Active Objects and Distributed Components: Theory and Implementation

Denis Caromel, Ludovic Henrio, and Eric Madelaine

INRIA Sophia-Antipolis, I3S, Université de Nice Sophia-Antipolis, CNRS
{denis.caromel,ludovic.henrio,eric.madelaine}@sophia.inria.fr

Abstract. To achieve effective distributed components, we rely on an active object model, from which we build asynchronous and distributed components that feature the capacity to exhibit various valuable properties, as confluence and determinism, and for which we can specify the behaviour.

We will emphasise how important it is to rely on a precise and formal programming model, and how practical component systems can benefit from theoretical inputs.

1 Introduction

Component models and frameworks have been in use for some years now. This is especially the case for *distributed components* that attempt to handle the inherent complexity of managing distributed systems. However, underlying languages do not seem to feature a strong and adequate programming model with respect to concurrent and distributed behaviour. The communications between distributed entities often take place with a weak semantics. For instance in Java RMI (Remote Method Invocation), the framework does not specify if the servers are executing the incoming calls in parallel or one after another. In C, C# and Java, the concurrency primitives are very low level, with a recognised difficulty to master the correctness of programs, even at the level of a simple, non-distributed program. When you put the two together, distribution and concurrency, the composition does not hold a clear, easy to grasp, semantics. One has to deal with the complexity of such under-specified features, and the behavioural combinatory explosion that occurs when put together.

Moreover, managing parallel and distributed software is now a basic requirement of any programming language. The slowing down of Moore's law, leading to the advent of multi-core processors, is dramatically increasing the pressure on programmers to introduce parallel decomposition in their applications, leading to both distribution and concurrency. Such solution-domain parallelism amplifies the intricacy of code-level behaviour, leading to even vaguer behaviour. The approach taken here is to limit concurrency to concurrent accesses between remote locations: visible concurrency is limited to the one entailed by distribution. However, at the middleware level, several threads have been introduced to introduce parallelism, with the application still behaving as if the *activity* was both the unit of distribution and of concurrency (each active object runs a single service thread).

F.S. de Boer et al. (Eds.): FMCO 2007, LNCS 5382, pp. 133–152, 2008.
© Springer-Verlag Berlin Heidelberg 2008

These shortcomings prevent us from having a clear semantics at the level of programming the distributed interactions, and in turn preclude from having precise semantics at the component level. When it comes to composite components, composing primitive components made of programming-language code, the semantics issue is even tenser as the imprecision composes into more imprecision. How would it be possible to promote properties at the level of compositions, when we do not have them straight at the inner level? This article advocates the strong need of a simple and sound programming model integrating *distribution, concurrency, and parallelism*, in order to benefit from *soundness and properties* at the level of distributed components.

This paper starts by presenting an active object model featuring asynchronous communications with first-class futures — futures that can be transmitted before having their values. This model is implemented in the ProActive Parallel Suite, available as Open Source within the ObjectWeb Open Source community (http://proactive.ow2.org). An interactive environment, developed as Eclipse plugins, eases the visualisation and control of applications. The next section presents ASP, a generalisation of the ProActive model. Together with a formal semantics, theoretical results on determinacy are detailed. The following section introduces asynchronous distributed components that rely on active objects: primitive components are made of active objects, and the membranes of composite are specified and implemented with active objects as well. An on-going work aiming at defining a joint European component model for Grid computing (GCM) will be summarised. Finally, the paper concludes with challenges at hand with component systems, especially work related to capturing behavioural properties: current work aiming at specifying the architectural and behaviour of components, and guaranteeing their correct behaviour by model-checking methods will be introduced.

Along the course of this article, we would first like to demonstrate how important it is to rely both on practical and theoretical approaches in order to tackle the complexity of today's large-scale distributed systems. The second statement has more to do with a technical orientation: active objects provide a powerful sound foundation for both understanding and programming distributed component systems.

2 Asynchronous Distributed Objects

In order to deal with components, a precise and adequate programming model is needed to adequately build primitive programs to be used as building blocks at composition time. The paper [5] defines ProActive, an object-oriented programming model for concurrent, parallel, and distributed systems.

2.1 Principles

We summarise here the key features of the ProActive programming model:

– *asynchronous calls*, for the sake of hiding latency and decoupling client-server interactions,

- *first-class futures*, for the sake of passing the results of asynchronous calls to other distributed objects without forcing useless synchronisations, also avoiding deadlocks – futures are indeed single assignment variables,
- *wait-by-necessity*, for the sake of using as much as possible data-flow synchronisations of parallel entities,
- *collective synchronisation operations*, for the sake of manipulating synchronisations as first-class entities, e.g., blocking on the availability of all futures in a vector,
- *service primitives*, for the sake of programming in a flexible manner the inner synchronisation of activities,
- *typed asynchronous groups*, for the sake of enabling asynchronous remote method invocations on a group of entities, also a way towards parallel component invocation.

The communication paradigm of ProActive is strongly similar, and somehow inspired by, the actor paradigm [2]. Indeed, active objects communicating by requests and serving them one after the other are similar to actor communicating by messages received in a mailbox and treating them one after the other. More precisely, the active object paradigm can be described as follows. Only active objects can be referenced remotely. A method call on an active object is asynchronous; such a call is stored as a request in a request queue. After a while the active object decides to serve the request[1] and evaluates a result for it. While the result is not computed yet, a *future* [24,31] represents the result of an asynchronous method call. When the result has been computed, it is returned to all the objects holding a reference to the corresponding future.

More recently, programming paradigms relatively similar to ProActive have been developed in different contexts, among them one can distinguish Creol [25,19] and AmbientTalk [20]. Also, X10 [18] can be considered as closed to the ProActive language except that activities in X10 are multi-threaded and X10 does not support futures to our knowledge; whereas ProActive is conceived to have mono-threaded activities which ensures most of its properties and simplifies the programming of active objects.

