Behavioural Semantics for Asynchronous Components

Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine, Alexandra Savu

Behavioural Semantics for Asynchronous Components

Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine, Alexandra Savu

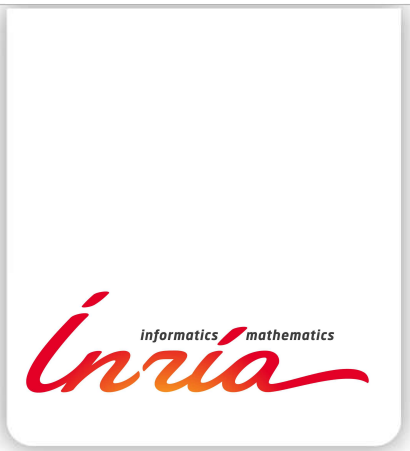
RESEARCH REPORT

N° 8167

December 2012

Project-Team Oasis

ISSN 0249-6399 ISRN INRIA/RR--8167--FR+ENG



Behavioural Semantics for Asynchronous Components

Rabéa Ameur-Boulifa*, Ludovic Henrio†, Eric Madelaine†, Alexandra Savu†

Project-Team Oasis

Research Report n° 8167 — December 2012 — 61 pages

Abstract: Software components are a valuable programming abstraction that enables a compositional design of complex applications. In distributed systems, components can also be used to provide an abstraction of locations: each component is a unit of deployment that can be placed on a different machine. In this article, we consider this kind of distributed components that are additionally loosely coupled and communicate by asynchronous invocations.
Components also provide a convenient abstraction for verifying the correct behaviour of systems: they provide structuring entities easing the correctness verification. This article aims at providing a formal background for the generation of behavioural semantics for asynchronous components. We use the pNet intermediate language to express the semantics of hierarchical distributed components communicating asynchronously by a request-reply mechanism. We also formalise two crucial aspects of distributed components: reconfiguration and one-to-many communications. This article both demonstrates the expressiveness of the pNet model and formally specifies the complete process of the generation of a behavioural model for a distributed component system. The behavioural models we build are precise enough to allow verification by finite instantiation and model-checking, but also to use verification techniques for infinite systems.

Key-words: Behavioural specification, software components, asynchronous communications, futures

This work was partially funded by the ANR Blanc International project MCorePhP, and by the FUI project CloudForce

* Institut Telecom, Telecom ParisTech, LTCI CNRS, Sophia-Antipolis, France
† INRIA-I3S-CNRS, University of Nice Sophia Antipolis, France

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Sémantique Comportementale pour Composants Asynchrones

Résumé : Les composants logiciels fournissent une abstraction de programmation intéressante pour la conception modulaire d'applications complexes. Dans les systèmes répartis, les composants peuvent également être utilisés pour fournir une abstraction de la localisation des processus: chaque composant est une unité de déploiement qui peut être placée sur une machine différente. Dans cet article, nous considérons ce type de composants distribuées, faiblement couplés et communiquant par des appels asynchrones.

Les composants fournissent également une abstraction commode pour vérifier le bon comportement des systèmes: ils fournissent un concept structurant qui facilite la vérification de ses propriétés. Cet article vise à fournir un support formel pour la génération de la sémantique comportementale des composants asynchrones. Nous utilisons le formalisme intermédiaire pNet pour exprimer la sémantique des composants hiérarchiques distribués communiquant de manière asynchrone par un mécanisme de requêtes. Nous formalisons également deux aspects fondamentaux des composants distribués: la reconfiguration et les communications de groupe. Cet article d'une part démontre l'expressivité du modèle pNet et d'autre part spécifie formellement le processus complet de la génération du modèle comportemental d'un système de composants distribués. Les modèles de comportement que nous construisons sont suffisamment précis pour permettre la vérification par instanciation finie et model-checking, mais aussi pour utiliser des techniques de vérification de systèmes infinis.

Mots-clés : Spécifications comportementales, composants logiciels, communications asynchrones, futures

1 Introduction

Ensuring the safety of distributed applications is a challenging task. Not only the network and the underlying infrastructure are not reliable, but already without failures, applications are more complicated to design because of the multiple execution paths possible. To ensure the safety of distributed applications, we propose to use formal methods to be able to verify the correct behaviour of distributed applications. In order to verify properties of programs, it is necessary to choose a programming abstraction that is convenient enough to program, but also that provides enough information to be able to check the properties of the program. We adopt a programming model that is expressive enough to program complex distributed applications but with some constraints enabling the behavioural verification of these application. Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed in component applications, dependencies are defined together with provided functionalities by means of provided/required ports; this improves the program specification and thus its re-usability. Several effective distributed component models have been specified, developed, and implemented in the last years [22, 33, 11, 10] ensuring different kinds of properties to their users. Component models have been chosen as the target programming model for many developments in formal methods, mostly because additionally to the valuable software engineering methodology they ensure, components also provide structural informations that facilitate the use of formal methods: the architecture of the application is defined statically.

However, defining statically the exact structure of the application is sometimes too restrictive. Indeed, especially in a distributed setting, applications must evolve at runtime in order to adapt to changes in the execution environment or to provide improved functionalities. Some component models keep a trace at runtime of the component structure and their dependencies. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities: the application can be adapted by changing some of the components taking part in the composition or changing the dependencies between the involved components. Reconfigurations consist in changing at runtime the component structure, by adding or removing components in the system, or by changing the way components are bound together. In distributed systems, reconfiguration can also be used at runtime to discover services and use the most efficient service available. Also, as some distributed components will naturally migrate from one location to another, they will change their execution environment and may have to adapt to the new execution platform they are moved to. Concerning formal verification, keeping trace of the component structure when building the model of the application not only allows us to build the model in a compositional manner, but also allows us to encode reconfiguration procedures and to verify the properties of the system when some reconfigurations occur.

In this work, we focus on one distributed component model, the GCM (Grid Component Model [10]). This component model originates from the Grid computing community, it is particularly targeted at composing large-scale distributed applications. Its reference implementation GCM/ProActive relies on the notion of active objects, and ensures that, during execution, each thread is isolated in a single component. Because of active objects, components that usually structure the application composition also provide the structure of the application at runtime, in terms of location and thread (a single applicative thread manipulates the state of the component). We call this kind of components *asynchronous components* because they are loosely coupled entities communicating by an asynchronous request-reply mechanism. All those aspects facilitate the use of formal methods for ensuring safe behaviour of applications, but they also require specific developments to produce a formal model of an application built from such components.

This article formalises the construction of a behavioural model for ProActive/GCM components. It describes formally how we can generate a behavioural model in terms of *parameterised*

Networks of synchronised automata (pNets) [6] from the description of the architecture of a GCM/ProActive application and the description of the behaviour of each service method implemented by the programmer. In other words we formalise the automatic construction of the behavioural model for communication, management, and composition aspects.

Our behavioural models are *parameterised*: they can be viewed as a structured composition of labelled transition systems (LTS) that can use parameters/variables. Each pNet is either formed of other pNets or is a single LTS. Parameters can be used as local variables inside a LTS; but they can also be used to define families of pNets of variable size, and to specify the way events occurring in different pNets are synchronised. Once the parameterised behavioural model generated, we can for example generate a finite instance of the model that can be checked against correctness formulas using a model-checking tool. But our behavioural model is richer than what can be checked by finite-state model-checkers and other verification techniques could also be used.

For specifying the interaction between the service methods, we generate behavioural models encoding the following features:

- Futures: futures are frequently used in active object languages, they are place-holders for results of asynchronous invocations, called requests here. We encode in our models the mechanisms for dealing with futures (Section 4.1.5) and the transmission of futures references between components (Section 5.1).
- Component composition: from an ADL (architecture description language), we generate the synchronisations corresponding to the communications that can occur between the different components. We distinguish two cases: primitive components which are the leaves of the composition tree and composite components that are built from other components.
- Primitive components: at the leaves of the hierarchy, from the definition of the service methods, we specify a component able to receive requests and serve each of them one after the other. When a request service terminates, a reply is sent back to the component that emitted the request. The crucial parts composing the model of a primitive component are: the request queue, the handling of communications for sending requests and replies, the futures and their management (Section 4.1).
- Composite components (composites, for short): as our component model is hierarchical, a component can be built from the composition of other components; as composites are instantiated at runtime, it is necessary to specify their behaviour in our model too (see Section 4.2). Each composite is in fact implemented as an active object and thus the internal structure of a composite is very similar to the one of a primitive component.
- Reconfiguration: as stated earlier, reconfiguration plays a major role in distributed systems. This article also provides a behavioural model for reconfigurable components (Section 5.2). To enable the verification of component reconfiguration, we rely on an extended ADL defining all the configurations that will be taken into account in our model.
- One-to-many communications: distributed systems often rely on some form of multicast communications between one emitter and a set of registered receivers. For encoding such cases, the GCM component model defines multicast interfaces. We also formalise the generation of models for these patterns of communication in Section 5.3.

This article is built upon previous works of the authors. The definition of pNets has already been formalised in [6]; in this article we provide a more concise and simpler definition so that this article is self-contained. Concerning the modelling of component features, modelling of primitive

and composite components, and of binding controllers has been described in [6]; [20] provides a study of behavioural models for first-class futures; multicast communications were modelled in [2, 15]. Compared to those previous works, this article first proposes a model aggregating all those features of the GCM component model. More important, this article fully formalises the modelling process, which is a necessary step for the automatic generation of behavioural models for component systems. Also this article is the first one to propose a model for handling of futures in composite components. Overall this article builds upon the individual use-cases studied in our previous works to fully formalise the generation of behavioural models for asynchronous components.

Our behavioural specification is particularly adapted to the reasoning on GCM components but our approach is also applicable to other component and programming models. The component structure of GCM is quite similar to the one of Fractal [16] and SCA [11]; the runtime behaviour of components uses active objects/actor-like computations, which is similar to Creol [29], AmbiantTalk [24], and JCobox [35]. The generation of behavioural models for those frameworks can be adapted from the results of this article. Consequently, this article also shows that the pNets formalism is adapted to the behavioural specification of systems using those framework.

This article is organised as follows. Section 2 gives a brief overview of the GCM component model defined in [10] and its implementation inside the ProActive library; it also gives the abstract syntax we use for the definition of component systems and defines the set of components we consider as *well-formed* in Section 2.3. Section 3 provides a definition of the pNets formalism [6]. Then Sections 4 and 5 contain the main contribution of the paper; they present respectively the basic behavioural model for GCM components and more complex features, namely first-class futures, reconfigurations, and one-to-many interfaces. This paper concludes with an example of building the model of a use-case in Section 6, a comparison with related works in Section 7 and a conclusion. Appendices describe the semantics of pNets, a summary of main behavioural semantic functions, and some details on the model of the example use-case.

2 The Grid Component Model: GCM

GCM has been proposed in the CoreGrid Network of Excellence, it is an extension of the Fractal component model [17, 18] to better address large-scale distributed computing. GCM builds above Fractal and thus inherits its hierarchical structure, the enforcement of separation between functional and non-functional concerns, its extensibility, and the separation between interfaces and implementation.

Figure 1 shows the basic component structure provided by GCM. It introduces most of the terminology used to describe GCM components and their composition. Interfaces are annotated with the type of their methods. Among the notions presented in the figure only multicast interfaces interfaces are specific to GCM.

Fractal does not impose any granularity for the components, but the existence of composite bindings and some of the features of the model suggest a rather fine grained implementation: a primitive component should contain a small number of objects. Overall, the GCM has been conceived with a granularity that is somehow in the middle between small grain Fractal components and coarse grain component models, like CCM where a component is of a size comparable to an application or a service. Somehow, GCM has been conceived thinking of components of the size of an MPI process, though it can be used to define much finer or coarser grain components. In ProActive/GCM the primitive components (and the composite ones too) have this intermediate size: they contain an activity, i.e. an active object, its dependencies, a request queue, and a thread. Somehow, this paper relies on the fact that components are used as structuring enti-

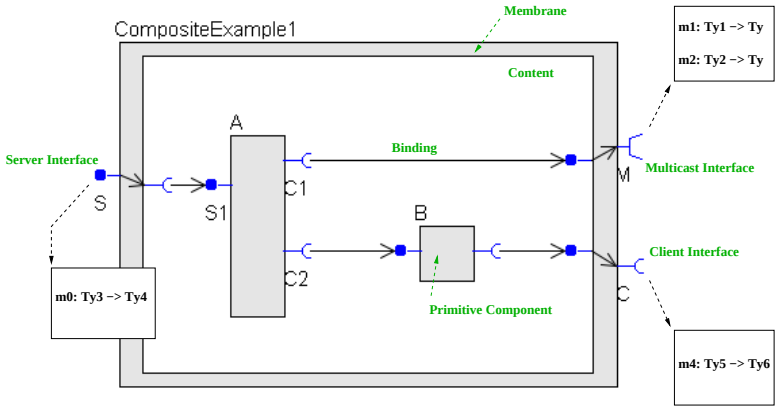


Figure 1: A typical GCM assembly

ties that specify the different threads of the applications and the set of objects manipulated by each of those threads. It is very interesting and convenient to use formal methods to check the properties of GCM applications, as the component structure gives crucial informations on the concurrency that occurs at runtime.

GCM and Fractal come with an ADL (architecture description language) providing a textual (XML-based) way of describing component assembly. Such a description of the application architecture (either textual or graphical) is the starting point of our work. Starting from the architecture description and the description of the behaviour of each component, we build a formal description of the behaviour of the whole application.

2.1 A reference implementation for GCM

ProActive/GCM is a reference implementation of the GCM component model. It is based on the ProActive Java library and relies on the notion of active objects. It is important to note that each component corresponds at runtime to an active object and consequently each component can easily be deployed on a separate JVM and can be migrated. Of course, this implementation relies on design and implementation choices relatively to the purely structural definition provided by the model.

One of the main advantages of using active objects to implement components is their adaptation to distribution. Indeed, by nature active objects provide a natural way to provide loosely coupled components. By loose coupled components, we mean components responsible for their own state and evaluation, and only communicating via asynchronous communications. Asynchronous communications increase and automate parallelism; and absence of sharing eases the design of concurrent systems. Additionally, loose coupling reduces the impact of latency, and limits the interleaving between components. Finally, independent components also ease the autonomic management of component systems, enabling systems to be more dynamic, more scalable and easily adaptable to different execution contexts. That is why we think that a distributed component system should rely on such loosely coupled asynchronous components. That is thus the reason why we think active objects are particularly adapted to implement a distributed component model.

2.2 Informal semantics of asynchronous components

This section describes briefly an informal semantics of GCM/ProActive components. The general principle is that interaction between components is limited to communications, and more precisely to a request/reply mechanism. A more formal and general semantics can be found in [27].

Communications The basic communication paradigm we consider is asynchronous message sending: upon a communication the message is enqueued at the receiver side in a queue. To prevent shared memory between components, messages can only transmit parameters which are copied at the receiver side; no object or component can be passed by reference. This communication semantics is similar to messages in an actor model. We call *requests* messages sent between components. References to components cannot be passed as request parameters.

We call our component model asynchronous because communication does not trigger computation on the receiver side immediately, it just enqueues a request. To allow for transparent asynchronous requests with results, we use transparent first-class futures. The promise for a reply to a request is created automatically when the request is sent, we call it a *future*. For accessing the value of a future, the caller must wait until the request is treated and the result sent. When the request is finished, the result is automatically sent to replace all the references to the corresponding future. Futures are said to be *first-class* if they can be transmitted between components.

Component behaviour The primitive components encapsulate the business code. They generally serve requests in the order they arrived, providing answer for all the requests they receive. They can call other components by emitting a request on one of the client interfaces.

In ProActive/GCM, each component is mono-threaded: a single request is served at a time and no internal concurrency occurs inside a component. However, a component always accepts the reception of a future value or of a request.

While primitive components contain the application logic, composite components have a predefined behaviour because they are only used as composition tools and the programmer expects them to only transmit the requests according to the specified composition. Composites serve requests in a FIFO order, delegating requests to the bound components or to the external ones. Globally, a request emitted by the client interface of a primitive component will be sent unchanged to the server interface of the primitive component that is bound to it, following one or several bindings (several bindings are used when the bounding of a composite is crossed). A composite performs no computation: it only delegates requests.

Collective interfaces GCM components mostly use three kinds of interfaces: singleton, multicast, and gathercast. Singleton interfaces are used to perform one-to-one communications as described above; a singleton client interface must be bound to a single server interface. Multicast interfaces allow one component to be bound to several others, with a configurable communication semantics; in general, a call from a multicast client interface is broadcasted to all the server interfaces bound from it. Gathercast interfaces are somehow symmetrical to multicast ones; they allow many components to be bound to a server interface and come with a synchronisation pattern: when all the client interfaces bound to a gathercast interface have emitted a request, those requests are collected and transmitted as a single one to the component having the gathercast interface.

Components featuring the semantics defined above are loosely coupled; they are better adapted to a distributed setting and easier to program safely by the limited concurrency they

allow. However, the semantic of such components rely on several notions, like for example request queues and futures, that have to be specified when building a behavioural model for those components. This article specifies how those notions can be formally defined as pNets, and how those definitions can be used to build the behavioural model of a component-based application.

2.3 Component Definition

This section defines a hierarchical structure for representing components, we define a syntax for describing the different elements of a GCM component assembly. We also define a set of auxiliary functions that will help us manipulate the component structure, and finally we define what component systems we consider to be well-formed, these are the systems for which we are able to build a behavioural model.

The definitions below rely on several predefined structures. *Type* represents a type, we have several kind of names (even if there is no need to distinguish them strictly): *Name* is an interface name, *CName* and C are component names, *MName* is a method name.

2.3.1 Syntax and Notations

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

For example $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3; exceptionally, such mappings with only a few entries will also be denoted $(3{\mapsto}a)$ in the following. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over I" when formally we should speak of multisets, and still better write "$x \in A_i^{i\in I}$" to mean $\exists i \in I.\, x = A_i$. An empty family is denoted [] (it can be defined as $a_i^{i\in\emptyset}$).

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

2.3.2 Interfaces

Let *SItf* be the description of a service interface, it is characterised by the interface name, the interface cardinality and the signature of one or more service methods. In the same way, *CItf* is the description of a client interface, containing one or more client methods. We use *Itf* to range over interfaces that can be either client or server ones.

$$SItf \quad ::= \quad (Name, Card, MSignature_i^{i\in I})_S \tag{1}$$

$$CItf \quad ::= \quad (Name, Card, MSignature_i^{i\in I})_C \tag{2}$$

$$Itf \quad ::= \quad SItf \mid CItf \tag{3}$$

Concerning cardinalities, client interfaces can be either *singleton* (which means the client interface of the current component is bound to one server interface of another component), or

multicast (meaning that the client interface is bound to more than one components through service interfaces on these components). In GCM, interfaces can also be of cardinality "gathercast" but this case is not treated in this article.

$$Card ::= singleton \mid multicast$$

A method signature, *MSignature*, consists of a method name, an argument type[1], and a return type:

$$MSignature ::= MName : Type \rightarrow Type$$

A *method definition* consists of a *MSignature* and the behaviour of this method, which is a pNet (pNets will be defined in Section 3). M_i range over method definitions. $Signature(M_i)$ returns the signature.

2.3.3 Components

From those definitions we define components, which can be either primitive or composite ones.

$$\begin{aligned} Comp ::= \; & CName < SItf_i^{i \in I}, CItf_j^{j \in J}, M_k^{k \in K} > \\ & \mid CName < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} > \end{aligned}$$

A *primitive component* consists of a name *CName*, a set of Server Interfaces S_{Itf}, a set of Client Interfaces C_{Itf}, and a set of method definitions $M_k^{k \in K}$, where $M_k \in MSignature \times pNet$.

Figure 2 illustrates a simple configuration of a primitive component. It exposes two service interfaces, which can receive requests. It has also a single client interface showing 2 client methods. The corresponding pNet system will be drawn in Figure 4.

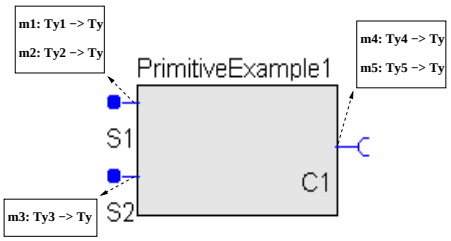


Figure 2: Simple Primitive Component

A *composite component* (a composite, for short) consists of a set of sub-components exporting some server interfaces, some client interfaces, and bindings. A *binding* connects two interfaces of two components, either sub-components of the same composite, or one is a sub-component and the other is the composite. A binding is thus a pair of qualified names. Each qualified name is made of two parts, the first one is either the name of a (sub)component plus the name, or *This* to mean the considered composite component, the second one is the name of an interface.

In the composite component definition, the interfaces are the external ones, the ones visible by the outside world.

$$Binding ::= (QName, QName) \quad (4)$$

$$QName ::= This.Name \mid CName.Name \quad (5)$$

[1] it is not restrictive to consider methods with a single arguments as we have no restriction on type complexity

Figure 1 shows a composite component containing two inner composite components. For example ($A.C1$, $This.M$) is one of the bindings of this component. The interface M is a multicast interface, it can emit two kinds of requests: *m1* and *m2*, with the types defined in the figure.

We define a function Interfaces that given a component returns the indexed set of its interfaces, and a function Name that returns the name of its argument that can be a component, a method, or an interface.

Membrane and internal interfaces In Fractal and GCM, the frontier of a composite component is called a membrane and can intercept incoming calls; it deals with the component management; additionally GCM provides the capacity to specify the content of a membrane as a component composition. Roughly, for each server interface of a composite component accessible from the outside, there is an internal client interface. Here we chose to have no membrane and not specify explicit internal interfaces, more precisely the membrane has no content and the internal interfaces match exactly the external ones. For each interface declared in the composite component definition, a symmetric internal interface is created with the same name and a symmetric role (server or client) as illustrated in Figure 3. By convention, a server interface of cardinality multicast stands for a single server interface, associated with an internal multicast interface.

