one *interface* element included in the *composite* ADL (line 4). Its symmetric counterpart is not specified and automatically handled by GCM/ProActive. Next, the *component* elements at lines 6 and 7 are references to their actual ADLs. Lines 9 and 10 specify the two bindings of this architecture using the familiar **this** and **.** notations. For last, line 12 specifies the *controller* as the default composite component controller.

The remaining pieces for the specification of the *Composite* architecture concerns the ADLs of its two subcomponents: *Master* and *Slave*. The *Slave* architecture has already been discussed (see Listing 2.1), its only modification concerns the classpath of its *name* definition. Regarding the *Master* component, its ADL is depicted by Listing 2.3.

```
1 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
    EN" "classpath://org/objectweb/proactive/core/component/adl/xml/
    proactive.dtd">
2
3 <definition name="org.objectweb.proactive.examples.userguide.
    components.adl.composite.adl.Master">
4   <interface signature="org.objectweb.proactive.examples.userguide.
      components.adl.composite.Runner" role="server" name="runner"/>
5   <interface signature="org.objectweb.proactive.examples.userguide.
      components.adl.composite.Itf1" role="client" name="i1"/>
6
7   <content class="org.objectweb.proactive.examples.userguide.
```

```
        components.adl.composite.MasterImpl"/>
8
9   <controller desc="primitive"/>
10 </definition>
```

Listing 2.3: GCM/ProActive *Master* component ADL

As expected, it follows the same structure as the *Slave* ADL. Indeed, it mainly differs on the inclusion of a client interface (line 5).

In this Section we only covered the fundamentals of the ProActive middleware. The interested reader is pointed to ProActive's programming manual for more discussion on its intricacies [7].

2.3 pNets: A formalism for defining behavioural semantics

pNets stands for *parametrized Network of synchronized automata*. It provides an intermediate and general formalism aimed at the specification and synchronisation of the behaviour of a set of automata. Further, it is mainly aimed towards two goals: provide a formal foundation to the model generation principles for distributed component frameworks — such as GCM —, and to propose an expressive, yet concise, machine-oriented model that can be used as an internal format for software tools.

The first definition of pNets was published in [8]. Here, we restrict ourselves to a more succinct discussion, focusing on the material relevant for the understanding of this thesis.

A pNet is a hierarchical structure whose leaves are *pLTS*s or *queues*. A pLTS is a labelled transition system with variables. It can possess guards and assignments on its transitions. It is formally defined as follows.

Definition 2.3.1 (pLTS)**.** *A pLTS is a parametrized LTS defined by a tuple* $pLTS \triangleq (P, S, s_0, L, \rightarrow)$ *where*

- P *is a finite set of parameters, from which we construct the term algebra* $\mathcal{T}_P$*, with parametrized actions* $\mathcal{A}_P$*, the parametrized expression* $\mathcal{E}_P$*, and the boolean expressions* $\mathcal{B}_P$*.*

- *S is a set of states. For each state $s \in S$, variables of s are global to the pLTS.*
- *$s_0 \in S$ is the initial state.*
- *L is the set of labels of the form $(\alpha, e_b, (x_j := e_j)^{j \in J})$, where $\alpha \in \mathcal{A}_P$ is a parametrized action, $e_b \in \mathcal{B}_P$ is a guard, and the variables $x_j \in P$ are assigned the expressions $e_j \in \mathcal{E}_P$.*
- *$\rightarrow \subseteq S \times L \times S$ is the transition relation.*

Before proceeding to the remaining definition of a pNet structure, let us define the *sort* of a pNet. Basically, it is its signature: the set of actions it can perform, that is, the set of labels of its transitions. We define it by the following function $Sort : pNet \rightarrow \mathcal{A}_P$.

$$Sort(p) = \begin{cases} L & \text{for } p = (P, S, s_0, L, \rightarrow) \\ \{?Q\_m_i \mid m_i \in \mathcal{M}\} \cup \{!Serve\_m_i \mid m_i \in \mathcal{M}\} & \text{for } p = Queue(\mathcal{M}) \\ L & \text{for } p = (P, L, pNet_i^{k \in K}, SV_k^{k \in K}) \end{cases}$$

Its definition should pose no doubt. For the cases of being a leaf, if it is either a pLTS or a queue. For the former it simply returns its set of labels L. For the latter it returns a set of actions depending on $\mathcal{M} \subseteq \mathcal{T}_P$. Basically, a $Queue(\mathcal{M})$ can be seen as an infinite pLTS modelling the behaviour of a First In, First Out (FIFO) queue. Its semantics is simply the en-queueing and de-queueing of $m_i \in \mathcal{M}$. The last case concerns the recursive constructor of the pNet structure, where we also simply return its set of labels L.

Let us now see a formal definition for pNet.

Definition 2.3.2 (pNet)**.** *A pNet is a hierarchical structure defined by $pNet \triangleq pLTS \mid Queue(\mathcal{M}) \mid (P, L, pNet_i^{i \in \mathcal{I}}, SV_k^{k \in K})$ where*

- *P is a finite set of parameters, from which we construct the term algebra $\mathcal{T}_P$, with parametrized actions $\mathcal{A}_P$.*
- *$L \subseteq \mathcal{A}_P$ is the set of labels actions of the pNet.*
- *$\mathcal{I} \in \mathcal{I}_P$ is the set of over which sub-pNets are indexed, $\mathcal{I} \neq \emptyset$.*

- $pNet_i^{i \in \mathcal{I}}$ *is the family of sub-pNets.*
- $SV_k^{k \in K}$ *is a set of synchronization vectors* ($K \in \mathcal{I}_P$). $\forall k \in K, SV_k = \alpha_j^{j \in J_k} \rightarrow \alpha'_k$. *Each synchronization vector verifies:* $\alpha'_k \in L, J_k \in \mathcal{I}_P, \varnothing \subset J_k \subseteq I$, *and* $\forall j \in J_k . \alpha_j \in Sort(pNet_j)$.

Being a hierarchical structure, a pNet composes sub-pNets and expresses by its synchronization vectors how the sub-entities are synchronized. For instance, $SV_k = \alpha_j^{j \in J} \rightarrow \alpha'_k$ means that each sub-pNet can perform the action α_j, resulting in a global action labelled α'_k.

2.3.1 Behavioural semantics for GCM/ProActive applications

Section 2.1 presented the GCM, and its reference implementation was discussed in Section 2.2. Here, we show how to use pNets to give a formal semantics to GCM/ProActive applications.

As an illustrative example, the internals of a GCM primitive component featuring three service methods — m_1, m_2 and m_3 — and two client methods — m_4 and m_5 — are depicted by Figure 2.11.
Invocation on service methods — $\mathsf{Q\_m}_{i,\ i \in \{1,2,3\}}$ — go through a Queue, that dispatches the request — Serve_m* — to the Body. Serving the request consists in performing a Call_m* to the adequate service method, represented by the $\mathcal{M}_i$ boxes in the figure. Once a result is computed, a synchronized R_m* action is emitted. This synchronization occurring between the service method and the Body stems from the fact that GCM primitive components are mono-threaded. Moreover, the careful reader will notice the $fid_{i,\ i \in \{1,2,3\}}$ in the figure. These are the previously discussed *futures* (see Section 2.2), and act as promises for replies, leveraging asynchrony between components.

Service methods interact with external components by means of client interfaces. This requires obtaining a proxy — GetProxy_m*, $\mathsf{New\_m}_{i,\ i \in \{4,5\}}$ — in order to be able to invoke client methods — $\mathsf{Q\_m}_{i,\ i \in \{4,5\}}$. The reply — $\mathsf{R\_m}_{i,\ i \in \{4,5\}}$ — goes to the proxy used to call the external component. Then, a GetValue_-$\mathsf{m}_{i,\ i \in \{4,5\}}$ is performed in order to access the result in the method being served.

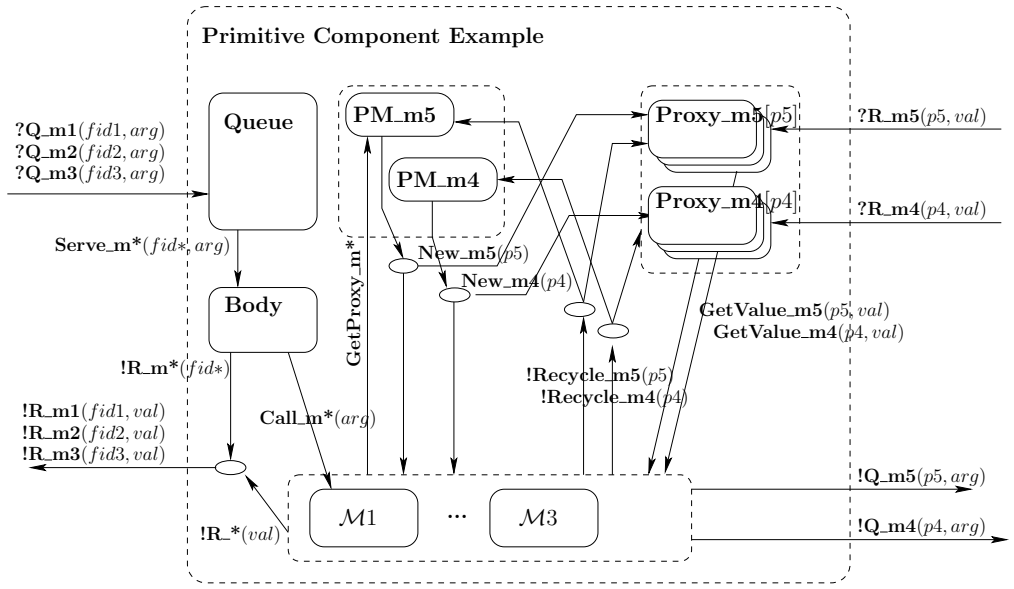

Figure 2.11: pNet representing a primitive component

Finally, Recycle_m$_{i,\ i\in\{4,5\}}$ actions can be performed in order to release the proxies.

The behaviour of the Queue and the Body elements should pose no doubt. The former acts as a queue with a FIFO policy, raising an exception if its capacity is exceeded. The latter dispatches the requests to the appropriate method and awaits its *return*, thus preventing the service of other requests in parallel.

The handling of proxies however, is not as straightforward and deserves a closer look. Figures 2.12 and 2.13 illustrate the behaviour of the Proxies and Proxy Managers, respectively. Upon reception of a New_m$_i$ action, a Proxy waits for the reply of the method invoked with it — R_m —, making thereafter its result available — GetValue_m. As soon as the reply is received, the Proxy can potentially be recycled through a Recycle_m action.

The behaviour of the Proxy Manager is slightly more elaborated. It maintains a *pool* of proxies, keeping track of those available and those already allocated. On the reception of a GetProxy_m action, it activates a new proxy — New_m — if there is one available. Should that not be the case, an Error(NoMoreProxy) action is emitted. As expected, a Recycle_m action frees a previously allocated proxy.

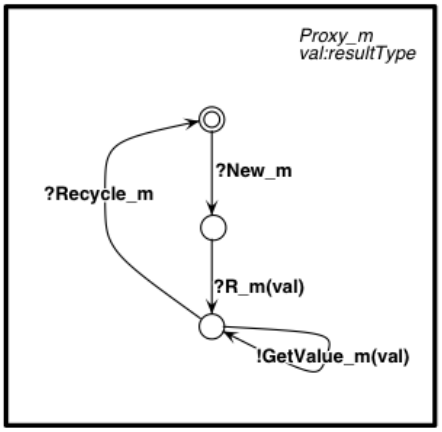

Figure 2.12: Behaviour of proxy

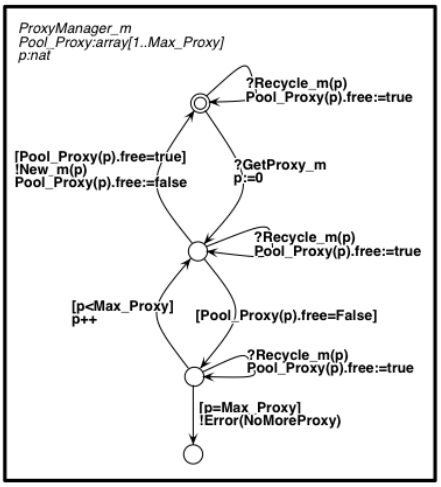

Figure 2.13: Behaviour of the proxy manager

2.3.2 Coping with structural reconfigurations

As mentioned in Section 2.1, the GCM also contemplates non-functional aspects such as structural reconfigurations. This means that the architecture of the application can evolve at runtime (e.g. by establishing new bindings, removing existing ones, ...).

For GCM applications *bind* and *unbind* operations are handled by the component owning the *client* interface that is supposed to be reconfigurable. This should come as no surprise, indeed, it follows the same spirit as in object-oriented languages: an object holds the reference to a target object; it is this object that must change the reference it holds. Moreover, we recall that client interfaces can be of singleton or multicast cardinality. The former is bound to at most one server interface, while the latter needs to potentially handle several recipients. Therefore, their reconfiguration is dealt in different manners.

Let us first illustrate how a reconfigurable client singleton interface is modelled in *pNets*. A component is equipped with a *binding controller* interface to bind and unbind its client singleton interfaces. As depicted by Figure 2.14, the binding controller pLTS attached to each of them controls their bindings.

Indeed, we allow for reconfigurations by defining two new request messages for the *binding* and *unbinding* of interfaces. These are delegated to a binding controller

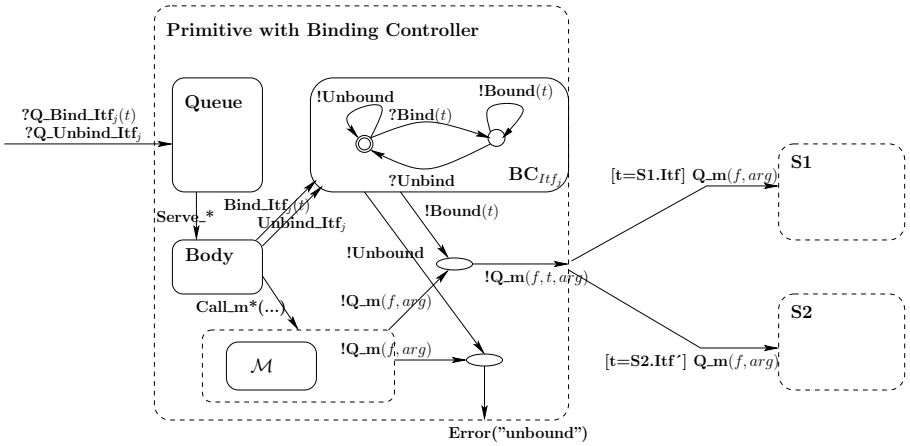

Figure 2.14: Binding controller

that upon method invocation over these reconfigurable interfaces will check if they are indeed bound, emitting an error if it is not the case. Moreover, the target of the invocation is decided by checking its passed reference. For this reason one must know statically what are the possible target interfaces that a reconfigurable interface can be bound too.

Reconfigurations involving multicast interfaces require keeping track of a set of recipients. Figure 2.15 depicts the pNet specificities concerning the handling of such interfaces.

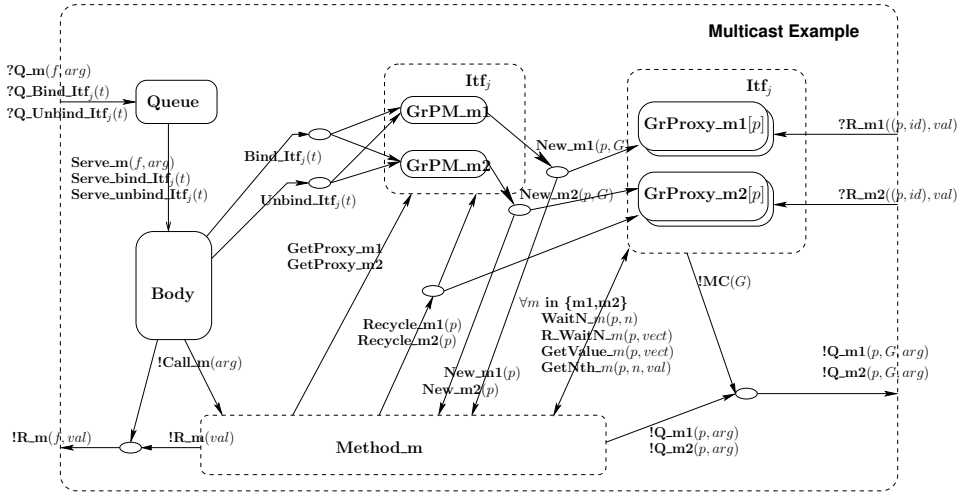

Figure 2.15: pNet example for reconfigurable multicast interface

In short, the machinery involved for dealing with this kind of interfaces mainly differs from reconfigurable singleton interfaces in that we must keep track of the target's connectedness status. Indeed, the emission of a new proxy — $\mathsf{New\_m}_{i,i\in\{1,2\}}$ — is synchronized in a similar manner, however we also transmit the current status of the multicast interface (i.e. the G variable in the figure). This status will be taken into account when invoking one of the client methods — $\mathsf{Q\_m}_{i,i\in\{1,2\}}$. In practice, G is a boolean vector whose element's valuation determine the interface's connectedness.

In this section we succinctly presented pNets mainly focusing on its application to the formal specification of GCM applications. It should be noted however, that pNets is also suited for the specification of other types of systems. For a more detailed account on its intricacies the interested reader is pointed to its authoritative reference [9].

2.4 The Fiacre specification language

Fiacre [10] stands for *Format Intermédiaire pour les Architectures de Composants Répartis Embarqués*. It is a formal language that allows to model systems, in particular, embedded and distributed systems, for formal verification and simulation purposes.

It is built around two notions: *processes* and *components*. The former describes the behaviour of sequential components. It is defined by a set of states, each associated with a program made of standard programming language constructs (loops, if-then-else, ...), non-deterministic constructs, communication events, and jumps to subsequent states. The latter describes the composition of processes. It is defined as a parallel composition of components and/or processes communicating through ports and shared variables.

For the sake of clarity, let us see a simple example[2]. First, we define an *union* type as depicted by Listing 2.4.

```
1 type Order is union
2     makeSandwich
```

[2]The described example is a slight adaptation from the online Fiacre tutorial: `http://projects.laas.fr/fiacre/doc/fiacre-tutorial.html`

```
3  | makeCoffee of 0..2
4  end
```

Listing 2.4: The Order type

Basically, the above Order type encodes two types of orders: make a sandwich (line 2), that takes no parameter as we assume there is only one kind of sandwich, and to make coffee (line 3), that takes a value ranging over [0..2] symbolizing that there are three kinds of coffee.

Let us now see a first Fiacre process. Listing 2.5 encodes the beginning of the Slave process.

```
5  process Slave [ready : none, order : in Order] is
6      states startSlavery, waitOrder,
7             makingLatte, makingEspresso, makingCapuccino,
8             makingSandwich
9      var tmp : Order
```

Listing 2.5: The Slave process (part 1/2)

The Slave process takes two parameters, often called *labels*. As we shall see later, these are used to make processes communicate together, or more precisely, *synchronize*.

At line 5, we can see that the first parameter, ready, is declared with type none. It is a predefined type meaning that no value can be attached to it. The second parameter, order, is of Order type. Moreover, the in keyword stands for the fact that it can only receive orders, and not emit them.

From line 6 to line 8 we define all the states of the Slave process. Further, a local variable, tmp, of type Order, is defined at line 9. We can now proceed to the the process behaviour. Listing 2.6 depicts its definition.

```
10 from startSlavery
11   ready;
12   to waitOrder
13
14 from waitOrder
15   order?tmp;
16   case tmp of
17    makeCoffee(0) -> to makingLatte
18  | makeCoffee(1) -> to makingEspresso
```

```
19  | makeCoffee(2) -> to makingCapuccino
20  | makeSandwich -> to makingSandwich
21 end
22
23 from makingLatte
24   to startSlavery
25
26 from makingEspresso
27   to startSlavery
28
29 from makingCapuccino
30   to startSlavery
31
32 from makingSandwich
33   to startSlavery
```

Listing 2.6: The Slave process (part 2/2)

Its understanding should pose no doubt, as the Fiacre specification language is rather intuitive and concise.

From its initial state, startSlavery, upon reception of a ready message (lines 10-11), the process proceeds to the waitOrder state (line 12). From there, it awaits for an order message, and stores its value on the tmp variable (lines 14-15). Then, depending on the value of tmp, it proceeds to the corresponding state (lines 16-21). Indeed, the remaining definition of the Slave process is rather trivial: from any attained state, it simply returns to the initial state startSlavery.

Let us now see the Master process. It is depicted by Listing 2.7.

```
34 process Master [ready : none, order : out Order] is
35     states start, ordering
36
37     from start
38         ready;
39         to ordering
40
41     from ordering select
42           order!makeCoffee(0)
43       [] order!makeCoffee(1)
44       [] order!makeCoffee(2)
45       [] order!makeSandwich
```

```
46        end;
47        to start
```

Listing 2.7: The Master process

Its definition is straightforward. From its initial state, start, it synchronizes with the Slave process through the label ready, and proceeds to the state ordering (lines 37-39). Then, it non-deterministically selects one order to emit (lines 41-46), and finally, returns to the initial state (line 47).

The last step is to specify how these two processes synchronize. This is depicted by Listing 2.8.

```
48 component CruelWorld is
49     port ready : none,
50          order : Order
51
52    par * in
53       Slave  [ready, order]
54    || Master [ready, order]
55    end
56 CruelWorld
```

Listing 2.8: Process synchronization

Labels are declared in components with the port keyword (lines 49-50). Then, we simply indicate the process synchronization (lines 52-55).

This simple example should be sufficient to provide an insight on the Fiacre specification language. For more details on its semantics the interested reader is pointed to [10].

2.5 A brief overview of the Coq Proof Assistant

The Coq Proof Assistant [11] is a system that implements the Calculus of Inductive Constructions [12] that itself combines both a *higher-order logic* and a functional programming language. In short, it goes beyond the capabilities of standard programming languages/environments by allowing the programmer to write logical definitions, write functions, and develop proofs involving these definitions and functions.

Indeed, this is also the case for other interactive theorem provers such as Isabelle/HOL [13] and PVS [14]. However, Coq distinguishes itself from these systems in one fundamental aspect: it is based on a *intuitionistic* logic[3]. This means that the *law of excluded middle* — for any proposition $\mathcal{P}$, $\mathcal{P} \vee \neg\mathcal{P}$ holds — is not assumed as an axiom. In practice, this entails that whenever trying to prove a formula $\mathcal{P} \vee \neg\mathcal{P}$ we need to show which one from $\mathcal{P}$ or $\neg\mathcal{P}$ actually holds.

In the remaining of this section we delve into Coq's realm and discuss its relevant features for the understanding of this thesis.

2.5.1 Data types

Coq comes with very little features built-in. For instance, the customary data types *boolean*, *integer*, *string*, etc, are not intrinsic to the system, but defined in the standard library. The point here is that even these canonical data types can be seen as ordinary user code.

For instance, in Coq natural numbers are inductively defined as follows:

```
1 Inductive nat : Set :=
2     O : nat
3   | S : nat -> nat
```

An inductive type is defined by a number of constructors, each with its own arity. For `nat`, these are `O`, with arity equal to zero, and `S`, with arity equal to one. Intuitively, it means that a natural number is either *zero* or the *successor* of some natural number. The above simple definition is therefore enough to model infiniteness of natural numbers.

A value of an inductive type is a composition of its constructors. Thus, the expression `S (S (S O))` is indeed a valid expression representing the natural number 3. In fact, the syntactical objects `0`, `1`, `2`, ... are seen as mere notations by Coq. Their sole purpose is to improve readability.

In a similar fashion, the standard library also provides a definition for the list data type.

```
1 Inductive list (A : Type) : Type :=
2   nil : list A
```

[3] Also referred also *constructive logic* in the literature.

```
3 | cons : A -> list A -> list A
```

A list is also defined by two constructors: nil and cons. These, possess arity zero and two, respectively. The former represents an empty list — and thus needs no argument —, while the latter a non-empty list. More precisely, cons h tl should be read as a list with *head* element h, and tl as its *tail* list.

The careful reader may wonder about the parameter A of the list data type. This stems from the fact that this definition is *polymorphic*. Indeed, it defines a list without explicitly stating the type of its elements. The sole implicit constraint is that they all possess the same type.

As an example, the expression cons 2 (cons 1 (cons 0 nil)) stands for a list of naturals, whose elements are 2, 1 and 0. For the sake of readability, there are also notations for easing the definitions of lists. For instance, the expression (2 :: 1 :: 0 :: nil) is equivalent to the one above.

2.5.2 Functional programming in Coq

Having understood the basics of Coq's data types, we can now proceed to writing functions over these data types.

Adding natural numbers can be seen as rather *intriguing*. This is directly related to how natural numbers are defined in Coq. The implementation of the plus function is given below.

```
1 Fixpoint plus (n m : nat) {struct n} : nat :=
2   match n with
3   | 0 => m
4   | S p => S (plus p m)
5   end
```

The first thing to note is the use of the keyword Fixpoint. This means that we are defining a recursive function. Looking at its signature we see that the function plus takes parameters n and m, both explicitly specified as being of type nat. Then, we find a rather suspicious *{struct n}*. In Coq, one can only write terminating functions. However, in general, determining whether a function terminates is *undecidable*. The struct construct specifies the argument that structurally decreases at each recursion step. For the plus function, every recursive call is applied on

p, which is a strict sub-term of S p. Thus, this reduction is monotone and will inevitably reach the O constructor, at which point the function terminates. Moreover, pattern-matching is exhaustive — all nat constructors are contemplated —, and therefore all cases are always covered. Indeed, the actual algorithm of addition is a bit surprising. If n is equal to zero we can simply return m. Otherwise, it must be the case that n is the successor of some number p, and we can simply return the successor of the addition between p and m.