The features above propose a disciplined way to manage parallelism, and many user operations are achieved in a parallel way without the burden to explicitly build complex synchronisations. Nevertheless, the programming model features a few fundamental properties:

- no interleaving within user code, each primitive component (resp. each active object) is mono-threaded, both concurrency and distribution are the result of the component (resp. active object) composition.
- no sharing of objects between concurrent threads,
- no call-backs, they are replaced by the use of future, which makes programs better structured

[1] Each active object either specifies its service policy using the *Serve* primitive, or uses the default FIFO policy.

Parallelism of operations seems to conflict with the property of not having interleaving within user code. Indeed, parallelism usually leads to interleaving of actions when conducted within a single address space. However, we rely here on the design and implementation of parallel operations within the middleware that have no consequences, whatsoever, for the user. This parallelism is risk-free, intrinsically acting towards confluence, because it does not produce any observable interleaving. Several harmless optimisations are indeed located at various places within ProActive's implementation, e.g., group communications, future updates with automatic continuation. They are increasingly becoming more important with the advent of multi-core processors.

2.2 Environment

The ProActive implementation comes with an environment for deploying, monitoring, and managing distributed applications, based on active objects. For example, Fig. 1 shows a screenshot of IC2D, an application for monitoring the execution of a ProActive application.

Fig. 1. Screenshot: monitoring a distributed application

Fig. 2 shows a screenshot of the ProActive scheduler. The scheduler is an application written in ProActive that is also part of the ProActive environment, and can be used as a tool administrating the deployment and the maintenance of a list of jobs over various platforms and infrastructures (Grid or P2P infrastructure).

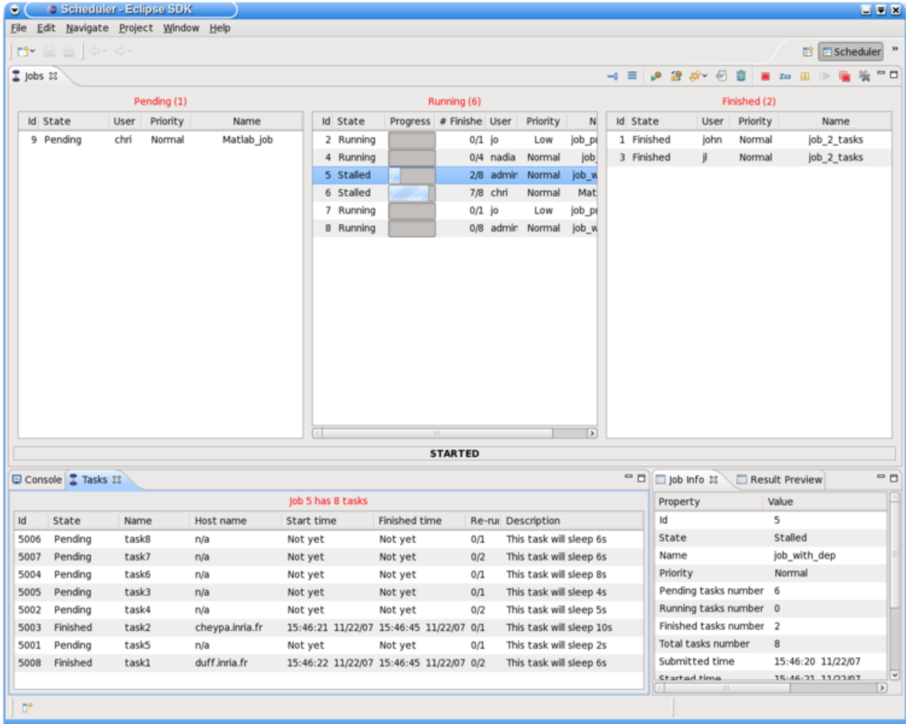

Fig. 2. Screenshot: ProActive scheduler

3 Calculus: Asynchronous Sequential Processes (ASP)

The ASP [17,16] calculus provides a generalisation of the ProActive programming model. It relaxes a few implementation decisions, and provides understanding, and proofs of confluence and determinacy for asynchronous distributed systems. The ASP calculus, is an extension of the **imp**ς-calculus [1,23] with two primitives (*Serve* and *Active*) to deal with distributed objects.

We present here the semantics of ASP in a slightly different version – but equivalent – from our previous publications [17,16]. This new version mainly comes with a more compact syntax. The resulting semantics is more compact than the one presented in our previous publications, but a little further from the implementation concerns. We hope this shorter version will make the semantic rules easier to read. The equivalence between the two versions is trivial, because this new semantics expresses, almost exactly, the same rules on a different syntax.

Concerning related works, futures have been formalised in several settings, generally functional-based [29,19,21]; those developments rely on explicit creation of futures by a thread creation primitive, in a concurrent but not distributed setting. Research on languages ensuring confluence has a long history,

the results which are the closest to the ones on ASP are probably the Process Networks [26] and linear types [27].

3.1 Syntax

We first define a syntax for ASP programs: the terms defined as source code that correspond to the ProActive code are defined in Fig. 3, excluding the underlined terms. Compared to the **imp**ς-calculus we added a parameter to methods; we added a primitive *Active* for creating an activity (i.e., a location containing: an active object, some passive ones, plus a queue for incoming requests); we also added *Serve* that filters the unserved receive requests (that are in the queue), and takes the first one corresponding to the filter given as argument.

$a, b \in L ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition,
$\mid a.l_i$	field access,
$\mid a.l_i := b$	field update,
$\mid a.m(b)$	method call,
$\mid \mathit{clone}(a)$	shallow copy,
$\mid \mathit{Active}(a, m)$	activates a. m defines the service policy
$\mid \mathit{Serve}(M)$	serves a request among the set M of method labels, $M = \{m_1, \ldots, m_k\}$
$\mid \underline{\iota}$	<u>location in store</u>
$\mid \underline{\alpha}$	<u>activity reference</u>
$\mid \underline{f}$	<u>future reference</u>

Fig. 3. Sequential syntax for ASP (underlined terms only occur at run-time)

In the following, l_i range over field labels, m_j over method names, x_i and y_i over variables, and a and b over terms.