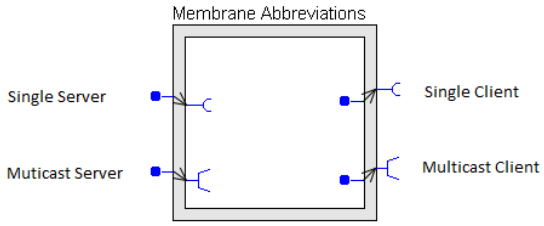


Figure 3: Abbreviations for matching external/internal interfaces

2.3.4 Auxiliary functions

We first define an auxiliary function that takes the symmetric of an interface. It takes an interface and returns the same interface with a symmetric role:

$$\text{Symm} : \mathit{Itf} \rightarrow \mathit{Itf}$$

$$\text{Symm}(\mathit{Name}, \mathit{Card}, \mathit{MSignature}_i^{i\in I})_S = (\mathit{Name}, \mathit{Card}, \mathit{MSignature}_i^{i\in I})_C$$
$$\text{Symm}(\mathit{Name}, \mathit{Card}, \mathit{MSignature}_i^{i\in I})_C = (\mathit{Name}, \mathit{Card}, \mathit{MSignature}_i^{i\in I})_S$$

We then rely on an auxiliary function Get that returns the interface inside a composite component that corresponds to a qualified name. If the qualified name is of the form *CN.IN*, then Get returns the external interface named *IN* of the inner component of name *CN*. Else, the qualified name is of the form *This.IN*; then the inner interface of the composite component is returned, it is the symmetric of the external interface of name *IN*.

$$\text{Get} : \mathit{QName} \times \mathit{Comp} \rightarrow \mathit{Itf}$$

$$\frac{k \in K \qquad \text{Name}(Comp_k) = CN \qquad Itf \in \text{Interfaces}(Comp_k) \qquad \text{Name}(Itf) = IN}{\text{Get}(CN.IN, CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_l^{l\in L} >) = Itf}$$

$$\frac{i \in I \qquad SItf_i = (IN, Card, MSignature_l^{l\in L})_S}{\text{Get}(This.IN, CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_l^{l\in L} >) = \text{Symm}(SItf_i)}$$

$$\frac{j \in J \qquad CItf_j = (IN, Card, MSignature_l^{l\in L})_C}{\text{Get}(This.IN, CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_l^{l\in L} >) = \text{Symm}(CItf_j)}$$

2.3.5 Well-formed components

In our semantics, we only want to deal with components that are correctly formed, for this we define a predicate *WF* that indicates whether a component is well-formed. We suppose there is a sub-typing relation $\trianglelefteq$, and that this relation is classically extended to method signatures, and to families of method signatures. In practice, *WF* is slightly more restrictive than what could be expected of a correct component definition, or what can be found in [28]; this is because we prevent bindings from having the same component as source and destination, and we prevent two bindings originating from the same multicast interface from having the same destination component. Building the behavioural models in those two cases is slightly more complicated; we will come back on this restriction and how to overcome it in Section 5.4.

We first define the predicate UniqueItfNames that takes a set of server interfaces and a set of client interfaces and returns true if no two of these interfaces have the same name.

$$\text{UniqueItfNames}(SItf_i^{i\in I}, CItf_j^{j\in J}) \Leftrightarrow \begin{cases} \forall i, i' \in I.\, i \neq i' \Rightarrow \text{Name}(SItf_i) \neq \text{Name}(SItf_{i'}) \ \wedge \\ \forall j, j' \in J.\, j \neq j' \Rightarrow \text{Name}(CItf_j) \neq \text{Name}(CItf_{j'}) \ \wedge \\ \forall i \in I, j \in J.\ \text{Name}(SItf_i) \neq \text{Name}(CItf_j) \end{cases}$$

Then a primitive component is well-formed if all its interfaces have distinct names and all the interfaces declared in the server interfaces have a corresponding method definition (modulo sub-typing).

$$WF(CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >) \Leftrightarrow \begin{cases} \text{UniqueItfNames}(SItf_i^{i\in I}, CItf_j^{j\in J}) \ \wedge \\ \forall i \in I.\ \text{if } SItf_i = (Name, Card, MSignature_l^{l\in L})_S \\ \quad \text{then } \forall l \in L.\, \exists k \in K.\, Signature(M_k) \trianglelefteq MSignature_l \end{cases}$$

Finally a composite component is well-formed if all its sub-components are well-formed, all the bindings defined bind an existing client to an existing server interface of a compatible type[2], all sub-components have distinct names, and no two bindings start from the same client interface except if this interface is multicast. We additionally require that no binding has the same component as source and destination: there is no binding looping back *directly* to the same component. Finally, no two bindings can have the same multicast[3] interface as source and the same component as destination. The two last conditions of the definition are the ones specific to our model, they are not generally required by the GCM model.

[2] Those interfaces are found thanks to the Get function: they are either interfaces of sub-components or internal interfaces of the composite component.

[3] the fact that the interface must be multicast follows one of the preceding requirements

$$WF(\,CName < SItf_i^{i\in I},\, CItf_j^{j\in J},\, Comp_k^{k\in K},\, Binding_l^{l\in L} >) \Leftrightarrow$$
$$\left\{\begin{array}{l}
\mathrm{UniqueItfNames}(SItf_i^{i\in I},\, CItf_j^{j\in J}) \wedge \\
\forall k, k' \in K.\, k \neq k' \Rightarrow \mathrm{Name}(Comp_k) \neq \mathrm{Name}(Comp_{k'}) \wedge \\
\forall k \in K.\, WF(Comp_k) \wedge \\
\forall (Src, Dst) \in Binding_l^{l\in L}. \\
\quad \exists Name,\, Name',\, Card,\, Card',\, M,\, M',\, MSignature_l^{l\in M},\, {MSignature'}_l^{l\in M'}. \\
\qquad \big(\mathrm{Get}(Src) = (Name, Card, MSignature_l^{l\in M})_C \wedge \\
\qquad\quad \mathrm{Get}(Dst) = (Name', Card', {MSignature'}_l^{l\in M'})_S \wedge \\
\qquad\quad {MSignature'}_l^{l\in M'} \trianglelefteq MSignature_l^{l\in M} \wedge \\
\qquad\quad Card \neq Multicast \Rightarrow \nexists Dst'.\, Dst' \neq Dst \wedge (Src, Dst') \in Binding_l^{l\in L}\big) \quad \wedge \\
\forall (C.CI, C'.SI) \in Binding_l^{l\in L}.C \neq C' \wedge \\
\forall C, CI, C', C'', SI, SI'.(C.CI, C'.SI) \in Binding_l^{l\in L} \wedge (C.CI, C''.SI') \in Binding_l^{l\in L} \Rightarrow C' \neq C''
\end{array}\right.$$

3 pNets: a formalism for defining behavioural semantics

3.1 Term algebra

Our models rely on the notion of parametrized actions. We leave unspecified the constructors of the algebra that will allow building actions and expressions used in our models, let us denote Σ the signature of those constructors. Let $\mathcal{T}_P$ be the term algebra of Σ over the set of variables P. We suppose that we are able to distinguish inside $\mathcal{T}_P$ a set of *action terms* (over variables of P) denoted $\mathcal{A}_P$ (*parametrized actions*), a set of *expression terms* (disjoint from actions) denoted $\mathcal{E}_P$, and, among expressions, a set of *boolean expressions* (guards) denoted $\mathcal{B}_P$. For each term $t \in \mathcal{T}_P$ we define $fv(t)$ the set of free variables of t. For $\alpha \in \mathcal{A}_P$ we also suppose that there is a function $iv(\alpha)$ that returns a subset of $fv(\alpha)$ which are the input variables of α, i.e. the variables newly defined by reception of their value during the action α.

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

For example the actions of Milner's *Value-passing CCS* [32] correspond to the following algebra: terms are τ, $a(x)$ for input actions, $\overline{a}(v)$ for output actions. Then $fv(a(x)) = iv(a(x)) = \{x\}$, whereas $iv(\overline{a}(v)) = \emptyset$.

3.2 The pNets Model

The first comprehensive definition of pNets was published in [6]. The definition we give below is simpler because it does not include transducers and each hole contains a single pNet. The semantics given here is thus shorter but exhaustive; it provides a rigorous formal basis for expressing the semantics of the GCM features. In this section, we define the structure of pLTSs, pNets and Queues, and define their operational semantics.

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

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

- *P is a finite set of parameters, from which we construct the term algebra $\mathcal{T}_P$, with the parametrized actions $\mathcal{A}_P$, the parametrized expressions $\mathcal{E}_P$, 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 $\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle$, 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 $e_j \in \mathcal{E}_P$. Variables in $iv(\alpha)$ are assigned by the action, other variables can be assigned by the additional assignments.*

- *$\rightarrow \subseteq S \times L \times S$ is the transition relation.*

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

pNets are constructors for hierarchical behavioural structures: a pNet is formed of other pNets, or pLTSs at the bottom of the hierarchy tree. Message queues can also appear in leaves of a pNet system. A composite pNet consists of a set of pNets exposing a set of actions, each of them triggering internal actions in each of the sub-pNets. The synchronisation between global actions and internal actions is given by a *synchronisation vector*: a synchronisation vector synchronises one or several internal actions, and exposes a single resulting global action.

Definition 2 (pNets) *A pNet is a hierarchical structure where leaves are pLTSs (or queues defined below): $pNet \triangleq pLTS \mid Queue(\mathcal{M}) \mid \langle\!\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle\!\rangle$ 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 of global actions of the pNet.*

- *$I \in \mathcal{I}_P$ is the set over which sub-pNets are indexed, $I \neq \emptyset$.*

- *$pNet_i^{i\in I}$ is the family of sub-pNets.*

- *$SV_k^{k\in K}$ is a set of synchronisation vectors ($K \in \mathcal{I}_P$). $\forall k \in K, SV_k = \alpha_j^{j\in J_k} \rightarrow \alpha'_k$. Each synchronisation vector verifies: $\alpha'_k \in L$, $J_k \in \mathcal{I}_P$, $\emptyset \subset J_k \subseteq I$, and $\forall j \in J_k.\ \alpha_j \in \mathrm{Sort}(pNet_j)$.*

For each pNet, we define a function sort ($Sort : pNet \rightarrow \mathcal{A}_P$). The sort of a pNet is its signature: the set of actions that a pNet can perform, that is to say the set of labels of its transitions, more formally:

$$\mathrm{Sort}(\langle\!\langle P, S, s_0, L, \rightarrow\rangle\!\rangle) = L \qquad \mathrm{Sort}(\langle\!\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle\!\rangle) = L$$

A pNet composes sub-pNets and expresses by its synchronisation vectors how the different sub-entities are synchronised. $SV_k = \alpha_j^{j\in J} \rightarrow \alpha'_k$ means that each of the sub-pNets can performs synchronously the action α_j; this results in a global action labels α'_k.

When $I = [1..n]$ we denote the pNet as $\langle\!\langle P, L, pNet_1, \ldots, pNet_n, SV\rangle\!\rangle$, and each synchronisation vector as: $\langle \alpha_1, \ldots, \alpha_n\rangle \rightarrow \alpha$. In that case, elements not taking part in the synchronisation are denoted $-$ as in: $\langle -, -, \alpha, -, -\rangle \rightarrow \alpha$.

Queues We also define a particular pNet called $Queue(\mathcal{M})$; it models the behaviour of a FIFO queue. It can be considered as an infinite pLTS with a set of actions depending on the chosen term algebra and of the set of enqueue-able elements $\mathcal{M} \subseteq \mathcal{T}_P$. We suppose that the term algebra has two specific constructors $?Q$ and $!Serve$[4] such that for all set of variables P, $\forall m_i \in \mathcal{M}. !Serve\_m_i \in \mathcal{A}_P \wedge ?Q\_m_i \in \mathcal{A}_P$. Then the queue pNet offers the following actions: $L = \{?Q\_m_i | m_i \in \mathcal{M}\} \cup \{!Serve\_m_i | m_i \in \mathcal{M}\}$. The behaviour of a queue is only FIFO en-queueing/de-queueing of messages.

$$\mathrm{Sort}(Queue(\mathcal{M})) = \{?Q\_m_i | m_i \in \mathcal{M}\} \cup \{!Serve\_m_i | m_i \in \mathcal{M}\}$$

Whenever pNets will be encoded by (ultimately finite) automata structures for model-checking, pNet Queues will naturally be represented by finite automata. However, in order to be able to address more general approaches, and in particular specific model-checking algorithms for unbounded channels, we need to keep a high-level representation of Queues. From these abstract Queues, we will be able to generate both regular representation (for unbound queues), and finite representation (for explicit-state model-checking).

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

$$\overleftarrow{\langle\!\langle P, PN_i^{i\in I} \rangle\!\rangle} \triangleq \langle\!\langle P, V, PN_i^{i\in I}, \{\alpha_j^{j\in J} \to \alpha_j^{j\in J} | \alpha_j^{j\in J} \in V\} \rangle\!\rangle$$
$$\text{where } V = \{\alpha_j^{j\in J} |\, J \subseteq I \wedge \forall j \in J.\, \alpha_j \in \mathrm{Sort}(PN_j)\}$$

This supposes that the elements of V belong to the term algebra and more precisely are action terms.

If all the elements of the family are identical, then we simply write $\overleftarrow{\langle\!\langle P, PN^I \rangle\!\rangle}$.

In fact, the definition of pNets shown here is a "simplified" version of pNets [6] that is convenient for providing a concise formal definition of both pNets themselves and the component specification in terms of pNets. Especially concerning families of pNets, it is not reasonable, in practice, to define all the possible synchronisation vectors inside a family. In [6], we defined a version of pNets where the families are flattened in the enclosing pNet and only the used synchronisations are instantiated. Though more efficient in practice this notation was more complex, that is why a simpler definition is presented in this paper. Section 6 will show an optimised instantiation of the produced pNet structure and synchronisation vectors.

An operational semantics for pNets is given in Appendix A.

3.3 Assumptions on the term algebra

Let us consider several aspects of the term algebra we might use in the description below, those aspects are not related to the pNets semantics but rather to the way we use it.

In our term algebra, we have three basic kinds of actions: *input actions* of the form $?a(x_1, ..., x_n)$ where x_i are input variables, *output actions* of the form $!a(v_1, .., v_n)$, where v_i are values (expressions), and *synchronised actions* of the form a only used for observation purposes. Our actions

[4] We chose constructors coherent with the term algebra we will use in this paper to simplify notations.

can also be parameterized by one or several arguments thus they can be of the form $a(arg)$ or $a(arg, p)$.

In order to express synchronisation vectors of families of pNets, we must allow families of actions to be considered as actions themselves. More precisely, if a and a_i are actions, then actions can be of the form $i{\mapsto}a$ to allow the sub-pNet at index i to perform an action, also we use $i \in I{\mapsto}a_i$ to express the family of actions $a_i^{i\in I}$ triggering a synchronous action on all the sub-pNets indexed in I, or $i{\mapsto}a, j{\mapsto}b$ to synchronise two elements of the family.

Synchronised actions (prefixed neither by ! nor by ?) are not meant to be used anymore for synchronisation purposes, they should just be visible at the top-level of the pNet hierarchy. Consequently, we define an operator that takes an indexed set of pNets and returns the synchronisation vectors that should be included in the parent pNets to allow the visibility of synchronised actions:

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

In the following, those synchronisation vectors dedicated to observation will be *implicitly* included as synchronisation vectors of all our pNets. This means that for all pNets written in the following of this paper, Observe($pNet_i^{i\in I}$) is considered to be included in the set of synchronisation vectors of the considered pNets (where $pNet_i^{i\in I}$ is the set of sub-pNets of the new pNet).

In this paper, we do not use explicitly internal action (τ transition). If the action algebra contains τ transitions, then we would use weak bisimulation notions to deal efficiently with the pNets. More precisely, upon composition of pNets, a sub-pNet will be allowed to perform additional (invisible) τ transitions. In other words, τ would behave similarly to the synchronised actions defined above.

4 Behavioural semantics for GCM components

This section defines formally the behavioural semantics for the component model defined in Section 2.3. It shows how to build pNets from the specification of a hierarchy of components. We organise this section as follows. We first give a behavioural semantics for primitive components including simple future proxies, and the behaviour and synchronisation of the different elements of the primitive component. We then describe the behavioural semantics for composite components, which compose the semantics of their sub-components synchronising the request and replies of the sub-components between themselves and with the external components. For that, we will need to define a new kind of future proxies for handling the delegation mechanism that occurs in the composite components.

Term algebra The term algebra we use is a set of parameterised actions; actions will typically be of the form *Serve_m* for m a method label as defined below. Parameters will be either values (method parameters denoted *arg* or computed results denoted *val*), or future identifiers (denoted either p, or f, or *fid*). We suppose a set P is given, it is the set of all parameters potentially used in the different pNets; it will be used in all the pNets expressed in the rest of this paper.

In all this document, we use, as action parameters, two variables *arg* and *val* that (implicitly) range over the set of *values*, this set of values being purposely undefined. In an object-oriented language, those values should be an abstraction of objects (potentially containing other objects). It could be defined depending on the *type* of the value but we will not consider this aspect here. In our previous works, the abstract domain for values could be reduced to a few elements in order to generate a finite instantiation of the pNet and verify its behaviour by model-checking techniques.

We rely on a predefined set *Labels*, the set of action labels that can be used in our pNets, it should depend on the term algebra (it is possible to generate this set from the labels used in each pNet).

Labels for identifying methods Inside our actions, we need identifiers for methods that are more precise than simple method names. We define thus *MethodLabels* as a set of method labels, where a method label encompasses a method name, a signature, and the interface the method belongs to, plus possibly other meta-informations. Most of the following can be read as if *MethodLabels* were just method names, however at some specific points and to disambiguate different methods, the other informations encoded in *MethodLabels* are also necessary. m_i range over such method labels.

A function MethLabel : $Itf \rightarrow \mathcal{P}(MethodLabels)$ is defined, where MethLabel(Itf_i) returns the set of *MethodLabels* corresponding to the methods of interface Itf_i. MethLabel is also defined for sets of interfaces (union of method labels for each interface). Conversely, for a given method label m, $Itf(m)$ returns the interface of the method.

Behavioural semantics The behavioural semantics of components is expressed under the form $[\![Component]\!]$. It relies on the use of several auxiliary functions for expressing the semantics of specific parts of the components: the behaviour of the *service* of one request (for a primitive component), the behaviour of the *body* of the component serving requests one after the other, a *proxyManager* for managing the available future proxies, the behaviour of each future *proxy*, and finally a delegation behaviour used when a composite component *delegate*s the service of a request to another component. The signature of all these functions is summarised in Appendix B.

4.1 Semantics of primitive components

Primitive components are the leaves of the hierarchy; they contain the applicative code from which more complex components, and thus more complex behaviours can be built. This section gives a behavioural semantics for GCM primitive components, able to receive requests, to serve them in a FIFO order by executing a service method, and to send requests to the external world. Additionally to the global structure of a primitive component and the synchronisation of its sub-entities, this section defines pLTSs describing the behaviour of a FIFO service policy, of proxies for handling futures, and of managers for pools of future proxies. Considering the features we model, we think that our work provides a reliable basis for the behavioural specification of asynchronous components communicating by asynchronous requests and futures. This section also shows that pNets provide a convenient abstraction for modelling this behaviour.

4.1.1 Illustrative Example

We first illustrate and explain the structure of the behavioural semantics of primitive components based on the component shown in Figure 2. Figure 4 illustrates the structure of the pNet expressing the semantics of the component. It illustrates both the global structure of the pNets represented by boxes, and the synchronisation vectors represented by arrows (an ellipse is used when a synchronisation vector involves more than two processes). Note that the direction of an arrow is purely conventional, but goes, as much as possible, from an emission action to a reception action, intuitively following the data flow.

A primitive component can receive incoming requests ($?Q\_m_i$) that are stored in the *Queue* pNet and then served by the *Body* pLTS. The service consists in triggering a $Call\_m_i$ to the

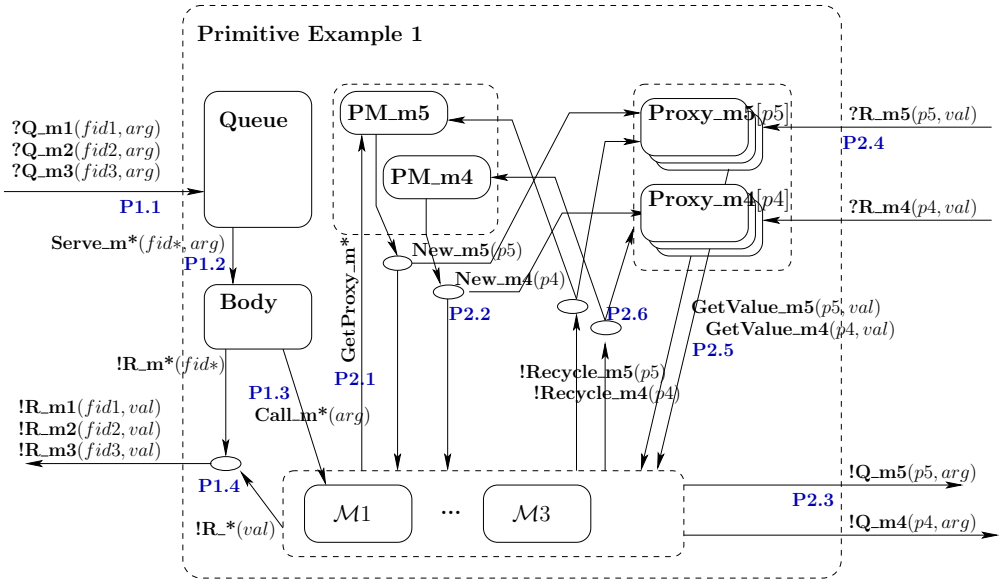


Figure 4: pNet for the Simple Primitive Component from Figure 2

adequate service method, called $\mathcal{M}i$ in the figure. Once a result is computed for the request, a $!R\_m_i$ action is emitted with the right future identifier *fid* and result value *val*.

The service method can call external components through client interfaces. Each method of each client interface is equipped with a proxy manager PM_m_i on which the caller can perform a *GetProxy*. Upon request, a fresh future proxy $Proxy\_m_i$ is allocated and returned by a New_m_i action that acts as a response to the *GetProxy*; there is a family of future proxies for each method of each client interface. Then the outgoing call is emitted with the reference to the corresponding proxy sent as parameter ($!Q\_m_i$). Finally when a result is computed the reply $?R\_m_i$ is received by the adequate proxy and the result can be accessed by $GetValue\_m_i$ actions performed by some service methods. Then, service methods can emit *Recycle* actions that are sent to the adequate proxy and proxy manager.

4.1.2 pNets and Synchronisation Vectors

This section formalises and generalises the principles depicted in the previous section. Primitive components encode the business code of the application. Consequently, the behaviour of primitive components include the behaviour of "service methods": those methods represent the code written by the programmer, it is the only part for which we do not specify the generation of the behavioural model. We suppose that for each service method m_l, for a set of methods $M_k^{k\in K}$ defined by the component, $[\![m_l, M_k^{k\in K}]\!]_{service}$ provides a pNet expressing the behaviour of this service method. Concerning the rest of the behaviour of a primitive component below, it is computed by the rules shown in this section.

$$\frac{\begin{array}{c} m_l^{l\in L} = \text{MethLabel}(SItf_i^{i\in I}) \\ \mathcal{Q} = Queue(m_l^{l\in L} \cup NF) \qquad \mathcal{B} = [\![m_l^{l\in L}, NF]\!]_{body} \qquad \forall l \in L. \mathcal{SM}_l = [\![m_l, M_k^{k\in K}]\!]_{service} \\ \forall j \in J.\ \text{let } m_n^{n\in N} = \text{MethLabel}(CItf_j) \text{ in} \\ \text{for all } n \in N \text{ let } \mathcal{F}_n = \langle\!\langle P, [\![m_n]\!]^{\mathbb{N}}_{proxy} \overleftarrow{\rangle\!\rangle} \text{ in} \\ \mathcal{P}_j = \langle\!\langle P, \mathcal{F}_n^{n\in N} \overleftarrow{\rangle\!\rangle} \text{ and } \mathcal{PM}_j = \langle\!\langle P, [\![m_n]\!]^{n\in N}_{proxyManager} \overleftarrow{\rangle\!\rangle} \\ SV = SV_S(m_l^{l\in L}) \cup SV_C(CItf_j^{j\in J}, L) \end{array}}{\begin{array}{c} [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!] = \\ \langle\!\langle P, Labels, \mathcal{Q}, \mathcal{B}, \overleftarrow{\langle\!\langle P, \mathcal{SM}_l^{l\in L} \rangle\!\rangle}, \overleftarrow{\langle\!\langle P, \mathcal{PM}_j^{j\in J} \rangle\!\rangle}, \overleftarrow{\langle\!\langle P, \mathcal{P}_j^{j\in J} \rangle\!\rangle}, SV \rangle\!\rangle \end{array}}$$

Note that P and *Labels* are fixed sets defined globally above. The pNet corresponding to a primitive component is made of:

- A queue able to receive incoming requests: it can enqueue a request on a method label of one of the server interfaces, we use here a pNet queue constructor. The queue is also given a set of method labels *NF*, it contains the set of non-functional requests that can be accepted by the component; we will use this set in Section 5.2 and 5.3. For the moment, $NF = \emptyset$.

