Asynchonous Distributed Components: Concurrency and Determinacy

Denis Caromel and Ludovic Henrio

CNRS – I3S – Univ. Nice Sophia Antipolis – INRIA Sophia Antipolis
Inria Sophia-Antipolis,2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis Cedex
{caromel, henrio}@sophia.inria.fr

Abstract. Based on the **impς**-calculus, ASP (Asynchronous Sequential Processes) defines distributed applications behaving deterministically. This article extends ASP by building hierarchical and asynchronous distributed components. Components are hierarchical - a composite can be built from other components, and distributed - a composite can span over several machines. This article also shows how the asynchronous component model can be used to statically assert component determinism.

1 Introduction

The advent of components in programming technology raises the question of their formal ground, intrinsic semantics, and above all their compositional semantics. It represents a real challenge as practical component models are usually quite complex, featuring distribution over local or wide area networks. But, few formal models for component were proposed so far [4, 20, 3, 14]. Since the first ideas about software components, usually dated in 1968 [1], the design of a reusable piece of software has technically evolved. From the first off-the-shelf modules, a component has become a complex piece of parameterized code with attributes to be set. Its behavior can be adapted with various non functional aspects (life-cycle, persistence, etc.). Finally, such piece of code is to be deployed in a hosting infrastructure, sometimes it can also be retrieved for replacement with a new version. In recent years, one crucial new aspect of component has been introduced: not only the interfaces being offered are specified, but also the needed interfaces.

A first key aspect of our work is to take into account this feature: the model being proposed allows to specify that a software components *provides* well defined interfaces, and *requires* well defined services or interfaces. A second and important contribution is to take into account components that are distributed over several machines. A given component can span as a unique entity over several hosts in the network. This work go further than a distributed-component infrastructure just allowing two components to talk over the network. Finally, the components being proposed are hierarchical (allowing a compositional spec-

Please use the following format when citing this chapter:

Caromel, D., Henrio, L., 2006, in International Federation for Information Processing, Volume 209, Fourth IFIP International Conference on Theoretical Computer Science-TCS 2006, eds. Navarro, G., Bertossi, L., Kohayakwa, Y., (Boston: Springer), pp. 165–183.

ification and verification of the behavior of large scale systems), communicating with remote method invocations (versus raw messages), and as much as possible decoupled (asynchronous to scale over large area networks).

When building some kind of component calculus, one has the option to start from scratch, or on the contrary to rely as much as possible on syntax and semantics of a programming calculus. This paper clearly takes the latter approach, relying as much as possible on a long history of research on concurrent and distributed calculi. It is in accordance with the practical situation where component infrastructure is usually added on top of a programming language. The main contributions of this paper are:

- a formalization of a component model featuring distribution, asynchrony, and hierarchical composition; with two translations defining the semantics;
- usage of components as a convenient abstraction for statically ensuring determinism, which, to our knowledge, is a totally novel approach.

This article is first a direct formalization of the component model implemented in ProActive [5, 11]. More generally, our distributed component model is *minimally characterized by asynchronous components, hierarchy, no shared memory, and a single threaded lowest level of components*; thus, it can be adapted to turn any object model into distributed decoupled components communicating by structured method calls.

Taking advantage of ASP and its properties [10], summarized in Section 2, this article provides a formal syntax for the description of distributed components in Section 3. Then, Section 4 shows an example of a deterministic component. Two translational semantics are given in Section 5. Finally, components provide a suitable abstraction for statically identifying deterministic programs as shown in Section 6.

2 Background

2.1 Some Related Works

ASP is based on the untyped imperative object calculus of Abadi and Cardelli [2], with a local semantics inspired from [15]. Futures [16, 13] are used to represent awaited results of remote calls, determinism is strongly related to process networks [17], and linear channels [18]. A comparison of ASP with other calculi can be found in [10, 9].

Components over Actors are presented in [4], compared to our work, Actor components neither are hierarchical nor benefit from the notion of futures. Moreover, the communication and evaluation model of Actors cannot guarantee the causal ordering and determinism properties featured by ASP. [3] focuses on the definition of connection and interactions, and on the specification on the behavior. Connectors having their own activity it is impossible to adapt our determinism properties to Wright.

Stefani et al. [6, 20] introduced the kell calculus that is able to model components and especially sub-components control. We rather demonstrate how to build distributed components that behave deterministically and for which the deterministic behavior is statically decidable. Moreover, the properties shown here rely on properties of communications and semantics of the calculus that are not ensured directly by the kell calculus, and its adaptation would be more complicated than the new calculus presented here. However, those two approaches being rather orthogonal, one could expect to benefit of both by adapting a kell calculus-like control of components with an (adaptation of) ASP as the underlying calculus. Bruneton, Coupaye and Stefani also proposed a hierarchical component model: Fractal [12], together with its reference implementation Julia [7]. Our work can also be considered as a foundation for distributed Fractal components, focusing on the hierarchical aspect rather than on the component control.