For last, the signature contains the return type. As expected, the plus function returns a nat. It should be noted that Coq's type system is powerful enough to automatically infer all this typing information. The (unnecessary) extra verbosity in this example is included for the sake of explanation. In the following we shall omit typing information whenever it is obvious from the context.

Defining functions over lists follow the same principle. For instance, the rather common function computing the length of a list is defined as follows.

```
1 Fixpoint length (A : Type) (l : list A) : nat :=
2   match l with
3   | nil => 0
4   | cons h tl => S (length tl)
5   end.
```

Basically, at each recursion step we define the length of the list as being the successor of the length of its tail. The recursion is performed on the tail, and thus a strict sub-term. Eventually, we reach an empty list, at which point we simply return zero, and the function terminates.

Another peculiar aspect of Coq is that every defined function must be *total*, i.e. a function that is defined for all inputs. This seems a strong restriction as there are several functions that are *partial* by nature. For instance, a function returning the head of a list: what can be returned in the input list is empty? In standard programming languages, one usually handles such situations by raising an exception, but this is not possible in Coq.

One way to deal with these cases is through the option type.

```
1 Inductive option (A : Type) : Type :=
2   Some : A -> option A
3 | None : option A.
```

The best way to understand its use is to see it in action. Below, we exemplify it with the head function. Notice that we use the Definition keyword, and not Fixpoint. This is due to the fact that this function is not recursive.

```
Definition head (A:Type) (l: list A) : option A :=
  match l with
      nil       => None
  | cons h t => Some h
  end.
```

Two other approaches to cope with partial functions are the use default values, and *dependent types*. The former consists simply in associating a predefined return value for the "undesired" inputs. Clearly, this solution is not ideal, and sometimes even infeasible. The latter represents a more powerful feature of Coq. Basically, a dependent type is a type that depends on a value. For instance, we can define the vector type, that characterizes lists of a specific length.

```
Inductive vector (A : Type) : nat -> Type :=
| Vnil : vector A 0
| Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Such type definition ensures that a list of type (vector n) has always length **n**. This facilitates the definition of partial functions such as the one returning the head of a vector.

```
Definition vhead (A:Type) (n:nat) (v: vector A (S n)) :=
match v with
  Vcons el n v' => el
end.
```

Note that we need not to cope with the case of a empty vector, as this is directly discarded by typing! Indeed, dependent types are an extremely powerful feature. In the context of this thesis however, we do not make an extensive use of it. The interested reader is pointed to Chlipala's excellent book [15] for a more in-depth account on the subject.

2.5.3 Proving properties

Another main ingredient of Coq is about proving properties. First, we need to know how to define predicates: basically, they can be inductively defined in the

same manner as data types, except that their *sort*, i.e. type of their type, is Prop. As an example, consider a definition regarding the *parity* of a natural number.

```
1 Inductive is_even (n:nat) : Prop :=
2   | Base: n = 0 ->
3           is_even n
4   | Step: is_even (n - 2) ->
5           is_even n.
```

Its interpretation should pose no doubt. It is composed by two constructors, Base and Step, that can be read as follows. A natural number **n** is *even* if it is equal to zero, or if **(n-2)** is even.

Using this predicate constructors one can perform proofs. For instance, let us see how one could prove that 4 is even.

```
1 Lemma four_is_even:
2    is_even 4.
3 Proof.
4   apply Step.
5   apply Step.
6   apply Base.
7   reflexivity.
8 Qed.
```

Proving properties in a interactive system like Coq requires convincing it by applying a sequence of *tactics*. For this proof, we start by applying the Step constructor (line 4), at which point Coq will demand the proof of is_even (4-2). We therefore apply once again the same constructor (line 5), and get confronted with the establishment of is_even (4-2-2). Thus, we apply the Base constructor (line 6) and end up with goal of demonstrating 4 - 2 - 2 = 0, that can be trivially proved by reflexivity (line 7).

The above predicate and proof were rather simple. Let us now see a more interesting predicate attesting that two lists are a permutation of each other.

```
1 Inductive permut (A : Type) : list A -> list A -> Prop :=
2   | p_nil  : permut nil nil
3   | p_skip : forall (x : A) (l1 l2 : list A),
4                 permut l1 l2 ->
5                 permut (x :: l1) (x :: l2)
6   | p_swap : forall (x y : A) (l1 : list A),
```

```
7                  permut (y :: x :: l1) (x :: y :: l1)
8   | p_trans : forall l1 l2 l3 : list A,
9                   permut l1 l2   ->
10                  permut l2 l3 ->
11                  permut l1 l3.
```

This predicate is slightly more elaborated and requires a closer look. Indeed, in the realm of formal methods it is common that syntax verbosity get in the way of the understanding of concepts. It should be noted that such inductive predicates can be seen by a set of rules. For instance, the permut predicate is defined by the more readable and familiar rules depicted by Table 2.1.

$\dfrac{}{\text{permut nil nil}}\ p\_nil$	$\dfrac{\text{permut l1 l2}}{\text{permut (x :: l1) (x :: l2)}}\ p\_skip$
$\dfrac{}{\text{permut (y :: x :: l1) (x :: y :: l1)}}\ p\_swap$	$\dfrac{\begin{array}{c}\text{permut l1 l2}\\ \text{permut l2 l3}\end{array}}{\text{permut l1 l3}}\ p\_trans$

Table 2.1: Rules for the permut predicate

Using this set of rules, proving that two lists are a permutation of each other boils down to building a proof tree establishing that fact. Below, we exemplify with the proof that (1::2::3::nil) is a permutation of (3::1::2::nil).

$$\dfrac{\dfrac{\dfrac{}{\text{permut (2 :: 3 :: nil) (3 :: 2 :: nil)}}\ \text{(p\_swap)}}{\text{permut (1 :: 2 :: 3 :: nil) (1 :: 3 :: 2 :: nil)}}\ \text{(p\_skip)} \qquad \dfrac{}{\text{permut (1 :: 3 :: 2 :: nil) (3 :: 1 :: 2 :: nil)}}\ \text{(p\_swap)}}{\text{permut (1::2::3::nil) (3::1::2::nil)}}\ \text{(p\_trans)}$$

The proof starts by applying a p_trans, that naturally yields two subgoals. It should be noted that the application of this constructor requires the instantiation of l2. In this case, it is instantiated by (1::3::2::nil). The first subgoal (on the left) is handled by applying a p_skip, which produces the goal of proving the permutation of the lists' tails. This is proved by applying p_swap and this subgoal is therefore discharged. The second subgoal (on the right) is directly solved by applying a p_swap. And the proof terminates.

Indeed, the graphical intuition provided by a proof tree can be of great value. However, with more complex goals it rapidly becomes an exercise of style. The equivalent proof in Coq is shown below.

```
1 Lemma permut_example:
2   permut (1::2::3::nil) (3::1::2::nil).
3 Proof.
4   apply p_trans with (l2:=1::3::2::nil).
5   + apply p_skip.
6     apply p_swap.
7   + apply p_swap.
8 Qed.
```

As expected, the proof follows the same approach — the + symbols on the proof script serve the purpose of highlighting the subgoals. Moreover, it should be noted that it has the advantage of being machine checked. i.e. Coq makes sure no mistake was made during the proof.

2.5.4 Extracting certified programs

As seen above, Coq is equipped with a functional programming language containing most standard programming facilities, and even some rather sophisticated features (i.e. dependent types). Indeed, despite some "restrictions"[4] — i.e. total functions, termination, ... —, it allows the user to write rather complex programs. The flagship example of this is CompCert [16], a formally verified compiler.

However, Coq is not suitable to execute programs, at least not with the expectation of meeting real world demands, as it does not provide an efficient environment. The reason why one would write programs in Coq is that it offers the possibility to prove properties about them. Then, one can use Coq's *extraction* mechanism to obtain semantically equivalent code for efficient programming languages like OCaml and Haskell. Indeed, Coq ensures that the program's behaviour is preserved during the extraction, and therefore we say that such programs are *certified*.

The extraction mechanism opens the door for the building of highly reliable

[4]Depending on the reader, these "restrictions" may actually be seen as features! We include the quotes in order to avoid any disgust.

code. Yet, one should be aware of some particularities of this process to fully benefit from it. For instance, dependent types are not supported in a programming language like OCaml. Below, we recall the vhead function (on the left) and show its extraction in OCaml (on the right).

```
Definition vhead (A:Type) (n:nat)
           (v: vector A (S n)) :=
match v with
  Vcons el n v' => el
end.
```

```
let vhead n = function
  | Vnil ->
      Obj.magic (fun _ -> id)
  | Vcons (el, n0, v') ->
      el
```

(Un)Surprisingly, the extracted vhead contains the pattern matching for both constructors of the vector type. This is the case as there is no support for dependent types in OCaml, and as such the non-existence of a vhead call with Vnil as argument cannot be guaranteed by typing.

In short, it is the responsibility of the user to make sure all requirements are met before calling an extracted function. Thus, a good practice to follow when extracting code is not to use dependent types in the functions that will act as the program's interfaces. In the same spirit, the proved properties should avoid to make assumptions on their input domain, as only the computational part is extracted, all logical objects are discarded.

Naturally, only a small account of Coq's capabilities were covered in this section. For an authoritative report on its foundations and features the interested reader is pointed to [12]. There is also *Software Foundations* [17], a book that is literally a Coq script, focusing on the principles of programming and on the development of reliable software. For a shorter introduction written in a tutorial style we recommend Bertot's *Coq in a Hurry* [18].

✠✠✠

In this chapter we discussed the relevant technical background for the understanding of this thesis.

In the following chapter we present an industrial case study on the formal specification and verification of a GCM/ProActive application: The HyperManager.

Chapter 3

The HyperManager

"Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works."

Steve Jobs

This chapter discusses the formal specification and verification by model-checking of a reconfigurable monitoring application — The HyperManager. The *pNets* formalism is employed as a means to describe its behaviour and dynamic topology.

Section 3.1 presents The HyperManager application. The overall methodology employed for this case study is discussed in Section 3.2. Section 3.3 and Section 3.4 detail the formalization aspects and proven properties. Final remarks are discussed in Section 3.5

The results discussed in this chapter were partially presented as an industrial case study at the Jiangxi Normal University, China, for the 10^{th} *International Symposium on Formal Aspects of Component Software* [19].

Contents

As discussed in Section 1.2, Spinnaker is a French collaborative project between INRIA and several academic and industrial partners whose overall goal aims at promoting the widespread adoption of RFID-based technology. In this context, our contribution came with the design and implementation of a non-intrusive, flexible and reliable solution that can integrate itself with other already deployed systems.

Specifically, we developed the The HyperManager, a general purpose monitoring application with autonomic features. This was built using GCM/ProActive. For the purposes of this project, it had the goal to monitor the E-Connectware (ECW)[1] framework in a loosely coupled manner.

For the sake of clarity let us describe one of the real life scenarios faced in a industrial context. An hotel needs to keep track of their bed sheets life-cycle. Every bed sheet contains an embedded RFID sensor chip that uniquely identifies it. At every shift, the hotel maids go through all the rooms recovering them to a laundry cart. By reaching the end of the rooms' corridor, a device running the ECW Gateway software captures the bed sheets' identifiers. For each corridor there might be several laundry carts and one device running the ECW Gateway. After receiving the bed sheets' identifiers the ECW Gateways emit this information along with their own identifier to yet another physical device running the ECW Server. Once the information reaches the top of this hierarchy it can be used to whatever purpose, namely bed sheets traceability.

Abstracting away this particular scenario, one can see it in a hierarchical manner as depicted by Figure 3.1.

[1] `http://www.tagsysrfid.com/Products-Services/RFID-Middleware`

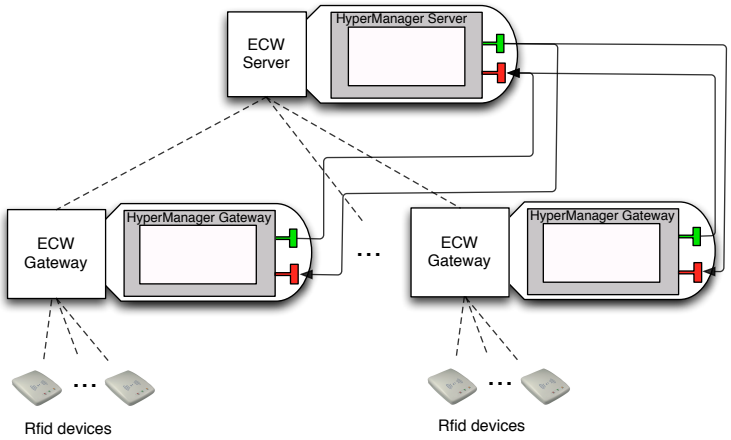

Figure 3.1: Hierarchical representation of our case study

Regarding the previously described scenario, this hierarchical view should pose no doubt. For each of the N floors of the hotel there are M laundry carts that interact in a *one-to-one* style with a gateway. On the other hand, the gateways communicate with the server on a *n-to-one* style. Moreover, there is also the need to cope for possible maintenance issues. For instance, in the case of malfunction of some device running the ECW Gateway, it may be required to replace it or add a new one in order to avoid any overloading.

The architecture depicted by Figure 3.1 also includes THE HYPERMANAGER application. Indeed, it is deployed alongside the pre-existent distributed system, performing its monitoring on all ECW components. Moreover, the careful reader may notice that the flow of requests go both from the HyperManager Server to the HyperManager Gateway, and vice-versa. Indeed, these follow the *pull* and *push* styles of communication, respectively. More details regarding these mechanisms will be discussed at a later stage.

3.1 The HyperManager architecture

THE HYPERMANAGER is a general purpose monitoring application that was developed in the context of the Spinnaker project. The goal was to deliver a modular solution that would be capable of monitoring a distributed application and react to certain events. As such, THE HYPERMANAGER is itself a distributed application,

deployed alongside the target application to monitor.

Generally, when performing a monitoring task in an application one may consider two types of events: *pull* and *push*. The former stands for the usual communication scenario where the server regularly *pulls* information from the client. The latter however, is when the client *pushes* data to the server. Both styles of communication are employed in The HyperManager application.

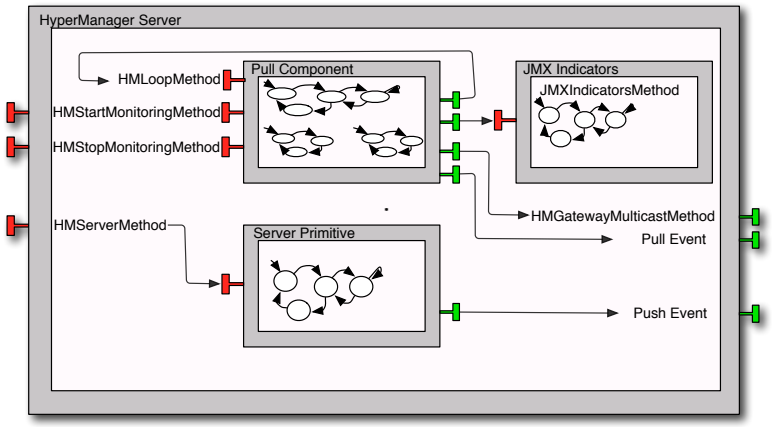

Figure 3.2: HyperManager server component

As illustrated by Figure 3.2, the HyperManager Server component features three primitive components that are responsible for the application logic. Each possesses one or several service methods.

The JMX Indicators component features only one service method: it accepts requests about a particular Java Management Extensions (JMX)[2] indicator and replies with its status. This encapsulates business code and interacts directly with the ECW.

The Pull Component includes three service methods and four client interfaces. As the component's name indicates, it is responsible for *pulling* information and emitting it as *pull events*. The service methods HMStartMonitoringMethod and HMStopMonitoringMethod are responsible for starting and stopping the *pulling* activity, respectively. Typically, these are the methods called by an administrator. The remaining service method, HMLoopMethod, may pose some doubt. Indeed, it is called from one of its own client's interface. Being a GCM/ProActive application,

[2]JMX is the standard protocol used for monitoring Java applications.

it follows the active object paradigm where explicit thread creation is discouraged. As such, making a method *loop* is achieved by making this method send itself a request before concluding its execution.

While in the monitoring loop, the HMLoopMethod method *pulls* information regarding its own local JMX indicators and those of its gateways via a multicast client interface. The last remaining client interface serves the purpose of reporting the *pulled* information as *pull* events.

Last, the Server Primitive component receives *push* information from the HyperManager Gateways — typically to alert the occurrence of some anomaly — and emits it as *push* events.

The description of the HyperManager Gateway component follow the same spirit. Figure 3.3 depicts its constitution.

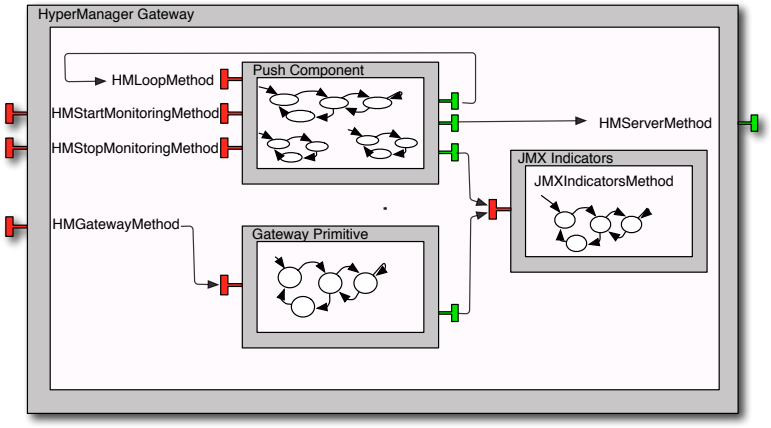

Figure 3.3: HyperManager gateway component

It is also composed of three primitive components. As expected, the JMX Indicators component has the same semantics as described above.

The Push Component features the same service methods as the Pull Component. Its semantics however, are slightly different. While *looping* it will check for the status of its JMX indicators, and communicate with The HyperManager server if some anomaly is encountered — which will then trigger a *push* event.

As for the Gateway Primitive component, its sole purpose is to reply to the *pulling* requests from the HyperManager Server component.

3.2 Formal specification and verification methodology

Regarding our modelling and verification *workflow*, the first step is to build the behavioural models by encoding the involved processes in the Fiacre specification language. Naturally, this step can be partially automated as processes such as the Queue, Body, etc, are computable, i.e., there is an algorithm that generates their behaviour. For instance, Listing 3.1 depicts the specification of the Queue process for the JMX Indicators component in Fiacre. It should be noted that we need a finite representation of such process: we illustrate this by a Queue bounded at two requests.

```
1  process Queue [ Q_JMXIndicatorsMethod: in
       future_id#JMXIndicatorsMethod_args_type,
2   Serve_JMXIndicatorsMethod: out
       future_id#JMXIndicatorsMethod_args_type,
3   Error:  errorMsgT ]
4  is
5    states SEmpty, S1, S11, SOutOfBounds, SError
6    var a0:future_id, a00:JMXIndicatorsMethod_args_type,
7          a1:future_id, a11:JMXIndicatorsMethod_args_type,
8          a2:future_id, a22:JMXIndicatorsMethod_args_type,
9          errmsg:errorMsgT:={parti=0,partst=UnknownErr}
10
11 from SEmpty
12     Q_JMXIndicatorsMethod?a0,a00; to S1
13
14 from S1 select
15       Serve_JMXIndicatorsMethod!a0,a00; to SEmpty
16    [] Q_JMXIndicatorsMethod?a1,a11      ; to S11
17 end
18
19 from S11 select
20       Serve_JMXIndicatorsMethod!a0,a00; a0:=a1; a00:=a11;  to S1
21    [] Q_JMXIndicatorsMethod?a2,a22; errmsg.partst:=PS11; to
         SOutOfBounds
22 end
23
```

```
24 from SOutOfBounds
25     Error!errmsg; to SError
```

Listing 3.1: Queue specification of the JMX Indicators component in Fiacre (bounded at two requests)

The JMX Indicators component possesses one service method, named JMXIndicatorsMethod, taking one argument of JMXIndicatorsMethod_args_type type. The three labels (lines 1-3) Q_JMXIndicatorsMethod, Serve_JMXIndicatorsMethod and Error represent enqueuing of a request to its service method, serving it, and throwing an exception due to exceeding of its capacity, respectively. From SEmpty, upon reception of a request the process goes to state S1 (lines 11-12), where it can serve the current request or receive another one (lines 14-17). The former case leads back to SEmpty. The latter takes the execution to S11 where it faces the same cases. On the one hand, another request will lead to SOutOfBounds (line 21), at which point the action Error!errmsg is emitted since the request queue's maximum capacity is reached — where errmsg is a variable holding information about the antecedent state — and the execution is transferred to the sink state SError (lines 24-25). On the other hand, since it exhibits a FIFO policy, serving a request (line 20) is performed on the initial call by emitting Serve_JMXIndicatorsMethod!a0,a00, and performing two assignments: a0:=a1 and a00:=a11. The variables a00, a11, a22 hold the request arguments by order of arrival. Thus, this simple mechanism ensure they are served in an adequate order, i.e., meeting the FIFO policy. Moreover, the variables a0, a1 and a2 contain future proxies, and are handled analogously. Naturally, these are implicit — transparent to the programmer —, but included into the formal model.

Let us now see how the body process is modelled. As an example, listing 3.2 depicts its specification for the JMX Indicators component.

```
1 process Body [ Serve_JMXIndicatorsMethod : in future_id#
    JMXIndicatorsMethod_args_type,
2  Call_JMXIndicatorsMethod : out future_id#
     JMXIndicatorsMethod_args_type,
3  R_JMXIndicatorsMethod : in future_id#
     JMXIndicatorsMethod_return_type ] is
4   states s0, s1, s11
5
```

```
6   var args   : JMXIndicatorsMethod_args_type,
7     reply  : JMXIndicatorsMethod_return_type,
8     fid    : future_id
9
10  from s0
11     Serve_JMXIndicatorsMethod ? fid, args ; to s1
12
13  from s1
14     Call_JMXIndicatorsMethod ! fid, args; to s11
15
16  from s11
17     R_JMXIndicatorsMethod ? fid, reply; to s0
```

Listing 3.2: Body specification of the JMX Indicators component in Fiacre

Its understanding should pose no doubt. Basically, it idles in state s0 until Serve_-JMXIndicatorsMethod?fid,args is activated, at which point the execution goes to s1 (lines 10-11). From s1 it emits Call_JMXIndicatorsMethod!fid,args, therefore starting the actual method execution, and moves to s11 (lines 13-14). Then, it just waits for the JMXIndicatorsMethod method finish its execution and return, at which point R_JMXIndicatorsMethod?fid,reply is received, and it goes back to s0 (lines 16-17).

Proxy and ProxyManager processes' behaviour can also be obtained in a systematic manner. Together with the Queue and Body processes they model all the internal intricacies of a GCM primitive component. The specification of the featured service methods however, is of specific nature, and thus a manual task.

Having the Fiacre sources files for all involved processes, we use the FLAC compiler[3] to produce LOTOS [20]. Then, we take advantage of the panoply of tools offered by the CADP suite [21] for model generation and formal verification by model-checking techniques. Figure 3.4 depicts the overall approach.

[3]Available online `http://gforge.enseeiht.fr/projects/fiacre-compil`

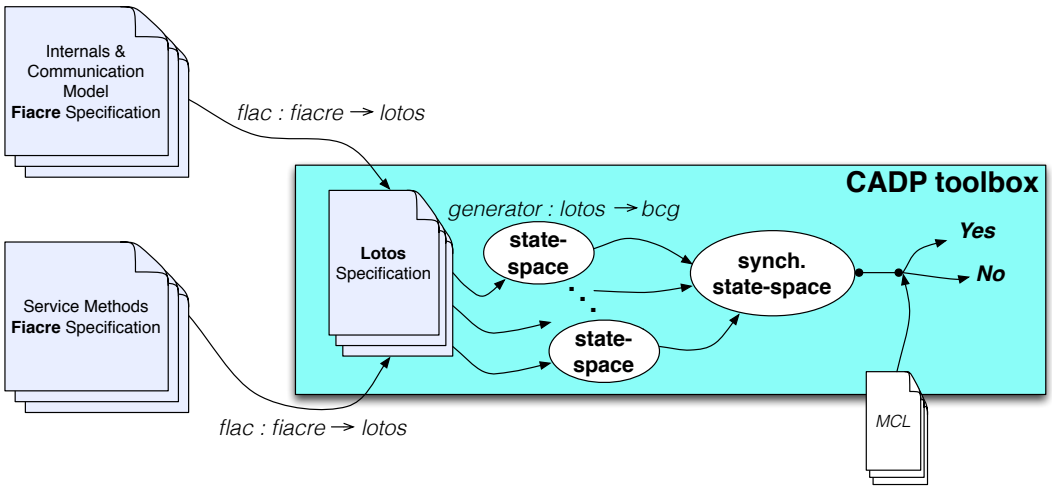

Figure 3.4: Formal specification and verification workflow

From CADP, we use generator for state-space generation. This process takes a plaintext .lotos file and yields a binary .bcg file encoding a Labelled Transition System (LTS) that models the process behaviour. This file is usually referred to as the state-space. Moreover, it should be noted that state-space generation can be performed in a distributed manner. CADP's distributor tool implements a distributed algorithm that can be executed on several machines. This can be of great value as each of them is used to generate and store a part of the LTS.

For the purposes of this thesis, we used distributed state-space generation by exploiting ProActive PACA Grid — a computing cloud operated by INRIA, University of Nice, and CNRS-I3S laboratory, and accessible through the ProActive Scheduler[4]. Nevertheless, it should be noted that parallel composition through synchronization vectors is a sequential process, and thus may be a bottleneck for large systems. Indeed, as we shall see later, dealing with large state-spaces was one of the main challenges of this case study.

For last, we use evaluator4 for model-checking state-spaces against Model Checking Language (MCL) [22] formulas — an extension of the alternation-free regular μ-calculus [23] with facilities for manipulating data.

In the following, to optimize the size of the model, composite components have no request queue and requests are directly forwarded to the targeted primitive

[4] `http://proactive.activeeon.com/index.php?page=proactive_5_min`

component. This has no influence in the system's semantics as the request queues of the primitive components are sufficient for dealing with asynchrony and requests from the subcomponents are directly dispatched too. We set the primitive components with re-entrant calls — Pull Component and Push Component — with a queue of size 2, and the remaining from the HyperManager Server and HyperManager Gateway composites with size 1 and 2, respectively. Furthermore, we set the future proxy domain to range over $\{0..1\}$, and consider the existence of two RFID devices per gateway.

3.3 The HyperManager as a formal methods case study

As mentioned above, The HyperManager is a GCM/ProActive application. A natural choice for its behavioural specification is therefore pNets. This task requires the specification of all service methods composing the application. These transparently interact with the component's internal intricacies — component queue, proxy manager, ... —, that are also included in the model.

In the following we dedicate Subsection 3.3.1, Subsection 3.3.2, and Subsection 3.3.3 to the discussion on the specification and verification by model-checking of the HyperManager Gateway component, HyperManager Server component, and overall system product, respectively. Next, Section 3.4 deals with the inclusion of structural reconfigurations in our model. Section 3.5 gives some final remarks regarding this case study.

Moreover, for more material on this case study — namely the specification's source files —, the reader is invited to check its companion website[5].

3.3.1 On HyperManager Gateway

JMX Indicators component

As seen above, the JMX Indicators primitive component only features one service method, JMXIndicatorsMethod, with JMXIndicatorsMethod_args_type and JMXIndi-

[5] http://www-sop.inria.fr/members/Nuno.Gaspar/HyperManager.php

catorsMethod_return_type as argument and return types, respectively. These are specified as shown by Listing 3.3.