- A body that will serve all the requests that can reach the queue, it will delegate the treatment of the request to the service methods $\overleftarrow{\langle\!\langle P, \mathcal{SM}_l^{l\in L} \rangle\!\rangle}$, the body constructor is also given the set *NF*.

- Service methods: there is one service method for each method label of a server interface, it encodes the business logic of the component.

- A family $\mathcal{PM}$ of proxy managers indexed both over the set of client interfaces and over the methods of those interfaces: those managers are responsible for allocating a new proxy when requested, and activating those newly created proxies.

- A family $\mathcal{P}$ of future proxies indexed over the set of client interfaces (J), the methods of those interfaces (N), and proxy indexes, i.e. integers ($\mathbb{N}$): a proxy is responsible for receiving the result of a request made towards another component; when the value of the result is needed by a service method, this method asks for the value to the adequate proxy.

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

The set of synchronisation vectors for a primitive component is built by two functions: SV_S that provides the set of synchronisation vectors corresponding to the server interfaces, and SV_C for the client interfaces. Each of those sets is defined as the smallest set verifying the constraints given in Table 1. Remember the synchronisation vector set also implicitly includes observation vectors defined in Section 3.3.

Let us explain briefly what are the synchronisation vectors generated by the inference rules, more precisely, we focus on the synchronisation vectors for the $GetProxy\_m_i$ actions, rule [P2.1]. One synchronisation vector for $GetProxy\_m_i$ is generated for each $l \in L$, for each $j \in J$, and for

Table 1: Server and client-side synchronisation vectors for primitive components. The synchronised sub-pNets occur in the following order: $\langle\!\langle\, Queue, Body, ServiceMethods, ProxyManagers, Proxies \,\rangle\!\rangle$

$$\frac{i \in L \qquad fid \in \mathbb{N}}{\begin{array}{ll}
\{\langle ?Q_m_i(fid, arg), -, -, -, - \rangle \rightarrow ?Q_m_i(fid, arg), & [1] \\
\ \langle !Serve_m_i(fid, arg), ?Serve_m_i(fid, arg), -, -, - \rangle \rightarrow Serve_m_i(fid, arg), & [2] \\
\ \langle -, !Call_m_i(arg), i{\mapsto}?Call_m_i(arg), -, - \rangle \rightarrow Call_m_i(arg), & [3] \\
\ \langle -, !R_m_i(fid), i{\mapsto}!R_m_i(val), -, - \rangle \rightarrow !R_m_i(fid, val) \} & [4] \\
\qquad\qquad \subseteq SV_S(m_i^{i \in L}) &
\end{array}} \text{ P1}$$

$$\frac{j \in J \qquad l \in L \qquad m_i^{i \in I} = \text{MethLabel}(CItf_j) \qquad i \in I \qquad p \in \mathbb{N}}{\begin{array}{ll}
\{\langle -, -, l{\mapsto}!GetProxy_m_i, j{\mapsto}i{\mapsto}?GetProxy_m_i, - \rangle \rightarrow GetProxy_m_i, & [1] \\
\ \langle -, -, l{\mapsto}?New_m_i(p), j{\mapsto}i{\mapsto}!New_m_i(p), j{\mapsto}i{\mapsto}p{\mapsto}?New_m_i \rangle \rightarrow New_m_i(p), & [2] \\
\ \langle -, -, l{\mapsto}!Q_m_i(p, arg), -, - \rangle \rightarrow !Q_m_i(p, arg), & [3] \\
\ \langle -, -, -, -, j{\mapsto}i{\mapsto}p{\mapsto}?R_m_i(val) \rangle \rightarrow ?R_m_i(p, val), & [4] \\
\ \langle -, -, l{\mapsto}?GetValue_m_i(p, val), -, j{\mapsto}i{\mapsto}p{\mapsto}!GetValue_m_i(val) \rangle \rightarrow GetValue_m_i(p, val), & [5] \\
\ \langle -, -, l{\mapsto}!Recycle_m_i(p), j{\mapsto}i{\mapsto}?Recycle_m_i(p), j{\mapsto}i{\mapsto}p{\mapsto}?Recycle_m_i \rangle \rightarrow Recycle_m_i(p)\ \} & [6] \\
\qquad\qquad \subseteq SV_C(CItf_j^{j \in J}, L) &
\end{array}} \text{ P2}$$

each $i \in I$. Each synchronisation vector synchronises one action $l{\mapsto}!GetProxy\_m_i$[5] of the sub-pNet containing the family of service methods, with one action $j{\mapsto}i{\mapsto}?GetProxy\_m_i$ of the sub-pNet containing the family of proxy managers (for each interface). As each of the synchronisation vectors of families of pNets triggers the action on the indexed element of the family, this line allows one action $!GetProxy\_m_i$ of one service method (indexed by l) to be synchronised with the action $?GetProxy\_m_i$ of the proxy manager indexed by i of the interface indexed by j. The action is globally visible as $GetProxy\_m_i$ and will be (implicitly) observable in all the hierarchy of pNets containing this one.

The set of service synchronisation vectors SV_S defined in rule [P1] encodes the following synchronisations: en-queueing an incoming request [P1.1], service of a request by the body [P1.2], the body calling a service method to serve a request [P1.3], and the service method providing a result for this served request [P1.4]. In the last case the result both notifies the body process and is returned to the outside of the primitive component. In all the actions, the method argument or the returned value is used as parameter, plus when necessary the identifier of the concerned future (*fid*).

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

- obtaining a new future proxy which involves a call to the proxy manager [P2.1] and another action [P2.2] for returning a fresh proxy identifier and activating the corresponding future proxy,
- the sending of a request from a service method to an external components [P2.3],
- the reception of a result by the future proxy [P2.4],
- the access to a future value [P2.5] from a service method, the future value is stored in the future proxy,

[5] $j{\mapsto}i{\mapsto}a$ should be read $j{\mapsto}(i{\mapsto}a)$

- the eventual recycling of a future proxy [P2.6].

The function SV_C receives as argument the set L of indexes over which service methods range. This argument is necessary because all service methods can perform some of the actions, like $GetValue\_m_i$.

Remember the interface called is encoded as part of the method label m_i, consequently the index of the concerned interface can be inferred from the *MethLabel* of the invoked method. Note the indexing of proxy managers (by interfaces and methods) and of proxies (by interfaces, methods, and proxy identifier).

4.1.3 Body

The body is a pLTS modelling the service of the different requests: for each service method, the body can dequeue a request corresponding to this method, delegate the service to the appropriate service method, wait for the computation of a result, and finally return this result before dequeueing a new request. It can be generated automatically from the set of service methods $m_i^{i\in I}$ and the set of non-functional method $m'_k{}^{k\in K}$. For simplicity, we suppose here methods m'_k have a single argument. $[\![m_i^{i\in I}, m'_k{}^{k\in K}]\!]_{body} = \langle\!\langle P, S, s_0, L, \rightarrow \rangle\!\rangle$ where:

- $S = \{s_0\} \cup \bigcup_{i\in I}\{s_i(fid, arg)\} \cup \bigcup_{i\in I}\{s'_i(fid)\} \cup \bigcup_{k\in K}\{s''_k(i)\}$
- $L = \bigcup_{i\in I}\{?Serve\_m_i(fid, arg), !Call\_m_i(arg), !R\_m_i(fid)\} \cup \bigcup_{k\in K}\{?Serve\_m'_k(i), !m'_k(i)\}$
- $\rightarrow = \bigcup_{i\in I}\{s_0 \xrightarrow{?Serve\_m_i(fid,arg)} s_i(fid, arg)\} \cup \bigcup_{i\in I}\{s_i(fid, arg) \xrightarrow{!Call\_m_i(arg)} s'_i(fid)\}$
  $\cup \bigcup_{i\in I}\{s'_i(fid) \xrightarrow{!R\_m_i(fid)} s_0\}$
  $\cup \bigcup_{k\in K}\{s_0 \xrightarrow{?Serve\_m'_k(i)} s''_k(i)\} \cup \bigcup_{k\in K}\{s''_k(i) \xrightarrow{!m'_k(i)} s_0\}$

For each method m, the body pLTS can always perform the three actions $?Serve\_m$, then $!Call\_m$ and then $!R\_m$. It also includes the services of the non-functional requests, which after the $?Serve\_m'$ action, only triggers a m' action. This body encodes a *mono-threaded* component behaviour where no two requests are served at the same time. This corresponds indeed to the behaviour of the ProActive/GCM framework, and more generally to the behaviour of active-objects or actors. Allowing the body to serve multiple requests at the same time would be quite easy but the resulting behaviour would be much more complex.

Figure 5 provides a graphical definition for the pLTS of the body defined above (in the rest of this paper we will express pLTSs graphically). The graph shows a body able to serve three functional requests (m_0, m_1, and m_2) and two non-functional ones ($bind\_Itf(i)$ and $unbind\_Itf(i)$). Those are the two kinds of non-functional requests that will be used for dealing with reconfiguration in Section 5.3[6].

4.1.4 Service Methods

The behaviour for each service method is expressed by a pNet, used when serving the corresponding request. This behaviour is either obtained by source code analysis, or provided by the

[6] Non-functional requests used in Section 5.2 are similar except that $unbind\_Itf$ has no parameter.

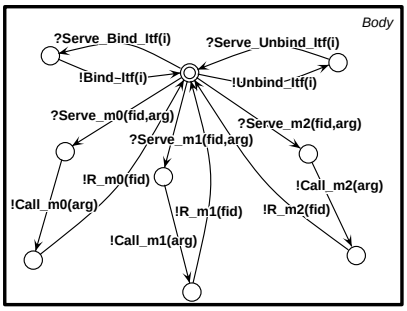


Figure 5: Graphical representation of the behaviour of the Body

user. It can for example be composed of the execution of several LTSs expressing the behaviour of each local method

4.1.5 Modelling of the Future Proxies

Communicating by asynchronous requests allows each component to execute asynchronously from the others. However it is commonly necessary to obtain a result for some of those asynchronous invocations. A convenient abstraction for dealing with response to asynchronous requests is the notion of futures. Technically, a future is often implemented by a proxy that represents the result and is accessible both locally to know whether the result came back, and remotely by the invoked component that wants to return the result. We represent those notions in our behavioural models. Such future proxies have to be instantiated upon need; thus, to allocate fresh futures, we use proxy managers that will be invoked before performing an asynchronous request. Finally, we leave the opportunity for the service methods to inform the manager that a future proxy is no longer useful and can be recycled; this behaviour is also encoded in our future proxies and managers.

Remember future proxies are families indexed by client interface index, method index, and future identifier; proxy managers are indexed by client interface index and method index. We propose a specification of manager and future proxy in Figure 6.

The behavioural semantics of the proxy manager is defined by the pLTS $ProxyManager\_m$ shown on the right side of Figure 6; it is denoted by $[\![m]\!]_{proxyManager}$. It maintains a list of available proxies and returns a fresh future (by a *New* action), or if there is no more fresh future, raises an error *NoMoreProxy*. Indeed, in our specification, we let future identifiers be indexed by $\mathbb{N}$ but if one wants to perform finite model-checking, a bound should be chosen on the size of each future proxy family, and in each proxy manager, Max_Proxy should be set to the chosen bound (which could be different for each manager).

Each proxy has a much simpler behaviour; $[\![m]\!]_{proxy}$ is defined by the pLTS $Proxy\_m$ shown on the left side of Figure 6. Once activated by a New_m action, it waits for the corresponding reply ($?R\_m(val)$). At this point, the proxy can be accessed to know the result of the request invocation, it continuously sends the result to the service methods by a $!GetValue\_m(val)$ output action until the proxy is recycled.

The proxies in Figure 6 are endowed with a $Recycle\_m$ transition, bringing back the proxy in its initial state. This is useful when information can be computed, e.g., by static analysis techniques, that the proxy will not be used anymore, so it can be made available again in the proxy pool of the ProxyManager. The $Recycle\_m$ event should be sent by the LTS modelling a

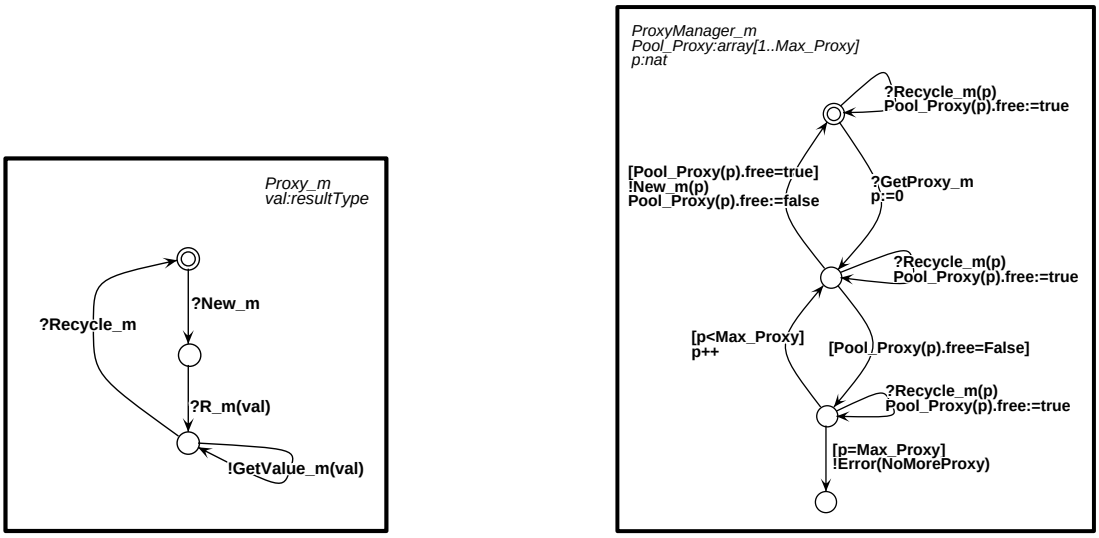


Figure 6: pLTSs for the Future Proxies and Proxy Managers

service method. When such an event is received by the ProxyManager, this sets the corresponding entry in the *Pool_ Proxy* to *free*. For generality, the proxy manager accepts *Recycle_ m* transitions in every state, even if for mono-threaded GCM components, this action can only be received when the proxy manager is in its initial state.

4.1.6 Dealing with stateful components

The semantics we presented here does not allow to store a state for the components, however it is easy to add component variables in our specification. Indeed it is sufficient to add an additional sub-pNet to the pNet of the primitive component that stores the value of the variables and accepts *set* and *get* actions. Those actions will be triggered by the service methods so synchronisation vectors for those setters and getters are also necessary, they relate service methods and the new sub-pNet dedicated to state management.

Such a state management pNet can also be used to express the behaviour of an *attribute controller*. In Fractal, an attribute is a configurable property of a component, it can be accessed and modified by setter and getter functions, exposed outside the component as a non-functional interface. The attribute controller interface is a non-functional interface exposed by the component. Attributes that can be stored and modified can be expressed in pNets by adding the getter and setter functions to the requests that can be enqueued, and by adding synchronisation vectors from the body to the sub-pNet dedicated to state management for getter and setter actions.

In the case of composite components (defined in the next section), a state management pNet can be necessary to configure the non-functional aspects (a composite component has no functional internal state). It can be expressed similarly.

4.2 Semantics of composite components

Hierarchical component models, like GCM, allow the specification of new components, based on the composition of others. Such a composition abstraction is very convenient when building large applications. We start from the hypothesis that composite components are defined statically by

some form of ADL (architecture description language) as defined in Section 2.3. The ADL is used to synchronise the emission and reception of communication actions between the sub-components. Additionally, a composite has a request queue for receiving requests coming from the outside or the inside of the component, it treats each of those requests by sending it to the adequate component or emitting a request to the outside world. For this the composite has future proxies but as the requests only transit through the component, we implement special future proxies that perform future redirection: when a future proxy is created, it receives the identifier for another future f' and when the reply will come back, it will be immediately re-sent as a reply for the future f'.

4.2.1 Illustrative Example

Figure 7 shows the pNets structure corresponding to the composite component of Figure 1. It illustrates the structure of the pNets we generate for specifying the behaviour of a composite component. We use it here to illustrate how we are able to generate behavioural models for GCM components.

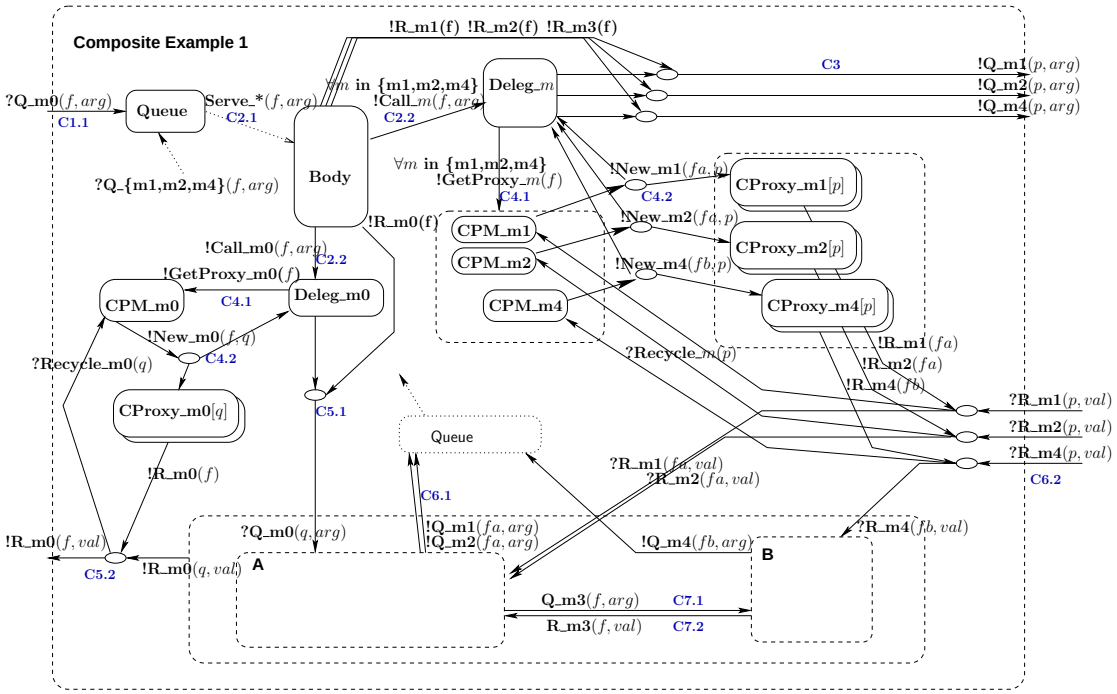


Figure 7: pNet for the Composite Component from Figure 1

Two sub-pNets A and B represent the behaviour of sub-components A and B. A queue pNet receives *?Q_ m0(f,arg)* requests where f is the future corresponding to the request and *arg* the value passed as argument. *Serve** communications allow the body to retrieve those requests, which will then be treated by the *Deleg_ m0* pNet, this pNet receives *Call* communications from the body and delegates the request to an inner component (here, A); during this process, a future proxy is created by the proxy manager (process *CPM_ m0*), the proxy (process *CProxy_ m0[q]*) is responsible for receiving the reply when A has finished the request treatment and for forwarding this result to the outside of the composite component: *R_ m0(q,val)* that becomes *R_ m0(f,val)*.

Note that this proxy encodes some basic form of first-class future: the future q corresponds to the same result as the future f.

Similarly, requests emitted by the inner components arrive in the queue (we draw two *Queue* boxes, but they correspond to the same element), they are then delegated to the outside world by a similar mechanism: a *Deleg_m* pNet delegates the call, and creates a future proxy, which will be responsible for sending back the result to the appropriate inner component. Here again the proxy manages the fact that both the future q and the future *fa* (or *fb*) represent the same result.

Finally, note the proxy structure we adopt: there is one proxy manager *CPM** for each method of each interface (proxy managers are both indexed over interfaces and over methods). Then each of those manager itself manages a family of proxies *CProxy**. Performing model-checking on those structures then requires a precise definition and optimisation of the number and size of those families.

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

4.2.2 Global structure

The semantics of a composite component is described below; the first difference compared to the semantics of primitive components is that it does not rely on service method specification, instead it delegates requests to sub-components, some of the sub-pNets of a composite component's pNet correspond to the behaviour of the sub-components. Like primitive components, the behaviour of composite components include a proxy manager and proxy families, but in the case of composites, we also need one future proxy family for each method of each *server interface*. Indeed the service of requests received on a server interface will be delegated to a sub-component, and thus a future is necessary to represent the result of such a delegated request. Note the use of the Symm function to take the symmetrical role of an interface and to transform those server interfaces into client ones. For similar reasons, the request queue can receive requests on all methods of all the interfaces of the composite component, both server and client (i.e., internal server) ones.