2.2 ASP Calculus: Syntax and Informal Semantics

The ASP calculus [10], is an extension of the **imp**ς-calculus [2, 15] with two primitives (*Serve* and *Active*) to deal with distributed objects. The ASP calculus is implemented as a Java library (ProActive [11]). ASP strongly links the concepts of thread and of object, it is minimally characterized by:

- *Sequential activities*: each object is manipulated by a single thread,
- Communications are *asynchronous method calls*, and
- *Futures* as first class objects representing awaited results.

$a, b \in L ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$	object definition,
$\mid a.l_i$	field access,
$\mid a.l_i := b$	field update,
$\mid a.m_j(b)$	method call,
$\mid clone(a)$	superficial copy,
$\mid Active(a, m_j)$	activates a. m_j defines the service policy
$\mid Serve(M)$	serves a request among the set M of method labels, $M = \{m_1, \ldots, m_k\}$

Fig. 1. ASP Syntax (l_i are fields names, m_j are methods names)

ASP is formalized as follows. An *activity* (denoted by α, β, γ, ...) is composed of a thread manipulating a set of objects put in a store. The primitive $Active(a, m)$ creates a new activity containing the object a which is said *active*, m is a method called upon the activity creation. Every request (method call) sent to an activity is actually sent to this master object. An activity also

contains the *pending requests* (requests that have been received and should be served later) and the computed results of the served requests (*future values*). $AO(\alpha)$ represents a reference to the remote active object of activity α. A *parallel configuration* (denoted by P, Q, ...) is a parallel composition of activities: $P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[\ldots] \| \ldots$ where a_α is the term currently evaluated in α, σ_α is the store (association between locations ι_i and objects), ι_α is the location of the active object inside σ_α, F_α is the list of calculated futures, R_α is the request queue, and f_α is the future corresponding to a_α.

Futures are generalized references that can be manipulated as local ones, they can be transmitted to other activities; and future identifiers are unique for the whole configuration. But, upon a strict operation (field or method access, field update, clone) on a future, the local execution is stopped until the value of the future is updated.

Calling a method on an active object atomically adds a new entry in a *request queue*, associates a *future* to the response and deep copies the argument of the request in the store of the destination activity. Deep copy allows one to prevent distant references to passive objects, synchronous request delivery ensures causal order between requests. The primitive $Serve(M)$ can appear at any point in the source code. Its execution stops the activity until a request on one of the methods of the set M is found in the request queue. The *oldest* such request is then removed from the request queue and executed (served).

Once the response to a request is computed, the corresponding value (*future value*) becomes available and every activity can get it. The futures associated with the currently served requests are called the *current futures*. Returning the value associated to a future (also called "updating a future"), consists in replacing reference to a future by a deep copy of the future value. We proved that the value of a future can be returned at any time without any consequence on the execution.

An operational semantics for ASP has been detailed in [10] and is denoted by $\longrightarrow$. It is based on a classical local reduction ($\rightarrow_S$) on ς-calculus terms [2]. This reduction specifies a *single* reduction point inside each activity which ensures a local sequentiality. $\mathcal{R}[a]$ denotes a reduction context, where the reduction point is inside a; thus $a_\alpha = \mathcal{R}[\iota.m_j(\iota')]$ means the next reduction of activity α will consist in performing a method call on the object referenced (locally) by ι; if moreover $\sigma_\alpha(\iota) = AO(\beta)$ then this is a remote method call to activity β. $\xrightarrow{*}$ denotes the reflexive transitive closure of $\longrightarrow$.

2.3 ASP Properties: Deterministic Objects Networks

This section presents the properties of the ASP calculus; mainly it recalls the definition of deterministic object networks which identifies a set of ASP terms that behave deterministically. Though DON terms are based on an intuitionist notion: "non-determinism only originate from conflicting requests"; ASP is the first calculus to feature such a property for concurrent imperative objects.

In the following, α_P denotes the activity α of configuration P. Without any restriction, and to allow comparison based on activities identifiers, we suppose that the freshly allocated activity names are chosen deterministically: the first activity created by α will have the same identifier for all executions.

Potential Services Let $\mathcal{M}_{\alpha_P}$ be an approximation of the set of M that can appear in the $Serve(M)$ instructions that the activity α may perform in the future. In other words, if an activity *may* perform a service on a set of method labels, then this set must belong to $\mathcal{M}_{\alpha_P}$:

$$\exists Q,\ P \xrightarrow{*} Q \wedge a_{\alpha_Q} = \mathcal{R}[Serve(M)] \Rightarrow M \in \mathcal{M}_{\alpha_P}$$

This set can be specified by the programmer or statically inferred.