```
type stability_type is union
  Stable
| Unstable
end

type device_id_type is union
   IdOne
 | IdTwo
end

type availability_type is union
   Available
 | Unavailable
end

type device_availability_type is record
  id:device_id_type,
  availability:availability_type
end

type JMXIndicatorsMethod_return_type is union
    memory_usage of stability_type
  | device_status of device_availability_type
end

type JMXIndicatorsMethod_args_type is union
    MemoryUsage
  | DeviceStatus of device_id_type
end
```

Listing 3.3: Specification for the JMXIndicatorsMethod argument and return types

For the sake of simplicity, we only model two types of JMX indicators: MemoryUsage and DeviceStatus. The latter takes into account an identifier, returning its availability status. This relates to the status of a RFID reader transmitting to the ECW Gateway. While the former accounts for the stability status of the memory.

Listing 3.4 illustrates the specification of the JMXIndicatorsMethod method behaviour.

```
1  process JMXIndicatorsMethod
2   [Call_JMXIndicatorsMethod: future_id#JMXIndicatorsMethod_args_type,
3   R_JMXIndicatorsMethod: future_id#JMXIndicatorsMethod_return_type] is
4
5   states s_init, s_mem, s_dev
6
7   var fid           : future_id,
8         device_id_var : device_id_type,
9       x             : device_availability_type
10
11  from s_init select
12      Call_JMXIndicatorsMethod ? fid, MemoryUsage          ; to s_mem
13    [] Call_JMXIndicatorsMethod ? fid, DeviceStatus(device_id_var) ; to
          s_dev
14  end
15
16  from s_mem select
17      R_JMXIndicatorsMethod ! fid, memory_usage(Stable); to s_init
18    [] R_JMXIndicatorsMethod ! fid, memory_usage(Unstable); to s_init
19  end
20
21  from s_dev
22      R_JMXIndicatorsMethod ? fid, device_status(x); to s_init
```

Listing 3.4: JMXIndicatorsMethod specification

Basically, it waits at the s_init state until it receives a request call regarding one of the JMX indicators (lines 11-13). Then, it goes to state s_mem or s_dev, depending on which JMX indicator is queried: a request to the memory usage takes the execution to the former, while a request to the device statuses takes the execution to the latter. From s_mem it randomly replies one of the possible values for the memory usage and goes back to s_init (lines 16-19). As expected, the same behaviour w.r.t. device statuses occurs from s_dev (lines 21-22). The careful reader may wonder about the rather different specification approaches between the reply of the two JMX indicators. Indeed, for the memory usage it exhaustively lists all possible reply values, and emits a non-deterministic choice among them.

For the devices statuses however, it simply replies with a non-instantiated variable **x**, that does the work of ranging over all possible reply values. Naturally, both approaches have the same semantics, it is merely a matter of style.

The Fiacre specification language allows for fairly simple and intuitive descriptions of automata-based models. Nevertheless, a more convenient representation for illustrating automata is in its graphical form. For instance, Figure 3.5 depicts the behaviour of the JMXIndicatorsMethod method.

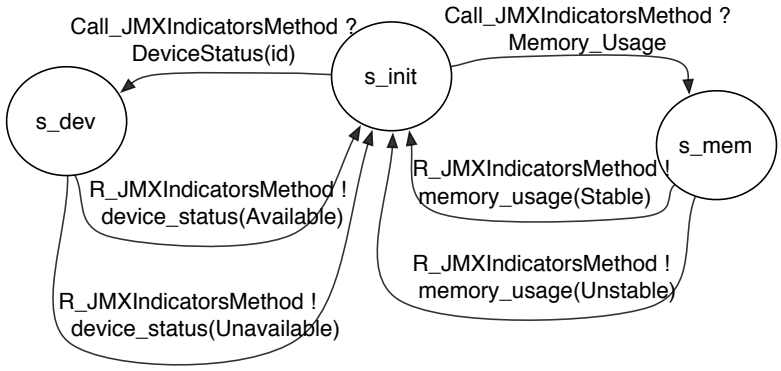

Figure 3.5: Behaviour of the JMXIndicatorsMethod method

Its understanding is straightforward, as it closely follows Fiacre's specification from Listing 3.4. There is, however, one small nuance: it does not include the future proxies. Indeed, these are related to the internal machinery of GCM components, and one would rather avoid such intricacies when specifying service methods. As such, they are omitted in the graphical representations: we consider these representations as an *user-version* LTS, i.e., typically what a user would specify without paying attention to the future proxies mechanism.

Gateway Primitive component

Let us now look at the behaviour of the HMGatewayMethod method. Listing 3.5 details its specification in Fiacre.

```
1 process GatewayMethod [Call_GatewayMethod: in
    future_id#gatewayMethod_args_type,
2  GetProxy_JMXIndicatorsMethod  : in none,
3  New_JMXIndicatorsMethod       : in proxy,
```

```
4  Q_JMXIndicatorsMethod          :  proxy#JMXIndicatorsMethod_args_type
       ,
5  GetValue_JMXIndicatorsMethod  : in proxy#
       JMXIndicatorsMethod_return_type,
6  Recycle_JMXIndicatorsMethod   : out proxy,
7  R_GatewayMethod               : out future_id#
       GatewayMethod_return_type] is
8
9   states s_init, s1, s2, s3, s4, s5, s6
10
11  var fid        : future_id,
12      args       : gatewayMethod_args_type,
13    p          : proxy,
14    jmx_reply  : JMXIndicatorsMethod_return_type
15
16  from s_init
17    Call_GatewayMethod ? fid, args;
18    to s1
19
20  from s1
21    GetProxy_JMXIndicatorsMethod;
22    to s2
23
24  from s2
25    New_JMXIndicatorsMethod ? p;
26    to s3
27
28  from s3
29    Q_JMXIndicatorsMethod ? p, args;
30    to s4
31
32  from s4
33    GetValue_JMXIndicatorsMethod ? p, jmx_reply;
34    to s5
35
36  from s5
37    Recycle_JMXIndicatorsMethod ! p;
38    to s6
39
```

```
40  from s6
41    R_GatewayMethod ! fid , jmx_reply;
42    to s_init
```

Listing 3.5: HMGatewayMethod specification

Despite its apparatus, this service method offered by the Gateway Primitive component has also a fairly simple semantics. It acts merely as a request forwarder for the JMX Indicators component. A far more convenient specification is given by Figure 3.6.

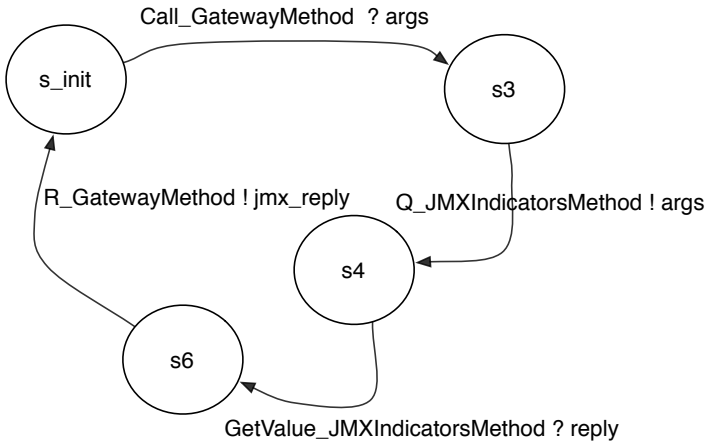

Figure 3.6: Behaviour of the HMGatewayMethod method

For this example the extra verbosity introduced by the handling of future proxies is even more evident. Indeed, all this machinery ends up obfuscating the method's actual behaviour. In the following, we constrain ourselves to the *user-version* LTS representation for the description of service method behaviour. Hopefully, this shall facilitate the understanding of the remaining specifications.

Push Component component

Regarding the Push Component, it is composed by three service methods: HMStartMonitoringMethod, HMStopMonitoringMethod and HMLoopMethod. Further, the global variable **started** is shared among them. For the sake of clarity communication actions are written in black, while *local* computations (e.g. guard

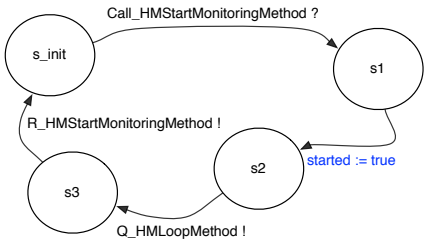

Figure 3.7: Behaviour of the HMStartMonitoringMethod method

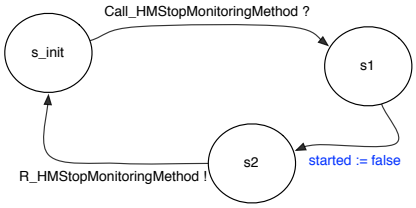

Figure 3.8: Behaviour of the HMStopMonitoringMethod method

evaluations, assignments) are written in blue. Their intended meaning should pose no doubt.

Figure 3.7 and Figure 3.8 depict the behaviours of the HMStartMonitoringMethod and HMStopMonitoringMethod methods, respectively. Basically, they are responsible for enabling/disabling the *looping* process of the HMLoopMethod. This is achieved by the shared variable **started**. On the one hand, invoking HMStartMonitoringMethod sets **started** to *true* and performs an invocation to HMLoopMethod. On the other hand, HMStopMonitoringMethod sets **started** to *false*.

It should be noted that these local computations involving the global variable **started** are merely syntactic sugar for the emission and reception of messages to another process. In practice, the involved labels are GuardQuery, GuardReply?b:bool, SetFalse and SetTrue. Their meaning should be obvious from their labels.

The last remaining service method to describe is the most interesting one — the HMLoopMethod method. Its behaviour is depicted by Figure 3.9.
The reception of Call_HMLoopMethod action starts the method execution, moving from state s_init to state s1. Then, the global variable **started** is checked. If it evaluates to *false* the method returns by emitting a R_HMLoopMethod message. Otherwise, the execution proceeds by querying the gateway's local JMX indicators — from s2 to s3 through Q_JMXIndicatorsMethod?jmx_args. The reply is received by GetValue_JMXIndicatorsMethod?jmx_reply and evaluated at state s4. If an anomaly is detected (e.g. unstable memory), it is reported to the HyperManager Server component through the message Q_ServerMethod!jmx_reply, thus moving from state s5 to state s6. If no anomaly is found, the execution proceeds directly to state s6, where a call to the method itself is performed: Q_HMLoopMethod. This places a request on the Push Component component queue — thus

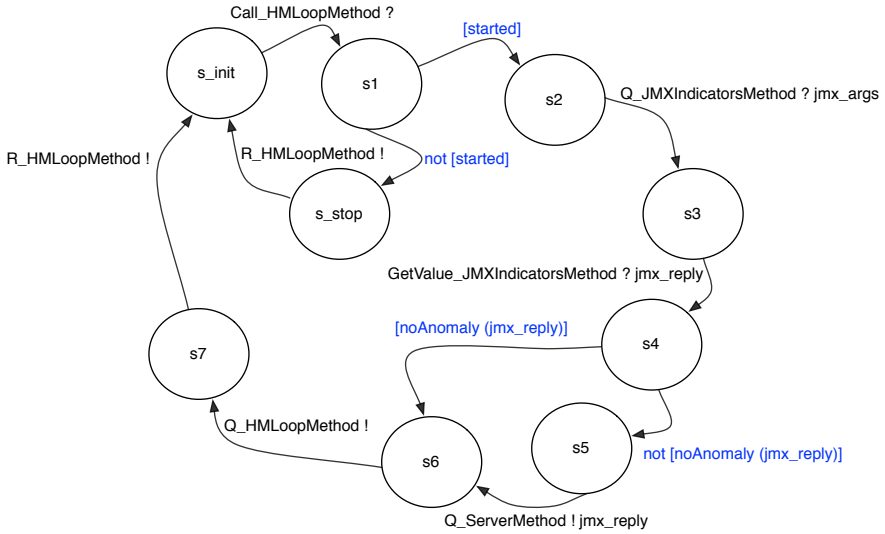

Figure 3.9: Behaviour of the HMLoopMethod method for the HyperManager Gateway

making the method *loop* — and transfers the execution to state s7. It should be noted that this request is scheduled as any other one: the FIFO policy dictates that requests are served by order of arrival, and therefore a potential Q_HMStopMonitoringMethod enqueued in the meantime would be served before this new Q_HMLoopMethod request. Finally, from state s7, the message R_HMLoopMethod is emitted, and the method execution concludes by going back to state s_init.

Model Generation and Proven Properties

Model generation is performed by synchronizing all the involved processes — GCM internals and service methods — of the HyperManager Gateway component. Table 3.1 illustrates its state-space information.

	States	**Transitions**	**File Size**
hmgateway.bcg	14.931.628	147.485.103	~ 295 mb

Table 3.1: State-space information for the HyperManager Gateway component

Having the state-space generated we can now prove some properties regarding the expected behaviour of the model. Specifying properties of interest in MCL

is a rather intuitive task due to its expressiveness and conciseness. Its main ingredients include patterns extracting data values from LTS actions, modalities on transition sequences described using extended regular expressions, and programming language constructs.

For instance, one could wonder about this rather unusual *looping* mechanism. Once setting the global variable started to *true* — accomplished by Q_HMStartMonitoringMethod —, the *looping* continues until a request to stop monitoring is received. That is, there is no execution path in which the global variable **started** evaluates to *false* without the occurrence of a Q_HMStopMonitoringMethod. Property 3.3.1 encodes this statement in MCL.

Property 3.3.1 (loops while variable started is true)**.**

```
[ "Q_HMStartMonitoringMethod" .
     (not "Q_HMStopMonitoringMethod")* . "GuardReply !FALSE" ] false
```

More precisely, Property 3.3.1 reads: all paths starting with the label Q_HMStartMonitoringMethod, followed by any other sequence of labels not including Q_HMStopMonitoringMethod, and ending by GuardReply !FALSE, cannot occur. Model-checking hmgateway.bcg against this formula demonstrates that it is indeed true.

Another property of interest concerns the overloading of the HyperManager Server component with unnecessary messages. We want to ensure that we cannot *push* data if not in the presence of an anomaly. This is modelled by Property 3.3.2.

Property 3.3.2 (no server overload)**.**

```
[ ((not "R_JMXIndicatorsMethod !memory_usage (Unstable)")* .
       "Q_ServerMethod.*")  |
  ((not "R_JMXIndicatorsMethod !device_status ((Unavailable, IdTwo))")* .
       "Q_ServerMethod.*") |
  ((not "R_JMXIndicatorsMethod !device_status ((Unavailable, IdOne))")* .
       "Q_ServerMethod.*")
] false
```

The above MCL formula states that all paths starting with any sequence of labels not including the reply of an anomalous JMX indicator — R_JMXIndicatorsMethod

!memory_usage (Unstable), R_JMXIndicatorsMethod !device_status ((Unavailable, IdTwo)) and R_JMXIndicatorsMethod !device_status ((Unavailable, IdOne) —, followed by a Q_ServerMethod.* — where * is a *wildcard* matching all possible arguments —, cannot occur. With no surprise, Property 3.3.2 is also proved to be satisfied by the model.

3.3.2 On HyperManager Server

JMX Indicators component

As seen for the HyperManager Gateway component, the HyperManager Server component also features a JMX Indicators component. This one however, is not endowed with JMX indicators for the RFID devices statuses.

Technically, we attach to the LTS modelling the behaviour of the JMX Indicators component (see Figure 3.5) a context that constraints its requests. This is achieved by synchronization with another LTS that solely emits requests regarding the memory status. This approach for contextual state-space generation is compositional and thus promotes reusability.

Server Primitive component

As seen above, upon detection of an anomaly, the HyperManager Gateway components *push* the relevant information to the HyperManager Server. Then, it is emitted as a *push* event. This behaviour is depicted by Figure 3.10.

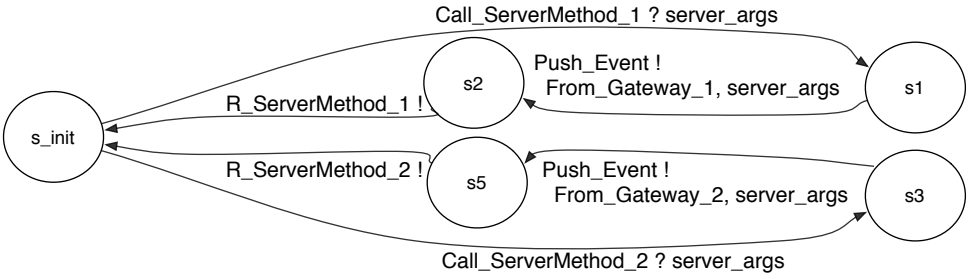

Figure 3.10: Behaviour of the HMServerMethod method

The careful reader will notice that the emitted event also contains the information regarding the HyperManager Gateway component from which the anomaly

originated. This should come as no surprise as there can be several of them, and properly identifying the source of an abnormal situation is of paramount importance. In our model, we consider the existence of two ECW/HyperManager gateways, and therefore we include specific labels — Call_ServerMethod_i$_{i\epsilon\{1,2\}}$ and R_ServerMethod_i$_{i\epsilon\{1,2\}}$ — for communicating with them.

Pull Component component

The Pull Component component is composed by three service methods: HMStartMonitoringMethod, HMStopMonitoringMethod, and HMLoopMethod. Indeed, these have the same names as the service methods for the Push Component component. In fact, their HMStartMonitoringMethod and HMStopMonitoringMethod methods possess the same semantics (depicted by Figure 3.7 and Figure 3.8). Their HMLoopMethod method however, behave differently. Figure 3.11 depicts its behaviour for the Pull Component.

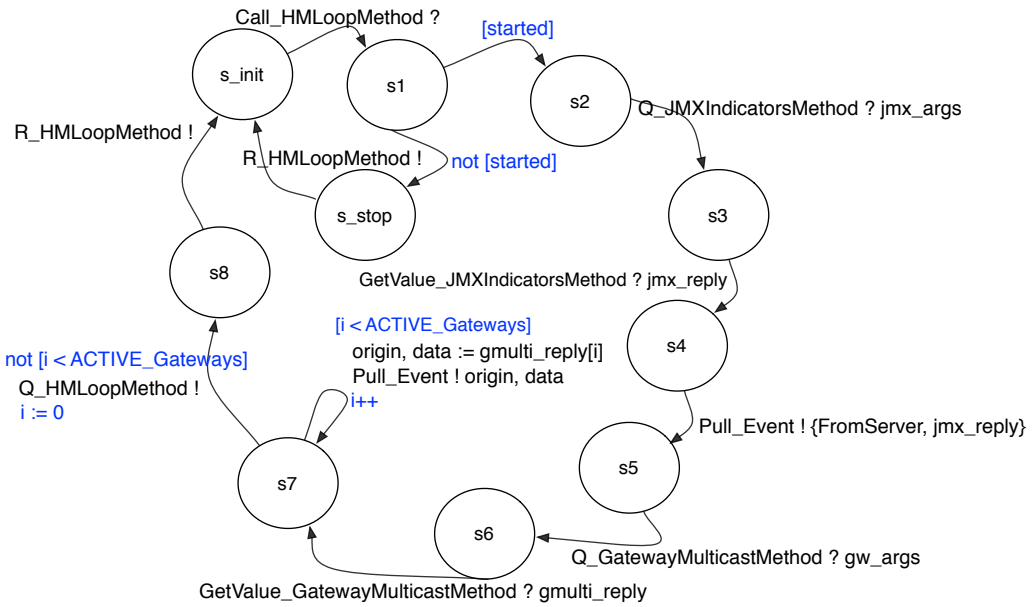

Figure 3.11: Behaviour of the HMLoopMethod method for the HyperManager Server

Its *looping* mechanism is also handled through a shared variable named **started**. While *looping*, the local JMX indicators are queried, and its reply emitted as *pull*

events. Moreover, through the multicast interface the monitoring information is *pulled* from the bound gateways. This, emits as many *pull* events as the number of bound gateways. Last, a request to itself is performed in order to continue *looping*.

Model Generation and Proven Properties

Table 3.2 illustrates the relevant information concerning HyperManager Server's state-space.

	States	**Transitions**	**File Size**
hmserver.bcg	12.787.376	187.589.422	~ 363 mb

Table 3.2: State-space information for the HyperManager Server component

It should be noted that we can also prove properties regarding the GCM internals. For instance, a rather trivial property we can expect to hold is that we can reach a state which exceeds the maximum capacity of the request queue. For the Server Primitive component, this can be modelled in MCL as demonstrated by Property 3.3.3

Property 3.3.3 (reachable exception)**.**

```
< true* . 'QueueException_ServerPrimitive !.*'> true
```

Formally, this property reads: from any state, there exists an execution path that can encompass the label QueueException_ServerPrimitive !.*. As expected, this property is proved to be satisfied by the model.

Another property regarding the GCM internals concerns the use of proxies. For instance, the HMLoopMethod needs to request a proxy in order to be able to invoke the service method from the JMX Indicators component. This is encoded as demonstrated by Property 3.3.4.

Property 3.3.4 (need for proxy)**.**

```
[ (not "GetProxy_JMXIndicatorsMethod.*")* .
            "Q_JMXIndicatorsMethod.*"] false
```

Formally, the above property reads: all execution paths starting with any sequence of labels not including GetProxy_JMXIndicatorsMethod.*, and followed by

Q_JMXIndicatorsMethod.*, cannot occur. With no surprise, model-checking this property confirms that it is satisfied by the model.

3.3.3 On System Product

We attempted to generate a system product constituted by two HyperManager Gateways and one HyperManager Server components. However, even on a machine with 90 GB of RAM, we experienced the so common state-space explosion phenomena. Indeed, as discussed in Section 3.2, while we can use distributed state-space generation, and thus spread the workload among several machines, synchronizations products remain a sequential task.

This arises often in the analysis of complex systems. To this end, *communication hiding* comes as an efficient and pragmatic approach for tackling this issue. Indeed, it allows the specification of actions that need not to be observed for verification purposes, thus yielding more tractable state-spaces.

Table 3.3 illustrates the effects of applying this technique to the model. The sole *communication actions* being hidden are the ones involved in (1) the request transmission from the Queue to the adequate method — Serve_ and Call_ —, (2) the proxy machinery —GetProxy_, New_ and Recycle_ —, and (3) finally in the *guard* of the *looping* methods — GuardQuery, GuardReply, SetFalse and SetTrue.

	States	**Transitions**	**File Size**
hmgateway-hidden.bcg	14.931.628	147.485.103	~ 287 mb
hmgateway-hidden-min.bcg	409.374	4.007.232	~ 8.5 mb
hmserver-hidden.bcg	12.787.376	187.589.422	~ 375 mb
hmserver-hidden-min.bcg	5.761.504	85.157.420	~ 179 mb
SystemProduct.bcg	342.047.684	3.026.114.393	~ 5.27 gb
SystemProduct-min.bcg	259.340.044	2.396.896.830	~ 4.83 gb

Table 3.3: State-space information for the overall system product

The lines suffixed by -hidden indicate the results obtained by *hiding* the mentioned *communication actions* in the HyperManager Gateway and HyperManager server state-spaces. For both, no effect is noticed on the size of the LTS. However, there is a decrease in the file size. This is due to the fact that the *hiding* process

yields several τ-transitions, which facilitates file compression. More importantly, this has the consequence of leveraging the subsequent *minimization* process: the entries suffixed by -min mean that minimization by branching *bisimulation* was applied. Indeed, we even obtain a reduction by two orders of magnitude (!) for the HyperManager Gateway state-space.

The HyperManager comes as a monitoring application that should be able to properly trace the origin of an anomaly. As such, one behavioural property that we expect to hold is that whenever an abnormal situation is detected by a HyperManager Gateway, it is *fairly inevitable* to be reported as a *push event* that correctly identifies its origin.

First, we shall use MCL's macro capabilities to help us build the formula:

```
macro GETVALUE_1_MEMORY () =
  "GetValue_JMXIndicatorsMethod_Push_1 !memory_usage (Unstable)"
end_macro

macro PUSH_1_MEMORY ()     =
  ("Push_Event (FirstGateway, UnstableMemoryUsage)")
end_macro
...
```

The above macros should be self-explanatory. The former represents the detection of an anomaly coming from the first HyperManager Gateway — the model is instantiated with two HyperManager Gateways, thus we differentiate their actions by suffixing them adequately. The latter stands for the emission of the *push* event corresponding to that anomaly. The macros for the remaining relevant actions are defined analogously.

Moreover, we define the following macro generically encoding the *fair inevitability*[6] that after an *anomaly* the system emits a *push*.