To delegate a request to an inner component or from an inner component to an external one, "delegation methods" are used, they are denoted $\mathcal{DM}$. Delegation methods transform a request into another and a special proxy for future is used in order to remember the relationship between the original future and the future of the new delegated request.

$$
\begin{array}{c}
m_l^{l\in L} = \biguplus_{j\in J} \mathrm{MethLabel}(SItf_j) \uplus \biguplus_{i\in I} \mathrm{MethLabel}(CItf_i) \qquad \mathcal{Q} = Queue(m_l^{l\in L} \cup NF) \\
\mathcal{B} = [\![m_l^{l\in L}, NF]\!]_{body} \qquad \forall l \in L.\, \mathcal{DM}_l = [\![m_l]\!]_{delegate} \qquad Itf_h^{h\in H} = (CItf_i^{i\in I}) \uplus (\mathrm{Symm}(SItf_j)^{j\in J}) \\
\forall h \in H.\ \ \text{let } m_n^{n\in N} = \mathrm{MethLabel}(Itf_h) \text{ in} \\
\text{for all } n \in N \text{ let } \mathcal{F}_n = \overleftrightarrow{\langle\!\langle P, [\![m_n]\!]^{\mathbb{N}}_{proxy} \rangle\!\rangle} \text{ in} \\
\mathcal{P}_h = \overleftrightarrow{\langle\!\langle P, \mathcal{F}_n^{n\in N} \rangle\!\rangle} \text{ and } \mathcal{PM}_h = \overleftrightarrow{\langle\!\langle P, [\![m_n]\!]^{n\in N}_{proxyManager} \rangle\!\rangle} \\
SV = \ SV_S(\mathrm{MethLabel}(SItf_i^{i\in I}), \mathrm{MethLabel}(CItf_i^{i\in I})) \ \cup \ SV_C(CItf_j^{j\in J}, Itf_h^{h\in H}) \\
\cup\ SV_B(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}) \\
\hline
[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_b^{b\in B} >]\!] = \\
\langle\!\langle P, Labels, \mathcal{Q}, \mathcal{B}, \overleftrightarrow{\langle\!\langle P, \mathcal{DM}_l^{l\in L} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle P, \mathcal{PM}_h^{h\in H} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle P, \mathcal{P}_h^{h\in H} \rangle\!\rangle}, \overleftrightarrow{\langle\!\langle P, [\![Comp_k]\!]^{k\in K} \rangle\!\rangle}, SV \rangle\!\rangle
\end{array}
$$

The body and the queue are similar to the ones of primitive components.

4.2.3 Future Proxies

The behaviour of future proxies for composite component is slightly different from the one of primitive ones, as illustrated in Figure 8: the process *CProxy_ m* in the figure gives the new value of the proxy semantics $[\![m]\!]_{proxy}$. The delegation methods create those proxy to remember the identifier of the future that the delegation method should serve. Consequently, the future proxy receives a future identifier and will return it upon need. The future proxy thus first receives a *New* action with a future identifier as parameter and then emits an *!R_ m(f)*. Such a proxy is somehow automatically recycled as, by construction, we know it is only used once.

4.2.4 Delegation Methods

The *Deleg_ m* process, also shown in Figure 8 expresses the generation of delegate methods: $[\![m]\!]_{delegate}$ is given by the pLTS *Deleg_ m*. This delegation process receives a *Call* invocation from the body, creates a future proxy, launches a remote invocation (either to an inner or to an external component) and finishes its execution. This way the composite component can continue its execution and serve another request, but the process of the future proxy is still running in order to redirect the reply towards the right future identifier. The *proxyManager* for composite component is not shown, indeed it is a direct adaptation of the primitive one (Figure 6): it behaves exactly the same except that the *GetProxy* action receives a future identifier as parameter, this parameter is then passed as argument in the *New* emission action (it will be used by the future proxy).

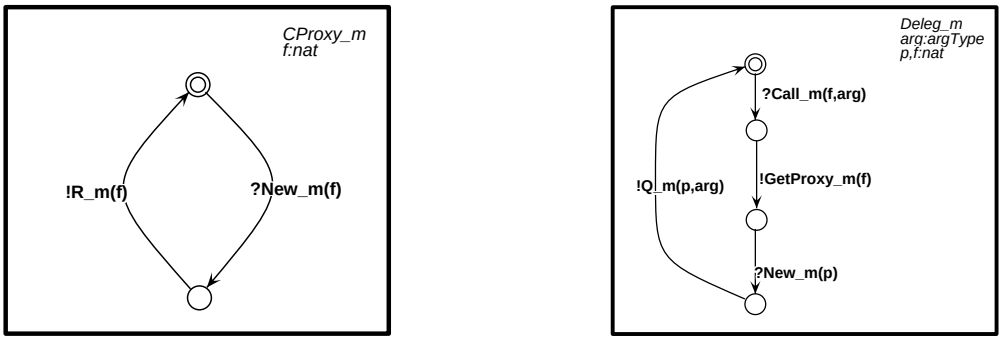


Figure 8: Auxiliary processes proxy and delegate of composite components

4.2.5 Synchronisation Vectors

Synchronisation vectors are now organised into three sets: server-side (SV_S), client-side (SV_C), and binding-related (SV_B) synchronisation vectors. The set of server and client synchronisation vectors is the smallest set verifying the rules given in Table 2 (and including the implicit observation synchronisation vectors).

The server-side synchronisation vectors are defined by rules [C1] and [C2]. [C1] allows external components to enqueue a request in the queue, for each method of a server interface of the composite (given as first argument of SV_S). Note that replies (*!R_ m*) are not part of this rule because they depend on the bindings of the component; consequently they are treated among the binding synchronisation vectors. Rule [C2] uses a bigger set of methods as it takes into account requests of the server interfaces and the client interfaces of the composite; indeed, remember client interfaces have an associated internal server interface accessible by the sub-components

Table 2: Server and client-side synchronisation vectors.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle Queue, Body, DelegationMethods, ProxyManagers, Proxies, Subcomponents \rangle\!\rangle$

$$\frac{i \in L \qquad f \in \mathbb{N}}{\langle ?Q\_m_i(f, arg), -, -, -, - \rangle \rightarrow ?Q\_m_i(f, arg) \in SV_S(m_l^{l \in L}, {m'}_l^{l \in L'})} \text{ C1}$$

$$\frac{(i \in L \wedge m = m_i) \vee (i \in L' \wedge m = m'_i) \qquad f \in \mathbb{N}}{\begin{array}{ll}\{\langle !Serve\_m(f, arg), ?Serve\_m(f, arg), -, -, - \rangle \rightarrow Serve\_m(f, arg), & [1] \\ \langle -, !Call\_m(f, arg), i \mapsto ?Call\_m(f, arg), -, - \rangle \rightarrow Call\_m(f, arg)\ \} & [2] \\ \subseteq SV_S(m_l^{l \in L}, {m'}_l^{l \in L'}) & \end{array}} \text{ C2}$$

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

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

of the composite. This second rule uses the second argument of SV_S, i.e., the list of client and server interfaces of the component. It deals with request service [C2.1], and subsequent calls [C2.2] to delegation pNets.

Client-side synchronisation vectors are expressed by rules [C3] and [C4] of Table 2. Similarly to the server case, [C3] is specific to external client interfaces (given as first argument of SV_C), whether [C4] is applicable to both external and internal client interfaces (the second argument of SV_C). Remember the internal client interfaces are the symmetric of server interfaces of the composite component. The first rule exports request sending (Q_m) sent by delegate methods to the external components. Note that delegate methods are indexed by the method labels of the interfaces: in the *pNet* definition $m_l^{l \in L} = \text{MethLabel}(SItf_j^{j \in J}) \uplus \text{MethLabel}(CItf_i^{i \in I})$, and thus each $l \in L$ is considered as equal to a method index k of a single interface i. Consequently, the request sending action ($!Q\_m_k$) is always issued by the delegate method indexed by k. Rule [C4] allows delegation methods to instantiate new proxies (by calls to the proxy manager and to the future proxies). Compared to the case of the primitive component, note the additional argument f passed to the proxy manager. This future identifier allows the future proxy (indexed p) to remember that the reply it will receive should be forwarded to the caller as the value for the future identifier f (and not p). In other words, the proxy remembers that the future p it will receive is in fact an alias for the future f. Similarly to primitive components, there are two actions for dealing with future proxy creations: *GetProxy* in [C4.1], and *New* in [C4.2].

Finally, the synchronisation vectors for the bindings of the composite component are shown in Table 3. There are three rules for building SV_B. Rule [C5] deals with import bindings, i.e. bindings from the composite component's internal client interfaces to inner components. Symmetrically, [C6] concerns export bindings, from inner components to the composite component's internal server interfaces. The last rule [C7] specifies synchronisations due to bindings between two inner components.

The first rule [C5] deals with import bindings. The first premise of the rule picks an import binding, the next premises find the concerned server interface of the composite component,

Table 3: Binding synchronisation vectors.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle$ *Queue*, *Body*, *DelegationMethods*, *ProxyManagers*, *Proxies*, *Subcomponents* $\rangle\!\rangle$

$$\frac{\begin{array}{c} k\in K \quad (This.SI, C.SI_2)\in Binding_b^{b\in B} \quad i\in I \quad SI=\mathrm{Name}(SItf_i) \\ C=\mathrm{Name}(Comp_k) \quad m_n^{n\in N}=\mathrm{MethLabel}(SItf_i) \quad n\in N \quad m'_n=m_n\{\!\{SI\leftarrow SI_2\}\!\} \quad q,\ f\in\mathbb{N} \end{array}}{\begin{array}{l} \{\langle -,!R\_m_n(f), n\mapsto !Q\_m_n(q,arg), -,-,k\mapsto ?Q\_m'_n(q,arg)\rangle \rightarrow Q\_m_n(q,arg), \quad [1] \\ \ \langle -,-,-,i\mapsto n\mapsto !Recycle\_m_n(q), i\mapsto n\mapsto q\mapsto !R\_m_n(f), k\mapsto !R\_m_n(q,val)\rangle \rightarrow !R\_m'_n(f,val)\} \quad [2] \\ \qquad\qquad \subseteq SV_B(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}) \end{array}} \text{ C5}$$

$$\frac{\begin{array}{c} k\in K \quad (C.CI, This.CI_2)\in Binding_b^{b\in B} \quad j\in J \quad CI_2=\mathrm{Name}(CItf_j) \\ C=\mathrm{Name}(Comp_k) \quad m_n^{n\in N}=\mathrm{MethLabel}(CItf_j) \quad n\in N \quad m'_n=m_n\{\!\{CI_2\leftarrow CI\}\!\} \quad p,\ f\in\mathbb{N} \end{array}}{\begin{array}{l} \{\langle ?Q\_m_n(f,arg), -,-,-,-,k\mapsto !Q\_m'_n(f,arg)\rangle \rightarrow Q\_m_n(f,arg), \quad [1] \\ \ \langle -,-,-,j\mapsto n\mapsto !Recycle\_m_n(p), j\mapsto n\mapsto p\mapsto !R\_m_n(f), k\mapsto ?R\_m'_n(f,val)\rangle \rightarrow ?R\_m_n(p,val)\ \} \quad [2] \\ \qquad\qquad \subseteq SV_B(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}) \end{array}} \text{ C6}$$

$$\frac{\begin{array}{c} k,k'\in K \quad (C.CI, C'.SI)\in Binding_b^{b\in B} \\ C=\mathrm{Name}(Comp_k) \quad C'=\mathrm{Name}(Comp_{k'}) \quad CItf'^{i\in I}_i=CItfs(Comp_k) \quad i\in I \\ CI=\mathrm{Name}(CItf'_i) \quad m_n^{n\in N}=\mathrm{MethLabel}(CItf'_i) \quad n\in N \quad m'_n=m_n\{\!\{CI\leftarrow SI\}\!\} \quad f\in\mathbb{N} \end{array}}{\begin{array}{l} \{\langle -,-,-,-,-,(k\mapsto !Q\_m_n(f,arg), k'\mapsto ?Q\_m'_n(f,arg))\rangle \rightarrow Q\_m_n(f,arg), \quad [1] \\ \ \langle -,-,-,-,-,(k'\mapsto !R\_m'_n(f,val), k\mapsto ?R\_m_n(f,val))\rangle \rightarrow R\_m_n(f,val)\ \} \quad [2] \\ \qquad\qquad \subseteq SV_B(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}) \end{array}} \text{ C7}$$

and the destination of the binding, i.e., a sub-component and its client interface. The only remaining non-trivial premise is $m'_n = m_n\{\!\{SI \leftarrow SI_2\}\!\}$; it replaces in m_n the occurrence of the interface named SI by the interface SI_2. Indeed, remember MethodLabels contain the name of the invoked interface, this name must thus be updated when a request/reply/... is transmitted from an interface to another[7]. Similar premises, renaming an interface name, will also be used in rules [C6] and [C7]. The first item of Rule [C5.1] synchronises the emission of a request by a delegate method with the inner component bound to the concerned internal client interface. This action is also synchronised with the proxy for future that will receive the result computed by the request. The case [C5.2] concerns the corresponding reply that is issued by the inner component, this reply is sent to the outside of the composite component. Note that the inner component emits a value for future q, that is directly synchronised with the future proxy number q of the composite component; the identifier of the future to be sent to the outside becomes f; it is retrieved from the future proxy, and an action $!R\_m'_n(f, val)$ is emitted. At the same time, a recycling action is triggered in the proxy manager.

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

Rule [C7] deals with bindings between two inner components. It considers a binding between an interface of component C and an interface of component C'. It finds k, the index of C, and k',

[7] at this point, other meta-informations encoded in the method label should also be updated.

the one of C'; the rule directly transfers requests [C7.1] and replies [C7.2] from one component to the other for all the methods of the client interface bound. Like in the preceding rules, the name of the interface is updated during transmission.

This section presented a behavioural semantics for hierarchical components communicating by asynchronous requests which are transmitted between components, where composite components just forward requests to the adequate destination. We encode replies by means of futures, and composite components act as reply forwarders, independently from the rest of the component; this avoids some of the limitations induced by the mono-threaded nature of our components. The next section will define more advanced features allowing more asynchronous or dynamic behaviours.

5 Advanced features

In this section we focus on three crucial advanced features of GCM components, which are general enough to be applicable to most other component models. First, we will introduce behavioural models for *first-class futures* in order to allow for more asynchrony between components. Second, we will define a model for *binding controllers* allowing for a more dynamic component model, where the bindings between components can be changed at runtime. Of course, this step supposes that (an abstract representation of the) potential bindings are known when the model is generated, or else we could not generate a full behavioural model. Usually, dynamic component reconfiguration also includes starting/stopping a component and adding a component inside another one. Concerning *start/stop* capacity, it is relatively easy to define a life-cycle pNet that would control the rest of the pNet. For the second case, as the specification of the component must be known at model-generation time, a component that is not yet added is similar to a component that is already here but totally unbound and stopped. This is why we only focus here on the specification of dynamic component bindings. Section 5.3 specifies the behaviour of *multicast interfaces equipped with a binding controller*. Those interfaces allow the easy definition of components performing some kind of group communications: multicast are one-to-many interfaces. One-to-many interfaces are frequently used in distributed systems where one computation/information is to be sent, sometimes cut into pieces, to several destinations. Finally, Section 5.4 will informally explain how to encode bindings that involve the client and the server interface of the same component, and then overcome the main limitation introduced in the definition of well-formed components, and.

In this section, we will specify new rules for the behavioural definition of those advanced features. In practice, we build new behavioural semantics by modifying the semantics defined in the preceding section, for this we use two additional operators on pNets:

- $\oslash$: Let *pNet* be a pNet and $\mathcal{A}$ be an indexed set of labels (strings); $pNet \oslash \mathcal{A}$ returns a pNet similar to *pNet* but with restricted synchronisation vectors. The synchronisation vectors of $pNet \oslash \mathcal{A}$ are the ones of of *pNet* except all the synchronisation vectors containing an element of $\mathcal{A}$ as part of their global synchronisation label. For example, if m belongs to $\mathcal{A}$ then all the vectors containing m in their global label will be removed, in that case the global labels concerned could be: Q_m, $!Q\_m$, $?Q\_m$, R_m, $Serve\_m$, ... Remember that method labels contain the name of the interface that contains the method and consequently, removing a method label cannot remove an action concerning another method with the same name in another interface.
- $\oplus$: For $I \in \mathcal{I}_P$ and $I' \in \mathcal{I}_P$ disjoint, let $pNet = \langle\!\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K} \rangle\!\rangle$ be a pNet and $pNet_i'^{i\in I'}$ be a pNet family (possibly empty). Let $SV_k'^{k\in K'}$ be synchronisation vectors

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

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

5.1 Semantics of First-class Futures

We call *first-class futures*, the futures that can be transmitted between components before their value is known: without first-class futures, a component must wait for the result of a request and perform a *GetValue* before being able to send this result to another component (inside a request parameter, or inside a request result). With first-class futures, a component can send a *generalised reference* to the future, i.e. a reference that uniquely represents the future in the whole component system. This way, once the result is computed it is sent to all the components that hold a (generalised) reference to the corresponding future.

In the model presented in Section 4, composite components could act as reply forwarders which corresponds to some kind of limited first-class futures. However, a service method of a primitive component that would return a future or send a future as request parameter has to wait until the adequate reply is received. This behaviour occupies the only thread of the primitive component and can create deadlocks.

We will consider in this section only the case when a future is transmitted as the single argument of a request. The case where a single future is returned as request result is similar, and closer to the reply forwarding mechanism implemented by the composite components. The case where a future is only a part of the transmitted object requires to reason on the abstract representation of the transmitted objects, which we do not do in this paper. Overall, we consider here the minimal case which is sufficient and general enough to understand the mechanism of the generation of behavioural models for first-class futures. A general study on the different kind of first-class futures, their identification and a few usage scenarios more complex than the one presented here can be found in [20]; however, compared to [20], the approach presented here provides a complete formal specification of the generation of behavioural models.

5.1.1 Principles and Illustrative example

Figure 9 is a simple scenario demonstrating the transmission of a future value as a parameter of a remote request. The principle of this scenario is the following. The component A first invokes a request on the component B (step 1); then the result of this invocation is sent as the single parameter of a request invocation to the component C (step 2). Possibly, the component C can try to access the future received, resulting in a wait-by-necessity (step 3). When the result is computed by B, it is returned to A and then to C (step 4), which releases the wait-by-necessity.

The pNet for the scenario described above is shown in Figure 10. The figure focuses on aspects related to the transmission of the future (denoted f in the code snippet, and indexed p as a local proxy) and its update. The generalised reference to the future is denoted *gf*. Generalised references, *gf*, now belong to the set of valid request arguments (denoted by the variable *arg*). The local future indexed p has a slightly different proxy able to emit a !*Forward*(*gf*, *val*) action. The generalised reference for the future is computed by composition of the *proxy index* p, and the *component holding the proxy and method label*, consequently the generalised reference *gf* is known and can be given at proxy creation time. Upon the invocation of *Q_m2*, *gf* is sent.

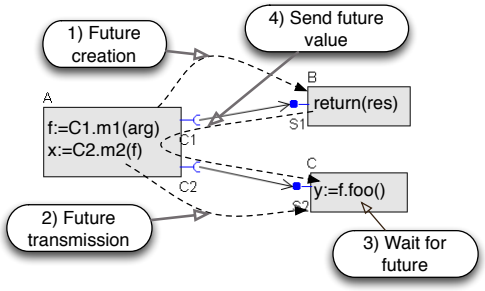


Figure 9: Scenario of first-class future transmission

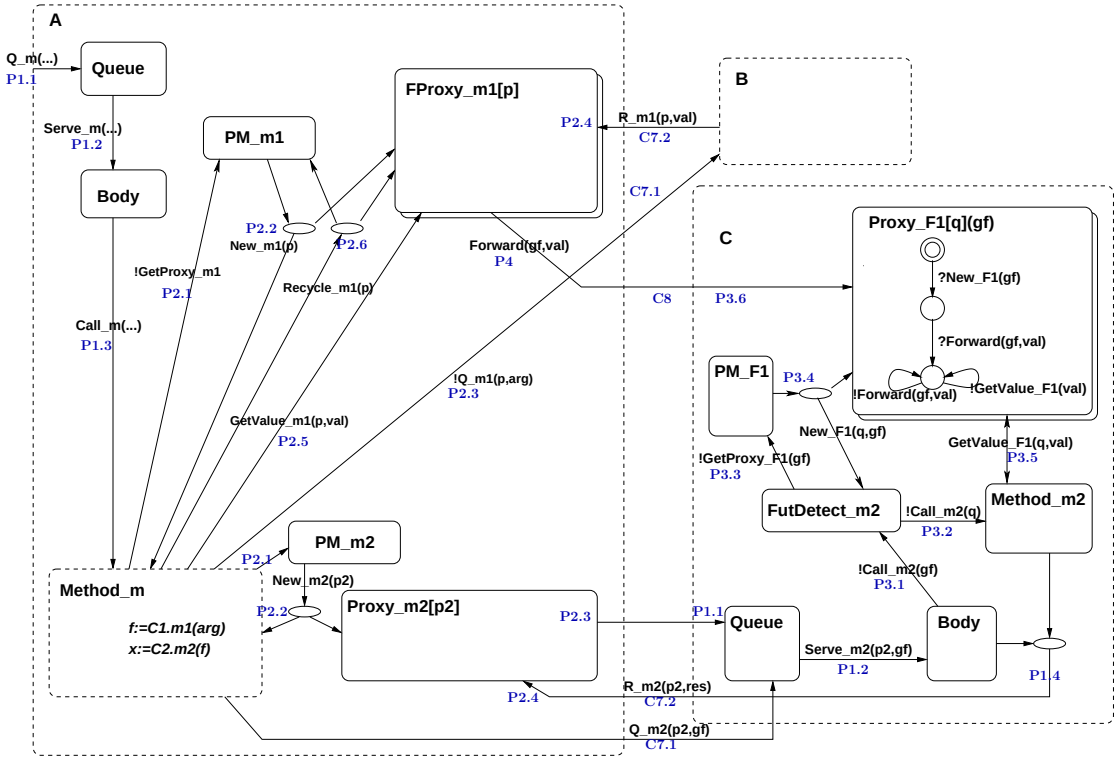


Figure 10: Transmitting a future as a method call parameter

When the request is handled by the component C, the call to the local method is intercepted by a specific pNet *FutDetect*. This pNet creates a local proxy for representing *gf* (*Proxy_F1*, indexed locally by *q*); *q* then replaces *gf* in the invocation. This local proxy for future waits for the *Forward* communication coming from the remote proxy, and can then be accessed by the service method *Method_m2*. We have a single *PM_F1* proxy manager for all the first-class proxies (*Proxy_F1*), it acts as any other proxy manager except that the first-class proxy is given the generalised reference *gf* when defining the semantics, and it transmits *gf* when creating the proxy. It thus has a similar behaviour to the proxy manager for a composite component; it does not correspond to any method name, we re-use the proxy manager semantics, giving it a pseudo

method name: $F1$. Consequently $PM\_F1 = [\![F1]\!]_{proxyManager}$.

It is crucial here to distinguish the two adjustments that have to be made to two different future proxies. First, all the future proxies must be able to send a *!Forward* action for forwarding the value they received, for them we will overload the existing proxy definition $[\![\ ]\!]_{proxy}$. Second, a new kind of proxy, *Proxy_F1*, needs to be created; those proxies act as a local future proxy for a proxy received as request parameter.

5.1.2 Primitive Components: pNets and Synchronisation Vectors

To handle first-class futures, it is not possible anymore to consider future identifiers as integers that are locally unique. Future identifiers need to be global references where unicity is guaranteed globally in the system. For this, we defined generalised references, where a generalised reference identifies uniquely a future proxy, it is thus defined by the tuple: component name, interface, method, future identifier. Let *GRef* be the set of all generalised references and let *gf* range over *GRef*. We can define a constructor of generalised references *GeneralisedRef*:

$$GeneralisedRef : CName \times Itf \times MethodLabels \times \mathbb{N} \rightarrow GRef$$

In this section we define $[\![\ ]\!]^{F1}$ the semantics of components with first-class futures. As shown in the illustrative example, the pNet of the primitive component must be extended with proxies for storing locally first-class futures received as parameters, called *Proxy_F1* in Figure 10 and in the rules below, and a proxy manager managing those proxies. The proxy manager for first-class futures is very similar to the ones shown before; like for composite components it receives a future reference and transmits it upon proxy creation, except this time the reference is a generalised one.