Interfering Requests Two requests on methods m_1 and m_2 are said to be *interfering* in α in a program P if they both belong to the same potential service, that is to say if they can appear in the same $Serve(M)$ primitive:

$$\text{Requests on } m_1 \text{ and } m_2 \text{ are interfering if } \{m_1, m_2\} \subseteq M \in \mathcal{M}_{\alpha_p}$$

Equivalence Modulo Replies $\equiv_F$, defined in [9], is an equivalence relation considering references to futures already calculated as equivalent to local reference to the part of store which is the (deep copy of the) future value.

More precisely, $\equiv_F$ is an equivalence relation on parallel configurations modulo the renaming of locations and futures and permutations of requests that cannot interfere. Moreover, a reference to a future already calculated (but not locally updated) is equivalent to a local reference to the (part of the store which is the) deep copy of the future value.

Deterministic Object Networks If two interfering requests cannot be sent to the same destination (β below) at the same moment then the program behaves deterministically. Of course, two such request would originate from two different activities (α_Q). "there is at most one" is denoted by $\exists^1$.

Definition 1 (DON) *A configuration P, is a* Deterministic Object Network *($DON(P)$) if it cannot be reduced to a configuration where two interfering requests can be sent concurrently to the same destination activity:*

$$P \xrightarrow{*} Q \Rightarrow \forall \beta \in Q,\ \forall M \in \mathcal{M}_{\beta_Q},$$
$$\exists^1 \alpha_Q \in Q, \exists m \in M, \exists \iota, \iota',\quad a_{\alpha_Q} = \mathcal{R}[\iota.m(\iota')] \wedge \sigma_{\alpha_Q}(\iota) = AO(\beta)$$

Theorem 1 (DON determinism).

$$\begin{cases} DON(P) \wedge \\ P \xrightarrow{*} Q_1 \wedge \\ P \xrightarrow{*} Q_2 \end{cases} \Rightarrow \exists R_1, R_2, \begin{cases} Q_1 \xrightarrow{*} R_1 \wedge \\ Q_2 \xrightarrow{*} R_2 \wedge \\ R_1 \equiv_F R_2 \end{cases}$$

$DON(P)$ ensures that, for all orders of request sending, we always serve the requests in the same order. Thus, provided no two requests can be sent at the same moment on the same potential service of a given destination, the considered program behaves deterministically. Section 6 will show how components can ensure this statically.

3 Distributed Components

This section demonstrates how to build hierarchical and distributed components upon ASP. The asynchronous components presented below interact with method calls in an object-oriented way. The component specification presented in this section can be viewed as an abstraction of a classical ADL (e.g. the Fractal ADL [12]).

Definition 2 (Primitive Component - Figure 2) *A* primitive component *is characterized with a component name* $Name$*, together with names for a set of* Server Interfaces *(SI), and a set of* Client Interfaces *(CI). We denote by Exported(PC) the set* $\{SI_i\}^{i\in 1..k}$ *and by Imported(PC) the set* $\{CI_j\}^{j\in 1..l}$.

$$PC ::= Name < \{SI_i\}^{i\in 1..k}, \{CI_j\}^{j\in 1..l} >$$

Primitive Component Activity: *To give functionalities to a PC, we attach to it an ASP term, say a, corresponding to an object to be activated and its dependencies (passive objects); the service method of a: srv (the method to be triggered on activation of a; a mapping from SIs to subsets of the served methods; and a mapping from CIs to names of fields of the object a, these fields will store references to components.* $\mathcal{M}$ *ranges over the set of method labels, and* $\mathcal{L}$ *over the set of field labels of a.*

$$PC_{Act} ::= Name_{Act} < a, srv, \varphi_S, \varphi_C >$$
$$\text{where } \varphi_S : Exported(PC) \rightarrow \wp(\mathcal{M}) \text{ and}$$
$$\varphi_C : Imported(PC) \rightarrow \mathcal{L} \text{ are total functions}$$

This definition requires that a content PC_{Act} is attached to each primitive component PC, this content consists of a single activity.

Composite components can be built by interconnecting other components – either primitive or composite – and exporting some SIs and CIs.

We suppose that for all components, every interface has a different name (but names could also be disambiguated by using qualified names).