Run-time syntax is also shown in Figure 3, but this time both underlined and non-underlined terms are included. Dynamically, one can refer to existing futures, activities or locations in a local store. Thus, we add three new distinct name spaces: activities ($\alpha, \beta, \gamma \in Act$), locations ($\iota$), and futures ($f_i$), and we let run-time syntax refer to them. Note that locations are local to an activity.

Substitution of variables by locations are denoted: $\{\!\{x_i \leftarrow \iota_i'^{i \in 1..n}\}\!\}$, and have the usual semantics (i.e., the substitution of the variable x do not enter the binder $\varsigma(x, t)$ or $\varsigma(t, x)$).

3.2 Store and Values

An *object* is said to be entirely *evaluated* if it is of the form: $[l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}]$, that is all its field have been evaluated and allocated in the store. o range over evaluated objects. A *value* is an evaluated object, a reference to a future, or a reference to an activity: $v ::= o \mid \alpha \mid f_i$

A store is a mapping from locations to values: $(\iota_i \to v_i)^{i\in 1..p}$, it is used to store objects, and modify them. It allows the expression of the imperative nature of ASP. We let $\sigma_\alpha, \sigma_\beta, \ldots$ range over stores.

Let $\sigma + \sigma'$ update the values defined in σ' by those defined in σ. It is defined on $dom(\sigma) \cup dom(\sigma')$ by

$$(\sigma{+}\sigma')(\iota) = \begin{cases} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{cases}$$

Let $\theta ::= \{\!\{\iota_i \leftarrow \iota_i'^{\,i\in 1..n}\}\!\}$ range over renaming of locations; $\sigma\{\!\{\iota_i \leftarrow \iota_i'^{\,i\in 1..n}\}\!\}$ is the store σ where each occurrence of ι_i is replaced by ι_i'.

We define a function *Merge* which merges two stores (it creates a new store, merging independently σ and σ' except for ι which is taken from σ'):

$$\begin{array}{l} Merge(\iota, \sigma, \sigma') = \sigma'\theta + \sigma \\ \quad \theta = \{\!\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma)\backslash\{\iota\}, \iota'' \text{ fresh}\}\!\} \end{array}$$

$copy(\iota, \sigma)$ designates the deep copy of store σ starting at location ι. That is the part of store σ that contains the object $\sigma(\iota)$ and, recursively, all (local) objects that it references.

Moreover, the following operator copies the part of the store σ starting at the location ι at the location ι' of the store σ':

$$Copy\&Merge(\sigma, \iota\,;\, \sigma', \iota') \triangleq Merge(\iota', \sigma', copy(\iota, \sigma)\{\!\{\iota \leftarrow \iota'\}\!\})$$

Those operators are used in the semantics for ASP given in Table 2.

For simplicity of notations, ι_0 is a reserved location in each store where the active object of the activity is stored.

3.3 Structure of Activities

When a request is finished, a result has been calculated for it. The corresponding value is associated to the future identifier for the request: $f_i \to \iota$ means that ι is the location of the value associated with the future f. We denote by F the list of computed futures; it is a list mapping future identifiers to value locations: $F ::= (f_i \to \iota_i)^{i\in I}$ where $I \subseteq \mathbb{N}$.

A current request is a term being evaluated together with the future to which it will be associated: $a \to f_i$ means that a is being evaluated, and when a result will be computed it will be associated with the future f_i. C is a list of current requests: $C ::= (a_i \to f_i)^{i\in J}$ where $J \subseteq \mathbb{N}$.

A pending request is a request that has been received but not served yet. It is denoted by $[m_j; \iota; f_i]$ and consists of:

- the name of the *target method* m_j (invoked method),
- the location of the *argument* passed to the request ι,
- the *future* identifier f_i which will be associated to the result.

R is a list of current requests: $R ::= [m_i; \iota_i; f_i]^{i\in K}$ where $K \subseteq \mathbb{N}$.

:: denotes the concatenation of lists, and appending an element to a list.

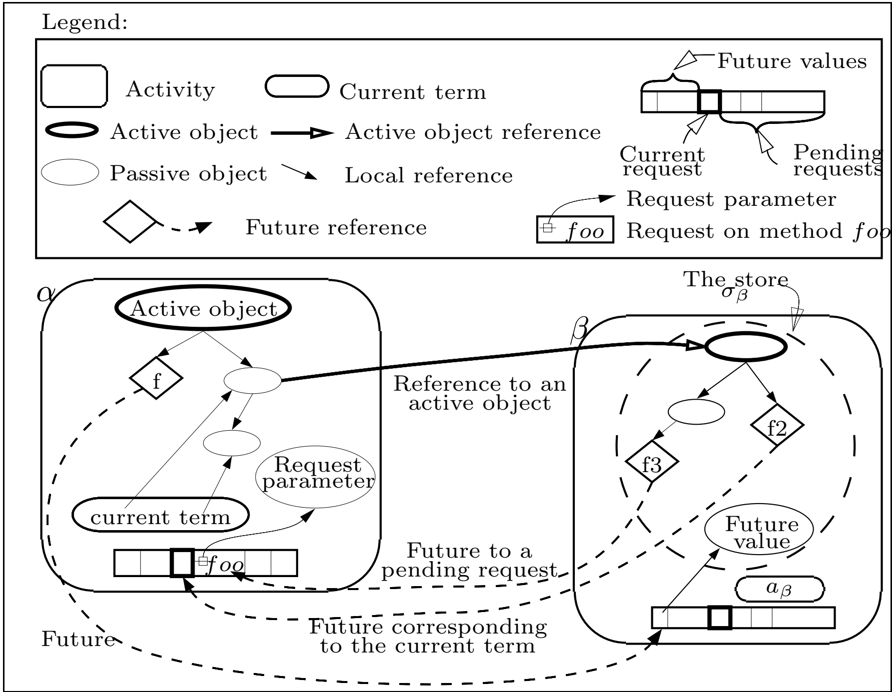

Fig. 4. Example of an ASP parallel configuration

An activity is simply formed of a name (α), a store (σ_α), and a request list containing finished, current, and pending requests ($F \cdot C \cdot R$) denoted by S.