Figure 11: Proxy for a first-class future and Future Detector machine

Figure 11 shows the pLTS of a first-class future proxy (a future that can be transmitted as request parameter), and of a *FutDetect* process that intercepts the local service of requests with a future as parameter. On the left part of the figure, the proxy for a future that can be transmitted as request parameter is a classical one except it can also send a *Forward* action (with the generalised reference as parameter) to transmit the future value outside the component. *FProxy_m* in the figure defines the new proxy creation semantics $[\![m, gf]\!]_{proxy}$; it is used each time a proxy for a future that can be transmitted as request parameter is instantiated. The future detector pLTS is shown on the right part of Figure 11. It creates a local proxy before delegating the call. *FutDetect_m* defines the pLTS denoted by $[\![m]\!]_{FutDetect}$ in the following.

We introduce a set L' that indexes the set of methods that can receive a future; L still indexes all the service methods. The service of each service method that can receive a future as

Table 4: Synchronisation vectors related to client and server interfaces of primitive components for first class futures.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies, FutDetect, PM\_F1, Proxies\_F1 \rangle\!\rangle$

$$\frac{i \in L' \qquad gf \in GRef \qquad q \in \mathbb{N} \qquad j \in L}{\begin{array}{ll}\{\langle -, !Call\_m_i(gf), -, -, -, i{\mapsto}?Call\_m_i(gf), -, -\rangle \rightarrow Call\_m_i(gf), & [1]\\ \langle -, -, i{\mapsto}?Call\_m_i(q), -, -, i{\mapsto}!Call\_m_i(q), -, -\rangle \rightarrow Call\_m_i(q), & [2]\\ \langle -, -, -, -, -, i{\mapsto}!GetProxy\_F1(gf), ?GetProxy\_F1(gf), -\rangle \rightarrow GetProxy\_F1(gf), & [3]\\ \langle -, -, -, -, -, i{\mapsto}?New\_F1(q), !New\_F1(q, gf), q{\mapsto}?New\_F1(gf)\rangle \rightarrow New\_F1(q, gf), & [4]\\ \langle -, -, j{\mapsto}?GetValue\_F1(q, val), -, -, -, -, q{\mapsto}!GetValue\_F1(val)\rangle \rightarrow GetValue\_F1(q, val), & [5]\\ \langle -, -, -, -, -, -, -, q{\mapsto}?Forward(gf, val)\rangle \rightarrow ?Forward(gf, val), & [6]\\ \langle -, -, -, -, -, -, -, q{\mapsto}!Forward(gf, val)\rangle \rightarrow !Forward(gf, val)\} & [7]\\ \qquad\qquad \subseteq SV_S^{F1}(m_i^{i\in L'}, L)\end{array}}\ \text{P3}$$

$$\frac{\begin{array}{c} j \in J \qquad m_i^{i\in I} = \text{MethLabel}(CItf_j) \\ i \in I \qquad \text{Futures for requests on } m_i \text{ can be sent as parameter} \qquad p \in \mathbb{N} \qquad gf \in GRef \end{array}}{\langle -, -, -, -, j{\mapsto}i{\mapsto}p{\mapsto}!Forward(gf, val), -, -, -\rangle \rightarrow !Forward(gf, val) \in SV_C^{F1}(CItf_j^{j\in J}, CName)}\ \text{P4}$$

parameter passes by an intermediate *FutDetect_m* process that creates a local proxy representing the transmitted future, it is denoted FD_l in the equation below:

$$\frac{\begin{array}{c} m_l^{l\in L} = \text{MethLabel}(SItf_i^{i\in I}) \\ m_l^{l\in L'} \text{ are the methods that can receive a future as parameter} \qquad PM\_F1 = [\![F1]\!]_{proxyManager} \\ \forall l \in L'.\ FD_l = [\![m_l]\!]_{FutDetect} \qquad SV^{F1} = SV_S^{F1}(m_l^{l\in L'}, L) \cup SV_C^{F1}(CItf_j^{j\in J}, CName) \end{array}}{\begin{array}{l}[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!]^{F1} = [\![CName < \overleftrightarrow{SItf_i^{i\in I}}, CItf_j^{j\in J}, \overleftrightarrow{M_k^{k\in K}} >]\!] \oslash call\_m_l^{l\in L'} \\ \qquad \oplus \langle\!\langle \langle\!\langle P, FD_l^{l\in L'} \rangle\!\rangle, PM\_F1, \langle\!\langle P, \overrightarrow{Proxy\_F1^{\mathbb{N}}} \rangle\!\rangle, SV^{F1} \rangle\!\rangle\end{array}}$$

The new pNet extends the old one thanks to the operator $\oplus$ defined previously. Concerning methods that can receive a future as parameter, direct *Call* invocations from the body to the service methods are removed by the $\oslash$ operator. The new synchronisation vectors are defined in Table 4.

There are seven entries in the new synchronisation vectors of rule [P3]. The two first ones, [P3.1] and [P3.2] intercept the invocation from the body to the service method, those invocations now are intercepted by the *FutDetect* process. [P3.3] and [P3.4] deal with the creation of proxies for futures received as argument, which is quite similar to a classical proxy creation, as shown in Section 4.1. All the service methods can access the *Proxy_F1* proxies by a *GetValue_F1*[8], which is expressed by [P3.5]. The new future proxies are updated by a *Forward* action instead of a R action for usual proxies [P3.6]. Finally, [P3.7] emits a forward from the first-class future proxy. Indeed, if the component that received a first-class future itself forwards the future to another component (inside a request parameter), it also needs to forward the future value when it is known.

The new synchronisation vectors on the client side of the primitive components allow the emission of *Forward* actions (rule [P4]): when a future proxy of kind *FProxy_m* emits a *Forward*, this *Forward* is sent to the outside of the primitive component.

[8] The future references can be transferred between two service methods only if the component is stateful

Table 5: Binding synchronisation vectors for first class futures.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle$ *Queue*, *Body*, *DelegationMethods*, *ProxyManagers*, *Proxies*, *Subcomponents* $\rangle\!\rangle$

$$\frac{\begin{array}{c}(C.CI, C'.SI) \in Binding_b^{b\in B} \\ C = \text{Name}(Comp_k) \qquad C' = \text{Name}(Comp_{k'}) \qquad CItf_i'^{i\in I} = CItfs(Comp_k) \qquad i \in I \\ CI = \text{Name}(CItf_i') \qquad gf \in GRef \qquad \exists m_n \in \text{MethLabel}(CItf_i').\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-,(k{\mapsto}!Forward(gf,val), k'{\mapsto}?Forward(gf,val))\rangle \rightarrow Forward(gf,val), \\ \in SV_B^{F1}(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K})\end{array}} \text{ C8}$$

$$\frac{\begin{array}{c}(This.SI, C.SI_2) \in Binding_b^{b\in B} \qquad i \in I \qquad gf \in GRef \qquad SI = \text{Name}(SItf_i) \\ C = \text{Name}(Comp_k) \qquad \exists m_n \in \text{MethLabel}(SItf_i).\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-,k{\mapsto}?Forward(gf,val)\rangle \rightarrow ?Forward(gf,val) \\ \in SV_B^{F1}(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K})\end{array}} \text{ C9}$$

$$\frac{\begin{array}{c}(C.CI, This.CI_2) \in Binding_b^{b\in B} \qquad j \in J \qquad gf \in GRef \qquad CI = \text{Name}(CItf_j) \\ C = \text{Name}(Comp_k) \qquad \exists m_n \in \text{MethLabel}(CItf_j).\ m_n \text{ can pass a future as parameter}\end{array}}{\begin{array}{r}\langle -,-,-,-,-,k{\mapsto}!Forward(gf,val)\rangle \rightarrow !Forward(gf,val) \\ \in SV_B^{F1}(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K})\end{array}} \text{ C10}$$

First-class proxies As shown in the illustrative example, the proxy of future that can be transmitted to other components is slightly different from the definition of Section 4.1.5: it can additionally emit a *!Forward* action. This action has a generalised referenced to the future as parameter, it can be computed locally as it is only formed of the local future identifier, the component, and the interface which are all known when the proxy is built. As mentioned above, for futures that can be transmitted, the $[\![\ ]\!]_{proxy}$ function is enriched as illustrated in Figure 11: $[\![m, gf]\!]_{proxy}$ is defined by the pLTS *FProxy_ m(gf)* in the figure. For the other futures, the definition of Section 4.1.5 is still valid.

5.1.3 Composite Components: New Synchronisation Vectors

Concerning composite components, the composition of the pNets is unchanged, we only need to add new synchronisation vectors for transmitting *Forward* actions between components, this is specified by the new behavioural semantics:

$$\begin{array}{l}[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_b^{b\in B} >]\!]^{F1} = \\ \qquad [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, Binding_b^{b\in B} >]\!] \\ \qquad\quad \oplus \langle\!\langle SV_B^{F1}(Binding_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K})\rangle\!\rangle\end{array}$$

The three rules of Table 5 define the transmission of forward communications. Rule [C8] deals with brother bindings: consider two components at the same level, if one forwards a future value, the other should receive it. Rule [C9] (resp. [C10]) transmits forward reception (resp. emission) along import (resp. export) bindings.

Note that each $gf \in GRef$ can easily be restricted; indeed the origin of the future (component, method label) is known and the future flow can be approximated.

5.2 Semantics of Binding Controllers

By nature, GCM inherits from Fractal reconfiguration capabilities. The component structure is known at runtime and can be introspected and modified dynamically. Mainly, Fractal and GCM provide capabilities for adding and removing components inside a composite component, and for changing bindings between components.

In the Fractal component model and in GCM, reconfiguration and in particular dynamic binding/un-binding of components is specified such that it is very close to what happens in object-oriented languages. In particular, the bind/unbind operations are addressed to the component that owns the *client interface to be reconfigured*, indeed if one thinks of a component as an object, it is the object that holds the reference to the target object. It is then this client interface that must change the reference it holds. In our behavioural semantics however, the synchronisation between components is done at the higher level in the hierarchy: this approach corresponds better to the definition of pNets and to the component structure. It is also more compositional: the behaviour of a component is totally independent from the component to which it is bound.

Here, we want to specify binding controllers that are similar to Fractal ones in order to verify real Fractal/GCM applications. For this we will have to re-introduce some form of reference in the components: for each reconfigurable client interface, we will define a bind operation that receives as parameter a reference to the server interface to be bound to this client interface (in practice this parameter can be the index in the list of interfaces/components that can potentially be bound). The "reference" will then be passed as parameter to the request emission, and at the higher level we will use this reference to send the request to the right target.

To define reconfigurable bindings, we introduce targets, denoted by t, that range over the set of qualified references to interfaces ($QName$). We will write $t = C.SI$ to check whether the reference contained in the variable t is the interface SI of the component C. In practice, t should store an index among the possible bindings, and $t = C.SI$ will check if the index t corresponds to the interface SI of the component with name C.

To deal with reconfigurable interfaces, a preliminary step is to know which interfaces can be reconfigured: those interfaces will be assigned a binding controller as explained above. Additionally, it is necessary to know statically which set of interfaces can be bound to this one in order to generate the possible behaviours of the component system. We suppose there is a set *TopBinding* that over-approximates the set of all bindings a component system can contain. A practical way to specify this set could be to extend the ADL with a new kind of binding, a *potential binding* that can be activated at runtime but is not active upon instantiation. The allowed reconfigurations would then only be the ones that activate a potential binding.

5.2.1 Principles and Illustrative Example

For encoding reconfigurable bindings in our behavioural specification, we follow the principles described below.

A component is endowed with a Binding Controller (BC) interface to bind and unbind its client interfaces to other components through primitive bindings.

The binding controller pLTS attached to each interface (see Figure 12) controls the bindings of a given client interface. It receives the control actions ($?Bind(t)$ and $?Unbind$) and emits status actions ($!Bound(t)$ and $!Unbound$) that are used to allow the component to forward the request ($!Q\_m(f, arg)$) to the appropriate bound interface. The target is, in a first time attached to the request emission. Then, the encompassing composite component will use this target to synchronise the adequate components. The distinguished action ($Error("Unbound")$) visible from the higher levels of hierarchy is triggered whenever a request is performed over an unbound interface. We symbolise the branching between the two components by some guarded invocation

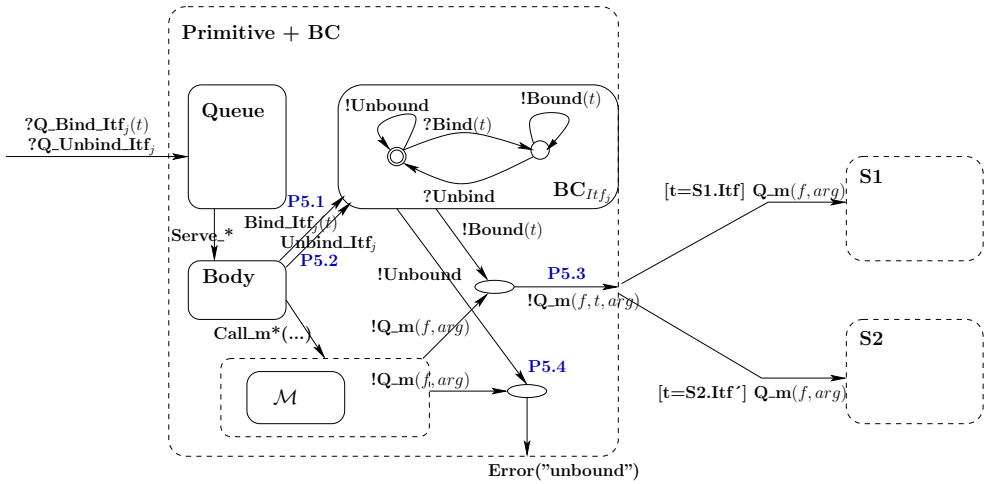


Figure 12: Adding a Binding Controller

in Figure 12; in the synchronisation vectors we will check in the premises of the rules whether $t = S1.Itf$ in order to trigger the right invocation.

5.2.2 Reconfigurable Primitive Components: pNets and Synchronisation Vectors

The semantics for reconfigurable primitive components supposes that the set of client interfaces that can be configured is known; it is denoted $CItf_j^{j\in J'}$, it is a subset of the set of client interfaces $CItf_j^{j\in J}$ of the component.

To provide reconfiguration capacities inside the pNet model for a primitive component, we add a family of sub-pNets encoding the binding controller. This family has as many elements as there are reconfigurable interfaces. Each binding controller is defined by the pLTS shown in Figure 12: $[\![\ ]\!]_{BC}$ is a function that takes a client interface, and returns a binding controller pLTS as described in Figure 12: the process BC_{Itf_i} in the figure defines the value of $[\![Itf_i]\!]_{BC}$. Additionally, the request emission for those interfaces has to be synchronised with the status of the binding controller. For this we remove ($\oslash$) the synchronisation vectors for emitting requests on those interfaces and replace them by the new ones defined below ($\oplus$). Finally, now that the bind and unbind requests are two new kind of non-functional requests, the set NF introduced in Section 4 is non-empty:

$$NF = \bigcup_{j\in J', t\in QName} \{Bind\_Itf_j(t), Unbind\_Itf_j\} \quad \text{where } CItf_j^{j\in J'} \text{ are the reconfigurable interfaces}$$

This implies that the request queue and the body of the primitive take into account binding requests. $[\![\ ]\!]^{BC}$ is the behavioural semantics of a component equipped with reconfigurable interfaces. The rule below defines this semantics for primitive components:

$$\frac{\begin{array}{c} m_l^{l\in L} = \text{MethLabel}(SItf_i^{i\in I}) \qquad CItf_j^{j\in J'} \text{ are the reconfigurable interfaces} \qquad j \subseteq J' \\ m_l^{l\in L'} = \text{MethLabel}(CItf_j^{j\in J'}) \qquad \forall j \in J'.\ BC_j = [\![CItf_j]\!]_{BC} \qquad SV^{BC} = SV_C^{BC}(CItf_j^{j\in J'}, L) \end{array}}{\begin{array}{l} [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!]^{BC} = \\ \qquad [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, M_k^{k\in K} >]\!] \oslash (!Q\_m_l)^{l\in L'} \oplus \langle\!\langle\!\langle P, \overleftrightarrow{BC_j^{j\in J'}} \rangle\!\rangle\!\rangle, SV^{BC} \rangle\!\rangle \end{array}}$$

Table 6: Client synchronisation vectors for reconfigurable primitive components.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies, BindingControllers \rangle\!\rangle$

$$
\begin{array}{l}
l \in L \qquad j \in J' \qquad m_i^{i \in I} = \text{MethLabel}(CItf_j) \qquad i \in I \qquad t \in QName \qquad f \in \mathbb{N} \\
\hline
\{\langle -, !Bind\_Itf_j(t), -, -, -, j{\mapsto}?Bind(t)\rangle \rightarrow Bind\_Itf_j(t), \quad [1] \\
\ \langle -, !Unbind\_Itf_j, -, -, -, j{\mapsto}?Unbind\rangle \rightarrow Unbind\_Itf_j, \quad [2] \\
\ \langle -, -, l{\mapsto}!Q\_m_i(f, arg), -, -, j{\mapsto}!Bound(t)\rangle \rightarrow !Q\_m_i(p, t, arg), \quad [3] \\
\ \langle -, -, l{\mapsto}!Q\_m_i(f, arg), -, -, j{\mapsto}!Unbound\rangle \rightarrow Error(\text{“Unbound”}), \quad [4] \\
\ \langle !Serve\_Bind\_Itf_j(t), ?Serve\_Bind\_Itf_j(t), -, -, -, -\rangle \rightarrow Serve\_Bind\_Itf_j(t), \quad [5] \\
\ \langle !Serve\_Unbind\_Itf_j, ?Serve\_Unbind\_Itf_j, -, -, -, -\rangle \rightarrow Serve\_Unbind\_Itf_j\} \quad [6] \\
\qquad\qquad \subseteq SV_C^{BC}(CItf_j^{j \in J'}, L)
\end{array}
\quad \text{P5}
$$

The new synchronisation vectors for client interfaces are defined by rule [P5] shown in Table 6. Items [P5.1] and [P5.2] transmit the *Bind* and *Unbind* actions from the body to the binding controller. In [P5.3], when a request on a method on the reconfigurable client interface is issued, the binding controller adds an additional parameter containing the target of the invocation. If there is no target, i.e. the interface is *Unbound*, then an error is raised [P5.4]. The two last items [P5.5] and [P5.6] synchronise the request queue with the body in order to serve bind and unbind requests. The target of the invocation will be used at the higher level in the hierarchy, as expressed below.

5.2.3 Reconfigurable Composite Components: pNets and Synchronisation Vectors

First of all, a composite component can be assigned binding controllers allowing its client interfaces to be bound to a new target. The introduction of such binding controllers is extremely similar to the case of the primitive components. We do not formalise it here, the only two technical differences are: the number of elements of the pNet is different, and the intercepted requests are emitted by the $Deleg\_m_i$ sub-pNets instead of the service methods. Also, as an internal client interface corresponds to each server interface, the server interfaces can be assigned a binding controller if they are reconfigurable. Considering their similarity with the case of primitive components, we do not describe formally here the binding controllers for composite components.

More interestingly, targets of invocations should be used by the synchronisation vectors generated by the potential bindings. We describe in Table 7 the synchronisation vectors corresponding to bindings that can be reconfigured, those synchronisation vectors are able to use the target reference sent as argument to address the request to the bound component. The new behavioural semantics thus requires to replace the items transmitting requests for methods for reconfigurable interfaces by the ones defined in Table 7.

$$
\begin{array}{l}
[\![CName < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, TopBinding_b^{b \in B} >]\!]^{BC} = \\
\quad [\![CName < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, TopBinding_b^{b \in B} >]\!] \\
\qquad \oslash \{Q\_m_n | m_n \text{ belongs to a reconfigurable interface}\} \\
\qquad \oplus \langle\!\langle SV_B^{BC}(TopBinding_b^{b \in B}, SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, CName) \rangle\!\rangle
\end{array}
$$

In fact the operator $\oplus$ is only used here to add new synchronisation vectors, no sub-pNet is added. The synchronisation vectors for replies are kept from the non-reconfigurable case, except that there must be one reply channel for each binding of *TopBinding* instead of each binding of

Table 7: Binding synchronisation vectors for reconfigurable composite components.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle$ *Queue*, *Body*, *DelegationMethods*, *ProxyManagers*, *Proxies*, *Subcomponents*, *BC*$\rangle\!\rangle$

$$\frac{\begin{array}{c}(\mathit{This}.SI, C.SI_2) \in \mathit{TopBinding}_b^{b\in B} \\ \mathit{This}.SI \text{ is reconfigurable} \quad i \in I \quad SI = \text{Name}(SItf_i) \quad C = \text{Name}(Comp_k) \\ m_n^{n\in N} = \text{MethLabel}(SItf_i) \quad n \in N \quad m'_n = m_n\{\!\{SI \leftarrow SI_2\}\!\} \quad q \in \mathbb{N} \quad t = C.SI_2\end{array}}{\begin{array}{c}\langle -, -, n{\mapsto}!Q\_m_n(q,t,arg), -, -, k{\mapsto}?Q\_m'_n(q,arg), -\rangle \rightarrow Q\_m_n(q,arg), \\ \in SV_B^{BC}(\mathit{TopBinding}_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, CName)\end{array}} \text{C11}$$

$$\frac{\begin{array}{c}(C.CI, \mathit{This}.CI_2) \in \mathit{TopBinding}_b^{b\in B} \\ C.CI \text{ is reconfigurable} \quad j \in J \quad CI = \text{Name}(CItf_j) \quad C = \text{Name}(Comp_k) \\ m_n^{n\in N} = \text{MethLabel}(CItf_j) \quad n \in N \quad m'_n = m_n\{\!\{CI \leftarrow CI_2\}\!\} \quad f \in \mathbb{N} \quad t = \mathit{CName}.CI_2\end{array}}{\begin{array}{c}\langle ?Q\_m'_n(f,arg), -, -, -, -, k{\mapsto}!Q\_m_n(f,t,arg), -\rangle \rightarrow Q\_m_n(f,arg), \\ \in SV_B^{BC}(\mathit{TopBinding}_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, CName)\end{array}} \text{C12}$$

$$\frac{\begin{array}{c}(C.CI, C'.SI) \in \mathit{TopBinding}_b^{b\in B} \quad C.CI \text{ is reconfigurable} \quad C = \text{Name}(Comp_k) \\ C' = \text{Name}(Comp_{k'}) \quad CItf_i'^{i\in I} = CItfs(Comp_k) \quad i \in I \quad CI = \text{Name}(CItf'_i) \\ m_n^{n\in N} = \text{MethLabel}(CItf'_i) \quad n \in N \quad m'_n = m_n\{\!\{CI \leftarrow SI\}\!\} \quad f \in \mathbb{N} \quad t = C'.SI\end{array}}{\begin{array}{c}\langle -, -, -, -, -, (k{\mapsto}!Q\_m_n(f,t,arg), k'{\mapsto}?Q\_m'_n(f,arg)), -\rangle \rightarrow Q\_m_n(f,arg), \\ \in SV_B^{BC}(\mathit{TopBinding}_b^{b\in B}, SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, CName)\end{array}} \text{C13}$$

Binding. There is no need to use addressing in this direction: a single component will necessarily send each reply because a single one received the corresponding request. The synchronisation vectors shown in Table 7 replace the ones of Section 4.2 for dealing with reconfigurable bindings. Now, SV_B receives an additional parameter: *CName*, the name of the composite component that contains the bindings. The rules of Table 7 are relatively straightforward: the target t is used and unified with one destination of a potential binding to trigger the right communication. Note that, in rule [C12], the encompassing composite is referred by its name, *CName*, and not by *This* because t is the "reference" that is stored in the binding controller of the contained component, and there is no reason to refer to the parent component by *This*.