Definition 3 (Composite Component) *A composite component is a set of components exporting some server interfaces* (ε_S)*, some client interfaces* (ε_C)*, and connecting some client and server interfaces (defining a partial binding* ψ*), only interfaces of the direct sub-components can be used:*

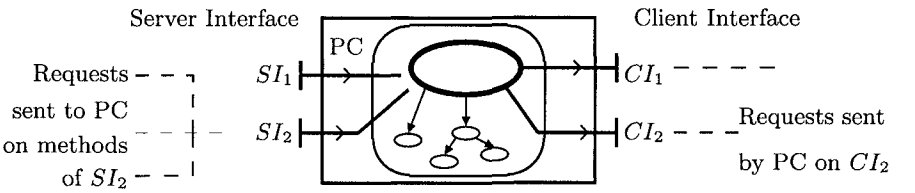


Fig. 2. A primitive component PC

$$CC ::= Name \ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg$$

Where a component C_i is either a primitive or a composite one: $C ::= PC \mid CC$, and each client interface CI inside CC can only be connected once, leading to the following definition:

$$\varepsilon_S : Exported(CC) \rightarrow \bigcup_{sc \in C_1 \ldots C_m} Exported(sc) \qquad \text{is a total function}$$

$$\psi : \bigcup_{sc \in C_1 \ldots C_m} Imported(sc) \rightarrow \bigcup_{sc \in C_1, \ldots C_m} Exported(sc) \qquad \text{is a partial function}$$

$$\varepsilon_C : \bigcup_{sc \in C_1 \ldots C_m} Imported(sc) \rightarrowtail Imported(CC) \qquad \text{is a partial surjective function}$$

Such that $dom(\psi) \cap dom(\varepsilon_C) = \emptyset$

We define: $Exported(CC) = dom(\varepsilon_S)$ *and* $Imported(CC) = codom(\varepsilon_C)$.

Defining ε_S as a function allows to export a given internal server interface as several external ones, but imposes each incoming request to be communicated to a single destination (each imported interface is bound to a single server interface of an internal component). Similarly, a client interface is exported only once for communications to have a single determinate destination: ε_C is a function (each client interface of an internal component is plugged at most once to an exported interface). ψ is a function so that internal communications are determinate too (each client interface of an internal component is plugged at most once to another internal server interface). And finally, also to ensure unicity of communication destination, ε_C and ψ have disjunct domain so that an internal client interface cannot be both bound internally and exported.

Correct Connections Figure 3 sums up the possible bindings that are allowed according to Definition 3. The component shown in the figure is a valid CC but not a DCC (DCC will be defined in Section 6.2, Definition 8).

Incorrect Connections Figure 4 shows the impossible bindings that correspond to the restrictions of Definition 3. The condition of Definition 3 that prevents the composition from being correct is written above each sub-figure.

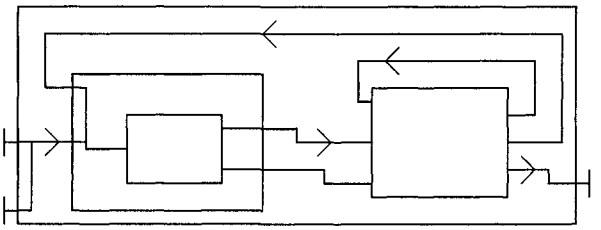


Fig. 3. A composite component

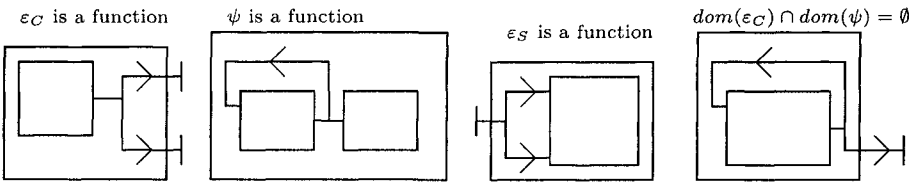


Fig. 4. Incorrect bindings between components

To conclude this section, we present two useful definitions: closed components that have no interface and form independent systems; and complete components for which all interfaces are either bound internally or exported: every request sent on a client interface has a destination and every server interface can at some point receive requests.

Definition 4 (Closed Component) *A component C is closed if it neither imports nor exports any interface:* $Imported(C) = \emptyset \ \wedge \ Exported(C) = \emptyset$

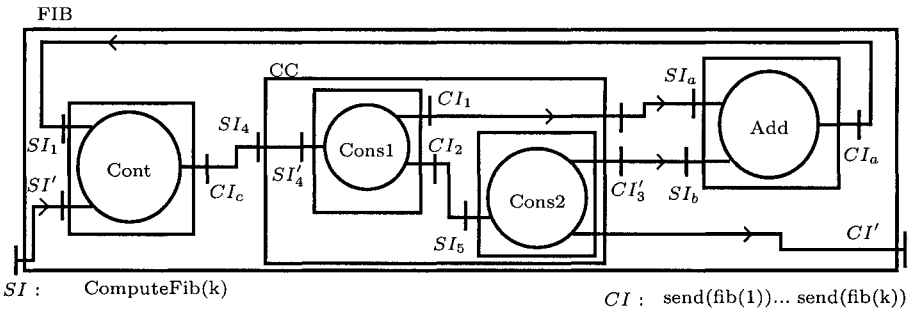
Definition 5 (Complete Component) *A primitive component is complete. A composite component* $Name \ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg$ *is complete if it consists of complete components and all its internal interfaces are plugged or exported:*

