An Asynchronous Distributed Component Model and Its Semantics

Ludovic Henrio, Florian Kammüller, Marcela Rivera

An Asynchronous Distributed Component Model and its Semantics

Ludovic Henrio[1], Florian Kammüller[2], and Marcela Rivera[1]

[1] INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
{lhenrio,mrivera}@sophia.inria.fr
[2] Institut für Softwaretechnik und Theoretische Informatik – TU-Berlin
flokam@cs.tu-berlin.de

Abstract. This paper is placed in the context of large scale distributed programming, providing a programming model based on asynchronous components. It focuses on the semantics of asynchronous invocations and component synchronisation. Our model is precise enough to enable the specification of a formal semantics. A variant of this model has been implemented, together with tools for managing components.
This paper explains why we consider that our component model is efficient and provides a convenient programming model. We show how futures play a major role for such asynchronous components, and provide a reduction semantics for the component model. This reduction semantics has been specified in the Isabelle theorem prover, and will be used to prove properties on the component model and its implementations.

1 Introduction

Component models provide a structured programming paradigm, and ensure a better re-usability of programs. Indeed application dependencies are defined together with provided functionalities by the means of provided/required ports; this improves the program specification and thus its re-usability. In distributed systems, this takes even more importance as the structure of components can also be used at runtime to discover services or adapt component behaviour. Several effective distributed component models have been specified, developed, and implemented in the last years [1, 2, 3, 4] ensuring different kinds of properties to their users. To be able to prove such properties, one must rely on some well defined semantics for the underlying programming language or middleware. This paper provides such a background for a category of component models.

This work is a study of asynchrony in component models. We present here a model for distributed components. This model is based on one key principle: *Components are the unit of concurrency.* More precisely, components only communicate by sending requests or results for those requests. We say that this model is asynchronous because requests can be treated in an asynchronous manner thanks to the introduction of *futures* (place-holders for request results). In order to prevent other communications or concurrency to occur, we require that *components do not share memory*, which ensures that components really are the

concurrency units. From a computational point of view, components are loosely coupled: the only strong synchronisation consists in waiting for the result of a request, and can be performed only when and where this result is really needed thanks to the use of futures.

Such components can then provide a convenient abstraction for distribution: each component can be placed on a different (virtual) machine. Indeed, the abstractions suggested above imply that each memory location is only accessible by one component, and thus it is easy to place each component on a different independent location. This makes our component model adapted to distribution.

This component model is closely related to the Grid Component Model (GCM). Indeed, this work can be considered as the GCM model, where communication is chosen to be a request / reply mechanism with futures. ProActive/GCM is a reference implementation of the GCM. Our objective is to provide a programming model more general than the one adopted in ProActive/GCM, but more precise than the strict GCM definition. Indeed, GCM provides a structural description of components. From this definition, we precise component composition and communication semantics; more precisely we define composite component behaviour and an asynchronous communication mechanism using futures. ProActive/GCM can also be considered as *a possible* implementation of our model where components are implemented as active objects. Our definition of components being both precise and formalised, we expect it to be a strong guide and a reliable basis for both component system implementation and formal tools.

Our components are loosely coupled, with a data-flow oriented synchronisation. While being a very convenient way of parallelising computations, loose coupling can raise issues when one wants to synchronise the management of several components. We will show in this paper some of the issues that can arise for synchronising the management of components, and some possible solutions.

We first detail the component model we suggest and explain why we think the proposed constructs are efficient (Section 2). After this introduction of our major concepts, we offer a comparison by summarising the main component models found in the literature (Section 3). Next, we introduce our formal model of asynchronous components (Section 4), and we present several implementation and component management issues (Section 5). Finally, we conclude this paper.

2 An Asynchronous Component Model

The GCM [4] is a component model defined by the European Network of Excellence CoreGrid. It extends the Fractal component model, by addressing Grid computing: it supports deployment, scalability, autonomic behaviour, and asynchronous communications. The GCM relies on the following aspects:

– *Fractal as the basis for the component architecture:* the main characteristics GCM benefits from Fractal are its hierarchical structure, the enforcement of separation between functional and non-functional concerns, its extensibility, and the separation between interfaces and implementation.

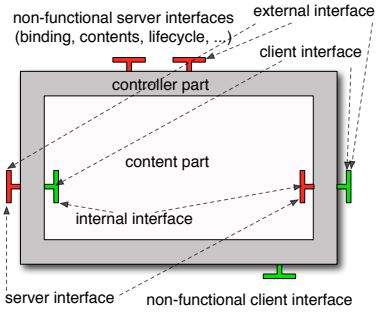

Fig. 1. A GCM component

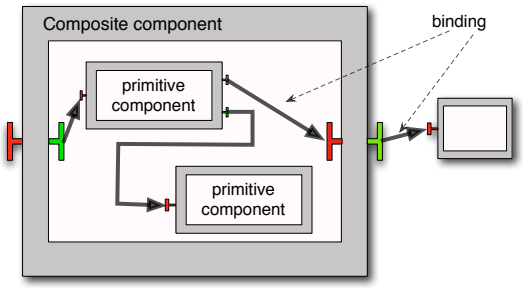

Fig. 2. A component system

– Communication Semantics: GCM components should allow for any kind of communication semantics (e.g., streaming, file transfer, event-based) either synchronous or asynchronous. However, for dealing with latency, asynchronous communication is preferred, and considered as the default.
– GCM supports collective communications – one-to-many, many-to-one, but also many-to-many.
– GCM also comes with a support for autonomic aspects and better separation of concerns (functional vs. non-functional).

In this section, we will first recall precisely the component structure of GCM components; then we will refine this model to define the semantics of our asynchronous components.

2.1 Component Structure

Let us start by the structure of the component model that is inherited from Fractal [5]. GCM is a hierarchical and reflective component model. A GCM component can be either *composite* (i.e. composed of subcomponents), or *primitive* (a basic element encapsulating the business code). A component comprises a content (providing the functional code) and a membrane (a container managing non-functional operations).

The interfaces are the access points to components. The components have *client interfaces* – emitting messages/invocations– and *server interfaces* – able to receive messages/invocations. A *binding* connects a client interface to a server interface (shown in Figure 2), with the implicit semantics that the message emitted by the client will be received by the server interface. Each client interface is bound to a single server interface. For composite components, if the interface is exposed to subcomponents this is an *internal* interface. If, on the contrary, the interface is exposed to other components this interface is an *external* interface. All the external interfaces of a component as well as its internal interfaces must have distinct names. Depending on its functionality, each interface is either functional or non-functional. Each internal functional interface must have a corresponding external interface of the same name. The implicit semantics is that

a call received on a server external (resp. internal) interface will be transmitted – unchanged – to the corresponding internal (resp. external) client interface. Among those notions, only non-functional client interfaces have been introduced in GCM compared to Fractal. A GCM component architecture can be described using an architecture description language (ADL).