Note that the initial state of the binding controllers is *unbound*. In practice, if the ADL of the system defines initial bindings, then it is necessary to change the coresponding initial states, either in the definition of the pLTSs of the binding controllers, or by initialising the request queues with a sequence of binding commands.

5.3 Multicast Interfaces

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

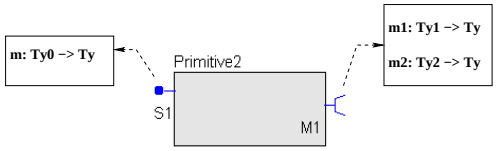


Figure 13: Example with a Multicast Interface

5.3.1 Principle of the approach

Figure 13 shows a primitive component with a multicast client interface. When a client interface is of type Multicast, it may have a variable number of outgoing bindings (it is bound to the server interfaces of several components). Invocations emitted by a multicast interface are broadcasted to all the server interfaces bound to it. In GCM, depending on the interface policy, arguments of requests emitted by the interface can be dispatched in a parameterisable manner. Here, we suppose that the argument is broadcasted to all the destination components. Then results will come back in an asynchronous way from the elements of the group. The encoding of multicast

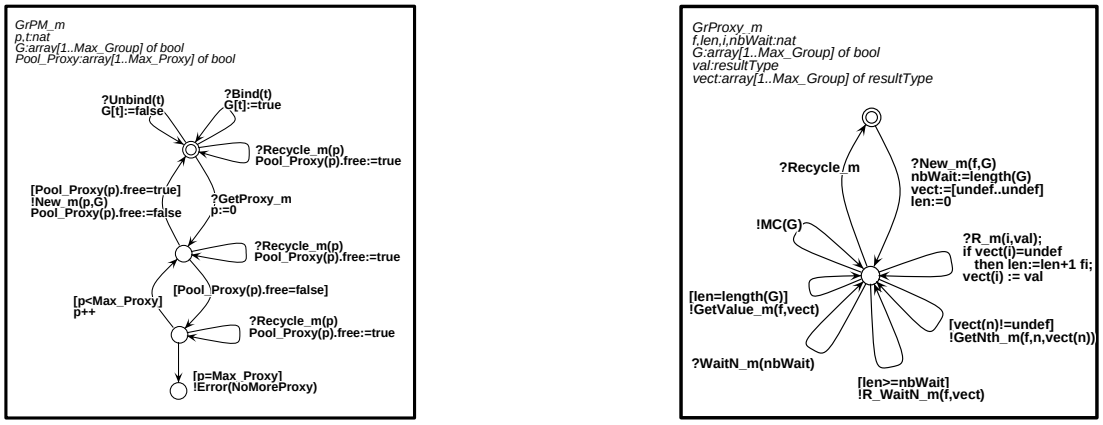


Figure 14: A Group Manager and a Group Proxy

interfaces relies on the two pLTSs of Figure 14 for dealing with the specific future proxies:

- one Group Proxy Manager for each method of each Multicast interface: the left part of Figure 14 shows the process $GrPM\_m$ that defines the value of $[\![m]\!]_{proxyManager}$ in case the method m belongs to a multicast interface. Compared to a classical proxy manager, each group proxy manager is also in charge of managing changes in the group content (bindings and unbindings). Each binding/unbinding operation targeted at a client multicast interface is thus broadcasted to all the group proxy managers of all the methods of this interface.

- for each method in the Multicast interface, an indexed family of Group Proxies: the right side of Figure 14 defines a process $GrProxy\_m$ that overrides the value of $[\![m]\!]_{proxy}$ in case m belongs to a multicast interface. Upon each request invocation, a corresponding proxy is activated and initialised. Each incoming reply to this request will update a result vector (additional conditions check that a given component does not reply twice). Results

can be accessed by the service methods that can query either the result vector totally filled (*GetValue_ m*), or a partially filled vector (*WaitN_ m*). As in the case of the standard future proxies, group proxies can be Recycled, whenever it can be decided that it will never be used again.

Note that in the Group Manager pLTS, the group variable G is modified by $!Bind$ and $!Unbind$ messages. Each time the Group Manager activates a new Group proxy ($!New\_m$ message), it sends a copy of the value of G, so that even if reconfigurations occur, each proxy keeps its own copy of Group, on which the invocation has been performed.

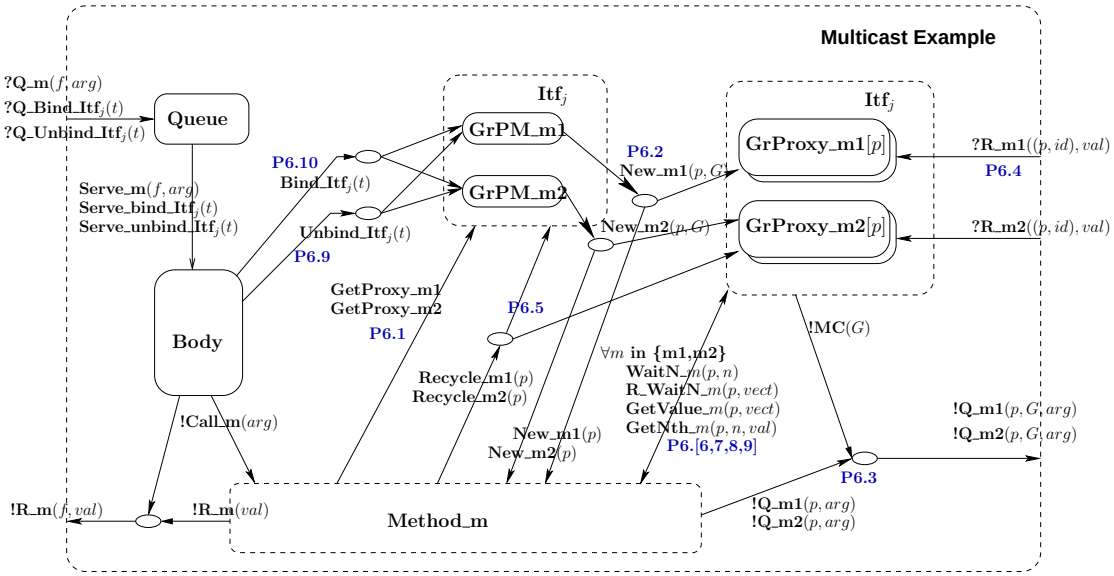


Figure 15: pNet model for Figure 13

The pNet of a primitive component with a multicast interface is shown in Figure 15, it corresponds to the primitive component that was shown in Figure 13. The figure shows the parts of the pNet that are specific to the handling of multicast interfaces. It shows that binding operations received in the queue are broadcasted to each group proxy manager of the targeted multicast interface. Then proxy creation (*New_ m*) is synchronised similarly to the case of usual interfaces except that the current status of the multicast interface (G) is transmitted to the created proxy. The main specificity of the synchronisation vectors for multicast interfaces is the fact that a request emission is also synchronised with the corresponding proxy that sends the group G targeted by the invocation ($!MC(G)$), and the outgoing request also sends G as argument. This argument containing the invoked group will be used at the higher level in the hierarchy by the encompassing composite component. Consequently, G *ranges over sets of qualified names.* We will also use variable g for members of G: g ranges over qualified names. Finally, actions for accessing the group proxy (*waitN_ m*, *GetValue_ m*, *GetNth_ m*) can be invoked by the service methods.

Figure 16 illustrates the principle of the synchronisation occurring at the higher hierarchical level, in the composite that contains the component with the multicast interface. Depending on the group targeted by the invocation, a different synchronisation vector is used, and the adequate server interfaces receive the invocation.

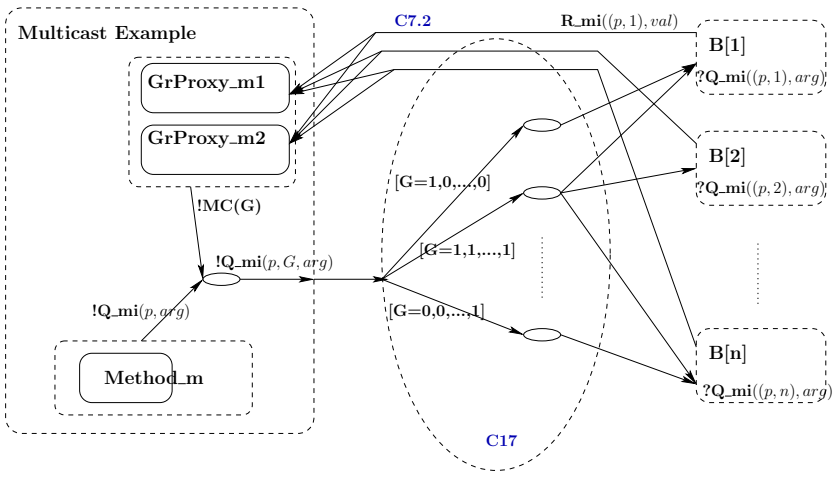


Figure 16: Dynamic Connector for a Multicast Interface

As for reconfigurable interfaces, the queue and the body of the components that contain multicast interfaces must be able to accept and handle bind and unbind messages. Like for binding controller, the set of non-functional requests now contain *Bind* and *Unbind* requests.

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

The way we handle reconfigurable multicast interfaces is relatively close to what we have shown in the previous section. However, there is no binding controller here. Instead, the role of the binding controller of the previous section is split between the proxy managers and the group proxies. The proxy managers receive *Bind*/ *Unbind* actions and store the current value of the group, whereas the group proxies are responsible for emitting the current value of the group at invocation. Another difference is to be noticed: in the case of a multicast interface, the unbind operation receives a target reference as a parameter. Indeed, contrarily to the usual binding controller, multicast interfaces are in general bound to several targets, it is thus necessary to transmit the reference of the target to be unbound.

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

Table 8: Synchronisation vectors for multicast client interfaces in primitive components. The synchronised sub-pNets occur in the following order:
$\langle\!\langle Queue, Body, ServiceMethods, ProxyManagers, Proxies \rangle\!\rangle$

$$j \in J \qquad l \in L \qquad m_i^{i \in I} = \text{MethLabel}(CItf_j) \qquad i \in I \qquad p \in \mathbb{N}$$
$$CItf_j \text{ is multicast}$$

P6

$\{\langle -, -, l{\mapsto}!GetProxy\_m_i, j{\mapsto}i{\mapsto}?GetProxy\_m_i, - \rangle \rightarrow GetProxy\_m_i,$ [1]
$\langle -, -, l{\mapsto}?New\_m_i(p), j{\mapsto}i{\mapsto}!New\_m_i(p,G), j{\mapsto}i{\mapsto}p{\mapsto}?New\_m_i(G) \rangle \rightarrow New\_m_i(p,G),$ [2]
$\langle -, -, l{\mapsto}!Q\_m_i(p,arg), -, j{\mapsto}i{\mapsto}p{\mapsto}!MC(G) \rangle \rightarrow !Q\_m_i(p,G,arg),$ [3]
$\langle -, -, -, -, j{\mapsto}i{\mapsto}p{\mapsto}?R\_m_i(id,val) \rangle \rightarrow ?R\_m_i((p,id),val),$ [4]
$\langle -, -, l{\mapsto}!Recycle\_m_i(p), j{\mapsto}i{\mapsto}?Recycle\_m_i(p), j{\mapsto}i{\mapsto}p{\mapsto}?Recycle\_m_i \rangle \rightarrow Recycle\_m_i(p),$ [5]
$\langle -, -, l{\mapsto}!WaitN\_m_i(p,nb), -, j{\mapsto}i{\mapsto}p{\mapsto}?WaitN\_m_i(nb) \rangle \rightarrow WaitN\_m_i(p,nb),$ [6]
$\langle -, -, l{\mapsto}?R\_WaitN\_m_i(vect), -, j{\mapsto}i{\mapsto}p{\mapsto}!R\_WaitN\_m_i(vect) \rangle \rightarrow R\_WaitN\_m_i(p,vect),$ [7]
$\langle -, -, l{\mapsto}!GetNth\_m_i(p,nb,val), -, j{\mapsto}i{\mapsto}p{\mapsto}?GetNth\_m_i(nb,val) \rangle \rightarrow GetNth\_m_i(p,nb,val),$ [8]
$\langle -, -, l{\mapsto}?GetValue\_m_i(p,vect), -, j{\mapsto}i{\mapsto}p{\mapsto}!GetValue\_m_i(vect) \rangle \rightarrow GetValue\_m_i(p,vect),$ [9]
$\langle -, !Bind\_CItf_j(t), -, j{\mapsto}(i' \in I{\mapsto}?Bind(t)), - \rangle \rightarrow Bind\_CItf_j(t),$ [10]
$\langle -, !Unbind\_CItf_j(t), -, j{\mapsto}(i' \in I{\mapsto}?Unbind(t)), - \rangle \rightarrow Unbind\_CItf_j(t),$ [11]
$\langle !Serve\_Bind\_CItf_j(t), ?Serve\_Bind\_CItf_j(t), -, -, -, - \rangle \rightarrow Serve\_Bind\_CItf_j(t),$ [12]
$\langle !Serve\_Unbind\_CItf_j(t), ?Serve\_Unbind\_CItf_j(t), -, -, -, - \rangle \rightarrow Serve\_Unbind\_CItf_j(t)\}$ [13]
$\subseteq SV_C^{MC}(CItf_j^{j \in J}, L)$

5.3.2 Multicast Interfaces for Primitive Components

The pNet of a primitive component with multicast interfaces is similar to the pNet without multicast interfaces, except that the proxy managers and the proxies for methods of multicast interfaces are replaced by the pLTSs defined in Figure 14: $[\![\ ]\!]_{proxyManager}$ is overloaded for multicast interfaces, it returns the classical proxy manager of Figure 6 for a singleton interface, and the new one of Figure 14 for a multicast interface, and similarly for $[\![\ ]\!]_{proxy}$. Additionally to these changes in the behavioural semantics of future proxies and their managers, the synchronisation vectors are modified as shown in the next rule: synchronisation vectors containing the name of a multicast interface are replaced by a new one. Note that, as method labels contain the name of the interface, $\oslash(CItf_j^{j \in J'})$ removes all the synchronisation vectors containing the name of a method m_i of an interface $CItf_j$ among its action labels.

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

The synchronisation vectors for the multicast interface of a primitive component are defined in Table 8, they correspond to Figure 15. We introduce a new variable *vect*, similarly to *arg* and *val*, *vect* ranges over arrays of values. In the rules, we write $(i' \in I{\mapsto}(?Bind(t)))$ to represent the family $?Bind(t)^{i' \in I}$; this represents a synchronisation vector that broadcasts the action $?Bind(t)$ to all the elements inside I, here all the proxy managers of the reconfigured interface.

Cases [P6.1] and [P6.2] are used to create a proxy: compared to Section 4.1, the content of the group targeted by the invocation (G) is transmitted to the proxy. The element [P6.3] expresses request emission, with the proxy emitting the adequate value of G. Compared to singleton interfaces, reply reception uses the fact that an index *id* is attached the future, and

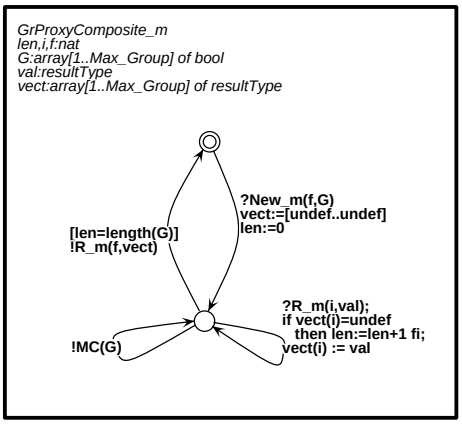


Figure 17: The Proxy of a Multicast interface inside a Composite component

transmits this index to the future proxy for multicast interface. Recycle [P6.5] is similar to the non-multicast case. Elements [P6.6] and [P6.7] are used for waiting for a given number, nb, of responses: first, nb is sent to the proxy ($WaitN\_m_i$ action), and then a reply is sent back to the service method by $R\_WaitN\_m_i$. The two next rules [P6.8] and [P6.9] do not require to work in a request/reply manner, the proxy can directly emit $GetNth\_m_i$ and $GetValue\_m\_m_i$ actions when they are enabled, i.e. when the necessary replies have arrived. The next two elements ([P6.10] and [P6.11]) deal with the reconfiguration of the multicast interface: they transmit bind and unbind orders from the body to all the group proxy managers corresponding to the reconfigured interface. The two last items [P6.12] and [P6.13] synchronise the request queue with the body in order to serve bind and unbind requests.

5.3.3 Multicast Interfaces for Composite Components

Concerning composite components, two aspects have to be added. First, composite components can also have multicast client interfaces. Compared to other reconfigurable interfaces, multicast interfaces have to deal with future proxies in a special manner, the behaviour of a composite multicast interface is different enough from the primitive ones to necessitate its complete specification. Second, composite components have to encode the synchronisation between multicast interfaces and the plugged components.

Concerning the first point, to be precise, multicast interfaces can be either external client interfaces, or internal client interfaces (that correspond to multicast external server interfaces). There is no multicast server interface, thus to state that an internal client interface is multicast, we tag as multicast the corresponding server interface (see Section 2.3.3).

Similarly to primitive components, the proxy managers and the future proxies are different for multicast interfaces compared to normal interfaces. The proxy manager for a method of a multicast interface is the same as the one for primitive components (see Figure 14). The group proxy is quite different and quite simpler than the primitive component case, it is shown in Figure 17: the process $GrProxyComposite\_m$ defines the behavioural semantics of the proxy for a multicast interface of a composite component: $[\![m]\!]_{proxy}$. After creation and emission of a $!MC(G)$ action, this future proxy accumulates replies and when all futures have been received, a $!R\_m(f, vect)$ action is emitted. Note that, as there is no application logic encapsulated in the composite component, a given policy must be chosen to know when a reply is issued from a multicast interface belonging to a composite. Here we choose to reply the whole vector of

replies when it is completely filled. It would also be possible to implement a different policy, for example return the most frequent result, this would be particularly adapted to the case where the multicast interface targets several replicas of the same component for redundancy purposes. The behavioural semantics of proxies and their managers feature the new semantics for multicast interfaces. Then, the behavioural semantics of a composite component supporting multicast interfaces is the following: it redefines the synchronisation vectors for transmitting request and replies concerning multicast interfaces, and the ones concerning the group proxies and their management.

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

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

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

$$[\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, TopBinding_b^{b\in B} >]\!]^{MC} = [\![CName < SItf_i^{i\in I}, CItf_j^{j\in J}, Comp_k^{k\in K}, TopBinding_b^{b\in B} >]\!] \oslash Q\_m_l^{l\in L} \oslash GetProxy\_m_l^{l\in L'} \oslash New\_m_l^{l\in L'} \oslash R\_m_l^{l\in L'} \oplus \langle\!\langle SV^{MC} \rangle\!\rangle$$

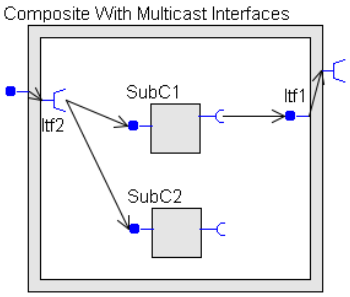


Figure 18: A Composite component with multicast internal and external interfaces, VCE graphics

Figure 19 illustrates the construction of synchronisation vectors for a composite component having one internal client multicast interface, and one external client multicast interface. The rules for generating the synchronisation vectors dealing with the multicast server and client interfaces of a composite component are shown in Table 9. The first rule [C14] defines the emission [C14.1] of a request on a multicast external client interface and the reception of a reply by this interface [C14.2]. The request is emitted by the delegation method indexed k, the proxy provides the target group G. The emission of request by *internal* multicast client interfaces will be described below as it depends on the bindings inside the composite component. The reply reception [C14.2] is similar to the reply reception in a primitive component [P6.4].

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

We now describe how we generate synchronisation vectors for bindings involving a multicast interface. When defining well-formed components, we required that there is no looping binding and two bindings from the same MC interface do not reach the same component; this allows

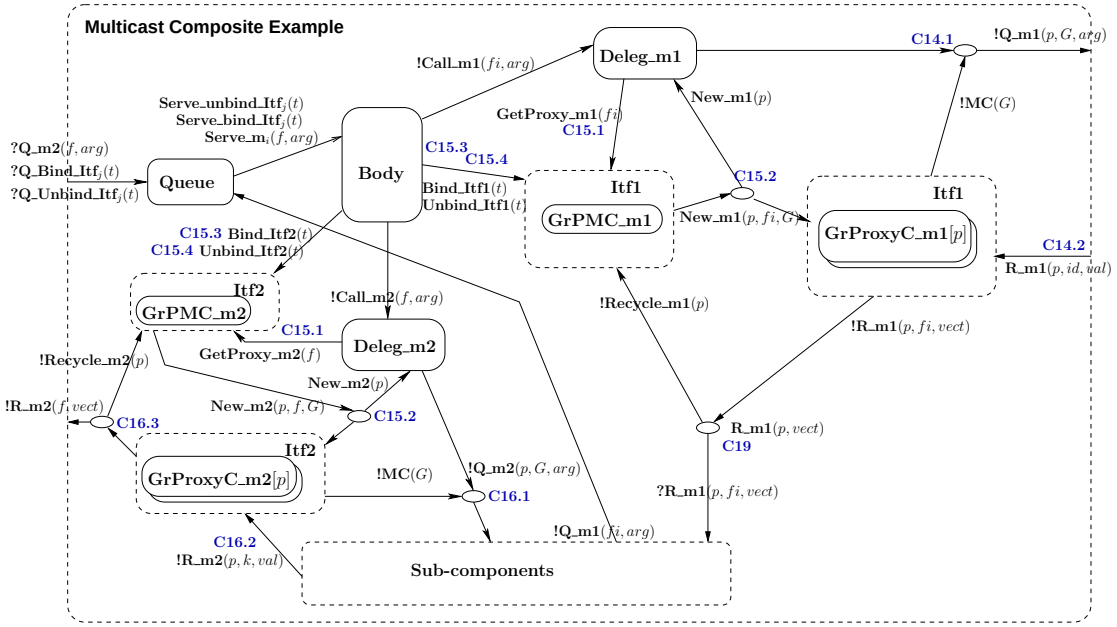


Figure 19: pNets for the Composite component with multicast internal and external interfaces

us here to write synchronisation vectors for expressing multicast interfaces. Indeed, those restrictions ensure that in each of the synchronisation vectors expressed below, each sub-pNet of the composite pNet performs a single action in a given synchronisation vector. Section 5.4 will discuss how to avoid this limitation.