$$S ::= F \cdot C \cdot R$$

A parallel configuration is a set of activities:

$$P, Q ::= \alpha[S_\alpha; \sigma_\alpha] \parallel \beta[S_\beta; \sigma_\beta] \| \ldots$$

Each future identifier is unique: it either belongs to the computed, current, or pending requests of a unique activity. Activities are unique too: there is a single activity with a given name. In practice the unicity of future (resp. activity) identifiers can be ensured by choosing as identifier a composition of the creator of the future[2] (resp. of the activity) with a unique local identifier. Note that activity names and future identifiers only appear at runtime and are used as references to activities or futures, e.g., for sending a request to an activity or receiving a reply from a future.

Configurations are identified modulo the reordering of activities. Figure 4 shows a parallel configuration of the ASP calculus. It shows two activities

[2] To better identify the request one might rather choose the identifier of the activity that treats the request.

α and β, bold ellipses are active objects, squares at the bottom are the requests (S), the bold square being the current requests (C), on the left are computed futures (F), and on the right pending requests (R). Future references are diamonds (and dotted arrows), whereas activity references are bold arrows, simple arrows are local references.

3.4 Contexts

Reduction contexts are expressions with a single hole ($\bullet$) expressing the part in the term where the reduction occurs. We define three reduction contexts:

- one that gives the reduction point in a sequential term,
- one that picks one of the futures an activity has computed, abstracting away the rest of the request list,
- one that gives the (unique) reduction point in the request list.

We first define sequential reduction contexts, allowing to pick the part of a term that is to be evaluated: they simply express a left-to-right call by value evaluation:

$$\begin{aligned}\mathcal{R} ::= \bullet \mid & [l_i = \iota_i, l_k = \mathcal{R}, l_{k'} = b_{k'}; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..k-1, k' \in k+1..n}_{j \in 1..m} \\ & \mid \mathcal{R}.m \mid \mathcal{R}.m(b) \mid \iota.m(\mathcal{R}) \mid \mathcal{R}.l := b \mid \iota.l := \mathcal{R} \mid \mathit{clone}(\mathcal{R}) \mid \mathit{Active}(\mathcal{R}, m)\end{aligned}$$

A future value context extracts one future value, corresponding to a finished request:

$$\mathcal{R}_f ::= F :: \bullet :: F' \cdot C \cdot R$$

A parallel reduction context extracts the current request actually served.

$$\mathcal{R}_c ::= F \cdot (\mathcal{R} \rightarrow f) :: C \cdot R$$

Actually, several requests are being served at the same moment, but only one is active. More precisely, when during the service of a request, a *Serve* primitive is encountered, the service is interrupted, and is stored and a new request, specified by the *Serve* primitive is served. The former current request will be restored when the new current one will be finished. The single point of reduction inside an activity is $\mathcal{R}_c$.

We denote by $\mathcal{R}[a]$ the term obtained by syntactically replacing the hole in the reduction context $\mathcal{R}$, by the term a; note that this substitution allows variables to be captured by a binder. Similarly, we use $\mathcal{R}_f[f \rightarrow \iota]$, and $\mathcal{R}_c[a]$.

3.5 Sequential Semantics

Table 1 recalls the semantics of the **imp**ς-calculus, in the form of a small-step operational semantics.

It has been slightly modified to take into account the second parameter of methods. The semantics do not have to take into account reduction contexts because they will already be used in the parallel semantics.

Table 1. Sequential reduction

STOREALLOC:

$$\frac{o \text{ is of the form } [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}] \quad \iota \notin dom(\sigma)}{(o, \sigma) \to_S (\iota, \{\iota \to o\} :: \sigma)}$$

FIELD:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n}{(\iota.l_k, \sigma) \to_S (\iota_k, \sigma)}$$

INVOKE:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..m}{(\iota.m_k(\iota'), \sigma) \to_S (a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}, \sigma)}$$

UPDATE:

$$\frac{\begin{array}{c}\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n \\ o' = [l_i = \iota_i; l_k = \iota'; l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..k-1, k' \in k+1..n}_{j \in 1..m}\end{array}}{(\iota.l_k := \iota', \sigma) \to_S (\iota, \{\iota \to o'\} + \sigma)}$$

CLONE:

$$\frac{\iota' \notin dom(\sigma)}{(clone(\iota), \sigma) \to_S (\iota', \{\iota' \to \sigma(\iota)\} :: \sigma)}$$

3.6 An Operational Semantics for the ASP Calculus

This section defines the semantics of the ASP calculus. The rules of Table 2 present the formal operational small step semantics of ASP, we explain briefly each of the rules below:

LOCAL performs a local reduction: each activity can perform a step of reduction as specified in Table 1, except that a reference to a future cannot be cloned.

NEWACT creates a new activity. m is the service method (first method executed). For simplicity, and because it is not restrictive in practice, m should have no argument. One could specify for example a FIFO service policy as follows:

$$\begin{aligned} Repeat(a) &\triangleq [repeat = \varsigma(x)a; x.repeat()].repeat() \\ FifoService &\triangleq Repeat(Serve(\mathcal{M})) \end{aligned}$$

where $\mathcal{M}$ is the set of all method labels defined by the concerned active object. Note that the reference to the created activity (γ) is stored in a new location, and thus $\sigma_\alpha(\iota)$ is still a passive object.