$$C_1, .., C_m \text{ are complete component } \wedge \ dom(\psi) \cup dom(\varepsilon_C) = \bigcup_{sc \in C_1 \ldots C_m} Imported(sc)$$
$$\wedge \ codom(\psi) \cup codom(\varepsilon_S) = \bigcup_{sc \in C_1 \ldots C_m} Exported(sc)$$

Non-complete components contain unplugged interfaces: some of the CIs of the sub-components must not be used (request without destination) or some of the SIs never receive any request (potential deadlock). As such it is reasonable to forbid them.

4 Example: A Fibonacci Component

Consider the Process Network that computes the Fibonacci numbers in [19]. Let us write an equivalent composite component as shown in Figure 5. Both *Cons1*



$$
\begin{aligned}
&FIB \ll Cont < \{SI_1, SI'\}, \{CI_c\} >, \\
&\quad CC \ll Cons1 < \{SI'_4\}, \{CI_1, CI_2\} >, Cons2 < \{SI_5\}, \{CI_3, CI''\} >; \\
&\qquad \{SI_4 \rightarrow SI'_4\}; \{CI_2 \rightarrow SI_5\}; \{CI_1 \rightarrow CI'_1, CI_3 \rightarrow CI'_3, CI'' \rightarrow CI'\} \gg, \\
&\quad Add < \{SI_a, SI_b\}, \{CI_a\} >; \\
&\quad \{SI \rightarrow SI'\}; \{CI_c \rightarrow SI_4, CI'_1 \rightarrow SI_a, CI'_3 \rightarrow SI_b, CI_a \rightarrow SI_1\}; \{CI' \rightarrow CI\} \gg \\
&Add_{Act} = <[n1 = 0, n2 = 0, out = []; \\
&\qquad serv = \varsigma(s, \_) Repeat(Serve(set1); Serve(set2); s.out.send(s.n1 + s.n2)), \\
&\qquad set1 = \varsigma(s, n) s.n1 := n, set2 = \varsigma(s, n) s.n2 := n \ ], \\
&\quad serv, \{SI_a \rightarrow \{set1\}, SI_b \rightarrow \{set2\}\}, \{CI_a \rightarrow out\} > \\
&Cons_1{}_{Act} = <[out = [], nxt = []; \\
&\qquad serv = \varsigma(s) \, (out.set1(1); nxt.send(1); Repeat(Serve(send))), \\
&\qquad send = \varsigma(s, n)(out.set1(n); nxt.send(n)) \ ] \\
&\quad serv, \{SI'_4 \rightarrow \{send\}\}, \{CI_1 \rightarrow out, CI_2 \rightarrow nxt\} >
\end{aligned}
$$

Fig. 5. A composite component for computing Fibonacci numbers

and *Cons2* forward their input to their two client interfaces (upon initialization they respectively send 1 and 0 to their client interfaces); they are merged in a composite component. *Add* simply sends on its output interface the addition of what the component receives on its two server interfaces. A controller *Cont* exports a server interface (*ComputeFib*(k)) taking an integer k and forwarding $k-1$ times its input on the other interface SI_1 to CI_c.

Primitive components for *Add* and $Cons_1$ are specified by Add_{Act} and $Cons_1{}_{Act}$, the others can be specified similarly (*Repeat* performs an infinite loop, ";" expresses sequential composition, both can be expressed directly in ASP). $Cons_2$ can be specified by renaming inputs and outputs of $Cons_1$.

Finally, the FIB composite component is built by interconnecting those components as shown and expressed in the figure. For example, requests sent by $Cons_1$ on CI_1 are first exported on interface CC'_1 of CC and then sent, according to the bindings of FIB, to the interface SI_a of *Add*. *Cons*2 sends *send* requests to the exported client interface, thus FIB produces $Fib(1) \ldots Fib(k)$.

5 Translational Semantics

This section gives two possible translational semantics for the component model, with ASP as the target calculus. The first one only instantiates primitive components and directly binds them but is not compositional. The second one instantiates an additional activity for each primitive and each composite component, it is defined recursively on the component structure. Both semantics first rely on a deterministic deployment phase; then, components can be started and communicate by asynchronous method calls. Both translations rely on the fact that the names of the interfaces are pairwise distinct, and thus a single component corresponds to each interface.

5.1 A Static Deployment

In the case of a *closed* component $CC = \text{Name} \ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg$, we define here a translation from this component system into an ASP configuration.