In order to define reconfigurable bindings for multicast interfaces, similarly to Section 5.2, we rely on *TopBinding*, the maximal set of bindings that can exist. Then we define four rules for building synchronisation vectors from *TopBinding*. For each of the three first rules, we build G_{max} the maximal set of qualified names that can be bound to the considered multicast interface. Then we consider all the possible subsets G of G_{max}; these are the possible sets on which the request invocations originating from the multicast interface can arrive. Note that SV_B has now *CName* as additional parameter, it is the name of the composite component that contains the bindings. We use two auxiliary functions for computing G_{max}, and for obtaining the index of the component inside a qualified name:

$$\text{Gmax}(\mathit{TopBinding}_b^{b\in B}, \mathit{QName}, \mathit{CName}) = \{C.\mathit{Itf} | (\mathit{QName}, C.\mathit{Itf}) \in \mathit{TopBinding}_b^{b\in B} \wedge C \neq \mathit{This}\}\{\!\{\mathit{This} \leftarrow \mathit{CName}\}\!\}$$

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

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

Table 10 shows rules for building synchronisation vectors related to bindings involving a multicast interface. Rule [C16] deals with the case when a server interface is multicast, or more precisely an internal client interface is multicast. The item [C16.1] in the synchronisation vector expresses request emission. Each request emitted by a delegation method is broadcasted to the bound interfaces, where the destination set G is taken from the adequate group proxy. For each destination of the invocation $g \in G$, the index of the target component is obtained thanks to the *Target* function; then $\text{Index}_G(g)$ is attached to the future identifier. Replies can originate

Table 9: Synchronisation vectors for multicast interfaces in composite components. The synchronised sub-pNets occur in the following order:
$\langle\!\langle$ *Queue*, *Body*, *DelegationMethods*, *ProxyManagers*, *Proxies*, *Subcomponents* $\rangle\!\rangle$

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

$$\frac{h \in H \qquad m_k^{k\in K} = MethLabel(Itf_h) \qquad k \in K \qquad f,p \in \mathbb{N} \qquad Itf_h \text{ is multicast}}{\begin{array}{ll}\{\langle -,-,k{\mapsto}!GetProxy\_m_k(f),h{\mapsto}k{\mapsto}?GetProxy\_m_k(f),-,-\rangle \rightarrow GetProxy\_m_k(f), & [1]\\ \langle -,-,k{\mapsto}?New\_m_k(p),h{\mapsto}k{\mapsto}!New\_m_k(p,f,G),h{\mapsto}k{\mapsto}p{\mapsto}?New\_m_k(f,G),-\rangle \rightarrow New\_m_k(p,f,G)\} & [2]\\ \langle -,!Bind\_Itf_h(t),-,h{\mapsto}(k' \in K{\mapsto}?Bind(t)),-\rangle \rightarrow Bind\_Itf_h(t), & [3]\\ \langle -,!Unbind\_Itf_h(t),-,h{\mapsto}(k' \in K{\mapsto}?Unbind(t)),-\rangle \rightarrow Unbind\_Itf_h(t), & [4]\\ \langle !Serve\_Bind\_Itf_h(t),?Serve\_Bind\_Itf_h(t),-,-,-\rangle \rightarrow Serve\_Bind\_Itf_h(t), & [5]\\ \langle !Serve\_Unbind\_Itf_h(t),?Serve\_Unbind\_Itf_h(t),-,-,-\rangle \rightarrow Serve\_Unbind\_Itf_j(t)\} & [6]\\ \subseteq SV_C^{MC}(CItf_j^{j\in J}, Itf_h^{h\in H}) & \end{array}} \quad \text{C15}$$

from each g member of G independently; overall, we build one reply vector for each element of G_{max}. The synchronisation vectors for replies are split into two vectors compared to singleton interfaces: the return of results from sub-components [C16.2] is done independently from the reply of the overall result [C16.3], the second only occurs when the vector of replies is filled. The last item of the first rule is in fact unrelated to bindings, it is however more natural to mention it here; it sends a reply out of the composite component when the vector of replies of a future proxy f of a multicast server interface has been completely filled. Method renaming (computation of m'_g from m_j) relies on a function *Itf* that returns the interface of a method label.

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

Rule [C18] deals with the case when a sub-component has a client multicast interface that sends a request to other sub-components, but also to the encompassing component. This rule applies when the invocation is performed on a target group G that contains *CName.Itf* where *CName* is the name of the composite component. Note that as bindings are reconfigurable, the composite component can be bound or not, and depending on whether it is bound, [C17] or [C18] applies. This rule only applies for request transmission, it is similar to the preceding rule except that the composite component also receives a request and that the set of destination sub-components is obtained from G', the elements of G that are not the composite component.

The last rule [C19] deals with the sending of replies from a multicast client interface of the composite component to a sub-component. As replies for multicast are bound similarly to replies

Table 10: Binding synchronisation vectors for multicast interfaces.
The synchronised sub-pNets occur in the following order:
$\langle\!\langle$ *Queue*, *Body*, *DelegationMethods*, *ProxyManagers*, *Proxies*, *Subcomponents* $\rangle\!\rangle$

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

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

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

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

for singleton interfaces, the only difference with the rules of Section 4 is when the client interface of the composite that sends the reply is a multicast interface. Indeed, when the interface is singleton the reply occurs as soon as one reply is received, and the proxy for future is used to rename the future identifier (see Section 4.2). In the case of a multicast interface, the reply occurs independently from external communications *when the vector of replies is entirely filled*. This is visible in the rule because the global action is just an observable action of the form R_m instead of a communication reception of the form $?R\_m$. Rule [C14.2] that specifies the reception of the reply $?R$ by the composite still applies for receiving replies from other components.

5.4 Dealing with Binding Loops

The definition of *Well-formed components* in Section 2.3 does not allow a client interface to be bound to a server interface of the same component (binding loop). Also, the same multicast interface cannot be bound twice to the same component. Those restrictions do not exist in the GCM specifications; we had to add them because synchronisation vectors do not allow two actions to occur simultaneously on the same sub-pNet. Additionally, performing this kind of bindings, especially binding loops, is sometimes useful in real applications. We explain below how to overcome this restriction; we focus on the binding loop case but a similar approach can be used to allow one multicast interface to be bound twice to the same component.

A first idea to encode a binding loop could be to add an additional pNet that would transmit its input to its output; such a pNet would intercept the outgoing request invocations (and replies) and send them back to the same sub-pNet. While this approach is feasible it would un-synchronise the communication and thus does not fit the original semantics for communications.

Instead, we should add an action to the pNet of the sub-component that performs both the sending and the reception of the request (it could be labelled $Q\&R\_m$). We thus add an action exported by the sub-component that at the same time enqueues and emits a request. For all interfaces of all components that support loop-bindings, such a compound action can be added.

One possibility would be to add a generic action product operator, as in the SCCS or Meije process algebras. But this would complexify too much the structure of actions, and the complexity of model-checking. Instead, we prefer to introduce local products in a very limited manner, when really required.

Concerning the restriction on multicast interfaces, a similar approach can be taken: a single action can be added that enqueues two requests at the same time (in an order arbitrarily chosen) and if the same component is bound twice, this compound action is triggered.

6 Full example

We will sketch here a small example illustrating the most important of the constructs defined in this article. This is an application called *"HyperManager"* which role is to monitor and control a pre-existing distributed application. The *HyperManager* itself (see Figure 20) is distributed, with a GCM composite encapsulating each of the original application legacy components, together with local monitoring and control GCM components. The application administrator can issue commands through a toplevel HyperManager component, for configuring the local legacy component, setting trace level attributes in the local monitors, and trace filtering rules in the global monitor, or performing reconfiguration operations by *Binding* or *Unbinding* local components from the global HyperManager.

This allows us to illustrate the construction of pNets for primitive and composite components, with future proxies, service and client delegation processes, attribute controllers, a multicast

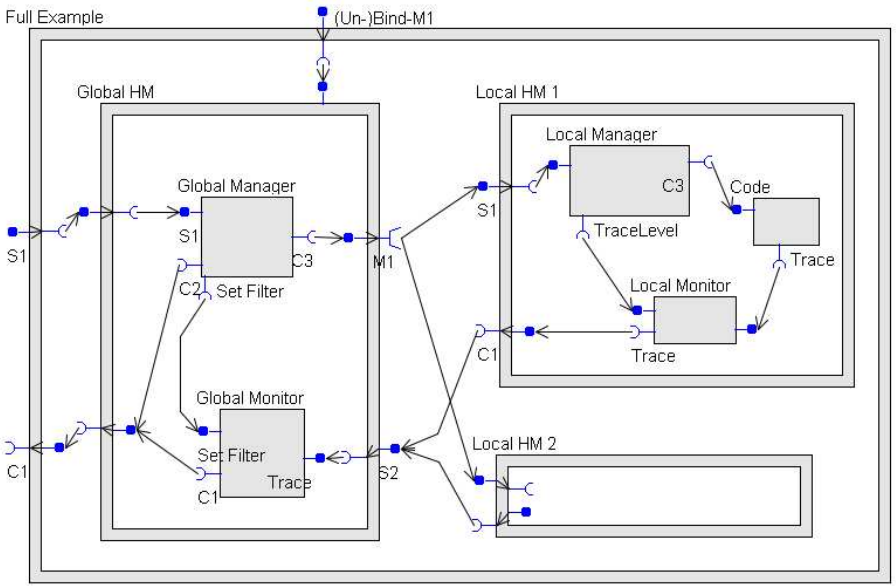


Figure 20: Component Structure of the Full Example

client interface, and reconfiguration of this multicast interface.

Additionally, this gives us the opportunity to discuss a number of implementation issues, and in particular of simplification of the generated pNet structure depending on the configuration of the GCM components considered. A brute application of the semantical rules from this article would produce an unnecessary number of management pNets, and we give example of optimisations that are applicable in a significant number of situations. A full description of the implementation of the pNets construction, and of the optimisations, is out of the scope of this paper.

6.1 Structure of the pNets semantics

We illustrate here, in an informal manner, the construction of the pNets semantics of the HyperManager example. We focus on the pNets hierarchy and their synchronisation vectors, defined in a graphical way (examples of the pLTS representing the basic blocks of the semantics will be given in Annex C). We give some figures about the complexity of this construction.

We start, in Figure 21, with the pNet structure representing the semantics of one of the (identical) *Local HM* components. Only the structure of the pNets is defined here, some of the pLTSs involved in the composition will be defined in appendix C. Each *Local HM* is a composite containing a business process represented here by a *Code* pNet, a *Local Manager* primitive component managing the requests coming from the global HyperManager, and a *Local Monitor* listening to monitoring messages coming from *Code*.

Remark that the semantic rules above defining the produced pNet structures systematicaly build an intermediate pNet containing all sub-pNets for the sub-component of a composite (resp. the sets of proxies, proxy managers, deleg methods, etc). These intermediate pNets simplify the writing of the rules, but are unnecessary when building a particular instantiation, and we have omitted them in the drawings.

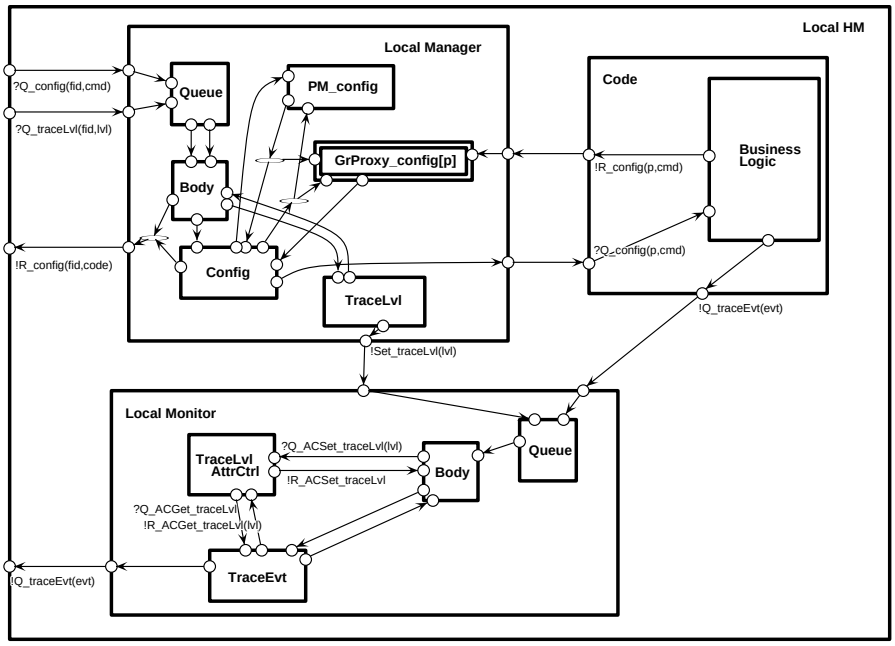


Figure 21: pNets for a Local HM composite

The `Local HM` *composite* has a very simple configuration: it receives requests on a single service interface, and directly passes these requests to the *Local Manager* sub-component, it has a single client interface whose methods have a *void* return type, and it has no reconfiguration features. This is a pattern that allows us to omit its whole control structure (Queue, Body, and proxies), and to synchronise directly the interface events with the corresponding inner pNets.

The `Local Manager` *primitive component* has a standard pNet structure, with:

- a service interface with 2 service methods *config* and *trace* that go through the Queue and Body pNets, before being routed towards their respective methods.
- a single client interface *Config* with a single method *config*, but that may be called several times from within the *config* method. This interface requires a proxy manager and a family of future proxies *Proxy_ config[p]*.
- the *config* service request requires a result code, so we have a returning *!R_ config* message, while the *trace* request simply forwards its command to the *Local Monitor* component, without requiring a return message.

The `Local Monitor` *primitive component* has a quite simple pNet structure: it has a queue listening to requests from the *Code*, but also to *Set_ traceLvl* requests addressed to the *TraceLvl* attribute controller (see Annex C, Figure 26). It has a single (client) interface, with a single method requiring no answer, so it needs no proxies.

The `Code` *object* is not a GCM component, but rather a simple encapsulation of legacy code. The API entry and return points are directly connected to the business code itself.

We move now to Figure 22, and explain the structure of the *Global HM* composite. This part features a complete pNet infrastructure in the composite membrane, and the management of a reconfigurable multicast client interface.

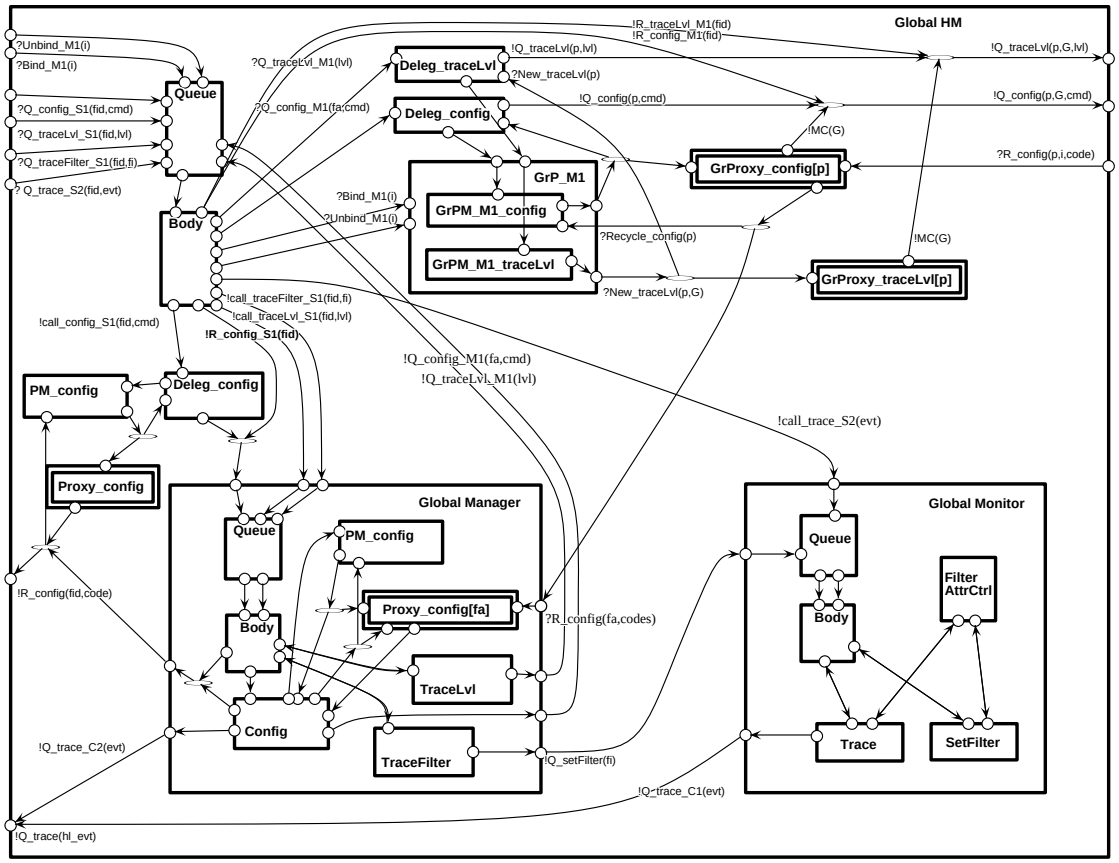


Figure 22: pNets for a Global HM composite

The `Global HM` *composite* has two sub-pNets for its internal sub-components, and a full set of pNets modelling its membrane, namely: a *Queue* receiving requests both from its service interface *S1* and from the *Global Manager* sub-component, a body dispatching these requests to the *Global Manager* or to the controllers (*Deleg, GrPM*) of the multicast interface *M1*. On the *M1* interface, only the *config* method requires an answer, and has an indexed family of group proxies, managing the *R_config* return messages. The *GrProxy_traceLvl* proxy role is only to store the group value (G) before sending the broadcasted request, so it only needs one instance, and no recycling. Last, the *config_S1* service method requires an answer, so it has a set of controllers (*Deleg, PM, Proxy*) managing this future.

The `Global Manager` *and* `Global Monitor` *primitive components* are very similar to the *Local Manager* and *Local Monitor*.

The last step is to assemble the toplevel composite, including the (dynamic) modelling of the multicast binding. In the current version of the Fractal/GCM ADL, we have no possibility for specifying this architecture in a parameterized manner: the (maximum) number of *Local HM* components must be known from the beginning. Only the group G may vary dynamically, by the effect of *Bind*/*Unbind* operations.

In Figure 23 we show this toplevel pNet, for a configuration with a maximum of 3 local components. As we do not show here the state machines (pLTSs) of the controllers, the initial

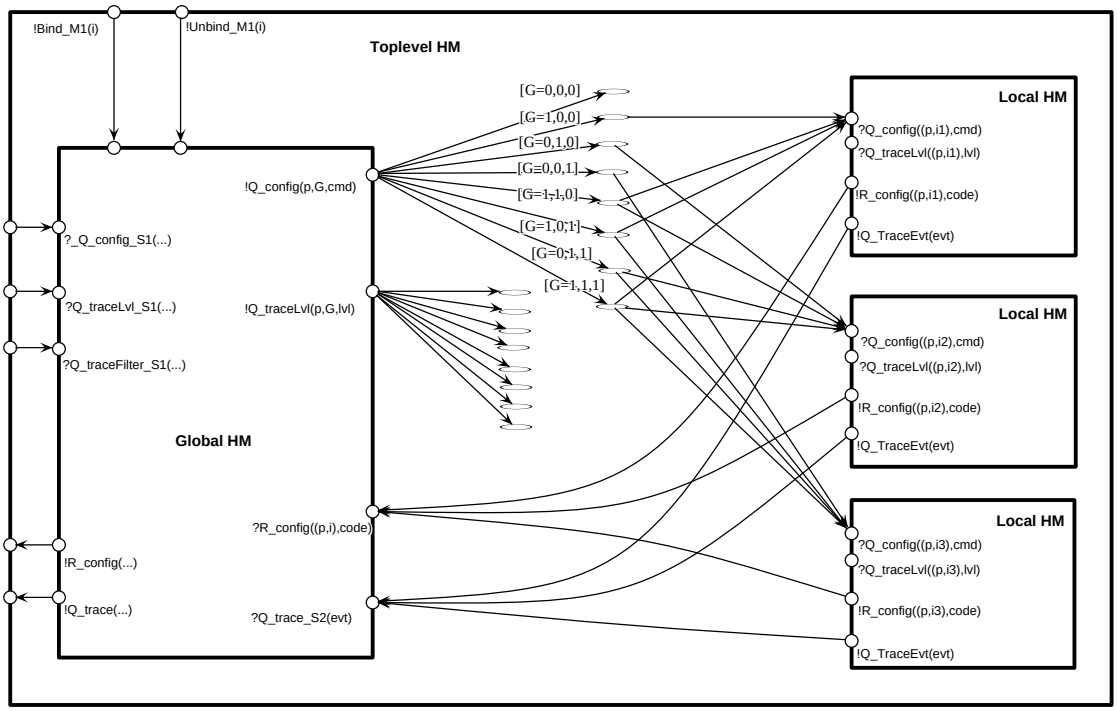


Figure 23: pNet for the toplevel composite component

state of our model is left undefined.

The `Toplevel HM` *composite component* has no need for membrane components, with an argument similar to the*Local HM* case: all requests on the service interface, including bind/unbind requests, are simply forwarded to the *Global HM* sub-component. The interesting part here is the dynamic multicast bindings between the *Global* and *Local HM* components[9]. All synchronisation vectors between the *!Q_ config* and *!Q_ traceLvl* ports of the *Global HM* and the corresponding input ports of the *Local HMs* are guarded by a predicate testing the value of the group parameter. Each of these synchronisation vectors models the synchronous sending of the messages to all currently connected *Local HMs*. The other way round, the *!R_ config* messages from each *Local HM* are synchronised with the corresponding port of the *Global HM*, with the index of the emitting group member inserted as explained in rule [C16].

6.2 Discussion

The implementation of this semantics will only be usable in practice if we are able to define significant optimisations on the generated pNet structure. We have sketched some of the main structural optimisations in this section, but the resulting model size is still quite big:
let p be the number of proxies per proxy family, and l be the number of local HMs, then we have $(21+2*p)+l*(10+p)$ basic pNets generated, and $(7+2*l+2*2^l)+(58+8*p)+l*(26+4*p)$ (parameterized) synchronisation vectors. For a typical small instantiation of $l := 3, p := 3$, we get 66 basic nets and 225 parameterized vectors, that is more or less 2 orders of magnitude bigger than the last use-case we published [2].