REQUEST sends a new request to an active object. It sends a deep copy of the parameter (at location ι'), and associates a new future f to this request. SERVE serves a new request. The current reduction is stopped and stored into the list

Table 2. Parallel reduction(unused variables are grayed)

LOCAL:

$$\frac{(a,\sigma)\rightarrow_S (a',\sigma') \qquad \nexists\iota,\ \big(a = clone(\iota)\wedge\sigma(\iota)=f_i\big)}{\alpha\big[\mathcal{R}_c[a],\sigma\big]\parallel P \longrightarrow \alpha\big[\mathcal{R}_c[a'];\sigma'\big]\parallel P}$$

NEWACT:

$$\frac{\gamma \text{ fresh activity} \quad \iota'\notin dom(\sigma) \quad \sigma'=\{\iota'\mapsto\gamma\}::\sigma \quad \sigma_\gamma = Copy\&Merge(\sigma,\iota\,;\,\emptyset,\iota_0)}{\alpha\big[\mathcal{R}_c[Active(\iota,m)];\sigma\big]\parallel P \longrightarrow \alpha\big[\mathcal{R}_c[\iota'];\sigma'\big]\parallel\gamma\big[\emptyset\cdot(\iota_0.m([])\rightarrow\emptyset)\cdot\emptyset;\sigma_\gamma\big]\parallel P}$$

REQUEST:

$$\frac{\begin{array}{c}\sigma_\alpha(\iota)=\beta \qquad \iota''\notin dom(\sigma_\beta) \qquad f \text{ fresh future} \qquad \iota_f\notin dom(\sigma_\alpha)\\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\iota'\,;\,\sigma_\beta,\iota'') \qquad \sigma'_\alpha=\{\iota_f\mapsto f\}::\sigma_\alpha\end{array}}{\alpha\big[\mathcal{R}_c[\iota.m(\iota')];\sigma_\alpha\big]\parallel\beta\big[S;\sigma_\beta\big]\parallel P\longrightarrow\alpha\big[\mathcal{R}_c[\iota_f];\sigma'_\alpha\big]\parallel\beta\big[S::[m;\iota'';f];\sigma'_\beta\big]\parallel P}$$

SERVE:

$$\frac{m\in M \qquad \forall[m';\iota';f_l]\in R,\ m'\notin M}{\begin{array}{c}\alpha\big[F\cdot\mathcal{R}[Serve(M)]\rightarrow f_i::C\cdot R::[m;\iota;f_k]::R';\sigma\big]\parallel P\longrightarrow\\ \alpha\big[F\cdot\iota_0.m(\iota)\rightarrow f_k::\mathcal{R}[[]]\rightarrow f_i::C\cdot R::R';\sigma\big]\parallel P\end{array}}$$

ENDSERVICE:

$$\frac{\iota'\notin dom(\sigma) \qquad \sigma'=Copy\&Merge(\sigma,\iota\,;\,\sigma,\iota')}{\alpha\big[F\cdot\iota\rightarrow f::C\cdot R;\sigma\big]\parallel P\longrightarrow\alpha\big[F::f\rightarrow\iota\cdot C\cdot R;\sigma'\big]\parallel P}$$

REPLY:

$$\frac{\sigma_\alpha(\iota)=f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta,\iota_f\,;\,\sigma_\alpha,\iota)}{\alpha\big[S;\sigma_\alpha\big]\parallel\beta\big[\mathcal{R}_f[f\rightarrow\iota_f];\sigma_\beta\big]\parallel P\longrightarrow\alpha\big[S;\sigma'_\alpha\big]\parallel\beta\big[\mathcal{R}_f[f\rightarrow\iota_f];\sigma_\beta\big]\parallel P}$$

REQUEST where $\alpha=\beta$:

$$\frac{\begin{array}{c}\sigma(\iota)=\alpha \qquad \iota'',\iota_f\notin dom(\sigma) \qquad f \text{ fresh future}\\ \sigma'=Copy\&Merge(\sigma,\iota'\,;\,\{\iota_f\mapsto f\}::\sigma,\iota'')\end{array}}{\alpha\big[\mathcal{R}_c[\iota.m(\iota')];\sigma\big]\parallel P\longrightarrow\alpha\big[\mathcal{R}_c[\iota_f]::[m;\iota'';f];\sigma'\big]\parallel P}$$

REPLY where $\alpha=\beta$:

$$\frac{\sigma(\iota)=f \qquad \sigma'=Copy\&Merge(\sigma,\iota_f\,;\,\sigma,\iota)}{\alpha\big[\mathcal{R}_f[f\rightarrow\iota_f];\sigma\big]\parallel P\longrightarrow\alpha\big[\mathcal{R}_f[f\rightarrow\iota_f];\sigma'\big]\parallel P}$$

of current requests (future f_i, expression $\mathcal{R}[[]]$), and the oldest request satisfying the labels specified in M is treated (future f_k, method m). If no such request is found, the activity is stuck until a matching request is found in the request queue.

ENDSERVICE occurs when the evaluation of a request is finished. It associates in the list of computed results, the current request response to the current future.

The evaluation that had been stopped at the beginning of the request is automatically restored (the second current request becomes first).

REPLY updates the value of a future. It can occur at any time provided an activity refers to a future for which the value has been computed by an(other) activity.

Note that futures remain in the F list, even when all the references to the future have been updated. No notion of garbage collection has been specified for futures in ASP, but it would be easy to adapt existing garbage collection techniques here.

3.7 Properties of the ASP Calculus

Overall, the ASP calculus provides a framework for understanding asynchronous distributed objects, and expressing the various potential implementation strategies that can be implemented in an active object middleware like ProActive. It allows the developer to study which implementation choices can be made without compromising the strong properties of determinacy ensured by the model.

Here we call determinism the fact that a program will always produce the same result (the same configuration), that is no concurrent actions have an impact on the program behaviour. More than determinism properties, our objective is to clearly identify the interferences that can be source of non-determinism. Consequently and more generally, we call partial confluence properties, the properties stating in which conditions two executions of the same programs will lead to the same result, i.e., to the same configuration. Determinism relies on a notion of equality between configurations: configurations are identified modulo alpha conversion[3], and modulo the dereferencing of futures already calculated (roughly, the same configuration before and after the application of a reply rule is considered as identical).