We denote $C \sqsubset CC$ the fact that a component C is *inside* another one CC:

$$C \sqsubset \text{Name} \ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg \Leftrightarrow \exists i \in 1..m,\ C = C_i \ \vee\ C \sqsubset C_i$$

The union of two disjunct partial function is denoted $\oplus$:

$$(f \oplus g)(x) = \begin{array}{l} f(x) \text{ if } x \in dom(f), \\ g(x) \text{ if } x \in dom(g) \\ \text{else undefined} \end{array}$$

For each SI of a composite component, ξ returns the primitive component interface which is (recursively) exported to it ($Id|_A$ is the identity function on A, $C \sqsubseteq C' \Leftrightarrow C = C' \vee C \sqsubset C'$). And symmetrically, μ recursively follows imported interfaces.

$$\xi_{PC} = Id|_{Exported(PC)}$$

$$\xi_{CC} : \bigcup_{CC' \sqsubseteq CC} Exported(CC') \rightarrow \bigcup_{PC \sqsubset CC} Exported(PC)$$

$$\xi_{\text{Name}\ll C_1,\ldots,C_m;\varepsilon_S;\psi;\varepsilon_C\gg} = (\xi_{C_1} \oplus \ldots \oplus \xi_{C_m}) \circ \varepsilon_S$$

Note that, if C is complete then ξ_C is total

$$\mu_{PC} = Id|_{Imported(PC)}$$

$$\mu_{CC} : \bigcup_{PC \sqsubset CC} Imported(PC) \rightarrow \bigcup_{CC' \sqsubseteq CC} Imported(CC')$$

$$\mu_{\text{Name}\ll C_1,\ldots,C_m;\varepsilon_S;\psi;\varepsilon_C\gg} = \varepsilon_C \circ (\mu_{C_1} \oplus \ldots \oplus \mu_{C_m})$$

Ψ_C defines all the bindings defined inside C:

$$\Psi_{PC} : \emptyset \to \emptyset$$

$$\Psi_{CC} : \bigcup_{C \sqsubseteq CC} \mathit{Imported}(C) \to \bigcup_{C \sqsubseteq CC} \mathit{Exported}(C)$$
$$\Psi_{\text{Name}\ll C_1,\ldots,C_m;\varepsilon_S;\psi;\varepsilon_C\gg} = \psi \oplus \Psi_{C_1} \oplus \ldots \oplus \Psi_{C_m}$$

In the general case, μ_C and Ψ_C are partial functions. In the case of a complete component C, for any client interface CI of C or a component inside C, either $\mu_C(CI)$ or $\Psi_C(CI)$ is defined.

Φ follows bindings, exportations and importations, to define the bindings between *primitive components*:

$$\Phi_C : \bigcup_{PC \sqsubseteq C} \mathit{Imported}(PC) \to \bigcup_{PC \sqsubseteq C} \mathit{Exported}(PC)$$
$$\Phi_C = \xi_C \circ \Psi_C \circ \mu_C$$

For a *complete closed* component C, Φ_C is a *total surjective* function.

We define below the deployment of the composite component CC: this static deployment creates as many activities as there are primitive components and binds their interfaces accordingly. Let PC_n range over primitive components defined inside CC: $PC_n = \text{Name}_n < \{SI_{ni}\}^{i \in 1..k_n}, \{CI_{nj}\}^{j \in 1..l_n} >$ s.t. $PC_n \sqsubset CC$; and $PC_{nAct} = \text{Name}_{nAct} < a_n, srv_n, \varphi_{S_n}, \varphi_{C_n} >$ range over their activities. We denote $N_s(SI_p)$, the index of the primitive component defining the interface SI_p: $N_S(SI_{ni}) = n$. The term defined in Figure 6 deploys the composite component CC defined above (the mutually recursive definition of activities *let rec... and...* can be built from core ASP terms). This deployment phase does not rely on any request and thus is entirely deterministic.

$$\begin{array}{l}
\textit{let rec } c_1 = \textit{Active}((a_1.\varphi_{C1}(CI_{11}) := c_{N_S(\Phi_{CC}(CI_{11}))}).\cdots.\varphi_{C1}(CI_{1k_1}) := c_{N_S(\Phi_{CC}(CI_{1k_1}))}, srv_1) \\
\textit{and } c_2 = \textit{Active}(a_2.\varphi_{C2}(CI_{21}) := c_{N_S(\Phi_{CC}(CI_{21}))}.\cdots.\varphi_{C1}(CI_{2k_2}) := c_{N_S(\Phi_{CC}(CI_{2k_2}))}, srv_2) \\
\textit{and } \ldots \\
\textit{and } c_n = \textit{Active}(a_n.\varphi_{Cn}(CI_{n1}) := c_{N_S(\Phi_{CC}(CI_n))}.\cdots.\varphi_{Cn}(CI_{nk_n}) := c_{N_S(\Phi_{CC}(CI_{nk_n}))}, srv_n)
\end{array}$$

Fig. 6. Deployment of a composite component

This is sufficient to give a semantics to the components with all useful connections bound; but, here, components are not runtime entities and this translation neither is modular, nor gives any way of manipulating dynamically the components (e.g. component reconfiguration is far from trivial). An active object representation of each composite component will make them accessible and reconfigurable at runtime.