A GCM component and its different parts are shown in Figure 1. Functional interfaces are shown horizontally, and non-functional ones vertically. Client interfaces are on the right (or bottom) and server on the left (or top). Note that each external functional interface has a corresponding internal one, whereas non-functional interfaces may not have any corresponding internal ones in case non-functional requests are treated by the membrane. Figure 2 shows a component assembly composed of two main components, the left one is a composite composed of two primitives; the figure also illustrates all the kind of bindings that can be encountered in a GCM component assembly.

Adaptation mechanisms are triggered by the control part of the components; we call this part *non-functional* (NF). This NF part, named *membrane*, is composed of controllers that implement NF concerns. The membrane is a set of (controller) components that can be (re)configured. These controllers can manage configurations and reconfigurations. Compared to Fractal, GCM gives a component structure to the membrane; moreover in GCM controllers inside the membrane can interact with the membranes of other components through bindings between NF interfaces.

Interface cardinality The interface cardinality indicates how many bindings can be made from or to this interface. We have three kinds of cardinalities: singleton, collection, and collective. Collection interfaces were defined in Fractal to let an interface be instantiated as many times as necessary. GCM defines collective interfaces: multicast (one-to-many) and gathercast (many-to-one).

A *multicast* interface is a client interface that transforms a single invocation into a list of invocations, forwarded in parallel to a set of connected interfaces. The result of an invocation on a multicast interface is a list of results. Invocation parameters can be distributed according to a distribution policy: for example, *broadcast* sends the same parameter to each of the connected server interfaces; and *scatter* strips the parameter so that the bound components work on different data. Distribution policy can also be customized.

Symmetrically, a *gathercast* interface is a server interface that synchronises a set of invocations toward the same destination. A gathercast interface coordinates a set of incoming invocations before continuing the invocation flow, forwarding a single invocation. This interface may define synchronisation barriers and may gather incoming data.

Formalising collection and collective interfaces is outside the scope of this paper and we will focus on singleton interfaces. Singleton cardinality is to our mind sufficient to express the crucial points of asynchrony, and many-to-many communications can be studied as an extension to this work.

2.2 Informal Semantics

We focus now on the semantics of our component model; for this we make a few additional assumptions compared to the GCM component model. First of all, we start from the point of view presented in the introduction: "components are the unit of concurrency" and components do not share memory. This way interaction between components is limited to communication.

Communication 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.[3] This communication semantics is similar to requests in an active object model like ASP [6], but also to communication in Actors [7], where messages are enqueued in the message delivery system of the destination.

We call *requests* the messages that are transmitted between components, and that can contain parameters also transmitted (copied) between components.

References to components cannot be passed between components, for example, method parameters cannot contain references to components. More precisely, in order to allow non-functional features to be aware of component structure and manage the component system, we restrict component manipulations to non-functional concerns.

Returning results We call our component model asynchronous because communication does not trigger computation on the receiver side immediately, it just enqueues a request. Such a mechanism can be implemented with synchronous or asynchronous communications. As in ASP and ProActive, the model defined in Section 4 relies on a rendez-vous (enqueueing a request is done synchronously but the receiver component is always ready to enqueue a request). Asynchronous invocations could be performed by enforcing request results to be returned by an explicit call-back mechanism, but we prefer handling results automatically in order to prevent business code from dealing with communication purposes.

To allow for transparent asynchronous requests with results, we use *futures*, first introduced in [8, 9]. A future is an empty object that represents the result of a computation and will be updated when the result is available. In our case, futures are a transparent and natural way to handle asynchronous requests: a future is automatically created when sending a request from a component to another, it represents the result of this request. Transparent futures come with a natural and automatic synchronisation called *wait-by-necessity*: futures can be safely transmitted between components or stored while the real value of the result is not needed. When the value is really needed the thread accessing the future is automatically blocked until the result is available.

[3] To be precise, only futures are passed by reference, because their value will be finally transmitted by a copy semantics.

Transmitting a future between components is not considered as an operation requiring the value. Consequently, the result or the parameters of a request can contain a future, or even can simply be a future. Consequently, several components in the system may have a reference to the same future, the component platform will then be in charge of updating all those references. *Updating* a future consists in replacing a future reference by the result for the corresponding request. We call those futures *first-class* because they can be transmitted between components as any other value.

Primitive component behaviour Let us now detail a behaviour for primitive components that will ensure asynchronous communications and future handling.

The primitive components encapsulate the business code, thus in our model we consider they can have, internally, any behaviour. They will serve requests in the order they wish, providing answer for all the requests they receive. They can call other components by emitting a request on one of the client interface. However, each primitive component must always be able to accept a request (that will just be enqueued in its request queue), and to receive a result (that will replace a future reference by the received value).

Figure 3 illustrates a primitive component and its behaviour. A primitive component consists of a request queue, a content, a membrane, and a result list. Its content contains the business code that serves the requests; requests arrive from the server interfaces on the left and are emitted by the client interface on the right. An incoming request is enqueued immediately, associated with its future identifier. Later this request is served and treated by the component content, possibly emitting new requests to the clients. When the service is finished and a value is calculated for its result, this value is stored in the result list, stating that the future for the request is mapped to this calculated value. The calculated value can itself contain references to other futures. Later, the result will be sent from the result list to the components that hold a reference to the corresponding future. As future references can spread in all the components, including requests, results, and current component states, received results are used to update future references in all parts of the component.

Mono-threaded components In our model, a given thread manipulates a single component, but nothing prevents our components from being multi-threaded. Even, a component can serve several requests at the same time.

However, like in ProActive/GCM, components can be chosen to be mono-threaded; this simplifies concurrency, as each component has a sequential behaviour but can create deadlocks. For example, if there is a cycle of dependencies between results of requests: in a subsystem with two components, `C1` and `C2`, a request `A`, computed by `C1` depends on the result of a request `B` to component `C2`, itself depending on the result of another request `C`, but awaited from `C1`. In that case, `C1` will be indirectly waiting for itself, which could only be resolved by a second thread in `C1`. Fortunately, most applications can be written without such cyclic dependencies, especially thanks to first-class futures.