Here are the main properties that were disclosed thanks to the formal ASP model:

- future updates can occur at any time, in any order, as such the delivery of replies can be implemented with an infinity of strategies, in any order,
- the execution of a system is characterised by the order of request senders.

Those properties are further used in order to characterise several sets of deterministic types of programs:

- determinacy of programs based on a dynamic property: a non-deterministic program is a program which can lead to a point where two activities can send at the same time a request to the same third activity;
- determinacy of programs communicating over trees (i.e., programs for which the dependence between activities form a tree).

[3] Alpha-conversion is applied on futures and variables, and activity names are chosen deterministically to simplify the correct formulation of confluence properties.

The determinism properties clearly result from the absence of shared memory between active objects, and the single-threaded nature of ASP. The interested reader could refer to [16] for a detailed descriptions of ASP properties, details on the equivalence relation on ASP terms, and some proofs.

The difficulty when trying to prove properties on specific programs is to statically approximate activities, method calls, and potential services. Shifting to components will provide a statically defined topology: the component structure defines the distribution/concurrency structure.

These properties have massively been used in the development of the ProActive library, for example when implementing future update strategies – as futures can be updated at any time, or fault-tolerance mechanism – as the above properties give a minimal characterisation of a given execution. Globally, the impact of the formal definition and proven properties of the ASP calculus upon the real implementation of the ProActive middleware has proven to be very strong, and influenced both correctness and efficiency.

4 Components

We would like to define a component in a broad sense as:

> *a software module, with a standardised description of what it needs and provides, its accepted parameters for configuration, and to be manipulated by tools for composition and deployment.*

The GCM (Grid Component Model) has been defined in [11,30] by the CoreGrid European Network of Excellence. The GCM is defined as an extension of the Fractal [12] component model, and provides the same basic structure (Figure 5). The main additions that have been made to Fractal in the GCM are 1-to-many

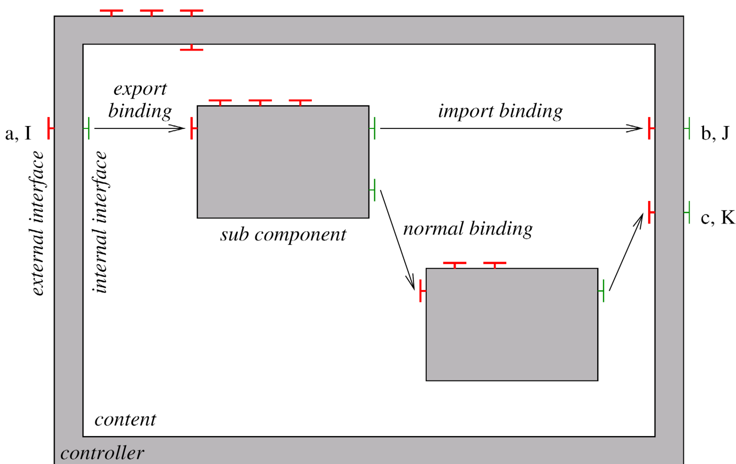

Fig. 5. A Fractal component

and many-to-1 communications, distribution, adaptive component control, and autonomic support.

A reference implementation of the GCM has been implemented in ProActive, overall the components depict the following characteristics:

- Primitive components featuring server and client interfaces,
- Composite components, allowing the hierarchical composition of primitive and composite components to build large and structured configurations,
- Interface specification including external languages such as: Java Interface, C++ .h, Corba IDL, WSDL, etc.
- Specification of Grid aspects such as: parallelism, distribution, virtual nodes, performance needs, QoS, etc.
- Multicast and Gathercast interfaces to manipulate parallel behaviours at the level of interface specification rather than hidden in the code,
- Component controllers, i.e., consider a controller as a sub-component, to provide dynamic adaptation of the component control,
- Autonomic components, the ability for a component to adapt to situations without relying on the outside.

Moreover, the GCM favors *asynchronous method calls*. By default, communications to the server interfaces are supposed to be non-blocking, as proposed in the ProActive implementation. Even in the case of methods returning non-void values, the caller is not supposed to be blocked during the method service. Together with the first-class futures, described above in the framework of ProActive and ASP, it provides the capacity to build both *structured and asynchronous* component configurations.

In the ProActive/GCM implementation, a primitive component is an active object together with passive ones, meaning that the component is the unit of concurrency and distribution. Indeed, as identified before, one of the difficulties towards deterministic distributed programs was to statically approximate activities, topologies, distributed method calls, and services. Shifting to configurations defined through components, and providing a statically defined topology, makes this static approximation a lot easier, very precise (e.g., activities and topologies are known exactly), and very practical. Indeed, the programmer has usually a clear idea about his program topology, therefore trying to discover it makes things unnecessarily complex and non-decidable. Instead of using the topology provided by the programmer, we take a stand to help the programmer achieve what he is willing to do, rather than trying to tell him from scratch the properties of his programs. Concurrency and behaviour are much easier to analyse as the distribution and remote communications are explicit: distribution is given by the component structures, and remote communications are exactly the ones following component bindings. Such explicitly-defined topology and dependencies also help a lot when analysing the behaviour of a component in isolation from its environment, and enhance the reusability of components.

Using the properties proved on the ASP calculus, it becomes possible to identify deterministic components in practice, first based on the detection of deterministic primitive components, further with the characterisation of deterministic

composition of primitive components. Overall, components provide a convenient abstraction for statically ensuring determinism.

5 VerCors: Behavioural Specification of Distributed Components

The effort described in [9,10] aims at *behavioural specification and verification* of asynchronous distributed systems; particularly, it deals with asynchronous distributed components based on active objects. That includes dealing with ProActive/GCM components as defined in the section above, specifying the structure and the visible behaviour of components, and generating behavioural models. The objective is then to check properties on this behaviour, using model-checking techniques.