```
macro FAIRLY_INEVITABLY_A_PUSH (ANOMALY, PUSH) =
       [ true* . "ANOMALY" . (not "PUSH")* ]
```

[6]We consider the definition by Queille and Sifakis [24]: a sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it.

```
            < (not PUSH)* . PUSH > true
end_macro
```

Having the macros defined, we can now write the formula of interest:

Property 3.3.5 (push fairly inevitable)**.**

```
(FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_MEMORY, PUSH_1_MEMORY) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_MEMORY, PUSH_2_MEMORY) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_DEVICE_1, PUSH_1_DEVICE_1) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_DEVICE_2, PUSH_1_DEVICE_2) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_DEVICE_1, PUSH_2_DEVICE_1) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_DEVICE_2, PUSH_2_DEVICE_2)
)
```

Basically, property 3.3.5 states that it is fairly inevitable that the appropriate push event is triggered after the detection of an anomaly. As expected, it is satisfied by the model.

3.4 The case study reloaded: on structural reconfigurations

As seen so far, The HyperManager acts as a monitoring application with two styles of communication: *pull* and *push*. However, it also needs to cope with structural reconfigurations. This means that at runtime the architecture of the application can evolve by, say, establishing new bindings and/or removing existing ones.

For GCM applications *bind* and *unbind* operations are handled by the component owning the *client* interface that is supposed to be reconfigurable. This should come as no surprise, indeed, it follows the same spirit as in object-oriented languages: an object holds the reference to a target object; it is this object that must change the reference it holds.

In our case study, these reconfigurations can occur both from the HyperManager Server — when *pulling* data from the bound gateways —, and from a HyperManager Gateway — when *pushing* data to the server. The difference lies in the fact

that the HyperManager Server communicates via a *multicast* interface, unlike the HyperManager Gateways that establish standard *1-to-1* communications.

3.4.1 On HyperManager Reconfigurable Gateway

In Subsection 2.3.2, we showed how a reconfigurable client interface is modelled in pNets. In particular, Figure 2.14 detailed all the internal machinery for singleton interfaces. For this case study, in practice, to the HyperManager Gateway model discussed in Subsection 3.3.1 we add the *non-functional* request messages Q_Bind_ServerMethod and Q_Unbind_ServerMethod.

Since we only have one reconfigurable interface we can avoid adding an explicit parameter — unlike shown in Figure 2.14, where we demonstrate a more general case. Moreover, since the HyperManager Gateways can only be bound to one target — the HyperManager Server —, the *binding controller* only needs to keep a state variable regarding whether it is bound or not.

As expected, these changes have a considerable impact on the size of the model. This is illustrated by Table 3.4.

	States	**Transitions**	**File Size**
hmgateway-reconfig.bcg	354.252.868	4.178.400.886	~ 8.45 gb
hmgateway-reconfig-min.bcg	354.104.012	4.176.956.686	~ 8.54 gb

Table 3.4: State-space information for HyperManager Gateway with reconfigurable interface

All the properties proven in Subsection 3.3.1 still hold for this new HyperManager Gateway model. However, for this new model we are more interested in addressing the reconfiguration capabilities. For instance, provided that the interface is bound, it will not yield an Unbound action upon method invocation.

Property 3.4.1 (Bound interface impossibility)**.**

```
< true* . "Q_Bind_ServerMethod" . (not "Q_Unbind_ServerMethod")*  .
"Q_ServerMethod"  .  (not "Q_Unbind_ServerMethod")* . "Unbound"  > true
```

Property 3.4.1 is proved *false*. This indicates that provided that the interface is

bound, a path yielding an Unbound action without the occurrence of a Q_Unbind_-ServerMethod cannot occur.

3.4.2 On HyperManager Reconfigurable Server

Table 3.5 demonstrates the impact of adding reconfiguration capabilities to the HyperManager Server model.

	States	**Transitions**	**File Size**
hmserver-reconfig.bcg	931.640.080	16.435.355.306	~ 32.93 gb

Table 3.5: State-space information for HyperManager Server with reconfigurable interface

The generated state-space for the HyperManager Server model nearly attained 1 billion states.[7] Our attempts to *minimize* it revealed to be unsuccessful due to the lack of memory. These were carried out on a workstation with 90 GB of RAM.

It is worth noticing that while we were not able to *minimize* the produced state-space, we were still able to model-check it against Property 3.3.3.

3.4.3 On Reconfigurable System Product

As seen in Subsection 3.3.3, building the product of the system already proved to be a delicate task. Abstraction techniques such as *communication hiding* were already required to build the system. Thus, it should come as no surprise that we face the same situation here.

However, it should be noted that the *hiding* process itself, produced little effect on the file size, and no effect on the state-spaces. It mainly acted as a means to leverage the subsequent *minimization* process, allowing for a very significant state-space reduction. Table 3.6 illustrates the results obtained by following the same approach as above.

[7] As mentioned in Subsection 3.3.2, for the HyperManager Server model, the JMX Indicators component is generated with a context not including the request of device statuses. Previous experiments not considering this context produced a HyperManager Server model with the following characteristics: 4.148.563.680 states, with 74.268.977.628 transitions, on a 154.2 GB file. It is interesting to note the huge impact that (the lack of) a contextual state-space generation on one of its components can provoke.

	States	Transitions	File Size
hmgateway-reconfig-hidden.bcg	354.104.012	4.176.956.686	~ 8.15 gb
hmgateway-reconfig-hidden-min.bcg	11.090.974	127.799.874	~ 283.5 mb
hmserver-reconfig-hidden.bcg	931.640.080	16.435.355.306	~ 31.28 gb

Table 3.6: State-space information for the reconfigurables HyperManager Server and HyperManager Gateway

We obtained a significant state-space reduction for the HyperManager Gateway model, but we were unable to *minimize* the HyperManager Server. Indeed, *communication hiding* may leverage state-space reduction, but still requires that the *minimization* process is able to run, therefore not solving the lack of memory issue. This is a rather embarrassing situation as we would expect a significant state-space reduction as well for the HyperManager Server.

While *communication hiding* revealed to be a valuable tool, *minimization* is still a bottleneck if the input state-space is already too big. Thus, we need to shift this burden to the lower levels of the hierarchy. Indeed, both HyperManager Server and HyperManager Gateway components are the result of a product between their primitive components. Moreover, these are themselves the result of a product between their internal processes — request queue, body, proxies — and service methods.

Table 3.7 illustrates the results obtained by hiding the same communication actions as in the above approaches, but before starting to build any product.

	States	Transitions	File Size
hidden-hmgateway-reconfig.bcg	3.483.000	43.193.346	~ 85.46 mb
hidden-hmgateway-reconfig-min.bcg	3.073.108	39.373.968	~ 83.95 mb
hidden-hmserver-reconfig.bcg	210.121.904	3.890.791.694	~ 7.52 gb
hidden-hmserver-reconfig-min.bcg	177.604.848	3.288.937.718	~ 6.61 gb
SystemProduct-reconfig.bcg	3.054.464.649	38.680.270.695	~ 74.16 gb

Table 3.7: State-space information for the reconfigurables HyperManager Server and HyperManager Gateway (second approach)

Indeed, following this approach proved to be effective as we were able to gen-

erate considerably smaller state-spaces for both the HyperManager Server and HyperManager Gateway, and also build the system product. Alas, system product's *minimization* still remained out of reach. Moreover, model-checking attempts at this approximately 3 billion states state-space also remained out of reach due to lack of computing resources.

Nevertheless, we are still in a position to model-check properties regarding structural reconfiguration for the HyperManager Server component state-space: hidden-hmserver-reconfig-min.bcg. For instance, *pulling* information via a *multicast* emission is now predicated with a boolean array whose element's valuation determines whether the HyperManager Gateway is bound or not. As an example, Property 3.4.2 depicts a rather simple *liveness* property.

Property 3.4.2 (can unbind gateways)**.**

```
<true* . "Q_GatewayMulticastMethod !.* !ARRAY(FALSE FALSE) !.*"> true
```

Formally, Property 3.4.2 states that from any state we can attain a Q_GatewayMulticastMethod !.* !ARRAY(FALSE FALSE) !*, i.e., we can indeed unbind the two HyperManager Gateways.

3.5 Discussion

In the realm of component-based systems, behavioural specification is among the most employed approaches for the rigorous design of applications. It leverages the use of model-checking techniques, by far the most widespread formal method in the industry. Yet, verification in the presence of structural reconfigurations remains still a rather unaddressed topic. This can be justified by the inherent complexity that such systems impose. However, reconfiguration plays a significant role for the increase in systems availability, and is a key ingredient in the autonomic computing arena, thus tackling its demands should be seen of paramount importance.

In this chapter we discussed the specification and formal verification of a reconfigurable monitoring application as an industrial case study. Several lessons can be drawn from this work.

The Spinnaker project gave us the opportunity to promote the use of formal methods within the industry. As expected, the interaction with our industrial part-

ners revealed to be a demanding task. Common budgetary issues (time allocation, hirings, ...) of such projects and their lack of prior formal methods' exposure were some of the barriers to overcome. This was further aggravated by the fact that software development was playing a little part in the overall project budget, and therefore was not a main priority.

Nevertheless our experience revealed to be fruitful. We were able to witness the general curiosity on the use of formal methods by the industry, and increase our understanding on the needs and obstacles for its broader adoption. Indeed, collaborative projects of this nature allow the industry to *test the waters* and expose researchers to real-world scenarios. However, bridging the gap between the industry's expectation and the current state of the art still remains as a challenge for the research community. To this end, recent work on Vercors [25] aims at bringing intuitive specification languages and graphical tools for the non-specialists.

Concerning our task at hand, modelling The HyperManager application including the intricacies of the middleware led us to a combinatorial explosion in the number of states. This, is further aggravated by the inclusion of reconfigurable interfaces. Even the use of compositional and contextual state-space generation techniques revealed to be insufficient. While this could be solved by further increasing the available memory in our workstation, it is worth noticing that this approach is not always feasible in practice. This bottleneck could be alleviated by performing the overall synchronization product in a *distributed* manner. Alas, this is not supported by the CADP toolbox. In addition, model-checking itself is also a sequential task... Indeed, in this case study we were often confronted with the physical limits of our computing resources. Further, we go beyond previous works [26] on the specification and verification of GCM applications by including reconfiguration capabilities. Investigating the feasibility of such undertaking was also within the scope of this work.

At last, as usual in the realm of formal verification, we conclude that abstraction is key. Taking advantage of CADP's facilities for *communication hiding*, one can specify actions that need not to be observed for the verification purposes, which further enhances the effects of a subsequent *minimization* by branching *bisimulation*. This illustrates the pragmatic rationale of formal verification by model-checking — the most likely reason behind its acceptance in the industry.

✠✠✠

In this chapter we presented an industrial case study concerning the formal specification and verification of a GCM/ProActive application: The HyperManager. The employed methodology along with its challenges and issues were discussed.

In the following chapter we introduce Mefresa, a mechanized framework for reasoning on software architectures.

Chapter 4

A mechanized framework for reasoning on software architectures

"If I had asked people what they wanted, they would have said faster horses."

Henry Ford

Contents

This chapter presents MEFRESA (***Me**chanized **F**ramework for **Re**asoning on **S**oftware **A**rchitectures*), a Coq framework for reasoning on software architectures. It focuses on the GCM, detailing how its specificities were mechanized in a proof assistant like Coq. Moreover, a simple operation language is proposed for the safe composition of GCM architectures. Concrete examples complete its presentation.

The results discussed in this chapter were partially published as a *short paper* at the proceedings of the *Conférence en IngénieriE du Logiciel* (CIEL'2012) [27] held at the *Université de Rennes*, and at the *International Journal of Parallel Programming* as a special issue of the *International Symposium on High-level Parallel Programming and Applications* (HLPP'2013) [28], held at the *Institut Henri Poincaré*, Paris.

Section 4.1 discusses the mechanization of the GCM with the Coq proof assistant. In particular, we show how we use inductive definitions to model the GCM core elements, and predicates to formally specify their structure and *typing* requisites. A language for conveniently manipulating GCM architectures is presented in Section 4.2. Examples on mechanically proven properties are illustrated in Section 4.3. For last, Section 4.4 gives some final remarks about this development.

For more details and release announcements of MEFRESA the interested reader is pointed to the website[1] of its development.

4.1 Mechanizing GCM with the Coq proof assistant

As discussed in Section 2.1, the GCM is an extension to Fractal. Both component models have their specification written in natural language, thus inherently ambiguous and open to interpretation. In fact, in [29], this liability has already been acknowledged by the people behind Fractal: *"However, there are aspects of the [Fractal] specification that remain decidedly insufficiently detailed or ambiguous"*.

[1] MEFRESA is available online at: `http://www-sop.inria.fr/members/Nuno.Gaspar/Mefresa.php`

To this end, they attempt to *"correct these deficiencies by developing a formal specification"* in Alloy [30].

The same reasoning applies to our case, except that we rely on the Coq proof assistant [11]. We take advantage of its high expressiveness and effective rationale to give a precise semantics to the structure of GCM applications, and provide the means to reason about its composition. Further, we are not limited to finite domains, we can define and prove general properties about parametrized structures.

In the following we devote the remaining of this Section to detailing the mechanization of the GCM with the Coq proof assistant.

4.1.1 Core elements

The GCM has three core elements: components, interfaces and bindings. Their structure and the way they interact are naturally encoded by means of inductive definitions.

Interface datatype

The interface datatype is depicted by Listing 4.1.

```
1 Inductive interface : Type :=
2     Interface : ident -> signature -> path           ->
3                 visibility -> role -> functionality  ->
4                 contingency -> cardinality           -> interface.
```

Listing 4.1: interface datatype

It is characterized by an *id* denoting its name, a *signature* corresponding to its java interface *classpath*, and a *path* identifying its location in the component's hierarchy (i.e. the component it belongs to). More precisely, these fields are specified as shown by Listing 4.2.

```
1 Inductive ident : Type :=  Ident : string -> ident.
2
3 Definition signature := string.
4
5 Definition path : Type = list ident.
```

Listing 4.2: ident, signature and path datatypes

The only subtlety concerns the path field. It should be noted that GCM is a hierarchical component model, and since by introspection an interface is able to identify the component it belongs to, a path identifying this component is necessary. Its definition (Line 5) is a list of identifiers, where these identifiers indicate the components that need to be traversed in the hierarchy to reach the component holding the interface.

The intended meaning of the remaining fields should pose no doubt. An interface is of internal or external *visibility*, possess a client or server *role*, is of functional or non-functional *functionality*, contains an optional or mandatory *contingency*, and its *cardinality* is of singleton, multicast or gathercast nature.

Listing 4.3 and Listing 4.4 illustrate the visibility and contingency datatypes, respectively.

```
1 Inductive role : Type :=
2   Client : role
3 | Server : role.
```

Listing 4.3: role datatype

```
Inductive contingency : Type :=
  Optional : contingency
| Mandatory: contingency.
```

Listing 4.4: contingency datatype

The remaining encodings are performed analogously. Moreover, comparing these values of these datatypes requires the explicit definition of such a function. For instance, Listing 4.5 depicts a boolean equality function for the role datatype values.

```
1 Function beq_role (ro1 ro2:role) : bool :=
2   match ro1, ro2 with
3     | Client, Client => true
4     | Server, Server => true
5     | _, _           => false
6   end.
7
8 Notation "ro1 '==' ro2" := (beq_role ro1 ro2).
```

Listing 4.5: Boolean equality function for the role datatype values

Basically, it is a rather simple function returning true if the two role values are equal, and false otherwise. The only doubt that may arise concerns the contents of line 8: it defines the infix operator == that can be conveniently used in place

of beq_role. As expected, other boolean equality functions and their respective convenient notation are defined analogously for other datatypes.

Component datatype

Let us now look at the two remaining core elements, component and binding. Listing 4.6 depicts the component datatype.

```
1 Inductive component : Type :=
2     Component : ident -> path -> implementationClass ->
3                 controlLevel -> list component -> list interface ->
4                 list binding -> component
```

Listing 4.6: component datatype

A component also possesses an identifier and a path indicating its level in the hierarchy. As expected, its implementationClass field is a string holding the java class of its implementation. Its controlLevel field determines its level of control: this feature however, is currently a mere *placeholder* with default value Configuration, it is only kept for the sake of possible future enhancements. A list of subcomponents, a list of interfaces and a list of bindings conclude its composition.

Before proceeding to the definition of bindings, let us discuss two auxiliary functions. A component is composed by several elements. It is therefore rather useful to make them easily accessible through *projections*. For instance, Listing 4.7 depicts such a projection for the component's identifier element.

```
1 Definition projectionComponentId (c:component) : ident :=
2   match c with
3     | Component id p cl _ lc li lb => id
4   end.
5
6 Notation "c '->id' " := (projectionComponentId c).
```

Listing 4.7: Projection for the component identifier element

Indeed, a simple pattern matching exposes the component's internal structure and allows to easily return the intended identifier value. Moreover, line 6 defines a convenient notation for its use: the expression c–>id stands for projectionComponentId c. Naturally, analogous projections are defined for the remaining elements (subcomponents, interfaces, etc).

Another useful function regards the indexation of components. Searching through a list for a component with a specific identifier is achieved by the function depicted by Listing 4.8.

```
1 Function get_comp (id:ident) (l:list component): option component :=
2     match l with
3       | nil     => None
4       | e :: r => if (e->id) == id then Some e else get_comp id r
5     end.
6
7 Notation "l '[' id ']'" := (get_comp id l).
```

Listing 4.8: Function to find a component by its identifier

At each recursion step the *head* element e of parameter l is checked for its identifier value. If it is equal to parameter id, then e is returned, otherwise it simply recurs on its *tail* list r (line 4). No component is returned in case l is empty (line 3). Further, line 7 defines a convenient notation for its use: the expression l[id] stands for get_comp id l.

Binding datatype

For last, bindings are connecting components together through their interfaces. Listing 4.9 depicts the binding datatype.

```
1 Inductive binding : Type :=
2     Normal : path -> ident -> ident -> ident -> ident -> binding
3   | Export : path -> ident -> ident -> ident -> binding
4   | Import : path -> ident -> ident -> ident -> binding.
```

Listing 4.9: binding datatype

As mentioned in Section 2.1, there are three types of bindings: normal, export and import (Figure 2.2, Figure 2.4, and Figure 2.3, respectively). Their path indicates the component they belong to, and their remaining ident constituents identify the involved components and interfaces. For the sake of clarity, Figure 4.1 depicts a component hierarchy including an example of each type of binding.

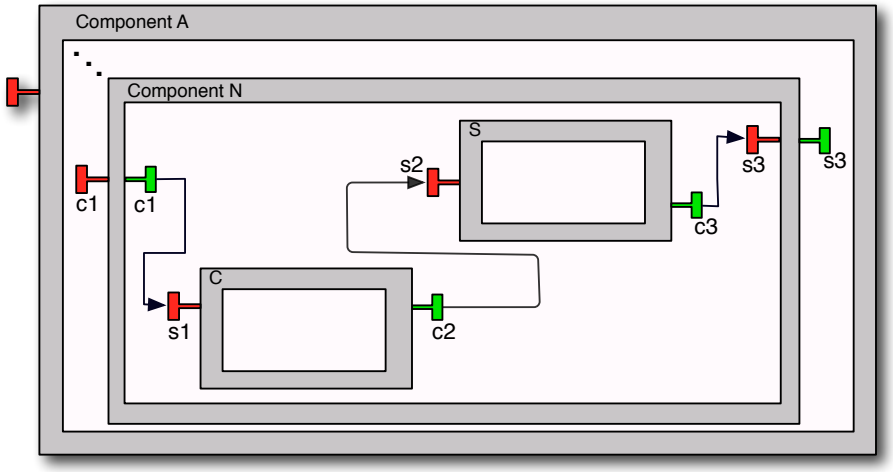

Figure 4.1: binding examples

From left to right, the three bindings illustrated by Figure 4.1 are depicted by Listing 4.10.

```
1 Export pb (Ident "c1") (Ident "C") (Ident "s1").
2 Normal pb (Ident "C") (Ident "c2") (Ident "S") (Ident "s2").
3 Import pb (Ident "s3") (Ident "S") (Ident "c3").
```

Listing 4.10: binding examples in MEFRESA

where **pb** is (Ident "Component A" :: ... :: Ident "Component N" :: nil), i.e., the adequate path for the bindings hold by the component named *"Component N"*.

A complete example

Having discussed the three core elements of the GCM, it still remains to see them all together at once. For the sake of clarity, in the following we show how the aforementioned component *"Component N"* is represented in MEFRESA. First, let us mechanize its subcomponent *"C"*. Listing 4.11 depicts such a task.

```
1 Parameter p   : path.
2 Parameter iclC : implementationClass.
3 Parameter sigS1 : signature.
4 Parameter sigC2 : signature.
5
6 Definition cpath : path := (app p (Ident "Component N"::nil)).
```

```
7
8  Definition C : component :=
9   Component (Ident "C") cpath iclC Configuration
10      (*subcomponents*)
11      nil
12      (*interfaces*)
13      (Interface (Ident "s1") sigS1 (app cpath (Ident "C"::nil))
14                 External Server Functional Mandatory Singleton ::
15       Interface (Ident "c2") sigC2 (app cpath (Ident "C"::nil))
16                 External Client Functional Mandatory Singleton :: nil)
17       (*bindings*)
18       nil.
```

Listing 4.11: Model for component *"Component N"* (part 1 of 3)

Lines 1-4 define some parameters for this example. Parameter **p** stands for the path of component *"Component N"*. Then, iclC, sigS1 and sigC2 stand for the implemenetation class of component *"C"*, signature of interface S1 and signature of interface C2, respectively. Line 6 defines the path for component *"C"* by appending the appropriate identifier. The definition of component *"C" per se* starts at line 8. Its understanding should pose no doubt. It is set with the default control level value Configuration, has no subcomponents — thus represented by an empty list (line 11) — features two interfaces adequately defined in lines 13-16, and has no bindings (line 18).

```
19 Parameter iclS : implementationClass.
20 Parameter sigS2 : signature.
21 Parameter sigC3 : signature.
22
23 Definition spath : path := cpath.
24
25 Definition S : component :=
26   Component (Ident "S") spath iclS Configuration
27      (*subcomponents*)
28       nil
29       (*interfaces*)
30       (Interface (Ident "s2") sigS2 (app spath (Ident "S"::nil))
31                  External Server Functional Mandatory Singleton ::
32        Interface (Ident "c3") sigC3 (app spath (Ident "S"::nil))
33                  External Client Functional Optional Singleton :: nil)
```

```
34        (*bindings*)
35        nil.
```

Listing 4.12: Model for component *"Component N"* (part 2 of 3)

Subcomponent *"S"* has the same structure and is naturally defined analogously. Its mechanization is depicted by Listing 4.12. Note that both components *"S"* and *"C"* possess the same path (line 23), we assign a new variable for the sake of clarity.

```
36 Parameter sigC1 : signature.
37 Parameter sigS3 : signature.
38
39 Definition pN : path := cpath.
40
41 Definition N : component :=
42   Component (Ident "Component N") p "_blank" Configuration
43    (*subcomponent*)
44    (C :: S :: nil)
45    (*interfaces*)
46    (Interface (Ident "c1") sigC1 pN
47                External Server Functional Mandatory Singleton ::
48     Interface (Ident "c1") sigC1 pN
49                Internal Client Functional Mandatory Singleton ::
50     Interface (Ident "s3") sigS3 pN
51                Internal Server Functional Optional Singleton   ::
52     Interface (Ident "s3") sigS3 pN
53               External Client Functional Optional Singleton   :: nil)
54     (*bindings*)
55     (Export pN (Ident "c1") (Ident "C") (Ident "s1") ::
56      Normal pN (Ident "C") (Ident "c2") (Ident "S") (Ident "s2") ::
57      Import pN (Ident "s3") (Ident "S") (Ident "c3") :: nil).
```

Listing 4.13: Model for component *"Component N"* (part 3 of 3)

For last, composite component *"Component N"* is illustrated by Listing 4.13. The careful reader may notice the definition of the variable pN (line 39). This, is subsequently used as path for component *"Component N"*'s interfaces and bindings. Indeed, all of its constituents (subcomponents, interfaces and bindings) possess the same path.

The remaining of its mechanization should pose no doubt. It mainly differs from the two precedent mechanizations (Listing 4.11 and Listing 4.12) in that it

also possesses subcomponents, interfaces of internal visibility, and bindings.

4.1.2 Well-formed component architectures

The above definition of the GCM core elements allowed us to see how to compose components. Indeed, by exploiting their hierarchical structure one can build arbitrary complex architectures. Yet, randomly constructing these by solely respecting the inherent typing rules for the component, interface and binding structures, is naturally insufficient w.r.t. the GCM specification compliance. In order to be certain that we are building valid GCM architectures we need to formalize its requirements.

Let us define a predicate concerning the *well-formedness* of a component. Listing 4.14 depicts such predicate.