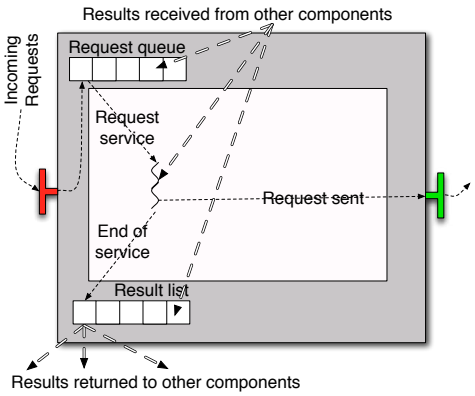

Fig. 3. Component behaviour

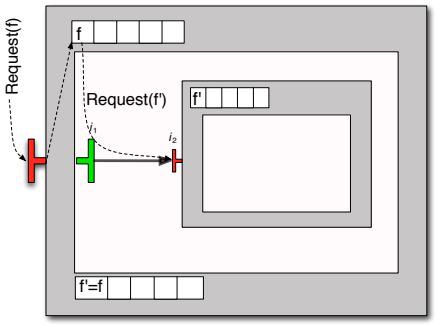

Fig. 4. Request delegation

Composite component behaviour To summarise, the behaviour of the primitive components is highly parameterised. We have just specified the handling of requests and futures in the preceding paragraphs. By contrast, composite components have a predefined behaviour.[4] Composite components serve requests in a FIFO order, delegating request to the bound components or to the external ones in a much transparent and natural way. 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 linked by a binding. More precisely, several bindings may be used and several composite components may be crossed.

This request transmission can rely on a mechanism similar to the handling of requests by primitive components; this mechanism is illustrated by Figure 4. Requests are dequeued in a FIFO order from the request queue. Consider one request (associated with the future f), suppose the request has been received from the outside of the composite, i.e., it was received on an external server interface. There is necessarily an internal client interface matching this external one. Handling the requests consists in sending another request from the internal client interface matching the interface that receives the request ($i1$). This request is sent to the interface bound to $i1$, that is $i2$ in the figure; this interface necessarily belongs to an inner component. This new request corresponds to a future f', and the result for the first one is just a reference to f', i.e. $f = f'$. In case the request was received from the inside of the composite, the mechanism is similar, with a new request sent from the matching external client interface.

An alternative approach would consist in implementing a delegation mechanism, like in allowing a component to delegate the calculation of a result to another component, like handlers of [10]. However, we did not choose this technique in order to avoid introducing a new mechanism, but also to ensure that the component calculating a value for a given future will not change along time.

[4] In the future, we want to study how the behaviour of the composite component could be changed safely but this is not the purpose of this paper.

3 Comparison with some Component Models

This section presents the main distributed component models, focusing on their main characteristics with respect to structure, distribution, and synchronisation. We summarise this comparison in Table 1.

CCA [1] aims at a minimal specification of component architecture for high-performance computing. CCA builds on core concepts, defining a component (the software entity), a framework (the container), and ports (the access towards the environment). The components are assembled at runtime connecting ports together, thanks to scripts that interact with the CCA framework. The container allows building, connecting and running components. Component composition is not hierarchical. CCA considers parallelism and distribution of data.

CCM (CORBA Component Model) [11, 2] is a specification for business components which can be distributed, heterogeneous, and implemented over different programming languages or operating systems. CCM components communicate through ports that can be interconnected. Also, the OMG D&C specification [12] supports hierarchical assemblies. All component instances are handled at runtime by their container. A fortitude of CCM is to provide a clear separation between functional and non-functional concerns.

SCA (Service Component Architecture) [3] provides a component-oriented programming model for building applications and solutions based on a Service Oriented Architectures. SCA provides a model for both the composition of services and the creation of service components, including the reuse of existing application functions within SCA composites. SCA is a hierarchical component model, but its component structure is not specified at runtime. Additionally, SCA components can be implemented with different languages such as Java, BPEL, and state machines.

ASP [6, 13] is the computation model behind ProActive. This calculus of active objects starts from $\varsigma_{\mathbf{imp}}$-calculus [14] extending it with explicit commands for activating an object and for serving requests. Activities contain an active object, possibly several passive objects, and are managed by only one thread. Communication is realized using an asynchronous request reply mechanism with futures. A request is associated to each future, and the request service aims at providing a result value to the future. Using a translational semantics, ASP components are defined as hierarchical combinations of activities.

Creol [15, 16] is a programming and modelling language for distributed systems based on active objects communicating via asynchronous method calls using futures. Creol's base language is – at least in more recent publications, e.g. [17] – an extended version of the functional ς-calculus [14]. Besides explicit distinction between fields and methods of objects the authors introduce classes and threads as first class citizens. Classes contain implementation of methods; threads are sequences of method calls referenced by futures; objects' fields contain the values resulting from thread evaluation. Concurrent access to objects is controlled by an explicit lock mechanism. The operational semantics is a reduction style structural operational semantics based on rewriting logic implemented in Maude which enables testing of model specifications [18]. Interfaces are integral part of

the Creol language. They describe the observable behaviour of objects using assumption guarantee specifications [19]. Traces of communication events between the object and its environment specify input and output behaviour based on visible parts of an object's features. More recently, [17] defines Creol components, and a framework for describing and testing them. The authors use a simple specification language over communication labels enabling the expression of component behaviour as a set of traces at the interfaces.

A Creol component is a collection of classes, objects, and threads where the threads are simply composed in parallel. Thus, components are not hierarchical. Threads – or sets of threads – define concurrent components. A thread never leaves the object in which it is defined. Thus objects are the unit of concurrency. Distribution is not given by explicit locations but using independent object evaluation and asynchronous method call invocations. In comparison to Creol, our approach is hierarchical. We use a separate level of component specification with an abstract behaviour model. We separate the structural component level from the program semantics.

FOCUS [20] is a framework for the systematic formal specification and development of distributed components communicating by asynchronous messages. Contrarily to other models, in this framework the basic notion is the stream. There are two types of streams: streams of actions (traces) and streams of messages. Streams of messages are used to represent *communication histories* of channels. The behaviour of a component is described by logical formulas specifying stream processing functions. Compared to this approach, our formalization focuses on components that could be imperative and can have a much richer behaviour, more difficult to specify, but more expressive. We expect to be able to prove automatically properties on component composition and component behaviour.