[9] for the sake of readability, we have omitted the connections for the *!Q_ traceLvl* broadcast messages

In this previous case-study, we showed how to combine 3 main techniques to master the state-explosion of such systems, namely data-abstraction, compositional construction/minimization, and distributed model-checking. Naturally all will be useful here, but we want to highlight the first one that will probably be very sensible here: many of the parameters that occur in the actions of our behaviour rules may be safely abstracted away in specific configurations (like the optimisations we described previously), others will have to be abstracted in a manner depending on the formulas we want to prove. Naturally, we want to validate formally all these optimisations.

7 Related Work

Component-based development has in recent years become an established approach. It shown successes in many application domains such as in distributed and embedded systems. There are several component models that are supplied for building complex systems: Fractal [17], Ptolemy [25], CCM [22, 33, 11], AADL, and GCM [10, 1]. But there are only a few that have a theoretical framework that allows reasoning about modelled systems and verification of their behavioural properties.

Some of the formal developments around components are done with objectives very different from ours. For example, the formalisation of the Fractal model in Alloy [31] brings several interesting properties, but they are mainly related to the model itself more than to the component applications. Similarly, a formal model of GCM has been specified in Isabelle/HOL [28], while it gives an interesting framework to reason on the component model and prove some of its properties, it is not adapted to prove the correct behaviour of applications. A formal framework for reasoning on futures has been defined in [23], but the authors did not provide, to our knowledge, the tools to use their equations in order to automatically or semi-automatically prove properties on programs. Behavioural specification, on the contrary, is better adapted to the correctness proofs for given applications. Among the researches dedicated to the component oriented behavioural verification that we are aware of, the closest are SOFA, Kmelia, STSLib, and BIP.

The SOFA system [19] is a development and verification framework for large-scale distributed software systems based on hierarchical components. It uses *behaviour protocols* [34] to specify interactions between components in terms of ordering of method invocation events. Behaviour protocols are also used, at each level of the component hierarchy, to define a black blox specification of the subsystem. The behaviour compliance and consent relations are defined on behaviour protocols based on their trace semantics, allowing to prove separately at each level of the hierachy the compliance of an implementation (called architecture) with its specification (protocol). Behavior protocols can also be encoded e.g. in Promela, that allows for classical LTL model-checking [30].

Kmelia [3, 5, 4] is a component specification model based on the description of complex services. Kmelia and its toolbox COSTO can be used to model software architectures and their properties, these models being later refined to execution platforms. It can also be used as a common model for studying component or service model properties (abstraction, interoperability, composability), using various verification toolsets, including CADP, MEC5, and Atelier-B. To our knowledge, though, there is no explicit behavioural semantics defined for Kmelia applications.

The STSLib library [26] provides a formal component framework that synthesizes components from symbolic protocols in terms of Symbolic Transition Systems (STS). Just as pNets, STS concisely represent infinite systems, however STS rely on Abstract Data Types (ADT) which are more expressive than the Simple Types used in pNets but less intuitive for software engineers. Both formalisms rely on (N-ary) synchronization vectors, but in STS they are static whereas

in pNets they are dynamic. STSLib synthesizes components based on their STS protocols; a controller interprets the STS protocol and data from which the ADT is implemented (and generated) in Java. The communication in STS components is rather low level ; both emitter and receiver must agree to exchange a message, and there is no explicit notion of required nor provided services.

On the implementation side, the two approaches are quite different: the implementation of STS simulates the synchronisation vectors that can be expressed in the specification, whereas in our approach, we write only the synchronisation vectors corresponding to the possible communications between components. Our specification language is more independent from the middleware, and it allows us to express complex synchronisations. This allows us to reason on efficient, expressive, and proved communication mechanisms. Overall, even if the pNet formalism is approximately at the same level of abstraction as STS, in our approach, the programmer is rather exposed to a higher-level composition framework, closer to his usual programming and composition concerns.

BIP [12, 7] is a formal framework that allows building and analysing complex component-based systems, both synchronous (reactive) or asynchronous (distributed) by coordinating the behavior of a set of primitive and heterogeneous components. A component's behaviour is described as a Petri net extended with data and functions, whereas coordination is described as interactions between components and scheduling policies between interactions. Even if the BIP framework allows powerful compositional reasoning on the system, it does not support the definition of parameterised components, nor does it allow explicit data transfer between components.

BIP is supported by a toolset including translators from various programming languages as Lustre and C into BIP, a compiler for generating code executable by a dedicated engine, and the verification tool D-Finder. This last tool ([13]) is not a generic model-checker, but a specific tool for deadlock detection and diagnosis, allowing to address systems of large size, as shown e.g. in [8, 14].

8 Conclusion

This article provides a formal framework for the generation of behavioural semantics of asynchronous distributed software components. Asynchronous software components provide a convenient programming abstraction for designing large-scale distributed systems, where each component acts as an autonomous entity, only communicating with the others by asynchronous communications. Behavioural semantics enable the generation of a model of the program behaviour; then its correctness can be verified using dedicated platforms, for example based on model-checking techniques. The main contributions of this article are:

- a minimal formal definition of pNets. This definition does not cover all aspects previously published, but constitutes a simpler core definition, self-contained and sufficient for this article.
- a formal definition of GCM components, their abstract syntax, and a well-formedness criteria, together with an informal description of their semantics,
- a precise formal definition of the behavioural semantics of GCM, in the form of structural rules constructing pNet models,
- an illustrative example sketching ideas for using these rules in practice.

Among these contributions, the main part concerns the behavioural specification of the most important Fractal/GCM features. For some of these, we have already described a behavioural semantics in previous conference papers, but only in an informal way. Here we have given a full definition, and a procedure for building their pNet models. This includes the basic structure for GCM components, request queues, body expressing the service policy, future proxies and proxy managers; it also defines the semantic artifacts needed for first class futures, and for group communication over multicast interfaces, including management of group reconfigurations.

The example described in the last section is extracted from a new (yet unpublished) industrial use-case, and features all the aspects listed above, except for first class futures. We also use this example to comment on the possible implementation of our semantic model generation, and to discuss a number of optimisation rules that would be important to take into account.

Naturally enough, we are currently working on a full implementation of this semantics in our VerCors specification and verification platform [21]. This effort is complemented by the case-study mentioned in the above paragraph, and will be clearly a difficult challenge in terms of state-explosion, and may be several orders of magnitude more complex than previous studies [2]. We also plan to extend this work in several directions. First there are still some GCM features that are not included in the current core semantics, and that are important in pratical applications. This is the case of Gathercast interfaces (see [9]), and the various policies for managing parameters distribution and results gathering. And there is much more to do about the reconfiguration features of GCM, either considering explicit reconfiguration scripts, or autonomic components.

But the pNets approach can easily be applied to other types of distributed languages or formalisms, in the way we have dealt with Active objects, Fractal components, and GCM. It encodes in a very flexible and expressive way any kind of process composition, communication and synchronisation, provided it stays within the family of first-order hierarchical structures. This rules out formalisms such as the Π-calculus or the chemical machine, but includes most component models (CCM, SCA, Creol, ...) and their implementations.

In a completely different direction, we plan to relate the behavioural semantics of this article with the more "operational" one defined in [9]. We have used the latter in our research on theorem-prover based formalisations and proofs for GCM, and relating the two semantics would give us powerful tools for dealing with properties of dynamicaly evolving applications, and in the longer term to combine model-checking and theorem proving methods to reason about realistic applications.

References

[1] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, and C. Zoccolo. Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components. In *Joint Workshop on HPC Grid programming Environments and COmponents and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15*, June 2006.

[2] R. Ameur-Boulifa, R. Halalai, L. Henrio, and E. Madelaine. Verifying safety of fault-tolerant distributed components. In *International Workshop on Formal Aspects of Component Software (FACS'11)*, Oslo, Sept 2011.

[3] P. André, G. Ardourel, and C. Attiogbé. Adaptation for hierarchical components and services. *Electron. Notes Theor. Comput. Sci.*, 189:5–20, 2007.

[4] P. André, G. Ardourel, and C. Attiogbé. Composing Components with Shared Services in the Kmelia Model. In *7th International Symposium on Software Composition, SC'08*, volume 4954 of *LNCS*. Springer, 2008.

[5] C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition (ETAPS/SC'06*, volume 4089 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[6] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed fractal components. *Annals of Télécommunications*, 64(1-2):25–43, 2009.

[7] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the bip framework. *IEEE Softw.*, 28(3):41–48, May 2011.

[8] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Felix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 631–635. IOS Press, 2008.

[9] 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. *Annals of Telecommunications*, 64(1):5–24, 2009.

[10] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and Ch. Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Télécommunications*, 64(1-2):5–24, 2009.

[11] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, and O. Hurley. SCA service component architecture, assembly model specification. Technical report, OSOA, March 2007.

[12] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3), 2010.

[13] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.

[14] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, June 2010.

[15] R. Ameur Boulifa, L. Henrio, and E. Madelaine. Behavioural models for group communications. In *WCSI-10: International Workshop on Component and Service Interoperability*, Malaga, Spain, 2010.

[16] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12), 2006.

[17] Eric Bruneton, Thierry Coupaye, M. Leclercp, V. Quema, and Jean Bernard Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[18] Eric Bruneton, Thierry Coupaye, and Jean Bernard Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004. `http://fractal.objectweb.org/specification/index.html`.

[19] T. Bureš, P. Hnetynka, and F. Plasil. SOFA 2.0: balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006, IEEE CS*, pages 40–48, Aug 2006.

[20] A. Cansado, L. Henrio, and E. Madelaine. Transparent first-class futures and distributed component. In *International Workshop on Formal Aspects of Component Software (FACS'08)*, Malaga, Sept 2008.

[21] A. Cansado and E. Madelaine. Specification and verification for grid component-based applications: From models to tools. In *FMCO*, pages 180–203, 2008.

[22] CCA forum. The Common Component Architecture (CCA) Forum home page, 2005. http://www.cca-forum.org/.

[23] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

[24] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[25] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[26] F. Fernandes and JC. Royer. The stslib project: Towards a formal component model based on sts. *Electronic Notes in Theoretical Computer Science*, 215:131–149, 2008.

[27] L. Henrio, F. Kammüller, and M. Rivera. An asynchronous distributed component model and its semantics. In F. de Boer, M. Bonsangue, and E. Madelaine, editors, *FMCO'08*, volume 5751 of *LNCS*, pages 159–179. Springer, Heidelberg, 2008.

[28] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. A framework for reasoning on component composition. In *FMCO 2009*, Lecture Notes in Computer Science. Springer, 2010.

[29] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: a types-safe object-oriented model for distributed concurrent systems. *Journal of Theoretical Computer Science*, 365(1 – 2):23 – 66, 2006.

[30] J. Kofron. Checking Software Component Behavior Using Behavior Protocols and Spin. In *proceedings of Applied Computing 2007*, Seoul, Korea, 2007.

[31] Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, 2008.

[32] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[33] Object Management Group, Inc. (OMG). *CORBA Component Model Specification*, omg headquarters edition, April 2006. `http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf`.

[34] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

[35] J. Schafer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

A An operational semantics for pNets

This appendix provides an operational semantics for the pNet model; it is based on a valuation domain for the variables of the pNet, that can be finite, infinite, or even contain new variables.

To give a semantics to pNets, we need a unique *valuation domain* $\mathcal{D}$. This domain can possibly be a countable instantiation domain for each variable. To simplify the semantics, we require that it is possible to decide whether a boolean expression in $\mathcal{D}$ is true, and to decide whether two expressions have the same value (e.g. when two action labels are the same). If we choose a finite domain for each variable and if each pLTS has a finite set of states, the semantics of the pNet will be a finite LTS that can be used in a finite-state model-checker.

We let $\phi = \{x_j \to V_j | j \in J\}$ be a valuation function where x_j range over variables of the considered pNet (each variable must be given a value), and $V_j \in \mathcal{D}$. Such a valuation maintains a mapping from variables to values. For a term $t \in \mathcal{T}_P$, $t\phi \in \mathcal{D}$ is the value of the term obtained by replacing each variable by their values given by ϕ. A valuation can be applied to expressions, actions, or even indexed sets. In all cases, the variables are replaced by their value and the new expressions are evaluated. The set of valuation functions, Φ, allows the precise definition of the state-space to be considered: only valuation functions such that $\phi \in \Phi$ are considered. We define an update operator $+$ on valuations, where $\phi_1 + \phi_2$ replaces some of the values defined in ϕ_1 by the ones in ϕ_2; ϕ_2 might also define new entries, formally:

$$\{x_j \to V_j | j \in J\} + \{x'_j \to V'_j | j \in J'\} = \{x'_j \to V'_j | j \in J'\} \cup \{x_j \to V_j | j \in J \setminus J'\}$$

Note that variables are used locally to each pNet/pLTS, it is thus possible to use qualified names to avoid collision of variable names in the valuation. To simplify notations, in the semantics we suppose that variable names are unique.

Consider a pNet *pNet* and an initial valuation $\phi_0 \in \Phi$ associating a value to each variable of the pNet. The semantics of *pNet*, starting from a valuation ϕ_0, is given by a LTS (or possibly a pLTS if $\mathcal{D}$ contains variables) where:

- states are hierarchical composition of product states of the sub-pNets, more precisely states are $S(pNet)$ where:

$$\begin{array}{l} S(\langle\!\langle P, S, s_0, L, \to \rangle\!\rangle) = \{(s, \phi) | s \in S \ \wedge \ \ \phi \in \Phi\} \\ S(\langle\!\langle P, L, pNet_i^{i \in I}, SV_k^{k \in K} \rangle\!\rangle) = \{\langle s_i \rangle^{i \in I\phi} | \ \phi \in \Phi \wedge \forall i \in I\phi.\, s_i \in S(pNet_i)\} \\ S(Queue(M)) = \{(M_j\phi_j)^{j \in [1..n]} | n \in \mathbb{N} \ \wedge \ \forall j.\, (M_j \in M \wedge \phi_j \in \Phi)\} \end{array}$$

- labels are $\{\alpha\phi | \alpha \in \mathrm{Sort}(pNet) \ \wedge \ \phi \in \Phi\}$;

- the initial state is the composition of initial states, $S_0(pNets)$ where:

$$\begin{array}{l} S_0(\langle\!\langle P, S, s_0, L, \to \rangle\!\rangle) = (s_0, \phi_0) \\ S_0(\langle\!\langle P, L, pNet_i^{i \in I}, SV_k^{k \in K} \rangle\!\rangle) = \langle S_0(pNet_i) \rangle^{i \in I\phi_0} \\ S_0(Queue(M)) = [\,] \end{array}$$

- and transitions are defined as the $[\![pNet]\!]$, the smallest set of transitions verifying the rules below.

$$\frac{\phi \in \Phi \qquad k \in K\phi \qquad \alpha_j^{j\in J} \to \alpha \in SV_k \qquad \forall j \in J\phi.\, \phi_j \in \Phi \wedge s_j \xrightarrow{\alpha_j \phi_j} s'_j \in [\![pNet_j]\!] \qquad \forall i \in I\phi \setminus J\phi.\, s'_i = s_i}{\langle s_i^{i\in I\phi}\rangle \xrightarrow{\alpha\phi} \langle {s'_i}^{i\in I\phi}\rangle \in [\![\langle\!\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle\!\rangle]\!]}$$

$$\frac{\phi\in\Phi \qquad s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle} s' \in \to \qquad iv(\alpha) = \{x'_i \mid i \in K\} \qquad \forall i \in K.\, V_i \in \mathcal{D} \qquad \phi' = \phi + \{x'_i \to V_i \mid i \in K\} \qquad e_b\phi' = True \qquad \phi'' = \{x_j \to e_j\phi' \mid j \in J\}}{(s,\phi) \xrightarrow{\alpha\phi'} (s', \phi' + \phi'') \in [\![\langle\!\langle P, S, s_0, L, \to\rangle\!\rangle]\!]}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n+1].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{?Q\_M_{n+1}\phi_{n+1}} (M_j\phi_j)^{j\in[1..n+1]} \in [\![Queue(M)]\!]}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{!Serve\_M_1\phi_1} (M_{j+1}\phi_{j+1})^{j\in[1..n-1]} \in [\![Queue(M)]\!]}$$

The most complicated part of the semantics is the way variables are dealt with in the pLTS: only input variables, and assigned variables are allowed to change value in the valuation function. This also applies (indirectly) to the queue where the valuation used in the action is in the *Serve* case constrained by the source state, and unconstrained in the case of a $?Q\_M...$ transition.

B Signature of Behavioural Semantics

This appendix summarises the signatures of the functions computing the behavioural semantics in this paper

Function	Signature	Description	page
$[\![\ ]\!]$	$Component \to pNet$	basic behavioural semantics	18,24
$[\![\ ]\!]^{F1}$	$Component \to pNet$	behavioural semantics for first-class futures	32
$[\![\ ]\!]^{BC}$	$Component \to pNet$	behavioural semantics for reconfigurable components	35,36
$[\![\ ]\!]^{MC}$	$Component \to pNet$	behavioural semantics of components with multicast interfaces	41,43
$[\![\ ]\!]_{service}$	$MethodLabels \times \mathcal{P}(MSignature \times pNet) \to pNet$	Service methods(not specified here)	
$[\![\ ]\!]_{body}$	$\mathcal{P}(MethodLabels) \times \mathcal{P}(MethodLabels) \to pNet$	The body: serves requests in a FIFO order	20
$[\![\ ]\!]_{proxyManager}$	$MethodLabels \to pNet$	Manages future proxies overloaded for multicast interfaces	22 38
$[\![\ ]\!]_{proxy}$	$MethodLabels \to pNet$ $MethodLabels \times GRef \to pNet$	future proxy overloaded for composite components overloaded for multicast interfaces overloaded for first-class futures	22 25 38,42 31
$[\![\ ]\!]_{FutDetect}$	$MethodLabels \to pNet$	pLTS detecting a future received as request parameter	31
$[\![\ ]\!]_{delegate}$	$MethodLabels \to pNet$	Delegation method (in composite components)	25
$[\![\ ]\!]_{BC}$	$CItf \to pNet$	binding controller	35

C Examples of instantiated pLTS

In Section 6 we gave the structure of the pNets hierarchy for the *HyperManager* example. Here we give some of the pLTS corresponding to this case-study.

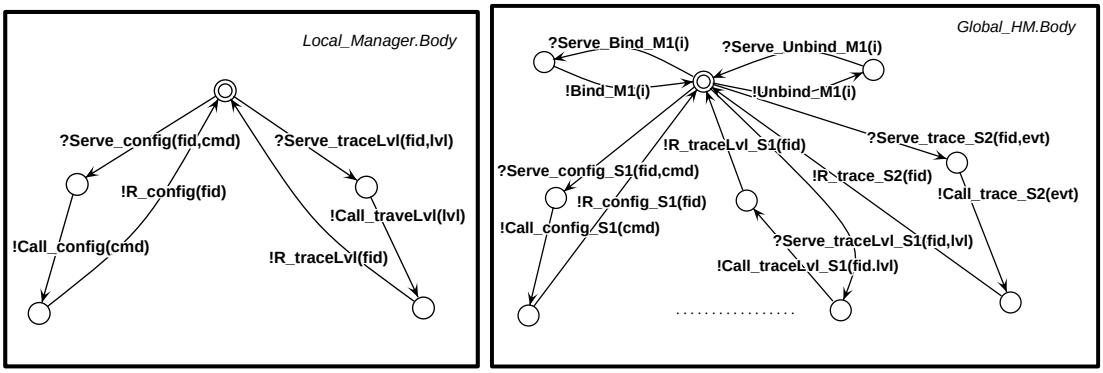


Figure 24: Bodies of the *Local Manager* (full) and *GlobalHM* (partial) components

In Figure 24(left) we have the instantiation of the *Body* of a *Local Manager* component (the one inside the *Local_ HM* composite). It only has two service methods *config* and *traceLvl*, and no non functional interface.

In Figure 24(right) the *Body* LTS is more complex, because the *Global HM* has a total of 4 service methods on its two service interfaces *S1* and *S2*, plus 2 outgoing methods on the multicast interface *M1* and 2 on *C1*. In addition we have two specific loops dealing with the Bind/Unbind requests managing the group at interface *M1*.

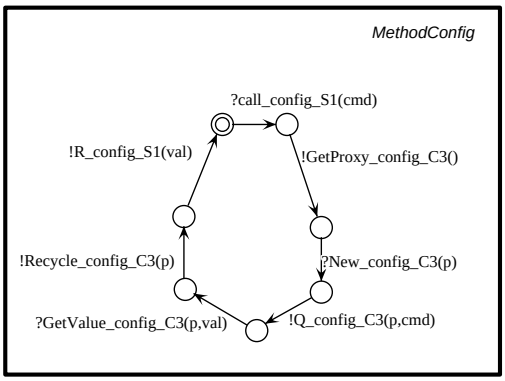


Figure 25: pLTS for the *config* method of the *Local Manager*

In Figure 25 we give an example of a service method behaviour, expressed as a pLTS. This is the *config* method of the *Local Manager* primitive component. It receives a request call on the service interface *S1* with argument *cmd*. Then it sends a similar request to the *Code* component, which is bound on the client interface *C3*; this one is a standard remote request call, requiring the activation of a future proxy: the pLTS here requires a proxy (from the proxy manager), and gets back its value in the *?New_ config_ C3* message, passes the future proxy id, together with the *cmd* argument, in the request *!Q_ config_ C3*, waits for the return of the future value,

signals that the proxy can be recycled, and terminates by replying to the original request with the message *!R_config_S1(val)*.

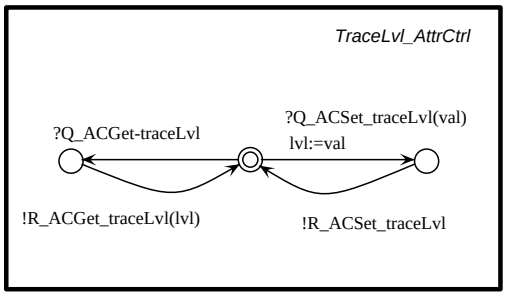
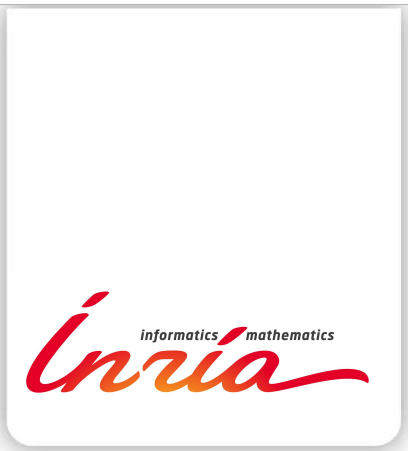


Figure 26: Attribute controller in the *Local Monitor*

Attributes controllers, in Fractal and in GCM, are linked to non-functional interfaces providing an external get/set access to local variables of a component. In Figure 26 we see an example of such a controller, for the variable *traceLvl* of the *Local Monitor*, that can be controlled (*set*) by the *Local Manager*.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399