```
1 Inductive well_formed_component c : Prop :=
2     | WellFormedComponent:
3       (forall c', In c' (c->subcomps) -> well_formed_component c') ->
4       unique_ids (c->subcomps)                                      ->
5       well_formed_interfaces (c->itfs)                              ->
6       well_formed_bindings  (c->bnds) (c->subcomps) (c->itfs)       ->
7       well_formed_component c.
```

Listing 4.14: well-formed component definition

Basically, a component is *well-formed* if all its subcomponents are individually well-formed (line 3), and do not possess common identifiers (line 4). Further, its interfaces and bindings must also be well-formed (lines 5-6).

The intended meaning of the unique_ids predicate should be clear from its name. Listing 4.15 depicts its definition.

```
1 Inductive not_in_l (i:ident) (l:list component): Prop :=
2   | NotInNil  : l = (nil: list component) ->
3                 not_in_l i l
4   | NotInStep : forall (c:component) (r:list component),
5                 l = c :: r              ->
6                 (((c->id) == i) = false) ->
7                 not_in_l i r            ->
8                 not_in_l i l.
9
```

```
10 Inductive unique_ids (l:list component) : Prop :=
11   | Unique_Base: l = nil -> unique_ids l
12   | Unique_Step: forall c r,
13                  l = c :: r          ->
14                  not_in_l (c->id) r ->
15                  unique_ids r        ->
16                  unique_ids l.
```

Listing 4.15: unique_ids predicate definition

It is composed by two constructors: Unique_Base and Unique_Step. The former simply states that an empty list has indeed unique identifiers. The latter covers the case when the parameter l is composed by a head element c and a tail list r (line 13). The subsequent two premisses ensure that c's identifier is not equal to the components' identifier in the tail r (line 14), and for last, the predicate itself is applied to the tail r (line 15). Concerning the not_in_l predicate, it is essentially defined in the same manner. The *NotInNil* constructor deals with the case when the parameter l is empty (lines 2-3). The NotInStep constructor deals with the more interesting case, when l is composed by a head element c and a tail r (line 5). c's identifier is checked against parameter i (line 6), and the predicate itself is recursively applied to the tail r (line 7).

Regarding component's interfaces, the sole requirement is that they are uniquely identifiable by their id and visibility values. Listing 4.16 formalizes this constraint.

```
1  Inductive not_in_l_pairs (i : ident) (v : visibility) l : Prop :=
2      NotInPairNil  : l = (nil:list interface) -> not_in_l_pairs i a l
3    | NotInPairStep : forall (int : interface) (r : list interface),
4              l = int :: r                                          ->
5              ((int->id) == i) && ((int->visibility) == v) = false ->
6              not_in_l_pairs i v r                                  ->
7              not_in_l_pairs i v l.
8
9  Inductive unique_pairs (li: list interface) : Prop :=
10     UniquePairsNil : li = nil -> unique_pairs li
11   | UniquePairsStep: forall int r,
12                      li = int :: r                                  ->
13                      not_in_l_pairs (int->id) (int->visibility) r ->
14                      unique_pairs r                                 ->
15                      unique_pairs li.
```

```
16
17 Definition well_formed_interfaces li : Prop := unique_pairs li.
```

Listing 4.16: well_formed_interfaces predicate definition

Indeed, the not_in_l_pairs predicate does the main work. The expression not_in_l_pairs i v l should be read as: there is no interface in **l** that possesses both the same identifier and visibility values as **i** and **v**, respectively. The constructor NotInPairsNil covers the case where **l** is empty — it is simply true by definition (line 2). The constructor NotInPairStep accounts for the case where **l** contains at least one element: a head element **int** followed by a tail **r** (line 4). The arguments **i** and **v** are checked against the identifier and visibility values of **int** (line 5), and finally the same process is applied to the tail **r**, making the predicate recur (line 6).

The understanding of the unique_pairs predicate should pose no doubt, it essentially follows the same rationale, and relies on not_in_l_pairs to achieve its intended significance.

Before further proceeding into well-formedness concerns, let us define an auxiliary function concerning the indexation of interfaces. Listing 4.17 defines a function retrieving one interface from a list of interfaces by its identifier and visibility values.

```
1 Function get_interface (id:ident) (v:visibility)
2           (l:list interface) : option interface :=
3   match l with
4     | nil    => None
5     | i :: r => if ((i->id) == id) && ((i->visibility) == v then
6                     Some i
7                 else
8                     get_interface id v r
9   end.
10
11 Notation "l '[' id ',' v ']'" := (get_interface id v l).
```

Listing 4.17: Function to find an interface by its identifier and visibility values

Its comprehension is straightforward. The *head* element i of parameter l is checked for its identifier and visibility values (line 5). Should they be equal to parameters id and v, respectively, interface i is returned (line 6). Otherwise, the function recurs in the *tail* r (line 8). Naturally, no interface is returned if l is empty (line 4).

Moreover, a convenient notation is defined (line 11): l[id, v] stands for get_interface id v l.

The last remaining piece concerns the well-formedness of bindings. Listing 4.18 illustrates its definition.

```
1 Definition valid_binding b lc li : Prop :=
2   match b with
3     | Normal p i j k l => normal_binding i j k l lc
4     | Export p i j k   => export_binding i j k lc li
5     | Import p i j k   => import_binding i j k lc li
6   end.
7
8 Definition well_formed_bindings lb lc li : Prop :=
9   forall b, In b lb -> valid_binding b lc li.
```

Listing 4.18: well_formed_bindings predicate definition

The careful reader may notice one particularity regarding the definition of the well_formed_bindings predicate: unlike well_formed_component and well_formed_interfaces, it takes more than one parameter into account. A binding is hold by a component and serves as a means for components to communicate with each other, and thus cannot exist by itself. As such, it should come as no surprise that its well-formedness depends on its surroundings, i.e. the components and interfaces involved in its definition.

Regarding the mechanization depicted by Listing 4.18, its understanding should pose no doubt. Basically, the predicate valid_binding is applied to all bindings $b \in lb$ (line 9). This, performs a case analysis on b and its validity is checked depending on whether it is a normal, export or import binding (lines 2-5). For instance, Listing 4.19 depicts the normal_binding predicate.

```
1 Function normal_binding (idC1 idI1 idC2 idI2:ident)
2                  (lc:list component)  : Prop :=
3  match lc[idC1], lc[idC2] with
4  | Some c1, Some c2 =>
5     match (c1->itfs)[idI1, External], (c2->itfs)[idI2, External] with
6       | Some i, Some i' => (i->role) = Client /\ (i'->role) = Server
7       | _, _            => False
8       end
9  | _, _ => False
```

```
10  end.
```

Listing 4.19: normal_binding predicate definition

A normal binding establishes a binding between two subcomponents, and is composed by four identifiers denoting (1) the name of the component holding the client interface, (2) the name of this client interface, (3) the name of the component holding the server interface, and finally (4) the name of this server interface. Therefore, checking its validity boils down to ensuring that these identifiers refer to existing client and server interfaces. The parameters idC1 and idC2 contain the name of the components involved in the binding. These are both supposed to be indexed in lc (line 3), if that is not the case it means that the binding is not valid and False is returned (line 9). If both components c1 and c2 are found, then idI1 and idI2 are supposed to be indexed in their list of interfaces, respectively. Further, since it is a normal binding, it implies that it is established between two external interfaces (line 5). Failure to find the referred interfaces means that the binding is not valid and False is returned (line 7). Otherwise, its validity depends on whether the interfaces i and i' are indeed of client and server roles, respectively (line 6).

As expected, the predicates export_binding and import_binding are defined analogously. The main difference lies in the fact that these bindings are not between two subcomponents, but between a parent component and one subcomponent (or vice-versa).

A first well-formedness proof

Having defined our well-formedness predicates it is time to see them in action. Let us retake our running example concerning component *"Component N"* discussed in Subsection 4.1.1 (see Listing 4.11, Listing 4.12 and Listing 4.13).

First, let us reason about its subcomponents well-formedness. Lemma 4.1.1 addresses the well-formedness of component *"C"*.

Lemma 4.1.1. *C_is_well_formed:*
well_formed_component C.
Proof.

We start by applying the WellFormedComponent constructor, which yields four subgoals to discharge. The first and second subgoals concern its subcomponents:

their individual well-formedness and the uniqueness of their identifiers. Since it has no subcomponents both premisses are trivially established.

The third subgoal regards the well-formedness of its interfaces. These need to be uniquely identifiable by the tuple composed by their identifier and visibility values. It is clearly the case as it only possesses two external interfaces named s1 and c2.

Finally, the fourth subgoal requires the well-formedness of its bindings. This is also trivially proved as it is a primitive component and thus has no bindings. And therefore the proof concludes. □

As expected, proving the well-formedness of component *"S"* follows the same rationale. Indeed, both possess the same structure. Regarding component *"Component N"*, its well-formedness is tackled by Lemma 4.1.2.

Lemma 4.1.2. *N_is_well_formed:*
well_formed_component N.
Proof.

Applying WellFormedComponent yields four subgoals to prove.

It possesses two subcomponents (component "C" and component "S"), both are well-formed as previously discussed. Further, their identifiers are different, and therefore the first two subgoals are easily discharged.

It features four interfaces. Two of external visibility, and two of internal visibility. Indeed, some share the same identifier, yet, these are of different visibility value, and thus still uniquely identifiable.

For last, its three bindings are established from and to existent components/interfaces, and from client to server. Thus, also well-formed, and the proof concludes. □

4.1.3 Well-typed component architectures

A component may be well-formed but still unable to start execution. Indeed, its well-formedness guarantees involve important aspects regarding its composition, yet, further insurances are needed for the overall good functioning of the system in terms of its application dependencies. These are dictated by simple typing rules (see [4, p. 22]).

An interface possesses cardinality and contingency attributes. These determine its supported communication model and the guarantee of its functionality availability, respectively. For instance, as dictated by the GCM specification, for proper system execution we must ensure that singleton client interfaces are bound at most once. Indeed, for client interfaces only those with multicast cardinality are allowed to be bound more than once.

Analogously, similar constrains apply to the interfaces' contingency attribute. An interface of mandatory contingency is guaranteed to be available at runtime. This is rather obvious for server interfaces as they implement one or more service methods, i.e., they do have a functionality of their own. Client interfaces however, are used by service methods that require other service methods to perform their task. It therefore follows that a client and mandatory interface must be bound to another mandatory interface of server role. As expected, interfaces of optional contingency are not guaranteed to be available.

MEFRESA captures these requirements by defining a well-typed predicate as depicted by Listing 4.20.

```
1 Definition well_typed (c:component) : Prop :=
2   sound_cardinality c /\  sound_contingency c.
```

Listing 4.20: well_typed predicate definition

Its definition is straightforward, it is a simple conjunction between two more interesting predicates: sound_cardinality and sound_contingency. Listing 4.21 depicts the latter.

```
1 Definition sound_contingency (c:component) : Prop :=
2         subc_client_external_mandatory_itfs_are_bound c /\
3         client_internal_mandatory_itfs_are_bound c.
```

Listing 4.21: sound_contingency predicate definition

As shown by the sound_contigency predicate we split its definition into subc_client_external_mandatory_itfs_are_bound and client_internal_mandatory_itfs_are_bound. Indeed, the former checks the external interfaces, while the latter the internal ones.

```
1 Inductive client_internal_mandatory_itfs_are_bound c : Prop :=
2  | CIMI_Bound:
```

```
3   (forall itf, In itf (c->itfs) -> (itf->role)  = Client ->
4    (itf->visibility) = Internal -> (itf->contingency) = Mandatory ->
5    forall li,
6     li = export_recipients (itf->id) (c->bnds) (c->subcomps) ->
7     (li <> nil) /\
8         (forall i, In i li -> (i->contingency) = Mandatory)) ->
9     (forall c', In c' (c->subcomps) ->
10        client_internal_mandatory_itfs_are_bound c')        ->
11    client_internal_mandatory_itfs_are_bound c.
```

Listing 4.22: client_internal_mandatory_itfs_are_bound predicate definition

Listing 4.22 depicts the client_internal_mandatory_itfs_are_bound predicate. Basically, we need to check that for all client, internal, and mandatory interfaces of some component **c** (lines 3-4), their interface recipients list originating from export bindings are non-empty — i.e. at least bound once — (lines 5-7) and of mandatory contingency (line 8). Naturally, the same applies recursively to the remaining of the component hierarchy (lines 9-10).

As expected, the subc_client_external_mandatory_itfs_are_bound predicate is defined analogously w.r.t. the subcomponents external interfaces. For the sake of completeness, Listing 4.23 depicts its definition.

```
1  Inductive subc_client_external_mandatory_itfs_are_bound c : Prop :=
2   | CEMI_Bound:
3   (forall c', In c' (c->subcomps) ->
4    (forall int, In int (c'->itfs) -> (int->role) = Client ->
5    (int->visibility) = External -> (int->contingency) = Mandatory ->
6    (*these type of itfs can only be in a Normal or Import binding*)
7
8    forall l l',
9     l=normal_recipients (c'->id) (int->id) (c->bnds) (c->subcomps) ->
10    l' = import_recipients (c'->id) (int->id)  (c->bnds) (c->itfs) ->
11    (app l l' <> nil) /\
12    forall i, In i (app l l') -> (i->contingency) = Mandatory))      ->
13   (*recursive call*)
14   (forall c', In c' (c->subcomps) ->
15            subc_client_external_mandatory_itfs_are_bound c')        ->
16   subc_client_external_mandatory_itfs_are_bound c.
```

Listing 4.23: subc_client_external_mandatory_itfs_are_bound predicate definition

A first well-typedness proof

Retuning to our running example from Subsection 4.1.1, let us discuss component *"Component N"*'s well-typedness.

First, we need to establish that component *"Component N"* has indeed a sound contingency. Lemma 4.1.3 addresses this concern.

Lemma 4.1.3. *N_has_sound_contingency:*
sound_contingency N.
Proof.

By unfolding the statement to prove, we can see that it is actually a conjunction of two predicates: subc_client_external_mandatory_itfs_are_bound and client_internal_mandatory_itfs_are_bound.

For the former we start by applying its constructor CEMI_Bound and get two subgoals to discharge. The first one requires that all subcomponents' client, external and mandatory interfaces are indeed bound, while the second one recursively requires the same for the component hierarchy inner levels. Component "Component N" contains two subcomponents: "C" and "S". Both possess one client interface, "c2" and "c3", respectively. Yet, only "c2" is of mandatory contingency, and it is indeed adequately bound through a normal binding to "s2" (which is also of mandatory contingency). Proving the second subgoal is straightforward: both subcomponents "C" and "S" are primitive, thus possess no subcomponents. And therefore subc_client_external_mandatory_itfs_are_bound is discharged.

Proving client_internal_mandatory_itfs_are_bound follows the same rationale. Only the interface "c1" from component "Component N" is concerned and it is indeed adequately bound to "s1". □

The demonstration that component *"Component N"* has a sound cardinality follows the same spirit. Basically, we need to exhaustively check that all concerned interfaces are meeting the cardinality-related requirements. For such simple component architecture, it is easy to see that it is indeed the case as all the present interfaces are of singleton cardinality and are all bound at most once.

Finally, we can show that component *"Component N"* is indeed well-typed. This is illustrated by Lemma 4.1.4.

Lemma 4.1.4. *first_well_typedness_proof:*
well_typed N.
Proof.

Establishing the well-typedness of component "Component N" boils down to proving that it has a sound contingency and a sound cardinality. This is clearly the case as previously discussed. □

4.1.4 Well-formedness and well-typedness decidability

A component that is both well-formed and well-typed is said to be ready to start execution. The definition of these predicates gives us the ability to interactively prove inside Coq that a component system can, or not, start its execution. The burden of such a demonstration for one component c will therefore inherently depend on the complexity of **c**. In fact, one may wonder about the feasibility of undertaking such proof, i.e., whether it is indeed possible to actually prove or disprove that **c** is well formed and well typed[2]. In other words, it is worth knowing that both predicates are *decidable*.

First, we need a predicate expressing decidability. Listing 4.24 depicts one such predicate.

```
1 Definition decidable (P:Prop) := P \/ ~ P.
```

Listing 4.24: decidable predicate definition

Its definition is rather simple: it takes one proposition and returns the disjunction of this proposition itself with its negation. For our purposes, we shall use it for our well-formedness and well-typedness predicates. Indeed, proving them decidable is of paramount importance, it ensures the existence of an algorithm capable of automatically determining their valuation.

Generally, there are two approaches for proving a predicate's decidability. On the one hand, one can attempt to prove it by exploiting the structure of the involved parameters, showing that for all of its constructors it is indeed possible to infer a *True* or *False* valuation. On the other hand, one can write a function that acts as its computational counterpart, i.e., it returns a boolean value, *true*

[2]It should be noted that we are in the context of Coq's constructive logic, theorems that are true in classical logic (i.e. the excluded middle) may not be provable in Coq.

or *false*, in accordance with the predicate's valuation. Then, one needs to show that this function is *sound* — whenever it returns *true* the predicate's valuation is *True* —, and *complete* — whenever the predicate's valuation is *True*, the function returns *true*. Such functions are often called *decision procedures*. Finally, proving the predicate's decidability has now become trivial since it is intrinsically related with this boolean function.

In the following, we use the first approach for demonstrating the decidability of the well_formed_component predicate, and the second approach for tackling the decidability of the well_typed predicate. As we shall see in detail in Chapter 5, this choice is related with software engineering aspects: we need the functional code computing the well-typedness of a component, while an analogous function for the well-formedness would be redundant.

Well-formedness decidability

As shown by Listing 4.14, the well_formed_component predicate is composed by other predicates: unique_ids, well_formed_interfaces, and well_formed_bindings. Naturally, proving their decidability is required in order to demonstrate that well_formed_component is decidable.

For instance, let us consider the unique_ids predicate. As shown by Listing 4.15, its definition includes the not_in_l predicate (see Listing 4.15). As such, the first step is to prove its decidability. This is demonstrated by Lemma 4.1.5.

Lemma 4.1.5. *not_in_dec:*

```
forall id l, decidable (not_in_l id l).
Proof.
```

We start by applying induction on the structure of l*, and thus obtain two subgoals. Both require the original goal but applied to an empty list, and to a list composed by a head element* c *and a tail* l*.*

The first subgoal is trivial. We recall that the decidable *predicate actually stands for a disjunction. In particular, for this case:* not_in_l id nil ∨ ¬not_in_l id nil*. Its left-hand side proposition its evidently true. Proving* not_in_l id nil *can easily be achieved by applying the* NotInNil *constructor, at which point we only need to show that* nil = nil*, which is discharged by reflexivity.*

The second subgoal requires the demonstration of decidable (not_in_l id (c::l)) *by assuming* decidable (not_in_l id l) *as induction hypothesis. At this point we can safely assume that* (c ->id) = id *is either true or false.*

For the first case, we can show that the right-hand side of our goal is true. Indeed, since (c ->id) *is equal to* id*, then it is easy to see that* not_in_l id (c::l) *is false. Thus, we need to prove the negation of false, which is a tautology.*

For the second case, we need to use the induction hypothesis. We know that (not_in_l id l) *is either true or false. If it this true, then we can show that* not_in_l id (c::l) *is also true. Indeed, since* (c ->id) = id *is false we have all the ingredients to discharge this subgoal by applying the* NotInStep *constructor. If it is false, then it follows that* not_in_l id (c::l) *is also false, as we have in context the negation of one of its premises. And thus the proof concludes.* □

Having demonstrated not_in_l's decidability, we are now in a position to address the unique_ids predicate. Its decidability is discussed by Lemma 4.1.6.

Lemma 4.1.6. *unique_ids_dec:*

```
forall (lc:list component), decidable (unique_ids lc).
Proof.
```

We start by applying induction on the structure of lc*. This, naturally yields two subgoals: we now need to demonstrate the original goal for an empty list, and then for a list composed by a head element* c *and a tail list* l*.*

The first subgoal is trivial, we can easily demonstrate unique_ids nil *by applying its* Unique_Base *constructor.*

The second subgoal is more elaborated. From the induction hypothesis we know that unique_ids lc *is either true or false. Further, from Lemma 4.1.5 we know that this is also the case for* not_in_l (c ->id) lc*. We need to consider all their pairwise combinations, and thus we have four cases to deal with.*

For the first case, we have both assumptions with a valuation of true. It this therefore easy to show that unique_ids (c :: lc) *holds by applying the* Unique_Step *constructor.*

The remaining three cases possess at least one assumption with a valuation of false. Thus, it is always possible to show that ¬ unique_ids (c :: lc) *holds, as*

we always have the negation of one of its premises in context. And thus we can conclude the proof. □

Proving the decidability of the well_formed_interfaces and well_formed_bindings predicates follow the same rationale. We account for all possible cases by recurring to structural induction, and reasoning symbolically on the parameters. However, it should be noted that well_formed_component c also possesses a premise involving a nested recursive call: $\forall c', \ c' \in (c.subcomps) \rightarrow well\_formed\_component\ c'$. Alas, coping with such definitions in Coq is somewhat cumbersome, and requires delving into its intricacies. In the following, we demonstrate an approach that can be seen as a general recipe for dealing with such cases.

First, let us prove an auxiliary lemma that is somewhat peculiar: it is possible to decide that all components in an empty list are well-formed. Lemma 4.1.7 demonstrates this.

Lemma 4.1.7. *well_formed_component_nil_dec :*
decidable (forall c, In c nil -> well_formed_component c).
Proof.

We can easily show the left-hand side of our goal: $\forall c, \ c \in nil \rightarrow well\_formed\_component\ c$. *This means that we have* $c \in nil$ *as an assumption, which is always false. Since from false anything follows, we can conclude the proof.* □

Having dealt with the case where the list of components is empty, let us now cope with the case where it is composed by a head element and a tail. For this case however, as demonstrated by Lemma 4.1.8, we shall use some parameters to establish the proof. Basically, the parameters include the decidability of the well-formedness of a component **c**, and the decidability of the well-formedness of all components in a list **lc**. Then, we want to show that all components in (cons **e** **lc**) are also well-formed.

Lemma 4.1.8. *well_formed_component_cons_dec*
(c : component)
(Hc : decidable (well_formed_component c))
(lc : list component)
(Hlc: decidable (forall e, In e lc -> well_formed_component e)):
decidable (forall e, In e (cons c lc) -> well_formed_component e).
Proof.

From the Hc *hypothesis we know that well_formed_component* **c** *is either true or false. If it is false, then it evidently follows that* $\neg(\forall e,\ e \in (cons\ c\ lc) \rightarrow well\_formed\_component\ e)$*. We only need to instantiate* **e** *with* **c** *to see that it is the case. If it is true however, we need to account for two other possibilities: from* Hlc *we know that* $(\forall\ e,\ e \in lc \rightarrow well\_formed\_component\ e)$ *is either true or false. If it is true, then we have all the necessary assumptions to prove that the left-hand side of our goal holds: if* **e** *is instantiated with* **c**, *then we can show it with* Hc, *otherwise if* **e** $\in$ lc, *then we can show it with* Hlc. *If it is false however, then we can show the right-hand side of our goal by analogous reasoning.* □

The last remaining ingredient before proceeding to the main proof is depicted by Lemma 4.1.9. Basically, we prove the decidability of the well-formedness of all components member of a list, while having as parameter the decidability of the well-formedness of all components. Indeed, it seems a rather strange lemma to prove. However, it is of paramount importance for the applicability of this approach. We exhibit its proof with the associated Coq script in order to further emphasize the peculiarity of the approach.

Lemma 4.1.9. *well_formed_component_lc_dec:*

```
2       (Hp : forall c, decidable (well_formed_component c)):
3 forall lc, decidable (forall c, In c lc -> well_formed_component c).
4 Proof.
5   exact (fix well_formed_component_lc_dec lc :=
6    match lc as lc return
7      decidable (forall c', In c' lc -> well_formed_component c') with
8       | nil        => well_formed_component_nil_dec
9       | cons c lc => well_formed_component_cons_dec c (Hp c) lc
10                                    (well_formed_component_lc_dec lc)
11   end).
12 Defined.  (*Qed would hide recursion to the caller*)
```

The first thing to note is the use of the fix construct (line 5). This allows to define anonymous recursive functions. Basically, the proof is achieved by defining a recursive function that pattern matches on its parameter lc and uses Lemma 4.1.7 and Lemma 4.1.8 to adequately cover all the cases (lines 8-10). For last, the careful reader may wonder about the use of *Defined* instead of *Qed* (line 12). This

makes Coq keep the proof of Lemma 4.1.9 — it exposes its recursive nature to a caller — while with *Qed* it would become *opaque*.

Finally, we are in a position to prove the decidability of the well_formed_component predicate for all components **c** as demonstrated by Theorem 4.1.1. We also exhibit its proof as a Coq script to illustrate how the approach described so far is put into practice.

Theorem 4.1.1. *well_formed_component_dec:*

```
2   forall c, decidable (well_formed_component c).
3 Proof.
4  fix 1; intros [i p ic cl lc li lb].
5  assert (IHp_dec :=
6      well_formed_component_lc_dec well_formed_component_dec lc);
7  clear well_formed_component_dec.
8  Guarded.
9
10 destruct IHp_dec; destruct (well_formed_bindings_dec lb lc li);
11 destruct (well_formed_interfaces_dec li);
12 destruct (unique_ids_dec lc);
13
14 (*The first subgoal is when all properties are Well-formed. The
15 others have at least one that isn't and are proved by congruence*)
16  [ left; apply WellFormedComponent; simpl; intros; auto | ..];
17  (right; intro Hwf; inversion Hwf; simpl in *; congruence).
18 Qed.
```

We start the proof by performing structural induction on the first argument through the fix tactic[3] and introduce in the context component **c**'s constituents (identifier, path, implementation class, ...) (line 4). The fix tactic also introduces a new hypothesis in the context, named as the current theorem, and with the same type as the goal. Indeed, this is an odd behaviour, and one may be tempted to use this new hypothesis to discharge the goal, but that would make Coq reject the proof. In short, proofs done with fix are required to satisfy a syntactic *guardedness* criterion, and attempting to directly discharge a goal with the hypothesis it introduces violates this criterion.

[3]Not to be confused with the previously mentioned fix construct for recursive functions.

We introduce the proof that all subcomponents of **c** are well-formed into our context and name it IHp_dec (lines 5-6). Further, we remove the hypothesis introduced by fix since we will not use it anymore (line 7). Then, we use the Guarded command to check that Coq's guardedness condition is satisfied (line 8). It is not a tactic, it does not affect the state of the proof. Its sole purpose is to inform about the compliance of this guardedness condition at the current state of the proof, rather than having to wait until its conclusion.

At this point, the main work for this proof is done. The remaining simply accounts for all possible cases regarding the well-formedness decidability of the involved predicates (lines 10-12), and discharges them adequately (lines 16-17).

Finally the decidability of our well_formed_component is proved. Undeniably, the burden of this approach is considerable. This is mainly due to the way Coq is able to cope with nested definitions such as the component datatype. Moreover, the behaviour of the fix tactic is rather undocumented, and is generally not recommended. In fact, even in *Cocorico*[4] it is stated that *"it is not advisable to use it (unless you know what you do)"*. Alas, there are situations where it is indeed required. Nevertheless, for more details on its intricacies the reader is pointed to Giménez's tutorial on recursive types [54] for a very brief discussion on its use.

Well-typedness decidability

Having demonstrated the well_formed_component predicate decidable, it is time to cope with the well_typed predicate. As mentioned earlier, we shall use a different approach to tackle this task: we define some functional code that act as the computational counterpart of the well_typed predicate.

Basically, we need a function that returns the boolean value *true* whenever a component is well-typed, and the boolean value *false* otherwise. Further, it must be able to do it for all components given as input. Listing 4.25 depicts such function.