The behavioural model generation is based on compositional modelling of primitive components using Parameterized Networks of Labelled Transition Systems (pNets [7,6]). pNets is a new model, created as a low-level formalism for expressing behavioural semantics of distributed systems, and as a compact and powerful internal format for verification frameworks. It has a hierarchical structure, where basic behaviours are (parameterized) labelled transition systems, and composition of subsystems is expressed by generic constructions in term of (parameterized) synchronisation vectors. Parameters are used to express value passing messages, but also parameterized topologies of systems. As such the pNets model unifies and extends the value-passing CCS of Ingolfsdottir and Lin [28], and the synchronisation networks of Arnold and Nivat [4]. In [6], we have shown how to use pNets for building models of active objects and of distributed components. The models define abstractions for the domains of the application parameters. This way, the models are suitable for use with various model-checking engines, either directly with engines able to deal with parameterized systems, or with finite-state model-checkers. The latter requires another abstraction of parameters, in which we define finite partitions of the domains. through another abstraction, using finite partitions of parameter domains, permitting the generation of finite state-spaces.

In a nutshell, while relying heavily on the results from the ASP formalisation, the approach adopted to achieve successful specification is rather practical. We use several sources of information from which we build the behavioural models: architecture described through ADL (Architecture Description Language) or through graphical diagrams, and behaviour using State Machine diagrams or static analysis of program source code.

The key features and properties coming from ASP that we model, and use, in the behavioural specification framework are:

- the synchronisation by wait-by-Necessity,
- the active objects without shared memory,
- the lack of user- and code-level concurrency and parallelism,

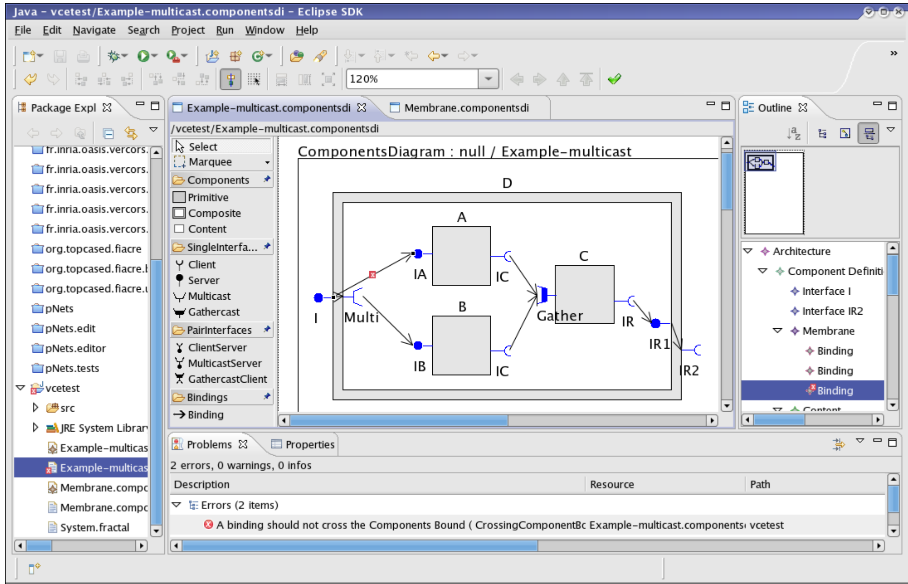

Fig. 6. VerCors Component Editor

- the atomicity of the rendez-vous protocol,
- the insensitiveness of programs w.r.t. distribution/location of activities within address spaces (JVMs).

On the practical side, we are building a specification and verification toolset for Fractal/GCM component systems called VerCors [8], that is freely available for research purposes in our website[4], and that is able to handle mid-size examples. For example in [13] we show how to find behavioural problems, and how to prove properties for a distributed cashdesks system built from over 15 components in 4 or 5 hierarchical layers, with a dozen parameter variables. At specification level, the VerCors platform includes diagram editors (Figure 6) for the architectural and behavioural definition of components [3]. From these diagrams, we build pNet models reflecting the behavioural semantics of the components in terms of communication between components; this includes control- and data-flow within components. This approach this allows us to build and to analyse behaviours of many levels of the ProActive/GCM framework; from active objects and hierarchical components, to non-functional features (deployment and reconfiguration) and group communication.

The model-checking part is done using existing and efficient engines (currently from the CADP toolset [22]). We generate explicit state-spaces both in a distributed and in a by-necessity (on-the-fly) manner. The properties addressed are temporal logic formulas, potentially including all safety properties in the regular μ-calculus. Simpler properties are directly accessible to the non-specialist user,

[4] `http://www.inria.fr/sophia/oasis/index.php?page=vercors`

including deadlock analyses, or reachability of predefined sets of events, typically deployment errors.

We are currently in the process of designing a specification language called Java Distributed Components (JDC)[15]. From JDC, we will allow the generation of both the behavioural models and the skeleton code of the implementation of components. This ensures, by construction, the correctness of the specification relatively to the implementation, and relieves us partially from the imprecision entailed by static analysis techniques.

We are also working on the inclusion of first class futures [14], and on the implementation in the platform of some specific infinite-state model-checking algorithms.

6 Conclusion: Practice in the ProActive Middleware

The ProActive middleware proposes a full-fledged environment with the programming of primitive code, the composition of such codes into composite components, the deployment on various practical infrastructures, and Graphical User Interface (Eclipse Plugin) to help programming, debugging and testing.

One of ProActive's key features is the combination of systematic asynchronous method-call, together with wait-by-necessity and first-class futures. At the level of components, it translates into the strong properties of large assemblage not being blocked by synchronous calls.

Within the GCM, collective operations, so far achieved at the level of programming, are being abstracted into elements of the interface. This shift first represents an achievement in terms of readability, and reuse. Second, functional methods can be used in various contexts, standard non-collective code and at the same time in powerful group interactions. Moreover, it also achieves an important rising with respect to the level of abstraction used by the programmer: interface versus the old API style for controlling parallelism, multicasting and synchronisations. Finally, it permits typing of collective behaviour.