The Relational Calculus of Object and Component Systems (rCOS) [21] is based on the Unifying Theory of Programming by He and Hoare supporting concurrency and relational refinement. In the rCOS component model [22] components are aggregations of objects; it uses required and provided interfaces together with contracts. rCOS has a rich and fine-grained object model but lacks – in comparison to our approach – the variety of hierarchical composition at the component level and consequently the explicitness of component interaction.

Fractal [5] and GCM [4] were presented in the preceding section. Let us simply recall their main differences. Contrary to Fractal, GCM specifies distribution aspects of the component model, and defines one-to-many and many-to-one communication, which are particularly efficient for distributed components. The GCM model also refines the structure of the membrane, and defines some controllers for autonomic behaviour.

A GCM reference implementation is based on ProActive [23]. In this implementation each component and each composite membrane is an active object. The controllers are encapsulated in the membrane which also dispatches functional calls to inner components. In this implementation, components communicate through asynchronous method calls with futures. Futures can be for-

warded to any component in a non-blocking manner. A property inherent to this implementation is the absence of shared memory between components, this leads to constraints but also greatly simplifies the reasoning about concurrency. The primitive components act as the unit of distribution and concurrency (each thread is isolated in a component).

Component Model	**Hierarchy**	**Distribution Unit**	**Concurrency**	**Communication**
CCA	no	Application dependent	Unspecified	Synchronous
CCM	yes	Application dependent	Unspecified	Synchronous or Asynchronous
SCA	yes	Unspecified	Unspecified	Call-and-return messages
ASP-component	yes	Active Object	Monothreaded Active Objects	Asynchronous, implicit futures
Creol	no	Object	Multi-threaded Active Objects	Asynchronous, explicit futures
Fractal(Julia)	yes	Unspecified	Multi-threaded Components	Synchronous
GCM	yes	Primitive Component	Unspecified	Request-Reply Paradigm
GCM (ProActive)	yes	Primitive Component	Unique control thread per component	Asynchronous, implicit futures

Table 1. Comparison of component models

In order to formalise Fractal components, several models and calculi have been designed, addressing different aspects. The Kell-calculus was introduced as a very general calculus able to represent component containment, control and passivation [24]. Then, this work was extended and adapted in order to deal with shared components [25]. In Fractal, a component is shared if it is the subcomponent of several different composite components. The formalism we present does not deal with component sharing. More recently, the Fractal component model has been formalised in Alloy [26]. This paper gives a very precise and unambiguous formalisation of Fractal component's structure and control. Compared to this framework, our work focuses on the asynchronous aspects of components, and somehow takes the decision of giving a less general semantics to components and component communications in order to provide a formal model of the interplay between component communication, component behaviour, their control and their structure.

Amongst the formal models for distributed computing, our work relies on the notion of futures and requests that have already been formalised, outside the context of components, see for example [27] in the context of Creol or [10] in the context of functional programming.

4 Formal Model

This section defines a semantics for our component model. It is being formalised in Isabelle/HOL [28].[5] This explains some design choices made here.

4.1 Structure and Notations

We let v_j, p_j range over values, f_j range over futures, i_j range over interfaces, N range over component names, and C over components. A list is denoted $[a_i]^{i\in 1..n}$. The operator # is the list append operation. $l \setminus f$ removes f from the list l, whatever its position is.

Component definition We build requests as triples (future identifier, parameter value, invoked interface): $R_j ::= [f_j, v_j, i_j]$. A result maps a value to an identified future: $F_j ::= [f_j, v_j]$. A component is either a primitive or a composite, each one has a state and a set of interfaces, a composite has additionally bindings and subcomponents (*subCp*): Prim[*itfs*, *PrimState*], Comp[*itfs*, *subCp*, *bindings*, *CompState*]. Enqueue(C, R) enqueues a request R in the request queue of the component C.

States Each state (*PrimState*, *CompState*) is a record containing a *queue*, and a list of computed results (*results*); additionally a primitive component state (*PrimState*) contains an internal state (*intState*), and a behaviour (*behaviour*). A behaviour is a labelled transition system between internal states where labels are actions defined below. An internal state contains a set of current requests (*currRq*), and a state referencing a set of futures.

s.queue returns the current queue of state s. The constituents of a state s, e.g. its queue, can be updated individually, for example $s(\!|queue := Q|\!)$ denotes a new state obtained by changing the queue of s to Q.

Subcomponents The set of subcomponents of a composite is a mapping from component names to components: $SubCp ::= [N \mapsto C]^{i\in 1..n}$. The subcomponent named N of the composite component Comp[*itfs*, *subCp*, *bindings*, *CompState*] is denoted *subCp*[N], and *subCp*[$N \mapsto C$] denotes a new set of subcomponents where C is the new component associated to the name N.

Bindings Each binding is of the form $[N.i_1, N'.i_2]$, if interface i_1 of component named N is plugged to the interface i_2 of N' (where N and N' can be *This* if the plugged interface is the composite component that defines the bindings).

Futures For any value, state, or component, futs(v) (resp. futs(s), futs(C)) represents the set of futures referenced by this element. We use a function *UpdFut* that is applied to values; *UpdFut*(v_i, f, v) replaces the future f – if present – in value v_i by v. Note that futs(*UpdFut*(v_i, f, v)) $\subseteq$ futs(v) $\cup$ futs(v_i) $\setminus \{f\}$ ($\setminus$ is the set subtraction). findRes(S, f) looks inside a component system S and returns the value which is the result corresponding to future f, if it is already computed.

[5] Prototype specification available at `www.inria.fr/oasis/Ludovic.Henrio/misc`

4.2 Local Actions

The behaviour of the primitive components is greatly customisable. For the purpose of the component model, we suppose this behaviour is specified by an (infinite) labelled transition system, it is denoted by $\mathcal{B}_C$ for a given primitive C. The actions of the primitive components are the labels of the transitions, and states are those of the primitive component. Actions of interest are the following:

NewService itfs p f dequeues a request on an interface of the set $itfs$ and starts serving it; f receives the future identifier and p the request parameter.
Tau is a non-observable action allowing to encapsulate internal behaviour.
Call i p f sends a request on interface i with parameter p; f receives the future identifier that corresponds to the request. i must be one of the client interfaces of the primitive component.
EndService f v finishes a service associating value v to future f; this action adds a new entry in the result list.
ReceiveResult f v receives a result value: future f is updated with value v. A primitive component must always be able to receive a future (if $f \notin \text{futs}(s)$ this action has no effect):

$$(\forall f, s, v.\, \exists s'.\, (s, \mathit{ReceiveResult}\ f\ v, s') \in \mathcal{B}_C)$$

Constraints on current requests The set *currRq* of requests currently handled by the primitive component changes only when a request is served (one current request added), or a service is finished (one current request removed). Additionally, one can only finish a service for a request that is current; this leads to the following constraints:

$$(s, \mathit{NewService}\ itfs\ p\ f, s') \in \mathcal{B}_C) \Rightarrow s'.\mathit{currRq} = f \# s.\mathit{currRq}$$
$$(s, \mathit{EndService}\ f\ v, s') \Rightarrow (f \in s.\mathit{currRq} \wedge s'.\mathit{currRq} = s.\mathit{currRq} \setminus f)$$

For all the other actions we have $(s, \mathit{action}, s') \in \mathcal{B}_C \Rightarrow s'.\mathit{currRq} = s.\mathit{currRq}$.