```
1 Definition well_typed_bool (c:component) : bool :=
2       sound_cardinality_bool c && sound_contingency_bool c.
```

Listing 4.25: well_typed_bool function definition

[4]Cocorico is the official Coq wiki, also known as *The nonterminating Coq wiki*: `http://coq.inria.fr/cocorico/Home`

Basically, it follows the same spirit as the well_typed predicate (see Listing 4.20), except that it is composed by boolean functions rather than predicates. With no surprise, the definition of the sound_contingency_bool function also follows the same spirit as the sound_contingency predicate (see Listing 4.21). Listing 4.26 depicts its definition.

```
1 Definition sound_contingency_bool (c:component) : bool :=
2  client_internal_mandatory_itfs_are_bound_bool c &&
3  subcomponents_client_external_mandatory_itfs_are_bound_bool c.
```

Listing 4.26: sound_contingency_bool function definition

Indeed, the actual work is carried by the two boolean functions client_internal_-mandatory_itfs_are_bound_bool and subcomponents_client_external_mandatory_-itfs_are_bound_bool — the computational counterparts of the predicates depicted in Listing 4.22 and Listing 4.23, respectively. Listing 4.27 depicts the former.

```
1 Fixpoint client_internal_mandatory_itfs_are_bound_bool c : bool :=
2  match c with
3    Component i p ic cl lc li lb =>
4     (client_internal_mandatory_itfs_are_bound_bool_one li lc lb) &&
5     (fix f_sub lc {struct lc} :=
6       match lc with
7         nil     => true
8       | sc :: r =>
9          client_internal_mandatory_itfs_are_bound_bool sc && f_sub r
10     end) lc
11 end.
```

Listing 4.27: client_internal_mandatory_itfs_are_bound_bool function definition

Its understanding should pose no doubt. Basically, it uses an auxiliary function client_internal_mandatory_itfs_are_bound_bool_one to check the interfaces at the current hierarchical level (line 4) and then recurs on the inner levels, i.e., on the subcomponents (lines 5-10).

The function responsible for checking each hierarchical level individually is also fairly simple. Listing 4.28 illustrates its definition.

```
1 Function client_internal_mandatory_itfs_are_bound_bool_one
2   (li:list interface) (lc:list component) (lb:list binding) : bool :=
3   match li[Client, Internal, Mandatory] with
```

```
4     nil     => true
5   | i :: r => check_if_recipients_are_mandatory_aux (i :: r) lb lc
6   end.
```

Listing 4.28: client_internal_mandatory_itfs_are_bound_bool_one function definition

The parameter li holds a list of interfaces. From these, it recovers those that are client, internal, and mandatory (line 3). If there are none meeting this criteria, it simply returns *true* (line 4): it means that there are no interfaces that need to be checked. Otherwise, there is the need to check if the recipients of these interfaces are indeed of mandatory contingency (line 5). This is achieved by yet another auxiliary function depicted by Listing 4.29.

```
1  Function check_if_recipients_are_mandatory_aux (li:list interface)
2                       (lb:list binding) (lc:list component) : bool :=
3    match li with
4      nil     => true
5    | i :: r => let lir := export_recipients (i->id) lb lc in
6                if not_nil lir && check_if_mandatory lir then
7                  check_if_recipients_are_mandatory_aux r lb lc
8                else
9                  false
10   end.
```

Listing 4.29: check_if_recipients_are_mandatory_aux function definition

In short, it recurs through all interfaces in parameter li, recovering their list of recipient interfaces involved in export bindings (line 5). These, must be non-empty and of mandatory contingency (lines 6-7) — this is the purpose of the not_nil and check_if_mandatory functions, respectively. Should that not be the case, the function returns *false* (lines 8-9). Reaching an empty list makes the function return *true* (line 4).

Defining a function that checks if the client, internal, and mandatory interfaces are adequately bound is not enough. We need to prove it correct, i.e., that it is indeed a reliable computational counterpart of the client_internal_mandatory_itfs_are_bound predicate. This is the purpose of Lemma 4.1.10. In order to deal with its nested recursive nature we employ the same approach as seen for the de-

cidability of the well_formed_component predicate. We omit the related auxiliary lemmas and go straight to the main proof.

Lemma 4.1.10. *cim_itfs_bound_correctness:*

```
forall c,
  client_internal_mandatory_itfs_are_bound_bool c = true <->
  client_internal_mandatory_itfs_are_bound c.
Proof.
```

We start by applying the fix *tactic and proceed by asserting the equivalence that we are trying to prove on the subcomponents. We do this in an analogous manner as discussed for the* well_formed_component_dec *theorem (see Listing 4.1.1).*

We want to prove a logical equivalence and therefore the proof is divided in two parts. The first part argues the soundness of the client_internal_mandatory_interfaces_are_bound_bool *function, i.e., whenever it returns the boolean value true, the predicate* client_internal_mandatory_interfaces_are_bound *has indeed a valuation of True. The second part concerns its completeness: it is always able to return the boolean value true, provided that the predicate has a valuation of True.*

For the first part, we start by applying the CIMI_Bound *constructor, thus obtaining two subgoals. For the first one, we need to establish that the concerned* interfaces *are indeed bound to recipients of* mandatory contingency*. Since we have as hypothesis* client_internal_mandatory_interfaces_are_bound_bool li lc lb = true*, it follows that* client_internal_mandatory_interfaces_are_bound_bool_one li lc lb = true *holds. By induction on* li*, and since we pattern match on the expression* li[Client, Internal, Mandatory]*, we can generalize and consider two cases: either it yields an empty list, or not. If it is an empty list, then we have a contradiction, since the* CIMI_Bound *constructor introduced a* client*,* internal *and* mandatory interface *variable int* ∈ *li in our context. Thus we can discharge this subgoal. if it is not an empty list, then we know that either* int *is the head of the list, or it is in the tail. It it is the head of the list, then we know that* not_nil lir && check_if_mandatory lir *is true — where* lir *is the list of recipients from* int: export_recipients (int->id) lb lc*. Thus, we have in context all the necessary assumptions to prove our subgoal. If it is in the tail of the list however, then we use the induction hypothesis, and can conclude this subgoal by unfolding and reducing the adequate definitions.*

For the second subgoal, we need to cope with the nested recursion on the subcomponents.

That is, we need to show that client_internal_mandatory_interfaces_are_bound c' *holds, for* $c' \in c.subcomps$*. From the beginning of this proof, we already asserted the equivalence for the sub*components*. Thus, we can apply this hypothesis and now need to show that* client_internal_mandatory_interfaces_are_bound_bool c' = true*, while having as an assumption that* client_internal_mandatory_interfaces_are_bound_bool c = true*. Since we know it holds for the parent* component **c***, then it must also hold for the sub*component **c'** *since the function definition recurs on all the* component *hierarchy. And thus we can conclude this subgoal.*

The second part of this proof regards the other implication: we now assume client_internal_mandatory_interfaces_are_bound c *and try to prove* client_internal_mandatory_interfaces_are_bound_bool c = true*. It should be noted that the function in our goal is defined by a boolean conjunction between* client_internal_mandatory_interfaces_are_bound_bool_one li lc lb *and the recursive call on the sub*components*. Thus, we need to prove that both functions return the boolean value true. For the first one, we start by applying induction on* li*, yielding two subgoals. The first subgoal is trivial: if* li *is empty, so is* li[Client, Internal, Mandatory]*. Thus, the function returns the boolean value true, and we are left to show that* true = true*, which is trivially proved by reflexivity. For the second subgoal we need to show that* client_internal_mandatory_itfs_are_bound_bool_one (a :: li) lc lb = true *holds. From the induction hypothesis we can conclude that* client_internal_mandatory_itfs_are_bound_bool_one li lc lb = true *holds, we now need to take care of the* interface **a***. We need to consider three cases. If* **a** *is not of* client*,* internal *and* mandatory *nature then it is irrelevant for the sake of this proof, and we can proceed to the subsequent case. If it is indeed of such nature, then it may or may not possess recipients of* mandatory contingency*. If it is the case, then we can show our goal by computation, i.e., we can reduce the term and eventually we get a* true = true *goal that is proved by reflexivity. If it is not the case, then we have a contradiction in our context w.r.t.* client_internal_mandatory_interfaces_are_bound c *premisses.*

*For the last part of this proof we need to show that the recursive call made on the sub*components *indeed returns the boolean value true. We start by applying induction on* lc*, which yields two subgoals. The first subgoal is where there is no sub*component*: in this case the function simply returns true and we can*

prove our goal by reflexivity. For the second subgoal we know there is at least one subcomponent and we can use the equivalence on the subcomponents and the induction hypothesis to establish our goal. And thus the proof finally concludes. □

With no doubt, the above proof is long and laborious. Yet, it only tackles one of the involved functions. Indeed, analogous proofs are carried for the functions subcomponents_client_external_mandatory_itfs_are_bound_bool and sound_cardinality_bool. We omit these proofs as they follow the same rationale.

Having established the equivalences between the relevant predicates and functions, we can demonstrate Theorem 4.1.2. We illustrate this proof with a Coq script in order to emphasize how the approach is put into practice.

Theorem 4.1.2. *well_typed_correctness:*

```
2   forall c, well_typed c <-> well_typed_bool c = true.
3 Proof.
4   intro c. unfold well_typed; unfold well_typed_bool.
5   rewrite sound_cardinality_correctness.
6   rewrite sound_contigency_correctness.
7   destruct (sound_cardinality_bool c);
8   destruct (sound_contingency_bool c); intuition.
9 Qed.
```

Indeed, proving Theorem 4.1.2 is mostly achieved by using previous lemmas. And finally, Theorem 4.1.3 shows the decidability of the well_typed predicate. We also demonstrate this proof as a Coq script as it further emphasizes how this general approach differs from the one employed for Theorem 4.1.1.

Theorem 4.1.3. *well_typed_dec:*

```
2   forall c, decidable (well_typed c).
3 Proof.
4   intro c. unfold decidable.
5   rewrite well_typed_correctness.
6   destruct (well_typed_bool c); auto.
7 Qed.
```

Finally the decidability of the well_typed predicate is demonstrated. Further, we also defined a sound and complete decision procedure for the well-typedness

valuation. Defining the computational counterparts of the well-typedness specification was indeed an elaborated task. To this end, recent work [31] aims at making such transformations automatic by extracting functional code from induction specifications and producing its proof of soundness. Yet, at the current stage of development it only works for a certain class of inductive specifications, and a completeness proof seems out of reach.

4.2 An operation language for composing architectures

In the previous section, we saw how to use the component datatype to model a component hierarchy. Yet, for more complex architectures, directly manipulating the component datatype is not very convenient. This is further exasperated if we consider reconfigurations: the software architecture evolves and one needs a simple and elegant solution for coping with such structural changes. Moreover, as mentioned in [32] composing an architecture in a arbitrary manner can lead to an "uncontrolled" architectural modification. Ensuring the consistency of the application's structure, both at deployment time and after performing a reconfiguration is thus of paramount importance.

To this end, we define an OPERATION language for manipulating GCM architectures. In the following, we dedicate the remaining of this section delving intro its intricacies.

4.2.1 Syntax and semantics

Before proceeding to the definition of our OPERATION language *per se*, let us introduce the notion of *state*. In most programming language definitions, a state is a structure that holds all the relevant information for a given program point, namely the mapping between the variables and their values. In our case, a state has the same shape as a component. Listing 4.30 depicts the state datatype.

```
1 Definition state := component.
2 Definition root := nil : list ident.
3 Definition empty_state : state :=
```

```
4        Component (Ident "Root") root "null" Configuration nil nil nil.
```

Listing 4.30: state datatype

The choice of the component structure to represent a state should come as no surprise. It holds all the relevant information w.r.t. the structure of a GCM application, and has the advantage of being "well-known".

As further demonstrated by Listing 4.30, and with no surprise, an empty state is a component that is located at the *root* of the hierarchy — and thus with a nil path —, and without any subcomponents, interfaces and bindings (lines 2-4).

Manipulating the state structure is achieved with our aforementioned OPERATION language. Its syntactic categories for building GCM architectures are defined by Listing 4.31.

```
1  Inductive operation : Type :=
2   | Mk_component : ident -> path -> implementationClass ->
3                    controlLevel -> list component -> list interface ->
4                    list binding -> operation
5   | Rm_component : path -> ident -> operation
6   | Mk_interface : ident -> signature -> path -> visibility ->
7                    role -> functionality -> contingency ->
8                    cardinality -> operation
9   | Mk_binding   : binding -> operation
10  | Rm_binding   : binding -> operation
11  | Seq          : operation -> operation -> operation
12  | Done         : operation.
```

Listing 4.31: operation datatype

The intended meaning of each constructor should raise no doubt. Basically, our OPERATION language allows to make and remove components, make interfaces, and make and remove bindings. Further, we can naturally compose operations as a sequence. For this, we define a convenient notation as depicted by Listing 4.32.

```
1  Notation "op1 ; op2" := (Seq op1 op2).
```

Listing 4.32: Notation for sequence of operations

Last, **Done** stands for the completed *operation* — it has the equivalent role of **skip** in standard programming language definitions.

The design of software architectures can be seen from a transition system point of view. One makes some operation **op**, in some state σ, and ends up with a reduced operation **op'** in some state σ**'**. This can be represented by the following manner.

$$\langle op, \sigma \rangle \longrightarrow \langle op', \sigma' \rangle.$$

Building GCM architectures will therefore require to define these transition rules for each constructor of our language. In other words, to define a semantics. Table 4.1 illustrates the semantics of our OPERATION language. The **.** notation is used for projections, and the involved definitions should have self-explanatory names. We use proof rules in order to ease readability. Basically, they represent the predicate step : (operation * state) → (operation * state) → Prop in MEFRESA, where each proof rule corresponds to a step constructor. We omit the constructor's names and number each rule for the sake of space.

$$\frac{\begin{array}{l} c = Component\ id\ p\ ic\ cl\ lc\ li\ lb \\ well\_formed\_component\ c \\ valid\_component\_path\ p\ \sigma \\ \forall\ c',\ c' \in (get\_scope\ p\ \sigma) \rightarrow (c'.id \neq id) \end{array}}{\langle \textbf{\textit{Mk\_component}}\ c,\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ add\_comp\ \sigma\ c \rangle}\ (1)$$

$$\frac{\begin{array}{l} i = Interface\ id\ s\ p\ v\ r\ f\ co\ ca \\ valid\_interface\_path\ p\ \sigma \\ c = get\_component\_with\_path\ p\ \sigma \\ \forall\ i',\ i' \in (c.interfaces) \rightarrow \\ \quad i'.id = id \rightarrow i'.visibility \neq v \end{array}}{\langle \textbf{\textit{Mk\_interface}}\ i,\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ add\_itf\ \sigma\ i \rangle}\ (2)$$

$$\frac{\begin{array}{l} binding\_is\_not\_a\_duplicate\ b\ \sigma \\ valid\_component\_binding\ b\ \sigma \end{array}}{\langle \textbf{\textit{Mk\_binding}}\ b,\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ add\_binding\ \sigma\ b \rangle}\ (3)$$

$$\frac{\begin{array}{l} valid\_component\_path\ p\ \sigma \\ component\_is\_not\_connected\ p\ id\ \sigma \end{array}}{\langle \textbf{\textit{Rm\_component}}\ p\ id,\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ rm\_comp\ \sigma\ p\ id \rangle}\ (4)$$

$$\frac{valid\_component\_binding\ b\ \sigma}{\langle \textbf{\textit{Rm\_binding}}\ b,\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ rm\_binding\ \sigma\ b \rangle}\ (5)$$

$$\frac{}{\langle \textbf{\textit{Done}};\ op_2,\ \sigma \rangle \longrightarrow \langle op_2,\ \sigma \rangle}\ (6)$$

$$\frac{\langle op_1, \sigma \rangle \longrightarrow \langle op_1', \sigma' \rangle}{\langle op_1;\ op_2,\ \sigma \rangle \longrightarrow \langle op_1';\ op_2,\ \sigma' \rangle}\ (7)$$

$$\frac{\langle op_1, \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}}, \sigma' \rangle}{\langle op_1;\ op_2,\ \sigma \rangle \longrightarrow \langle op_2,\ \sigma' \rangle}\ (8) \qquad \frac{}{\langle \textbf{\textit{Done}},\ \sigma \rangle \longrightarrow \langle \textbf{\textit{Done}},\ \sigma \rangle}\ (9)$$

Table 4.1: Semantics of our *operation* language

The rules from Table 4.1 dictate the premisses that need to hold in order to perform a step. Rule (1), whose constructor name is SMakeComponent, concerns making components. Naturally, it requires the component itself to be well-formed. Further, its path needs to be valid, i.e. to point to a pre-existent component in the hierarchy — this is the purpose of the valid_component_path : path → state → Prop. For last, its identifier must be different than the ones from the components at the same hierarchical level — as expected, the purpose of the get_scope : path → state → component function is to return the component indicated by the path

given as parameter. Performing this step yields a state where the component **c** is adequately added in state σ. This is the purpose of the add_comp : state → component → state function.

Rule (2), named SMakeInterface, follows the same spirit, the main difference is that an interface can share the same identifier with another one, provided that they have a different visibility value. The add_itf : state → interface → state function produces a state with the interface inserted in the hierarchy.

Rules (3) and (5), SMakeBinding and SRemoveBinding respectively, have the same requirement: the binding to be made/removed must be valid, i.e. can exist/exists in the hierarchy and will be/is of normal, import or export type. Further, for the creation of bindings, in order to prevent useless duplication we require that it does not already exists. As expected, the effects of the performed operation are accomplished by the functions add_binding : state → binding → state and rm_binding : state → binding → state, respectively.

Rule (4), named SRemoveComponent, concerns the removal of a component. For this, we need to provide a valid path and it must not be *connected*, that is, not bound to any other component. This removal occurs by the use of the rm_comp : state → path → ident → state function.

The remaining rules concern operation composition. Rule (6), named SSeqFinish, formalizes the intuitive idea that we can skip the idempotent Done operation. Next, rule (7), named SSeq1, precises that op_1 can itself be a composed operation. Orthogonally, rule (8), named SSeq2, is meant for the cases where op_1 is an operation that can be fully reduced in one step. Finally, rule (9) is named SDoneRefl and simply attests the reflexive nature of the step predicate.

The rules from Table 4.1 only define a one step transition. However, there are cases where we may want to reason about multiple steps transitions. This is naturally achieved by the reflexive transitive closure of our step predicate. Listing 4.33 depicts its definition in MEFRESA.

```
1 Definition relation (X:Type) := X -> X -> Prop.
2
3 Inductive rt_closure (X:Type) (R: relation X) (x:X) : X -> Prop :=
4   | rt_base : rt_closure R x x
5   | rt_step : forall (y z : X),
6               R x y ->
```

```
7                rt_closure R y z ->
8                rt_closure R x z.
9
10 Notation " op '/' s '~~~>*' op' '/' s' " :=
11                        (rt_closure step (op,s) (op',s')).
```

Listing 4.33: Reflexive transitive closure definition of step

First, we define the notion of relation (line 1). Basically, it is a predicate expecting two arguments of the same type. Its valuation dictates whether the arguments are related or not. Then, we proceed by defining a general notion of reflexive transitive closure (lines 3-8). It includes two constructors rt_base and rt_step. The former defines the reflexive case, i.e., when the two parameters are the same. The latter can be read as follows: if **x** and **y** are related (line 6), and there is a path from **y** to **z** (line 7), then there is a path from **x** to **z** (line 8). Finally, for convenience we define a notation for the reflexive transitive closure of the step predicate (lines 10-11).

A first architecture through an operation

Having seen the syntax and semantics of our OPERATION language, it is time to see it at work. We shall demonstrate its use on our running example. For this however, we shall assume that component *"Component N"* is located at the root of the hierarchy. We make this slight modification since we will be building a concrete component architecture and thus we cannot work with an uninstantiated path parameter — note that component *"Component N"*'s path was left as an undefined parameter in the previous examples.

First, let us demonstrate the operation to build the enclosing component, that is, *"Component N"*. Listing 4.34 depicts its definition.

```
1 Definition root : path := nil.
2
3 Parameter sigC1 : signature.
4 Parameter sigS3 : signature.
5
6 Definition build_N (id: ident) : operation :=
7   Mk_component id root "null" Configuration [] [] [];
8   Mk_interface (Ident "c1") sigC1 [id] External
```

```
9                 Server Functional Mandatory Singleton;
10  Mk_interface (Ident "c1") sigC1 [id] Internal
11                Client Functional Mandatory Singleton;
12  Mk_interface (Ident "s3") sigS3 [id] Internal
13                Server Functional Optional Singleton;
14  Mk_interface (Ident "s3") sigS3 [id] External
15                Client Functional Optional Singleton.
```

Listing 4.34: operation for the component *"Component N"*

Its understanding should pose no doubt. For the sake of clarity, we start by defining root as a nil path (line 1) — it emphasizes the idea that we are building this component at the top of the hierarchy. The build_N operation is parametrized by id in order to improve readability (line 6), and the implementationClass parameters sigC1 and sigS3 are left uninstantiated. Moreover, it should be noted that at this point we cannot establish the bindings as the subcomponents are yet to be made.

Listing 4.35 depicts the operation to make subcomponent *"C"*.

```
1 Parameter iclC : implementationClass.
2 Parameter sigS1 : signature.
3 Parameter sigC2 : signature.
4
5 Definition build_C (id: ident) (pN: path) : operation :=
6   Mk_component id pN iclC Configuration [] [] [];
7   Mk_interface (Ident "s1") sigS1 (app pN [id])
8                External Server Functional Mandatory Singleton;
9   Mk_interface (Ident "c2") sigC2 (app pN [id])
10               External Client Functional Mandatory Singleton.
```

Listing 4.35: operation for the component *"C"*

Its definition is rather straightforward. The only subtlety concerns the interfaces' path: the path of an interface should indicate the component it belongs to, thus we append pN to the identifier id.

The operation for creating subcomponent *"S"* follows the same spirit and his depicted by Listing 4.36.

```
1 Parameter iclS : implementationClass.
2 Parameter sigS2 : signature.
3 Parameter sigC3 : signature.
4
```

```
Definition build_S (id: ident) (pN: path) : operation :=
  Mk_component id pN iclS Configuration [] [] [];
  Mk_interface (Ident "s2") sigS2 (app pN [id])
               External Server Functional Mandatory Singleton;
  Mk_interface (Ident "c3") sigC3 (app pN [id])
               External Client Functional Optional Singleton.
```

Listing 4.36: operation for the component *"S"*

Finally, Listing 4.37 depicts the operation for establishing the bindings.

```
Definition bind_system (pN: path) : operation :=
 Mk_binding (Export pN (Ident "c1") (Ident "C") (Ident "s1"));
 Mk_binding (Normal pN (Ident "C") (Ident "c2")
                    (Ident "S") (Ident "s2"));
 Mk_binding (Import pN (Ident "s3") (Ident "S") (Ident "c3")).
```

Listing 4.37: operation for binding the components

For convenience, we put everything together in Listing 4.38. Basically, we simply use the sequence constructor to compose the overall operation.