5.2 A Compositional Translation

The compositional translational semantics adds one active object for each composite and for each primitive component. This translation does not suppose that any component is closed but requires that method names can be manipulated.

During the running phase, requests have to be dispatched between components: when a PC receives a *send* request from its contained active object, it serializes this request, forwards a *Call* request to the destination to which the CI is plugged. Then this method call may go through several CCs (first through CIs and then SIs). Finally, the *Call* request is received by a PC which de-serializes the request and calls a function on the contained active object.

Primitive Components Each PC is translated into a functional active object and a component active object.

The functional active object is built from the object specified in PC_{Act}, but every $\varphi_C(CI_j)$ field of the active object now references a passive object $CIobj_j$ and requests are sent through this object which acts as a proxy. The $CIobj_j$ serializes (builds a an object containing the method name) each request before forwarding it to the CI interface of the embedding PC (m_j methods range over the method of the interface CI).

$$CIobj_j \triangleq [PC = [], \forall m_j,\ m_j = \varsigma(s,x)s.PC.send(CI_j, m_j, x)]$$

$CIobj_j$ allows the component to systematically communicate using the encapsulating active object defined below.

The active object for the primitive component contains CI_j fields which store the destination component and interface to which they are plugged. Every request arriving at the CI_j interface has to be forwarded to the destination identified and stored inside the CI_j fields Figure 7 shows the object that is

$$
\begin{array}{l}
[\![\text{Name} < \{SI_i\}^{i\in 1..k}, \{CI_j\}^{j\in 1..l} > \text{ attached to the activity } \text{Name}_{Act} < a, srv, \varphi_S, \varphi_C >]\!] \triangleq \\
\quad let\ pc = Active(\\
\quad\quad [\forall j \in 1..l,\ CI_j = [CDest = [], IDest = []], \\
\quad\quad started = false, \quad act = []; \\
\quad\quad \forall j \in 1..l,\ setCI_j = \varsigma(s, CDest', IDest')(s.CI_j.CDest := CDest').IDest := IDest', \\
\quad\quad setact = \varsigma(s,a)s.act := a, \\
\quad\quad start = \varsigma(s)s.started := true, \\
\quad\quad Call = \varsigma(s, SI_i, m_j, x)s.act.m_j(x), \\
\quad\quad send = \varsigma(s, CI_j, m_k, x)s.CI_j.CDest.Call(s.CI_j.Idest, m_k, x), \\
\quad\quad srv = \varsigma(s)Repeat(if\ started\ then\ Serve(Call, send) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ Serve(setCI_1..setCI_l, setact, start)) \\
\quad\quad] \quad , srv)\ \ in \\
\quad let\ ao = Active((a.\varphi_C(CI_1) := (CIobj_1.PC := pc)\ldots).\varphi_C(CI_l) := (CIobj_l.PC := pc), srv) \\
\quad\quad in \quad\quad pc.setact(ao); pc
\end{array}
$$

Fig. 7. Primitive Component Deployment

instantiated for each primitive component, note that ao is initialized with the object containing the activity of the component in which $\varphi_C(CI_j)$ fields are replaced with $CIobj_j$ objects.

Composite Components Each CC contains the same CI fields as PCs, together with SI fields storing destinations to which received method calls must be forwarded.

For each composite component Name $\ll C_1, .., C_m; \varepsilon_S; \psi; \varepsilon_C \gg$, we define $N'_S(SI_i)$ the unique number such that $C_{N'_S(SI_i)}$ defines the server interface SI_i; and similarly $N'_C(CI_j)$ such that $C_{N'_C(CI_j)}$ is the sub-component containing the client interface CI_j. Figure 8 describes the instantiation of a composite component: it creates an activity for this component, binds the client interfaces according to ε_C and ψ, and the server interfaces according to ε_S.