Constraints on referenced futures Futures referenced by the internal state of a primitive component are also constrained. In general $(s, \mathit{action}, s') \in \mathcal{B}_C$ implies $\text{futs}(s') \subseteq \text{futs}(s)$, except when a new request is served or a result is received. In those cases, the request parameter or the result may contain new futures. Additionally, when a result is received, the future updated should not be referenced any more.

$$(s, \mathit{NewService}\ itfs\ p\ f, s') \in \mathcal{B}_C) \Rightarrow \text{futs}(s') \subseteq \text{futs}(s) \cup \text{futs}(p)$$
$$(s, \mathit{ReceiveResult}\ f\ v, s') \in \mathcal{B}_C) \wedge f \in \text{futs}(s) \Rightarrow \text{futs}(s') \subseteq (\text{futs}(s) \setminus \{f\}) \cup \text{futs}(v)$$

Moreover, sent values can only reference futures known by the internal state:

$$(s, \mathit{Call}\ i\ p\ f, s') \in \mathcal{B}_C) \Rightarrow \text{futs}(p) \subseteq \text{futs}(s)$$
$$(s, \mathit{EndService}\ f\ v, s') \Rightarrow \text{futs}(v) \subseteq \text{futs}(s)$$

Handling received values When an action receives a value, for example, *NewService itfs p f* receives p, the action must accept any value for parameter p and alter the internal state accordingly; p is, in fact, a variable that will, in turn, receive a value from the request queue. Similarly, f will receive the identifier of the future to handle. Instead of introducing variables and scoping, we simply chose to state that some of the parameters must be able to receive any value:

$$(s, \mathit{NewService}\ \mathit{itfs}\ p\ f, s') \in \mathcal{B}_C \Rightarrow \forall p', f'.\, \exists s'.\, (s, \mathit{NewService}\ \mathit{itfs}\ p'\ f', s') \in \mathcal{B}_C$$

This applies also for f in *Call i p f*: the future must be chosen fresh, and v in *ReceiveResult f v*: the received result is given by another component.

4.3 Semantics of the Component Model

The formal semantics of the component model defines a reduction relation $\rightarrow_R$ by a set of inductive rules. $S \vdash C \rightarrow_R C'$ if, in the component system S, the component C can be reduced to the component C'; S is the composite component containing all the components of the system. It is necessary to know the whole component system to retrieve request results and update futures. From $\rightarrow_R$, a reduction for the global component system can then be defined: $S \rightsquigarrow S' \Leftrightarrow S \vdash S \rightarrow_R S'$.

There is a second parameterised relation $\dashv i_1, f, v \mapsto$ allowing to express that a component is willing to emit a request, and must be matched with a reception action; statements of the form $\dashv i_1, f, v \mapsto$ used as hypotheses to the rules for composite components lead back to statements of $\rightarrow_R$. If $\vdash C \dashv i_1, f, v \mapsto C'$, then C emits a request on the interface i_1, with parameter v, and associated to a future f; after the emission, C becomes C'.

There are two kinds of reduction rules: the ones for primitive components (Figure 5), and the ones dealing with composite components (Figure 8).

In detail, the behaviour defined in primitive components determines the following rules of reduction.

- Tau: if the state s of a primitive component Prim$[\mathit{itfs}, s]$ contains a *Tau* transition from the internal state *s.intState* to another state s_2 then the component's internal state can be replaced by s_2. In Figure 3, this rule corresponds to internal transitions inside the content of the composite.
- RcvResultPrim: the primitive component's behaviour always contains a transition defining the reception of value v for future f, i.e. *ReceiveResult f v*, changing the internal state into the result state s_2 defined in the behaviour. The result value is found in the component system S; it is returned by the function findRes(S, f). The future is also updated in the request queue and the result list by the function *UpdFut*. After such a reception, the future f is not referenced anymore by the primitive. In Figure 3, this rule corresponds to the three thick arrows with "results received from other components".
- Call: the call to an interface i_1 with future f and parameter value v presupposes that the future f is fresh. Such a call transition in the behaviour of

$$\textsc{Tau}\quad \frac{(s.intState, Tau, s_2) \in s.behaviour}{S \vdash \mathrm{Prim}[itfs, s] \rightarrow_R \mathrm{Prim}[itfs, s(\!|intState := s_2|\!)]}$$

$$\textsc{RcvResultPrim}\quad \frac{\begin{array}{c}(s.intState, ReceiveResult\ f\ v, s_2) \in s.behaviour \\ \mathrm{findRes}(S, f) = v \qquad s.queue = [f_j, v_j, i_j]^{j\in 1..n} \qquad Q = [f_j, UpdFut(v_j, f, v), i_j]^{j\in 1..n} \\ s.results = [f_k, v_k]^{k\in 1..n'} \qquad R = [f_k, UpdFut(v_k, f, v)]^{k\in 1..n'}\end{array}}{S \vdash \mathrm{Prim}[itfs, s] \rightarrow_R \mathrm{Prim}[itfs, s(\!|intState := s_2, queue := Q, results := R|\!)]}$$

$$\textsc{Call}\quad \frac{(s.intState, Call\ i_1\ v\ f, s_2) \in s.behaviour \qquad f \notin \mathrm{futs}(S)}{\vdash \mathrm{Prim}[itfs, s] \dashv i_1, f, v \mapsto \mathrm{Prim}[itfs, s(\!|intState := s_2|\!)]}$$

$$\textsc{EndService}\quad \frac{(s.intState, EndService\ f\ v, s_2) \in s.behaviour}{S \vdash \mathrm{Prim}[itfs, s] \rightarrow_R \mathrm{Prim}[itfs, s(\!|intState := s_2, results := s.results\#[f, v]|\!)]}$$

$$\textsc{ServeNext}\quad \frac{\begin{array}{c}(s.intState, NewService\ itfs\ v\ f, s_2) \in s.behaviour \\ [f', v', i'] \in Q \Rightarrow i' \notin itfs \qquad s.queue = Q\#[f, v, i]\#Q'\end{array}}{S \vdash \mathrm{Prim}[itfs, s] \rightarrow_R \mathrm{Prim}[itfs, s(\!|intState := s_2, queue := Q\#Q|\!)]}$$

Fig. 5. Primitive Component Semantics

a primitive component creates now a parameterised reduction $\dashv i_1, f, v \mapsto$ of the primitive component because this call is passed on to the enclosing composite component. Upon synchronisation with the component bound to this one, the reduction will occur, modifying the internal state and storing locally the future f. In Figure 3, this rule corresponds to the "request sent" arrow sent to the client interface.