```
Definition build_running_example : operation :=
 let iN := Ident "Component N" in
 let iC := Ident "C"           in
 let iS := Ident "S"           in
 build_N iN; build_C iC [iN]; build_S iS [iN]; bind_system [iN].
```

Listing 4.38: Overall operation for component *"Compoennt N"*

So far, we only put together an operation, we do not possess an actual representation of the component architecture. For this, we must *reduce* the build_running_example operation. This is achieved demonstrating the statement depicted by Listing 4.39.

```
exists s, build_running_example / empty_state ~~~>* Done / s.
```

Listing 4.39: Logical statement regarding the reduction of build_running_example

Proving the above statement amounts to applying the appropriate rule for each step and establishing their premisses. Mixing deduction with computation solves this goal in rather a straightforward manner. To some extend, it is like using an interpreter where each step demands requirements that need to be satisfied in order to be allowed.

We omit the details of this proof as it mostly boils down to adequately apply the semantic rules shown in Table 4.1, and refer the reader to the online development for its details.

4.2.2 Building well-formed architectures

In the previous section, we saw how to build component architectures through an operation language. Further, we introduced the general notion of state — which is basically a component —, and in particular we defined an empty state (see Listing 4.30). Yet, while the definition of state possess the same structure of a component, we need a tailored well-formedness predicate for it. Indeed, this ensures the well-formedness of the complete component hierarchy starting from the root element. Listing 4.40 depicts its definition.

```
1 Definition well_formed (s:state) : Prop :=
2   match s with
3     | Component id p ic cl lc li lb =>
4       id = Ident "Root" /\ p = root /\ ic = "null" /\
5       cl = Configuration /\ li = nil /\
6       well_formed_component s
7   end.
```

Listing 4.40: well_formed predicate definition

In practice, manipulating a component architecture consists in interacting with the state's subcomponents — constituent lc —, and bindings — constituent lb. Thus, the remaining constituents are left untouched, that is, they should remain as they are in the empty state.

A first fact to establish concerns the well-formedness of the empty state. Lemma 4.2.1 addresses this point.

Lemma 4.2.1. *empty_state_is_well_formed:*
well_formed empty_state.
Proof.

We can split this proof into six subgoals. Naturally, the first five subgoals are trivially discharged by reflexivity. The remaining subgoal concerns the well-formedness of a component without subcomponents, interfaces and bindings. Apply-

ing the WellFormedComponent *constructor yields four subgoals. The first and second subgoals concern the well-formedness and the* identifier *unicity of the sub*components*. Since there are none to consider, both subgoals are trivially discharged. The third and fourth subgoals require the well-formedness of the* interfaces *and* bindings*, respectively. The same argument applies: there are none to check, and thus the proof concludes.* □

Another aspect discussed in the previous section was the possibility to reason on the overall execution of (sequence of) operations. This was achieved through the definition of the reflexive and transitive closure (see Listing 4.33) of the step relation defined by Table 4.1. To this end, an important theorem to prove is that reducing an operation in a well-formed state, ends up in a state that is also well-formed. Listing 4.41 depicts this theorem.

```
1 Theorem validity:
2   forall s       , well_formed s ->
3   forall op s', op / s ~~~>* Done / s' ->
4                      well_formed s'.
```

Listing 4.41: validity statement

Proving the above theorem is achieved by case analysis on the *operation* language constructors. For the constructors that actually cause an effect on the state — Mk_component, Mk_interface, ... — we need to show that performing one step yields a well-formed state. Thus, the proof delves into the effects caused by functions such as add_comp, add_itf, etc. The Done constructor is idempotent, and thus evidently maintains the state well-formed. For last, for the Seq constructor we need to explicitly deal with multiple-steps reductions. By induction on the reflexive transitive closure of step we get the base case that is trivially true by reflexivity. Further, for the induction step we use the induction hypothesis that requires to show the well-formedness of the attained intermediary state. Yet, we know this state was reached by a one step reduction. Thus, it is necessarily well-formed as we already showed that the remaining constructors always yield a well-formed state.

The details of this proof are rather elaborated as they mostly deal with the intricacies of the involved functions. The interested reader is referred to the online development for an exhaustive and document account of this proof.

Essentially, the above theorem gives the certitude that any composition of operations that meet the premisses will yield a well-formed component architecture. Moreover, it should be noted that we do not require the starting state to be empty. This is particularly relevant for structural reconfigurations, as these occur on an already deployed component architecture.

The reader may wonder about a similar theorem concerning the well-typedness. Clearly, it would not be possible to establish such a theorem. Indeed, it is easy to see that expecting the well-typedness to be maintained at every step of an operation would not be feasible. For instance, a component architecture is not well-typed right after adding an interface of mandatory contingency and client role.

The well_formed predicate concerns the structure and modular composition of a component architecture. Therefore, it is legitimate to expect that this property always holds. For instance, it is never reasonable to let a binding cross the component boundaries occur. The well_typed predicate however, is more related with a component architecture readiness for starting its execution. For instance, specifying an interface with a mandatory contingency and client role intuitively means that the application needs this interface to be bound for a correct behaviour. Further, deployment and structural reconfigurations occur while the application is stopped. Thus, we only need to ensure that a GCM application is well-typed right before it (re)starts its execution.

4.3 Proving Properties

In this section we illustrate some more concrete examples on the use of MEFRESA for proving properties. To this end, Subsection 4.3.1 focuses on aspects of the mechanized GCM specification, Subsection 4.3.2 deals with general properties regarding the operation language, and Subsection 4.3.3 discusses a previous case-study on autonomic computing through structural reconfigurations of a GCM application.

4.3.1 Meeting the Specification: Absence of *Cross-Bindings*

As mentioned in Subsection 4.1.1, there are three types of bindings allowed: *normal*, *export* and *import*. Regarding this, it is said that establishing one of these

types of bindings "*ensure that primitive bindings cannot cross component boundaries except through interfaces*" [1].

The first step in proving this property is to encode the notion of *cross-binding*. In MEFRESA, bindings are locally stored by the direct enclosing component. The involved components and interfaces are specified through their identifiers. Thus, in order to be crossing, a binding must specify a recipient that is not at the same hierarchical level. For instance, a cross-binding of *export* type is defined as follows:

```
Definition cross_export idI1 idC2 idI2 lc li : Prop :=
 (exists i, Some i = li[idI1, Internal] /\ (i->com) = Client) /\

 (~ exists c, Some c = lc[idC2] /\
    (exists i, Some i = (c->interfaces)[idI2, External])) /\

 (forall c, In c lc ->
    exists c', Some c' = (c->subComponents)[idC2] /\
    exists i, Some i = (c'->interfaces)[idI2, External] /\
              (i ->com) = Server).
```

A binding of *export* type is made from an internal client interface — this is what the first part of the above definition stands for. Next, since the binding is supposed to be *crossing* we must state that there exists no component and interface that match the identifiers *idC2* and *idI2*, respectively. This is due to the fact that we do not impose strict restrictions on the identification of our core elements (e.g. components can have the same identifier as long as they are in a different level of the hierarchy). Finally, the last part of the definition establishes the target of the binding: there exists a component, whose identifier is *idC2*, that is a subcomponent of the one being crossed, with an external server interface whose identifier matches *IdI2*.

The same reasoning is applied for the definition of cross-bindings of *normal* and *import* types.

Having defined the notion of cross-binding we shall rely on the following auxiliary lemma to complete the proof.

```
Lemma binding_cannot_be_crossing_if_valid:
  forall b s, valid_component_binding b s ->
              cross_binding b s -> False.
```

Essentially, the above lemma states that we cannot have a binding **b** in a state **s** that is both a valid binding and crossing at the same time — otherwise we could prove *false*.

Proving this lemma amounts to perform case analysis on the type of bindings. We know that bindings can be *normal*, *import* or *export*. Therefore, we need to show that each of these three types of bindings are always established within the bounds of components and being that the case, they cannot be crossing.

The property we want to prove here however is slightly different. We want to prove that meeting the specification, indeed we cannot have cross-bindings. In Mefresa, this boils down to the following theorem.

```
1 Theorem cross_binding_cannot_happen:
2   forall b s, well_formed s -> system_binding b s ->
3               cross_binding b s  -> False.
```

By definition, in the presence of a well-formed state **s** we can only have valid bindings. As such, any bindings **b** belonging to **s** must be valid. Since we know from our previously proved lemma bindings_cannot_be_crossing_if_valid that a binding cannot be valid and crossing at the same time, we can conclude the proof.

4.3.2 Supporting Parametrized ADLs

It is often the case that we want to build a distributed application that has several instances of the same component. For instance, in [26] we showed how to specify a GCM distributed application with fault-tolerance of Byzantine failures [33] that had several instances of the same primitive component.

For this kind of systems, it would be more convenient to specify it in a parametrized manner. To this end, coping with this parametrization amounts to defining a function that takes a component as a *template* and produces **n** instances of that component. For primitive components this is achieved as follows:

```
1 Function generate_from_template template n {struct n} :=
2  match template, n with
3    _                                  , 0   => nil
4  | Component i t p cl nil li nil      , S m =>
5    let li' := update_interfaces_path li (suffix_ident i n) in
6        (Component (suffix_ident i n) t p cl nil li' nil) ::
```

```
7         (generate_from_template template m)
8     | _, _ => nil
9   end.
```

At each recursion step we construct a new component whose identifier is the one from the template suffixed with the current value of **n**. Moreover, we also update the interfaces' path accordingly.

Naturally, provided that the given component is well-formed, we want to ensure that the list of generated components are also well-formed. Lemma 4.3.1 addresses this aspect. The well_formed_components : list component → Prop predicate simply requires that the list of components possess unique identifiers and each component is well-formed.

Lemma 4.3.1. *well_formed_template_generation:*

```
forall c, well_formed_component c ->
forall n, well_formed_components (generate_from_template c n).
```

Proof.

Proving the well-formedness of the generated components is obtained by induction on ***n****. If it is zero, then we get an empty list, which is evidently well-formed. If it is the successor of some natural* ***n*** *then we need to prove that the generation is well-formed for* ***(S n)*** *— the successor of* ***n*** *— given the well-formedness for* ***n*** *as the induction hypothesis. First, we need to establish that the components identifiers are unique, which can be derived by the fact that the suffix that is appended at each iteration is always strictly lower than the precedent, and thus different. Then, we need to prove that each generated component is individually well-formed. Unfolding one time the definition of the generation of* ***(S n)*** *components yields the first produced component whose identifier is* (SUFFIX_IDENT I (S N))*, appended with the recursive call on the argument* ***n****. The first component of this list is well-formed since it has no subcomponents and the interfaces remain well-formed after a path update. As for the tail of the list, it can be proved well-formed from the induction hypothesis. And thus the proof concludes.* □

4.3.3 Structural Reconfigurations

In the realm of autonomic computing one must be able to dynamically restructure the architecture of the application. In [32] we showed how structural reconfigurations were used in order to provide a cost-efficient solution for the saving of power consumption. A slightly simplified architecture[5] of the proposed GCM application is depicted by Fig. 4.2.

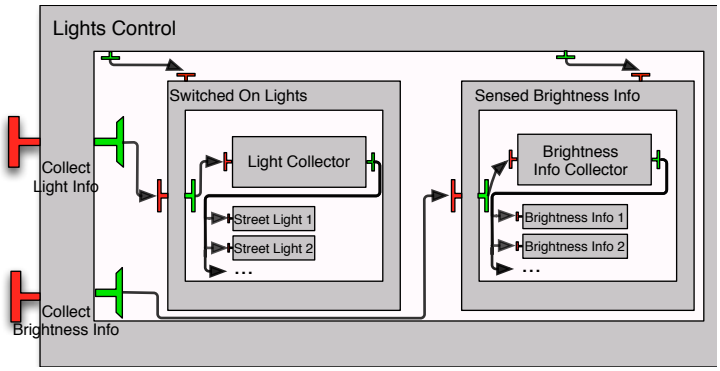

Figure 4.2: Use-Case Architecture

This use-case consists in an experimental component system in charge of switching on and off street lights, according to the perceived luminosity of the surroundings. Basically, in order to address the goal of saving energy, but at the same time offering an acceptable quality of service (i.e. an acceptable luminosity in the streets), the components *Street Light* and *Brightness Info* are added/removed accordingly. Moreover, as said in [32], the use-case starts with three *Brightness Info* components and zero *Street Light* component.

We model this scenario in MEFRESA by defining every component involved in the architecture. For instance, components *Light Collector* and *Street Light* are mechanized as follows:

```
1 Parameter slclass lcclass : implementationClass.
2 Parameter s1 s2 : signature.
3
4 Definition p1 := [Ident "Lights Control"; Ident "Switched On Lights"
    ].
```

[5] GCM components possess a *membrane* part that we do not model in MEFRESA. This however, should not be seen as too much of a shortcoming.

```
5
6  Definition LightComp : component :=
7    Component (Ident "Street Light") p1 slclass Configuration nil
8        [Interface (Ident "Get") "" (app p1 [Ident "Street Light"])
9                External Server Functional Mandatory Singleton] nil.
10
11 Definition LCC : component :=
12  Component (Ident "Light Collector") p1 lcclass Configuration nil
13      [Interface (Ident "Light Info") s1 (app p [Ident "Light
           Collector"])
14              External Server Functional Mandatory Singleton;
15       Interface (Ident "Collect Light Info") s2 (app p [Ident "Light
            Collector"])
16              External Client Functional Mandatory Singleton]
17      nil.
```

In the above definitions, **p1** holds the path of both components in the hierarchy. Furthermore, the remaining components are defined in a similar fashion. Considering the variable LightsControlUseCaseArchitecture as the structure holding the complete hierarchy, we can prove its well-formedness.

```
1 Lemma LightsControlUseCase_is_well_formed:
2   well_formed LightsControlUseCaseArchitecture.
```

The architecture from this use case indeed meets the specification, and proving this lemma poses no particular challenge as it has an explicit structure. The interesting aspect of this use-case comes from the fact that we need to add and remove components at runtime. Indeed, this scenario is yet another example of the usefulness of being able to cope with parametrized specifications.

Before proceeding to the structural reconfigurations, we shall use our *operation* language constructs to define new, higher-level constructs. For instance, we can easily define an operation to build several components at a time:

```
1 Fixpoint mk_components lc : operation :=
2  match lc with
3    nil     => done
4 | c :: r => mk_component c; mk_components r
5 end.
```

This new operation mk_components takes a list of components as argument and

produces a sequence of mk_component operations by means of our **;** constructor. Further, we shall also define an operation for the construction of a list of bindings.

```
Fixpoint mk_normal_bindings (p:path) (icc:ident) (iic:ident)
                            (ics:ident) (n:nat) (iis:ident) :=
 match n with
    0    => done
  | S m =>
    (mk_binding (Normal p icc iic (suffix_ident ics n) iis)) ;
    mk_normal_bindings p icc iic ics m iis
end.
```

Essentially, it is a set of bindings being made from the same client to several servers. The path **p** is given as pointer to the component in the hierarchy where the binding is going to be kept. Then, we specify the identifiers **icc** and **iic** that determine the component and interface from which the binding is made from. Since each binding is made to a different component, we need to generate different identifiers. This is achieved in the same manner as with the generate_from_template function defined above: at each recursion step we suffix **n** to **ics** so that we target different components for every binding built. Finally, the identifier **iis** gives the interface to which the binding is made to.

Having defined our new operations, we can now proceed to the specification of the reconfiguration to add *"Street Light"* components. Below, LightComp is a variable representing our *"Street Light"* component defined above.

```
Definition add_lights_reconfig (n:nat) :=
 mk_components (generate_from_template LightComp n);
 mk_normal_bindings p1
            (Ident "Light Collector") (Ident "Collect Light Info")
            (Ident "Street Light")  n (Ident "Get").
```

Basically, the above creates **n** components by using the *"Street Light"* component as template, and then establishes the required bindings. This, could for instance be used as follows:

```
Lemma adding_lights_red:
 exists s,
  add_lights_reconfig 3 / LightsControlUseCaseArchitecture ~~~>* Done
      / s.
```

```
4
5 Lemma adding_lights:
6  forall s,
7   add_lights_reconfig 3 / LightsControlUseCaseArchitecture ~~~>* Done
        / s ->
8   well_formed s.
```

Proving the above lemmas is achieved by the same reasoning techniques as discussed above. Regarding the first lemma, the key ingredient here to note is that components are created before establishing the bindings — and thus bindings will indeed succeed to be established — and they are well-formed since they are generated from a well-formed template component. As such, reducing this operation to **Done** is possible. It should be noted that while generalizing this lemma for **n** would require a proof by induction, it would still follow the same principle. As for the second lemma, it is a natural consequence of our *validity* theorem (see Listing 4.41).

4.4 Discussion

In this chapter we presented a framework to reason about the structure of GCM applications mechanized in the Coq Proof Assistant. The main novelty of our approach lies in the use of an operation language that allows to build software architectures that are *correct-by-construction*. While we lose some of the automation offered by the usual approach followed by the use of model-checkers, relying on the expressiveness of the Coq Proof Assistant allows us to reason on issues that cannot be addressed by usual model-checking techniques.

The choice of a *deep-embedding* rather than a *shallow* approach for the definition of our operation language is also worth discussing. By explicitly using a *datatype* for the encoding of our operation language there is a clear identification of its syntax, letting no doubt on how operations can be composed. Further, our goal here is also to formalize the GCM specification, a *deep-embedding* emphasizes the operational semantics for the building and reconfiguration of architectures.

Indeed, the definition of the GCM specification in natural language opens the door for several interpretations. Further, such a specification is supposed to be *flex-*

ible and allow various conceivable implementations. Nevertheless, the MEFRESA framework shows that it is possible and fruitful to formalize such an undertaking in a proof assistant like Coq. Further, is also provides a case-study on the mechanization of component models in the context of interactive theorem proving. In fact, the need for such an approach was already discussed in [29].

✠✠✠

In this chapter we presented MEFRESA, a Coq framework for the reasoning on software architectures.

In the following chapter we show its practical facet by demonstrating how we integrated it with the ProActive middleware, and the inherent benefits from this combination.

Chapter 5

Painless integration with ProActive

"Many will call me an adventurer - and that I am, only one of a different sort: one of those who risks his skin to prove his platitudes."

Che Guevara

Contents

Chapter 6

Mechanized behavioural semantics

"Yet, ..."

Nuno Gaspar

Contents

This chapter discusses the mechanization, in the Coq proof assistant, of a behavioural semantics based on the execution trace of synchronized labelled transition systems. Further, we show how it can be used in the context of GCM applications.

Section 6.1 presents the mechanization of LTS and their traces. We show how we can synchronize several LTS in Section 6.2. Then, we exemplify its use in the context of GCM applications in Section 6.3. For last, Section 6.4 discusses the final remarks about this mechanization.

6.1 Labelled transition systems, and traces

```
1 Definition state_mem : Type := list (string * nat) .
2
3 Inductive lts_state :=
4   | LTS_State : nat -> state_mem -> lts_state.
```

Listing 6.1: lts_state datatype

A lts_state is identified by a natural number, and holds an internal memory that is a simple mapping between strings to naturals. For the sake of simplicity we directly use strings to denote variables, and constrain ourselves to natural number values.

Next, let us now see the action datatype. Listing 6.2 illustrates its definition.

```
1 Inductive action : Type :=
2   Action : message -> assignments -> action.
```

Listing 6.2: action datatype

An action is composed by a message, and a set of assignments. As expected, the latter permits to specify the assignment of specific state variables to a transition. The former holds the label, its type of communication and a set of parameters. Listing 6.3 depicts its definition.

```
1 Inductive communication_type : Type :=
2   | Read:  communication_type
3   | Emit:  communication_type.
4
5 Inductive parameter : Type :=
6    Val: nat   -> parameter
7  | Var: string -> parameter.
8
9 Definition parameters := list parameter.
10
11 Inductive message : Type :=
12   | Message: string -> communication_type -> parameters -> message.
```

Listing 6.3: message datatype

A message can either be *reading* or *emitting* (lines 2-3). Typically, these are used to receive and transmit values, respectively. Naturally, the allowed parameters are either values — natural numbers — or variables (lines 6-7).

Finally, we can now see the LTS datatype as depicted by Listing 6.4.

```
1  Definition actions := list action.
2
3  Definition transitions := list(lts_state * action * lts_state).
4
5  Record LTS : Type := mk_LTS
6    { States      : lts_states          ;
7      Init        : lts_state           ;
8      Actions     : actions             ;
9      Transitions : transitions
10   }.
```

Listing 6.4: LTS datatype

Having defined the LTS datatype, we can now define traces. A trace is a sequence of actions resulting from the sequence of transitions taken by the LTS. It can either be finite or infinite. A finite trace means that a *sink* state was attained, thus the execution *deadlocks*. Listing 6.5 defines a LTS trace.

```
1  CoInductive LTS_Trace (A:LTS) : lts_state -> LList action -> Prop :=
2    | lts_empty_trace :
3        forall (m:message) (asgns:assignments) (q:lts_state),
4          In (Action m asgns) (@Actions A) ->
5          lts_target_state A q m = None ->
6          LTS_Trace A q LNil
7
8    | lts_lcons_trace : forall (q q':lts_state) (m:message)
9        (asgns:assignments) (l:LList action),
10         In (Action m asgns) (@Actions A) ->
11         Some q' = lts_target_state A q m ->
12         LTS_Trace A q' l                 ->
13         LTS_Trace A q (LCons (Action m asgns) l).
```

Listing 6.5: Trace definition for a LTS

A LTS_Trace is defined by a co-inductive predicate since we potentially deal with infinite sequences. The expression, LTS_Trace A q l, means that l is a trace in the LTS object A, starting from the state q.

The LTS_Trace predicate is composed by two constructors: lts_empty_trace (line 2) and lts_cons_trace (line 8). The former simply expresses that, from any state, all actions of the LTS yield no target state (lines 3-5). Indeed, the function

lts_target_state : LTS → lts_state → message → lts_state is responsible for computing the attained state from q with an action holding the message m. Thus, the constructor's conclusion is LTS_Trace A q LNil, since the trace from q is empty. The latter constructor however, demands the that we reach a state q' (line 11), and a trace from q' (line 12), in order to conclude LTS_Trace A q (LCons (Action m asgns) l) (line 13).

For the sake of clarity, Listing 6.6 depicts the lts_target_state function.

```
1  Function lts_get_target_state (ts:transitions) (st:lts_state)
2                                 (m:message) : option lts_state :=
3    match ts with
4      nil                              => None
5    | (q, Action me asgn, qs) :: r =>
6        if st === q && beq_message me m then
7          match st with
8            LTS_State _ q_mem =>
9              let q_mem_upd := process_assignments q_mem asgn in
10             Some (q_mem_upd->>qs)
11         end
12       else
13         lts_get_target_state r st m
14   end.
15
16 Definition lts_target_state (A:LTS) (st:lts_state) (m:message)
17                                              : option lts_state :=
18   match m with
19     Message "-" _ _ => Some st
20   | _ => lts_get_target_state (@Transitions A) st m
21   end.
```

Listing 6.6: lts_target_state function definition

The function pattern matches the message m and checks whether its label is equal to "-". If it is the case then it simply returns the current state st (line 19), otherwise it proceeds by calling the lts_get_target_state function (line 20). The is due to the fact that we consider a message with a "-" label as a special case. As we shall see in Section 6.2, this exception is related with the way we deal with synchronization between LTS.

The lts_get_target_state function starts by pattern matching on the LTS list of

transitions (line 3), returning no state if its empty (line 4). If it is not the case (line 5), then we check if the transition at the head of the list possesses a source state that matches the parameter state st, and if the message contained in its action is equal to the parameter message m (line 6). Should that be the case, then it simply process the assignments associated with the taken action (line 9), and updates the target state memory accordingly (line 10). As expected, this the the purpose of the process_assignment : state_mem → assignments → state_mem function, and ->> notation, respectively. Otherwise it recurs on the tail of the list of transitions (line 13).

6.2 Synchronization of LTS, and traces

In the previous Section we saw how to model a single LTS. However, it is often the case that we want to be able to have several LTS communicate with each other. This is achieved by synchronizing their actions.

First, let us formalize the notion of a synchronization vector. Listing 6.7 depicts its datatype.

```
1 Inductive SyncronizationElement : Type :=
2   | WildCard    : SyncronizationElement
3   | SyncElement : string -> SyncronizationElement.
4
5 Inductive SyncronizationOutput : Type :=
6   | GlobalAction: string -> SyncronizationOutput.
7
8 Inductive SynchronizationVector : Type :=
9   SyncVector: list SyncronizationElement * SyncronizationOutput ->
10              SynchronizationVector.
```

Listing 6.7: SynchronizationVector datatype

Basically, a synchronization vector is composed by a list of synchronization elements, and an output element (line 9). The former indicate the actions that need to be considered among the list of LTS being synchronized. The latter stands for the resulting global action. Both synchronization and output elements are represented by string values. For the sake of convenience we also permit synchronization elements to be defined through the WildCard constructor (line 2). Further we define a notation as depicted by Listing 6.8.

```
Definition sync_notation (a:string) : SyncronizationElement :=
  match a with
    "-" => WildCard
  | _   => SyncElement a
  end.

Notation "<< A , .. , B >> -> C" :=
 (SyncVector((cons (sync_notation A) .. (cons (sync_notation B) nil)
     .. ), GlobalAction C)).
```

Listing 6.8: A convenient notation for SynchronizationVector

Basically, it permits Coq to directly understand expressions such as the following one: << "x" , "y" , "-" , "z" >> -> "XYZ" — from left to right, four LTS objects are synchronized on their actions labelled "x", "y", any, and "z", respectively, yielding a global action labelled "XYZ".

Let us now define a type for a communicating network of LTS. We shall call this type Net. Its definition is depicted by Listing 6.9.