From an historical stand, *modules* then *objects* then *components*, components could be viewed as moving backward in programming evolution. We are moving to a more static topology, while we have shifted from module (static assemblage) to objects where the inter-connection between pieces of code is rather purely dynamic. With components, the interconnection is static, and can only move back to dynamicity using controllers at execution, like binding controllers. In other words, only some specific entities of the architecture authorises to master dynamicity. From this point of view, components can be viewed as *dynamicity under control!*

Why does it scale? Thanks to a few key features like typed, asynchronous (connection-less) communications – somehow RMI+JMS unified, with messages rather than long-living interactions.

Why does it compose? First, because it scales! Indeed one would not be able to scale up to very large component configurations without the benefits

of asynchronous method invocations. Second, the model composes because of its typed nature: remote method invocations typed with interfaces. One would not be able to check large systems without some of the guaranties given by a static type system. The absence of unstructured *call-backs* and *ports* makes a tremendous difference with respect to verifying a component system.

As much as possible, we try to use static relations provided by component configurations, avoiding a great deal of static analysis. We believe dynamicity has to be mastered in the future with appropriate controllers, such as binding controllers. As an envisioned perspective, specific properties demonstrated on such controllers can be further used into a dynamically evolving system to prove global properties needed in complex, adaptive reconfigurations.

To conclude, the strategy embraced for verifying real applications is to let the user provide as much information as possible rather than trying to discover non-decidable facts about the programs. We believe that it is impossible to *tell* the user what he is doing, but instead it is possible to *verify* automatically on his behalf what he thinks he is doing. *Rather checking than guessing what the user is doing*, that could summarise our current approach.

This paper presented ASP, a formal model to check general properties at the language level, VerCors, a behavioural specification platform allowing to model-check properties on specific applications, and ProActive/GCM, a middleware for active objects and distributed component implementing the corresponding programming model and benefiting from those formal specifications and verifications. Globally, our objective is to provide safe and efficient distributed hierarchical components that are easy to program, and to be able to guarantee the behaviour both of the middleware, and of the applications.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, New York (1996)
2. Agha, G.: An overview of actor languages. ACM SIGPLAN Notices 21(10), 58–67 (1986)
3. Ahumada, S., Apvrille, L., Barros, T., Cansado, A., Madelaine, E., Salageanu, E.: Specifying Fractal and GCM Components With UML. In: Proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007). IEEE, Los Alamitos (2007)
4. Arnold, A.: Finite transition systems. Semantics of communicating sytems. Prentice-Hall, Englewood Cliffs (1994)
5. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying, for the Grid. In: Grid Computing: Software Environments and Tools. Springer, Heidelberg (2005)
6. Barros, T., Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed Fractal components. Annals of Telecommunications (to appear, 2008); Research Report INRIA RR-6491, `https://hal.inria.fr/inria-00268965`
7. Barros, T., Boulifa, R., Madelaine, E.: Parameterized models for distributed java objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 43–60. Springer, Heidelberg (2004)

8. Barros, T., Cansado, A., Madelaine, E., Rivera, M.: Model checking distributed components: The Vercors platform. In: 3rd workshop on Formal Aspects of Component Systems, Prague, Czech Republic, ENTCS (September 2006)
9. Barros, T., Henrio, L., Madelaine, E.: Behavioural models for hierarchical components. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 154–168. Springer, Heidelberg (2005)
10. Barros, T., Henrio, L., Madelaine, E.: Verification of distributed hierarchical components. In: International Workshop on Formal Aspects of Component Software (FACS 2005). Macao, ENTCS (October 2005)
11. 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 (to appear, 2008)
12. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in java. Softw., Pract. Exper. 36(11-12), 1257–1284 (2006)
13. Cansado, A., Caromel, D., Henrio, L., Madelaine, E., Rivera, M., Salageanu, E.: A Specification Language for Distributed Components Implemented in GCM/ProActive. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008), http://agrausch.informatik.uni-kl.de/CoCoME
14. Cansado, A., Henrio, L., Madelaine, E.: Transparent First-class Futures and Distributed Components. In: 5th workshop on Formal Aspects of Component Systems, Malaga, Spain, ENTCS (September 2008)
15. Cansado, A., Henrio, L., Madelaine, E.: Unifying Architectural and Behavioural Specifications of Distributed Components. In: 5rd workshop on Formal Aspects of Component Systems, Malaga, Spain, ENTCS (September 2008)
16. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer, Heidelberg (2005)
17. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 123–134. ACM Press, New York (2004)
18. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 519–538. ACM, New York (2005)
19. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
20. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in AmbientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
21. Flanagan, C., Felleisen, M.: The semantics of future and an application. Journal of Functional Programming 9(1), 1–31 (1999)
22. Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter 4, 13–24 (2002)
23. Gordon, A.D., Hankin, P.D., Lassen, S.B.: Compilation and equivalence of imperative objects. FSTTCS: Foundations of Software Technology and Theoretical Computer Science 17, 74–87 (1997)

24. Halstead Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(4), 501–538 (1985)
25. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1–2), 23–66 (2006)
26. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
27. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. In: Conference Record of the 23rd ACM SIGACT-SIGPLAN (POPL 1996), St. Petersburg, Florida, January 21–24, pp. 358–371. ACM Press, New York (1996)
28. Lin, H.: Symbolic transition graph with assignment. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 26–29. Springer, Heidelberg (1996)
29. Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theoretical Computer Science 364(3), 338–356 (2006)
30. OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Innovative features of gcm (with sample case studies): a technical survey. Technical report, Deliverable D.PM.07 (September 2007)
31. Yonezawa, A., Shibayama, E., Takada, T., Honda, Y.: Modelling and programming in an object-oriented concurrent language ABCL/1. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming, pp. 55–89. MIT Press, Cambridge (1987)