EndService: the end of a service denotes that one of the current requests of a primitive component is finalised yielding value v. Hence, the respective primitive component's result list is extended by the pair $[f, v]$ where f is the future that corresponds to the finalised current request. After reduction the primitive component's current requests does not contain f anymore (see above). In Figure 3, this rule corresponds to the arrow.

ServeNext: finally, a *NewService* $itfs\ v\ f$ transition in a primitive component's behaviour leads to the creation of a new current request in the internal state of the component. The oldest request on the interface i is served. The parameter v matches the parameter of the first request in the request queue, and f, its corresponding future. The reduction updates the internal state by plugging in the target state s_2 of the behaviour's transition, and by popping off the head of the request queue. In Figure 3, this rule corresponds to the "end of service" arrow.

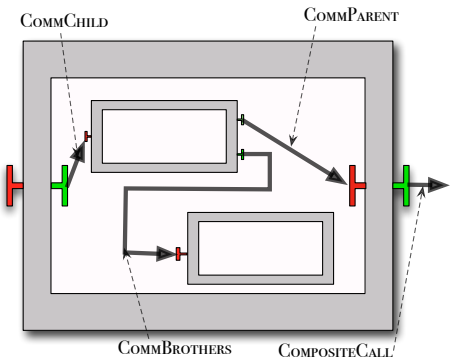

Fig. 6. Component Communications

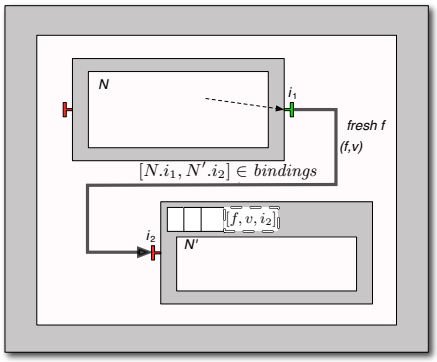

Fig. 7. Subcomponents communicate

The inductive rules for *composite components* determine how the service communication distributes on properly assembled systems. The first rule embeds subcomponent reduction in composite contexts; the second performs future updates inside composite components. The three Comm-rules, in the middle, define the communications transmitted by the different kinds of bindings inside the composite component; finally the last rule allows composite components to emit requests on their external client interfaces. In detail, the rules finalise the formal semantics as follows. Figure 6 illustrates the different kinds of communications expressed by the four last rules.

Hierarchy: this rule is a compositionality rule; it expresses that if a subcomponent $subCp[N]$ reduces in isolation to a component C then it does so as well in the context of a component hierarchy – given by updating $SubCp$ with $SubCp[N \mapsto C]$ in the context $\text{Comp}[itfs, SubCp, bindings, s]$.

RcvResultComp: this rule is very similar to the RcvResultPrim rule for primitive components. However, this one is simpler because the composite component does not have any internal state; only the request queue and the result list are updated by the received result.

CommBrothers, illustrated by Figure 7: a subcomponent N can pass a call to subcomponent N' inside the set of subcomponents $subComp$ of a composite component. The respective client interface of N, on which the call was emitted – $N.i_1$ – must be bound to the interface i_2 of the destination component – $N'.i_2$ – this binding must be stored in *bindings*. The call parameters f, v – parameterised in the request emission relation – are passed to interface i_2 of subcomponent N'. The operator *Enqueue* denotes that the request $[f, v, i_2]$ is properly added onto the request list of subcomponent N'. N is reduced simultaneously, sending a request.

CommParent, illustrated by Figure 9: if a subcomponent – a child – N of a composite component utters a request i_1, f, v to its parent component, then – similar to the previous rule – N is reduced simultaneously as it sends a request, and the request is added to the composite component's request

HIERARCHY

$$\frac{S \vdash subCp[N] \rightarrow_R C}{S \vdash \text{Comp}[itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[itfs, (subCp[N \mapsto C]), bindings, s]}$$

RCVRESULTCOMP

$$\frac{\text{findRes}(S, f) = v \qquad s.queue = [f_j, v_j, i_j]^{j \in 1..n} \qquad Q = [f_j, UpdFut(v_j, f, v), i_j]^{j \in 1..n} \qquad s.results = [f_k, v_k]^{k \in 1..n'} \qquad R = [f_k, UpdFut(v_k, f, v)]^{k \in 1..n'}}{S \vdash \text{Comp}[itf, subCp, bindings, s] \rightarrow_R \text{Comp}[itf, subCp, bindings, s(\!|queue := Q, results := R|\!)]}$$

COMMBROTHERS

$$\frac{[N.i_1, N'.i_2] \in bindings \qquad \vdash subCp[N] \dashv i_1, f, v \mapsto C \qquad SubCp' = subCp[N \mapsto C] \qquad C' = \text{Enqueue}(subCp'[N'], [f, v, i_2])}{S \vdash \text{Comp}[itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[itfs, SubCp'[N' \mapsto C'], bindings, s]}$$

COMMPARENT

$$\frac{[N.i_1, This.i2] \in bindings; \vdash subCp[N] \dashv i_1, f, v \mapsto C}{S \vdash \text{Comp}[itfs, subCp, bindings, s] \rightarrow_R \text{Enqueue}(\text{Comp}[itfs, subCp[N \mapsto C], bindings, s], [f, v, i_2])}$$

COMMCHILD

$$\frac{s.queue = [f, v, i_1]\#Q \qquad [This.i_1, N'.i_2] \in bindings \qquad f' \notin \text{futs}(S) \qquad C' = \text{Enqueue}(subCp[N'], [f', v, i_2]) \qquad s' = s(\!|queue := Q, results := s.results\#[f, f']|\!)}{S \vdash \text{Comp}[itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[itfs, subCp[N' \mapsto C'], bindings, s']}$$