```
Inductive Net : Type :=
  | mk_SingletonNet: LTS -> Net
  | mk_Net         : (list LTS * list SynchronizationVector) -> Net.
```

Listing 6.9: Net datatype

It is composed by two constructors: mk_SingletonNet and mk_Net. The former is used for simple models constituted with a single LTS, while the latter permits to specify communicating LTS by means of synchronization vectors.

For this new structure we need a new notion of state. Listing 6.10 depicts the pnet_state datatype.

```
Inductive net_state : Type :=
  | NetSingleton_State: lts_state      -> net_state
  | Net_State         : list lts_state -> net_state.
```

Listing 6.10: net_state datatype

Basically, both its constructors follow the same rationale: they keep track of the involved lst_states.

Before proceeding to the definition of traces for Nets, we need to consider how their transitions are performed. For this case, we need to consider the current net_-

state and the activated synchronization vector. Moreover, each synchronization element may permit more than one action to occur. Thus, a function computing the target net_states must also return the resulting global action. Listing 6.11 depicts such a function.

```
1  Fixpoint net_target_states (net_obj : Net) (q: net_state)
2             (sv:SynchronizationVector) : list (action * net_state) :=
3   match net_obj, q, sv with
4    mk_Net (llts, list_svs), Net_State sts, SyncVector (svs, svo) =>
5     let acts := allowed_actions_from_sv_element llts sts svs in
6     let acts' := combineN acts in
7     (fix fix_net_target_states (list_acts: list (list action)) :=
8      match list_acts with
9        nil       => nil
10     | acts :: r =>
11      let no_msg_states := get_target_lts_states sts llts acts in
12      let params_to_emit := message_parameter_to_emit no_msg_states in
13      let llts' := transmit_message params_to_emit no_msg_states in
14      let global_action := global_action_output svo acts in
15      (global_action, Net_State (llts')) :: fix_net_target_states r
16      end) acts'
17   | _, _, _ => nil
18   end.
```

Listing 6.11: net_target_states function definition

The above function may seem complicated at first sight, and requires a closer look. Basically, it starts by pattern matching on its Net parameter net_obj, net_state parameter q, and synchronization vector parameter sv (line 3). The function is supposed to be used with Net objects and net_state modelling systems with LTS synchronization. Should that not be the case, it simply returns an empty list (line 17). The function allowed_actions_from_sv_element : list LTS → list lts_state → list SyncronizationElement → list (list action) computes the actions that may occur at each LTS composing llts, taking into account its current lts_state held at sts and w.r.t the adequate synchronization element in svs (line 5). Then, it is necessary to generate all combinations of actions that may occur for each LTS (line 6). This is the purpose of the combineN : list (list action) -> list (list action) function. Next, it goes through all generated combinations, and gets for all LTS objects the target

states along with the message to synchronize (line 11). This is the purpose of the get_target_lts_states function. Then, the functions message_parameter_to_emit and transmit_message take care of the variables passing between the synchronized LTS objects. The function global_action_output returns the global action resulting from the synchronized action labels with their parameters (line 14). Finally, the computed global action along with the attained pnet_state is kept, and the function recurs (line 15).

Another useful function concerns the computation of the initial state of a Net. Listing 6.12 depicts its definition.

```
1  Definition init_net_state (net_obj: Net) : net_state :=
2    match net_obj with
3      mk_SingletonNet lts_obj => NetSingleton_State (@Init lts_obj)
4    | mk_Net (list_lts , _svs)  =>
5        let lq := (fix fix_init_lts_states list_lts :=
6            match list_lts with
7              nil          => nil
8            | lts_obj :: r => (@Init lts_obj) :: fix_init_lts_states r
9            end
10       ) list_lts in Net_State lq
11   end.
```

Listing 6.12: init_net_state function

Basically, the above function proceeds by pattern matching on the Net parameter net_obj (line 2). If it is a Net with a single LTS object, than it simply returns the adequate net_state constructor with the initial state of the LTS (line 3). Otherwise, it recursively gathers the initial state of each LTS objects composing net_obj into the local variable lq (lines 5-9), and returns the adequate net_state constructor with lq (line 10).

Let us now define a predicate indicating whether a state can attain another state. Listing 6.13 depicts its formalization.

```
1  Inductive attainable (A:Net) :
2                list net_state -> net_state -> net_state -> Prop :=
3    | Attain0 : forall q mem, q=q -> attainable A mem q q
4    | AttainN : forall qi qf mem,  ~ seen qi mem ->
5                forall list_lts svs, A = mk_Net (list_lts , svs) ->
6                forall qn sv, In sv svs ->
```

```
7              forall ga_lq, ga_lq = net_target_states A qi sv ->
8              In qn (get_list_snd ga_lq) ->
9              attainable A (qi::mem) qn qf ->
10             attainable A mem qi qf.
```

Listing 6.13: attainable predicate definition

It is composed by two constructors: Attain0 and AttainN. Intuitively, the former expresses the idea that a net_state can always attain itself (line 3). The latter is slightly more involved. Basically, in order for a net_state qi to attain net_state qf, there needs to be a net_state qn that belongs to qi's target states, and qf to be attainable from qn. The function net_target_states computes qi's target states (line 7) by using one of the specified synchronization vectors (lines 5-6). However, it returns a list of tuples associating a global action with a net_state. Thus, qn needs to belong to the list of elements at the right of each tuple. This is the purpose of the function get_list_snd : forall X Y : Type, list (X * Y) -> list Y (line 8). Then, qn needs to be able to attain qf (line 9). The careful reader may wonder about the second predicate parameter that stores the list of already seen net_states. There is no point in revisiting already seen net_states in order to check attainability. Thus, it is also required that the source net_state was not already previously seen (line 4). Naturally, this is the purpose of the negated predicate seen : net_state → list net_state → Prop.

As expected, traces for Nets are slightly more involved than traces for LTS. Listing 6.14 formalizes the notion of trace for the Net datatype.

```
1  CoInductive Net_Trace (A: Net) : net_state -> LList action -> Prop :=
2   | lts_trace: forall net_q lts_q,
3       net_q = NetSingleton_State lts_q ->
4       forall A',  A=mk_SingletonNet A' ->
5       In lts_q (@States A')            ->
6       forall l, LTS_Trace A' lts_q l   ->
7       Net_Trace A net_q l
8
9   | net_empty_trace: forall (q:net_state) list_lts list_sv,
10        A = mk_Net (list_lts, list_sv)                    ->
11        forall list_lts_states, q = Net_State list_lts_states        ->
12        indexed_membership list_lts_states list_lts                  ->
13        attainable A nil (init_net_state A) q              ->
```

```
14      (forall sv, In sv list_sv -> net_target_states A q sv = nil) ->
15      Net_Trace A q LNil
16
17  | net_lcons_trace: forall (q:net_state) sync_vec list_lts list_sv,
18       A = mk_Net (list_lts, list_sv)                          ->
19       forall list_lts_states, q = Net_State list_lts_states   ->
20       indexed_membership list_lts_states list_lts             ->
21       attainable A nil (init_net_state A) q                   ->
22       In sync_vec list_sv                                     ->
23       forall ga_lq', ga_lq' = net_target_states A q sync_vec  ->
24       forall ga q', In (ga, q') ga_lq' -> Net_Trace A q' l    ->
25       Net_Trace A q (LCons ga l).
```

Listing 6.14: Trace definition for Net

It is composed by three constructors: lts_trace, net_empty_trace and net_lcons_trace. The first constructor deals with singleton Nets, and thus relies on the previously discussed LTS_Trace predicate (lines 2-7). The second constructor is meant for empty traces, that is, when there is no synchronization vector that can be activated (line 14), and thus the execution is stuck (line 15). Two further requirements are specified. First, it is required that the current net_state q is attainable from the initial state of the Net object A (line 13). Second, the indexed_membership : list lts_state → list LTS → Prop predicate simply states that the i^{th} element of the first parameter belongs to the set of lts_states of the i^{th} element of the second parameter (line 12). For last, the third constructor deals with the most interesting case: there is a successful synchronization between the LTS composing the Net object A. Basically, it is necessary to establish a trace (line 24) from one of the net_states belonging to the list of tuples related the resulting global action with the target net_state (line 23), it

6.2.1 A first synchronization example

6.2.2 Master & Slave example

fiacre tutorial

6.3 Modelling GCM internals

6.4 Discussion

✠✠✠

In this chapter we presented the mechanization of a behavioural semantics based on the execution trace of synchronized labelled transition systems. Further, we exemplified its use in the context of GCM applications.

In the following chapter we discuss the works related with this thesis.

Chapter 7

Related Work

"If you know the enemy and know yourself you need not fear the results of a hundred battles."

Sun Tzu

Contents

7.1 On the HyperManager use case

[26] madelaine case study

Compositional Verification for Component-Based Systems and Application http://www-verimag.imag.fr/ sifakis/atva2008.pdf [34]

FACS: The maturity attained by the CADP toolbox made it a reference tool among the formal methods community. Several case studies have been published, namely industrial ones addressing other goals than verification. For instance, in [35] Coste et. al. discuss performance evaluation for systems and networks on

chips. More closely related with our work we must refer the experiments presented in [36]. A dynamic reconfiguration protocol is specified and model-checked, however their focus is on the reconfiguration protocol itself rather than reconfigurable applications.

Indeed, many works can be found in the literature embracing a behavioural semantics approach for the specification and verification of distributed systems. Yet, literature addressing the aspects of reconfigurable applications remains scarce.

Nevertheless, we must cite the work around BIP (Behaviour, Interaction, Priority) [37] — a framework encompassing rigorous design principles. It allows the description of the coordination between components in a layered way. Moreover, it has the particularity of also permitting the generation of code from its models. Yet, structural reconfigurations are not supported.

Another rather different approach that we must refer is the one followed by tools specifically tailored for architectural specifications. For instance, in [38] Inverardi et. al. discusses CHARMY, a framework for designing and validating architectural specifications. It offers a full featured graphical interface with the goal of being more *user friendly* in an industrial context. Still, architectural specifications remain of static nature.

Looking at the interactive theorem proving arena we can also find some related material. In [39] Boyer et. al. propose a reconfiguration protocol and prove its correctness in the Coq Proof Assistant [11]. This work however, focuses on the protocol itself, and not in the behaviour of a reconfigurable application.

7.2 On the Mefresa framework

Many approaches regarding the formalization of component models can be found in the literature. Yet, to the best of our knowledge this is the first work aiming at providing a mechanized framework that applies the *correct-by-construction* paradigm to the world of component-based engineering. Nevertheless, we must cite the work from Henrio et al. [40] on a framework for reasoning on the behaviour of GCM component composition mechanized in Isabelle/HOL [13]. Our approach is still considerably different in that we provide a semantics for an *operation* language for the building and reconfiguration of GCM architectures.

Another work involving the use of a proof assistant is the work by Johnsen et. al. on the Creol framework [41]. Creol focuses on the reasoning of distributed systems by providing a high-level object oriented modelling language. Its operational semantics are defined in the rewriting logic tool Maude [42]. This work however does not contemplate architectural reconfigurations and follows a methodology in the style of a Hoare Logic.

Indeed, while the use of proof assistants is gaining notoriety, model-checking is still the *de facto* formal approach for both industrial and academic undertakings. Usually, some form of state machine model is used for the design of the intended system, and then a carefully chosen model-checker as back-end is employed as a decision procedure. For instance, in [38] Inverardi et. al. discusses CHARMY, a framework for designing and validating architectural specifications. It offers a full featured graphical interface with the goal of being more *user friendly* in an industrial scenario. Still, architectural specifications remain of static nature. The BIP (Behaviour, Interaction, Priority) framework [37] formalizes and specifies component interactions. It has the particularity of also permitting the generation of code from its models.

Moreover, a formal specification of the Fractal Component Model has been proposed in the Alloy specification language [29]. This work proves the consistency of a (set-theoretic) model of Fractal applications. Their goal was to clarify the inherent ambiguities of the informal specification presented in [1]. Their specification however, is constrained by the first-order relational logic nature of the Alloy Analyzer. In fact, they point to the use of the Coq Proof Assistant in order to overcome this limitation.

For last, from a more engineering point of view, we can refer the work around FPath and FScript [43]. FPath is a domain-specific language for the navigation and querying of Fractal architectures. FScript embeds the FPath language and acts as a scripting language for the specification of reconfigurations strategies. The main goal of this work however is to alleviate the need to interact with the low-level API. Further, reliability of these reconfigurations is ensured by run-time checking, while we are more concerned on providing guarantees statically.

7.3 On Painless

Several proposals for ADLs can be found in the literature, each with their own particularities and attempting to address their specific application domain concerns.

The work around the ArchWare ADL [44] is of particular interest. They claim that *"software that cannot change is condemned to atrophy"* and introduce the concept of an *active software architecture* in order to address this challenge. Based on the higher-order π-calculus, it provides constructs for specifying control flow, communication and dynamic topology. Unlike Painless, its syntax exhibits an imperative style and type inference is not supported, thus not promoting concise specifications.

Nevertheless, it is sufficiently rich to provide executable specifications of active software architectures. Moreover, user-defined constraints are supported through the ArchWare Architecture Analysis Language. Yet, their focus is more aimed at the specification and analysis of the ADL, rather than actual application execution and deployment. In our work, the user solely defines the architecture of its application, structural constraints are implicit: they are within the mechanized GCM specification. Further, our tool support is tightly coupled with the ProActive middleware.

Also from the realm of process algebras, Archery [45] is a modelling language for software architectural patterns. It is composed by a core language and two extensions: Archery-Core, Archery-Script and Archery-Structural-Constraint. These permit the specification of structural and behavioural dimensions of architectures, the definition of scripts for reconfiguration, and the formulation of structural constraints, respectively. Moreover, a bigraphical semantics is defined for Archery specifications. This grants the reduction of the constraint satisfaction verification to a type-checking problem. However, this process is not guarantee to be automatic, and type-checking decidability remains as future work.

As expected, logic is also a natural formalism of choice. For instance, Gerel [46] is a generic reconfiguration language including powerful query constructs based on first-order logic. Further, its reconfiguration procedures may contain preconditions *à la* Hoare Logic [47]. These are evaluated by brute force. It is unclear how they

cope with the inherent undecidability of such task.

Furthermore, the use of a graph-based formalism for the specification of software architectures is also rather popular. For instance, in [48] two graph-based approaches and related tool support are compared. Another interesting aspect of this work regards the use of the Alloy Analyzer [30], and Maude [49] as a means to show the approaches' feasibility.

More recently, Di Cosmo et. al. defined the Aeolus component model [50]. Their focus is on the automation of cloud-based applications deployment scenarios. Nevertheless, their proposal is still loosely inspired by the Fractal component model [1] whose most peculiar characteristics are its hierarchical composition nature and reconfiguration capabilities. However, while both approaches permit architectural reconfigurations at runtime, its specification is not supported by their ADL, it solely contemplates deployment related aspects. Moreover, support for parametrized specifications is also not covered, forcing the software architect to explicitly define the application's structure.

Regarding Fractal, it is also worth noticing that they try to overcome the lack of support for reconfiguration specification through Fscript [51]. Fscript embeds FPath — a DSL for navigation and querying of Fractal architectures — and acts as a scripting language for reconfiguration strategies. These are not evaluated for their validity. Nevertheless, system consistency is ensured by the use of *transactions*: a violating reconfiguration is *rolled back*. Furthermore, an interesting feature is the support for application-specific architectural invariants.

To conclude, let us mention a survey regarding the approaches for the specification of dynamic software architectures [52]. These are classified into four categories: graph-based, process algebra-based, logic-based, and other. To this end, we place ourselves in the *other* category. Indeed, while formal logic is at the foundation of our approach, all the machinery involving requirement/constraint checking is implicit and done automatically by our Coq development. The software architect solely uses a language with a declarative trait.

Chapter 8

Final Remarks

"Yet, ..."

Nuno Gaspar

In this thesis

Appendix A

Appendix 1

Bibliography

[1] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," 2004. [Online]. Available: http://fractal.ow2.org/

[2] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez, "GCM: a grid extension to Fractal for autonomous distributed components," *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 5–24, 2009.

[3] ETSI Technical Committee GRID, "ETSI TS 102 830 V1.1.1 — GRID; Grid Component Model (GCM); GCM Fractal Management API," ETSI, Technical Specification, 2010. [Online]. Available: http://www.etsi.org/deliver/etsi_ts/102800_102899/102830/01.01.01_60/ts_102830v010101p.pdf

[4] ——, "ETSI TS 102 829 V1.1.1 — GRID; Grid Component Model (GCM); GCM Fractal Architecture Description Language (ADL)," ETSI, Technical Specification, 2009. [Online]. Available: http://www.etsi.org/deliver/etsi_ts/102800_102899/102829/01.01.01_60/ts_102829v010101p.pdf

[5] T. Gurock, "Active Objects and Futures: A Concurrency Abstraction Implemented for C# and .NET," 2007. [Online]. Available: http://blog.gurock.com/wp-content/uploads/2008/01/activeobjects.pdf

[6] L. Henrio, F. Huet, and Z. István, "Multi-threaded Active Objects," in *COORDINATION 2013*, C. Julien and R. De Nicola, Eds. Firenze, Italie: Springer, 2013, 15th International Conference on Coordination

Models and Languages, Florence, Italy, 3–6. [Online]. Available: http://hal.inria.fr/hal-00818482

[7] ActiveEon Company, in collaboration with INRIA, "ProActive Programming: Components. Version 5.4.0," ActiveEon S.A.S, Manual, 2014. [Online]. Available: http://proactive.activeeon.com/index.php?page=manual_proactive

[8] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine, "Behavioural models for distributed fractal components," *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 25–43, 2009.

[9] R. Ameur-Boulifa, L. Henrio, E. Madelaine, and A. Savu, "Behavioural Semantics for Asynchronous Components," INRIA, Research Report RR-8167, Dec. 2012. [Online]. Available: http://hal.inria.fr/hal-00761073

[10] B. Berthomieu, J. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat, "The syntax and semantics of FIACRE," RR, 2009.

[11] The Coq Development Team, "The Coq Proof Assistant Reference Manual," 2012. [Online]. Available: https://coq.inria.fr/

[12] Y. Bertot, P. Castéran, G. i. Huet, and C. Paulin-Mohring, *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*, ser. Texts in theoretical computer science. Berlin, New York: Springer, 2004, données complémentaires http://coq.inria.fr. [Online]. Available: http://opac.inria.fr/record=b1101046

[13] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[14] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, jun 1992, pp. 748–752. [Online]. Available: http://www.csl.sri.com/papers/cade92-pvs/

[15] A. Chlipala, *Certified Programming with Dependent Types*. MIT Press, 2011, http://adam.chlipala.net/cpdt/. [Online]. Available: http://adam.chlipala.net/cpdt/

[16] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

[17] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey, *Software Foundations*. Electronic textbook, 2014.

[18] Y. Bertot, "Coq in a hurry," *CoRR*, vol. abs/cs/0603118, 2006.

[19] N. Gaspar, L. Henrio, and E. Madelaine, "Formally Reasoning on a Reconfigurable Component-Based System — A Case Study for the Industrial World," in *International Symposium on Formal Aspects of Component Software (FACS 2013)*, Oct. 2013.

[20] T. Bolognesi and E. Brinksma, "Introduction to the iso specification language lotos," *Comput. Netw. ISDN Syst.*, vol. 14, no. 1, pp. 25–59, Mar. 1987. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(87)90085-7

[21] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes," in *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2011*. [Online]. Available: http://hal.inria.fr/inria-00583776

[22] R. Mateescu and D. Thivolle, "A model checking language for concurrent value-passing systems," in *Proceedings of the 15th international symposium on Formal Methods*, ser. FM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 148–164. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68237-0_12

[23] G. Winskel, "A note on model-checking the modal mu-calculus," *Theoretical Computer Science*, vol. 83, no. 1, pp. 157 – 167, 1991.

[24] J.-P. Queille and J. Sifakis, "Fairness and related properties in transition systems - a temporal logic to deal with fairness." *Acta Inf.*, vol. 19, pp. 195–220, 1983. [Online]. Available: http://dblp.uni-trier.de/db/journals/acta/acta19.html#QueilleS83

[25] O. Kulankhina, "A graphical specification environment for GCM component-based applications," INRIA, Ubinet Master internship report, 2013. [Online]. Available: http://www-sop.inria.fr/oasis/index.php?page=vercors

[26] R. A. Boulifa, R. Halalai, L. Henrio, and E. Madelaine, "Verifying safety of fault-tolerant distributed components," in *International Symposium on Formal Aspects of Component Software (FACS 2011)*, 2011.

[27] N. Gaspar and E. Madelaine, "Fractal à la Coq," in *Conférence en IngénieriE du Logiciel*, Rennes, France, Jun. 2012. [Online]. Available: http://hal.inria.fr/hal-00725291

[28] N. Gaspar, L. Henrio, and E. Madelaine, "Bringing Coq into the World of GCM Distributed Applications," *International Journal of Parallel Programming*, pp. 1–20, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0264-7

[29] P. Merle and J.-B. Stefani, "A formal specification of the Fractal component model in Alloy," INRIA, Rapport de recherche RR-6721, 2008.

[30] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149

[31] P.-N. Tollitte, D. Delahaye, and C. Dubois, "Producing certified functional code from inductive specifications," in *Certified Programs and Proofs*, ser. Lecture Notes in Computer Science, C. Hawblitzel and D. Miller, Eds. Springer Berlin Heidelberg, 2012, vol. 7679, pp. 76–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35308-6_9

[32] F. Baude, L. Henrio, and P. Naoumenko, "Structural reconfiguration : an autonomic strategy for gcm components," in *Proceedings of The Fifth Inter-*

national Conference on Autonomic and Autonomous Systems: ICAS 2009, 2009.

[33] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[34] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "Compositional verification for component-based systems and application," *IET Software*, vol. 4, no. 3, pp. 181–193, June 2010.

[35] N. Coste, H. Hermanns, E. Lantreibecq, and W. Serwe, "Towards performance prediction of compositional models in industrial gals designs." in *CAV*, ser. Lecture Notes in Computer Science. Springer, 2009, pp. 204–218.

[36] M. A. Cornejo, H. Garavel, R. Mateescu, and N. D. Palma, "Specification and verification of a dynamic reconfiguration protocol for agent-based applications." in *DAIS*, ser. IFIP Conference Proceedings, K. Zielinski, K. Geihs, and A. Laurentowski, Eds., vol. 198. Kluwer, 2001, pp. 229–244.

[37] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[38] P. Inverardi, H. Muccini, and P. Pelliccione, "Charmy: An extensible tool for architectural analysis," in *ESEC-FSE'05, ACM SIGSOFT Symposium on the Foundations of Software Engineering. Research Tool Demos*, September 5-9, 2005.

[39] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486791

[40] L. Henrio, F. Kammüller, and M. U. Khan, "A framework for reasoning on component composition," in *FMCO 2009*, ser. Lecture Notes in Computer Science. Springer, 2010.

[41] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1–2, pp. 23–66, Nov. 2006.

[42] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "The maude 2.0 system," in *Rewriting Techniques and Applications (RTA 2003)*, ser. Lecture Notes in Computer Science, R. Nieuwenhuis, Ed., no. 2706. Springer-Verlag, June 2003, pp. 76–87.

[43] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures." *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 45–63, 2009.

[44] R. Morrison, G. N. C. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. M. Greenwood, "Constructing active architectures in the archware adl," *CoRR*, vol. abs/1006.4829, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1006.html#abs-1006-4829

[45] A. Sanchez, L. S. Barbosa, and D. Riesco, "Bigraphical modelling of architectural patterns." in *FACS*, ser. Lecture Notes in Computer Science, F. Arbab and P. C. Ölveczky, Eds., vol. 7253. Springer, 2011, pp. 313–330.

[46] *Programming generic dynamic reconfigurations for distributed applications*, 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=152129

[47] C. A. R. Hoare, "An axiomatic basis for computer programming," *COMMUNICATIONS OF THE ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[48] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch Lafuente, "Concurrency, graphs and models," P. Degano, R. Nicola, and J. Meseguer, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Graph-Based Design and Analysis of Dynamic Software Architectures, pp. 37–56. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68679-8_4

[49] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 187–243, Aug. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(01)00359-0

[50] R. D. Cosmo, S. Zacchiroli, and G. Zavattaro, "Towards a formal component model for the cloud." in *SEFM*, ser. LNCS, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., vol. 7504. Springer, 2012, pp. 156–171.

[51] P.-C. David, T. Ledoux, T. Coupaye, and M. Léger, "FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures," *Annales des Télécommunications - Annals of Telecommunications*, vol. Volume 64, no. Numbers 1-2 / février 2009, pp. 45–63, Dec. 2008. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00468474

[52] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, ser. WOSS '04. New York, NY, USA: ACM, 2004, pp. 28–33. [Online]. Available: http://doi.acm.org/10.1145/1075405.1075411

[53] P. Merle and J.-B. Stefani, "A formal specification of the Fractal component model in Alloy," INRIA, Rapport de recherche RR-6721, 2008. [Online]. Available: http://hal.inria.fr/inria-00338987

[54] E. Giménez and P. Castéran, "A tutorial on [co-]inductive types in coq," 1998.

List of Acronyms

ADL Architecture Description Language 4

GCM Grid Component Model 3

API Application Programming Interface 10

FIFO First In, First Out 21

ECW E-Connectware 40

LTS Labelled Transition System 47

MCL Model Checking Language 47

RFID Radio-Frequency Identification 3

JMX Java Management Extensions 42