COMPOSITECALL

$$\frac{s.queue = [f, v, i_1]\#Q \qquad i_1 \text{is a client interface} \qquad f' \notin \text{futs}(S) \qquad s' = s(\!|queue := Q, results := s.results\#[f, f']|\!)}{\vdash \text{Comp}[itfs, subCp, bindings, s] \dashv i_1, f', v \mapsto \text{Comp}[itfs, subCp, bindings, s']}$$

Fig. 8. Semantics of the component composition

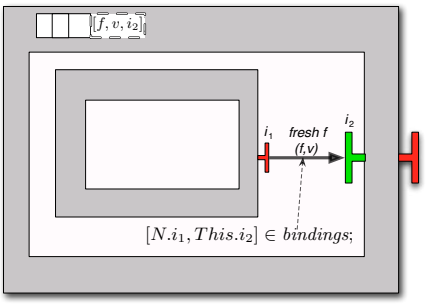

Fig. 9. COMMPARENT rule

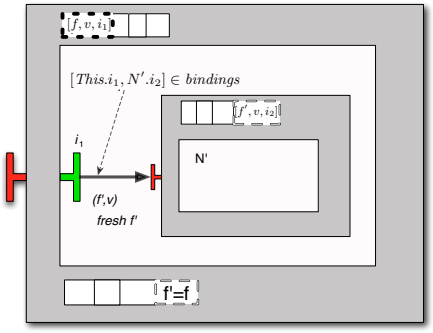

Fig. 10. COMMCHILD rule

queue. The bindings must bind the component N interface to the (inner server) interface of the parent.

CommChild, illustrated in Figure 10: this rule is the inverse case of the preceding one – a component communicates to a child – corresponds to a delegation of a request to subcomponents as shown in Figure 4. The parent component's request queue is reduced by its first element, a new future f' for the result of this request is created and added to the result list of the parent component, and the request – with the new future – is queued into the respective subcomponent. The subcomponent is determined using the bindings: if the original request was on the (external server) interface i_1 and $\mathit{This}.i_1$ is bound to $N.i_2$ then the request will be sent to the interface i_2 of the subcomponent N. The composite component records in its request queue that the result for the future f is in fact the newly created future f'.

CompositeCall: this rule explains how a call received by a subcomponent is emitted on the external client interface onto the context of the enclosing component. This rule corresponds to the Call rule for the primitive components. The first request f, v received on (internal server) interface i_1 is sent on the matching external client interface (with same name). This call will be matched against a Comm rule that will enqueue this request. A fresh future f' is found for this new request and the composite records that the value of f is in fact the future f'.

Figure 11 illustrates a sequence of rules allowing a client component Cli to send, on interface c, a request to the interface s of a component Srv; Srv is encapsulated in a composite component Cmp, thus the request transits by the interface i. The first reduction sends a request from Cli to the composite, then the request is delegated to Srv by the composite, with a new future f' aliased to f. Finally, Cli obtains a direct reference to future f' while Srv starts serving the request. The original configuration is of the following form (for the sake of exposition, we only mention internal states of primitives, and interface descriptions are omitted).

$$\begin{array}{l}\mathrm{Comp}[\emptyset, \mathrm{Cli} \mapsto \mathrm{Prim}[\{c\}, s_0^c],\\ \quad \mathrm{Cmp} \mapsto \mathrm{Comp}[\{i\}, \mathrm{Srv} \mapsto \mathrm{Prim}[\{s\}, s_0^s], \{[\mathit{This}.i, \mathrm{Srv}.s]\}, s_0]\\ \quad \{[\mathrm{Cli}.c, \mathrm{Cmp}.i]\}, s_0']\end{array}$$

5 Tools/Middleware

This section presents component management tools which are necessary to provide adaptation mechanism for distributed components. Proving the correctness of these tools could be a great opportunity for using the formal component model presented in the preceding section. We focus below on two aspects: stopping components, and component reconfiguration; tools for dealing with these two aspects have been implemented in the ProActive/GCM component platform, thus showing already their practical impact.

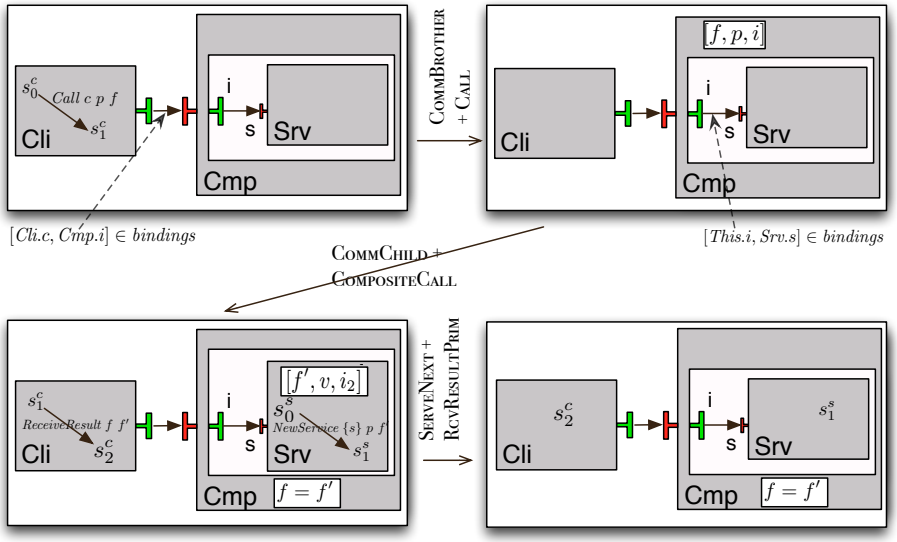

Fig. 11. Example of a Reduction (Hierarchy not mentioned)

5.1 Synchronisation and stop

Fractal component lifecycle proposes a *stop* action, and a stopped state. Existing frameworks for Fractal and GCM sometimes consider recursively stopping a component assembly. However, safely stopping asynchronous components cannot be addressed by stopping procedures proposed so far.

The paper [29] proposes an algorithm for stopping a GCM component system. This algorithm recursively stops a component together with all its subcomponents, and reaches a *safe state* where the component is idle and has no request to serve. This algorithm defines as *master component* the component that receives initially a stop request. The algorithm is split into two phases. In the first phase the master component marks all the requests it sends. This phase lasts until all the requests, for which the master awaits the results, are marked. In the second phase the master component blocks all the requests it receives (except the marked ones) and the inner components continue processing their requests. When all the components are idle, and all inner components have empty request queues, the components are stopped. Let us only focus here on the request marking mechanism. This mechanism is useful to identify re-entrant requests, and to avoid deadlocks involving such requests. Roughly, the algorithm relies on a propagation of marks: each request sent during the service of a marked request is marked too. The master does not propagate marks to its subcomponents; thus only requests outside the master component are marked.

The formal component model presented in this paper allows, for example, the identification of waiting states, of deadlocks, and of re-entrant requests; it will allow us to reason about such requests, and to prove the correctness of

the algorithm sketched above. This model also could verify properties on the algorithm termination and the state of the components when stopped.

5.2 Adaptation and reconfiguration

One of the main purposes of stopping a component system is to be able to reconfigure it in order to adapt it to a different execution context, or to provide new functionalities. Indeed, for safely reconfiguring a component system, a component assembly must be in a state where components are stopped, and considered as easily reconfigurable.

In Fractal and GCM, adaptation is performed by dynamic reconfiguration. For adaptivity purposes, the GCM extends the reconfiguration capabilities of Fractal to the non-functional aspects: the control part of a component can be reconfigured dynamically. Moreover, the GCM specifies interfaces for the autonomic management and adaptation of components. *Autonomicity* is the ability for a component to adapt to situations without relying on the outside. By default, support for some autonomicity concerns are implemented by GCM components with precise non-functional interfaces, but the model being extensible, autonomic behaviour can easily be improved, adapted, and extended.

The formal model presented above enables reasoning on the interplay between the component configuration and the communications. To our mind, it is a crucial tool to prove correctness of autonomic adaptation procedures. Also, in order to ease the development of adaptation procedures, we are developing a scripting reconfiguration language that can be interpreted in a distributed manner, and that can synchronise with communication events or component state.

6 Conclusions

In this paper we presented a model for asynchronous components. Compared to existing component models and language specifications, our work is focused on the interaction between the programming model and the component model. More precisely, we defined the structure of our component model: it relies on the notion of interfaces, separation of non-functional and functional aspects, hierarchy, and bindings transmitting communications. Then, we presented a coherent model for allowing components to communicate asynchronously through a request/reply mechanism, but also through the use of futures. The semantics of our model is flexible enough to allow for multiple implementations and design choices, like multi-threaded versus mono-threaded components, choice of a future update strategy, choice of one of several local programming models, etc. On the contrary the interplay between hierarchy, asynchrony, and communication is quite precisely defined.

The definition of the component model's semantics is precise enough for a formal specification of this semantics to be written, for example in a theorem prover like Isabelle/HOL. We expect this formal specification to allow us to prove properties on component systems, management protocols for those components,

or design choices of the different implementations of the model. Such a framework will provide a consequent step toward safe compositions of components, design of verification frameworks for asynchronous components, and safety of their management procedures.

On the other hand, we have taken care that the component model stays sufficiently abstract to be refined to different execution models. Since our component model abstracts from a concrete execution model it can be instantiated to others. One suitable execution model could be ASP, but also Creol is a candidate that could thereby be extended by hierarchical components. Even in the context of SOA, our model could enable SCA to be extended with a precise semantics for asynchronous communications.

References

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

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

[3] Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O.: SCA service component architecture, assembly model specification. Technical report (March 2007) `www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[4] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: GCM: A Grid Extension to Fractal for Autonomous Distributed Components. Annals of Telecommunications (accepted for publication, 2008)

[5] Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02). (2002)

[6] Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer-Verlag New York, Inc. (2005)

[7] Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1) (1997) 1–72

[8] Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proceedings OOPSLA'86. (November 1986) 258–268 Published as *ACM SIGPLAN Notices*, 21.

[9] Halstead, Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) **7**(4) (1985) 501–538

[10] Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theoretical Computer Science **364**(3) (November 2006) 338–356

[11] omg.org team: CORBA Component Model, V3.0. `http://www.omg.org/technology/documents/formal/components.htm` (2005)

[12] OMG: Deployment and configuration of component-based distributed applications, v4.0. Document formal/2006-04-02 Edition (Apr. 2006)

[13] Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (2004) 123–134

[14] Abadi, M., Cardelli, L.: A Theory of Objects. Springer-Verlag, New York (1996)
[15] Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. In: Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM 2004), IEEE press (September 2004) 188–197
[16] Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a types-safe object-oriented model for distributed concurrent systems. Journal of Theoretical Computer Science **365**(1 – 2) (2006) 23 – 66
[17] Grabe, I., Steffen, M., Torjusen, A.B.: Executable interface specifications for testing asynchronous creol components. Technical Report Research Report No. 375, University Of Oslo (July 2008)
[18] Meseguer, J.: Conditional reqriting logic as a unified model of concurrency. Journal of Theoretical Computer Science **96** (1992) 73 – 155
[19] Jones, C.B.: Development Methods for Computer Programs Including a Notion of Interference. PhD thesis, Oxford University, UK (June 1981)
[20] Broy, M., Dederich, F., Dendorfer, C., Fuchs, M., Gritzner, T., Weber, R.: The design of distributed systems - an introduction to focus. Technical Report TUM-I9202, Technische Univerität München (1992)
[21] He, J., Li, X., Liu, Z.: rcos: A refinement calculus for object systems. Theoretical Computer Science **365**(1–2) (2006) 109–142
[22] Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In Arbab, F., Sirjani, M., eds.: FSEN. Volume 4767 of Lecture Notes in Computer Science., Springer (2007) 191–206
[23] Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science and Technology **12**(1) (2006) 69–77
[24] Schmitt, A., Stefani, J.B.: The kell calculus: A family of higher-order distributed process calculi. Lecture Notes in Computer Science **3267** (Feb 2005) 146 – 178
[25] Hirschkoff, D., Hirschowitz, T., Pous, D., Schmitt, A., Stefani, J.B.: Component-oriented programming with sharing: Containment is not ownership. In Glück, R., Lowry, M.R., eds.: GPCE. Volume 3676 of Lecture Notes in Computer Science., Springer (2005) 389–404
[26] Merle, P., Stefani, J.B.: A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA (2008)
[27] de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In Nicola, R.D., ed.: ESOP. Volume 4421 of Lecture Notes in Computer Science., Springer (2007) 316–330
[28] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer-Verlag (2002)
[29] Henrio, L., Rivera, M.: Stopping safely hierarchical distributed components: application to gcm. In: CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance, New York, NY, USA, ACM (2008) 1–11