$$
\begin{array}{l}
[\![\text{Name} \ll C_1, .., C_m; ; \varepsilon_S; \psi; \varepsilon_C \gg]\!] \triangleq \\
\quad let\ c_1 = [\![C_1]\!]\ in \quad \ldots \quad let\ c_m = [\![C_m]\!]\ in \\
\quad let\ \text{Name} = Active([\\
\quad\quad \forall SI_i \in dom(\varepsilon_S),\ SI_i = [CDest = c_{N'_S(\varepsilon_S(SI_i))}, IDest = \varepsilon_S(SI_i)], \\
\quad\quad \forall CI_j \in codom(\varepsilon_C),\ CI_j = [CDest = [], IDest = []], \\
\quad\quad started = false; \\
\quad\quad \forall CI_j \in codom(\varepsilon_C),\ setCI_j = \varsigma(s, Cdst', Idst')((s.CI_j).CDest := Cdst').IDest := Idst', \\
\quad\quad \forall SI_i \in dom(\varepsilon_S),\ setSI_i = \varsigma(s, Cdst', Idst')((s.SI_i).CDest := Cdst').IDest := Idst', \\
\quad\quad Call = \varsigma(s, CI\_SI, m_j, x)s.CI\_SI.CDest.Call(s.CI\_SI.IDest, m_j, x), \\
\quad\quad start = \varsigma(s)s.started = true, \\
\quad\quad srv = \varsigma(s)Repeat(if\ started\ then\ Serve(Call) \\
\quad\quad\quad\quad\quad\quad else\ Serve(\forall CI_j \in dom(\varepsilon_C)\ setCI_j, \forall SI_i \in dom(\varepsilon_S)\ setSI_i, start)) \\
\quad]\quad\quad , srv)\quad in \\
\quad\quad \forall CI_j \in dom(\psi),\ c_{N'_C(CI_j)}.setCI_j(c_{N'_S(\psi(CI_j))}, \psi(CI_j)) \\
\quad\quad \forall CI_j \in dom(\varepsilon_C),\ c_{N'_C(CI_j)}.setCI_j(\text{Name}, \varepsilon_C(CI_j)); \\
\quad\quad c_1.start(); \ldots ; c_m.start(); \text{Name}
\end{array}
$$

Fig. 8. Composite Component Deployment

Once deployed, the main component has to be started:
$[\![\text{Name} \ll ... \gg]\!].start()$

The deployment phase relies on *setact*, and *setCI* requests but the order of these requests is always the same as first the *setact* requests are sent during the primitive component creation; and then the *setCI* are sent by the unique embedding composite component, and thus the deployment phase is deterministic.

This translation reveals the importance of the first class nature of futures. Indeed, every request transits through several primitive and composite components; if futures could not be transmitted between activities, then every component activity would be blocked as soon as a request transits through it, leading almost systematically to a deadlock. Of course, the first class nature of futures is also a major advantage from a functional point of view for both translations.

5.3 Perspective: Reconfiguration and Component Controllers

In the last translation extra activities are added (a kind of component membranes), and requests must transit through them. But this additional cost is

counterbalanced by a promising expressiveness: it permits to envision the dynamic manipulation of components and requests at execution. Indeed, the semantics only forwards *Call* and *send* requests but it could be extended in order to add non-functional behaviors to components (e.g., fault-tolerance, security), intercept requests, and perform treatments on transiting messages; or reconfigure them. Reconfiguration consists in providing primitives allowing to change dynamically ε_S, ε_C, or ψ for a given composite; with the last encoding, this can be realized by convenient calls of $setCI_j$ and $setSI_i$ methods at the same level as the reconfiguration occurs. Although very interesting, defining safe and coherent reconfiguration of a whole distributed component system is a challenging perspective that is beyond the scope of this paper.

6 Deterministic Assembly of Objects and Components

6.1 Static DON

Suppose one has, for each runtime object, a static approximation of the activity it belongs to, denoted $\dot{\alpha}$, $\dot{\beta}$, ... $o \in \alpha$ means o is an object stored in α. Let $Part(\dot{\alpha})$ be true if the abstract activity $\dot{\alpha}$ may dynamically be partitioned into several different activities:

$$Part(\dot{\alpha}) \Leftrightarrow \exists o, o', o \in \alpha, o' \in \gamma\, \alpha \neq \gamma \wedge \dot{\alpha} = \dot{\gamma}$$

In other words, $\forall \alpha, \neg Part(\dot{\alpha})$ iff some abstract activities can be merged to form a single activity at runtime, but no abstract activity is split dynamically. Then, an object which can be either active or passive should be considered statically as active. To summarize:

$$\forall \alpha, \neg Part(\dot{\alpha}) \Rightarrow (\alpha \neq \gamma \Rightarrow \dot{\alpha} \neq \dot{\gamma})$$

Moreover, let $\mathcal{G}(P)$ be an *approximated call graph*:
If a request on the method foo can be sent from o to o', and $o \in \dot{\alpha}$ and $o' \in \dot{\beta}$ then $(\dot{\alpha}, \dot{\beta}, foo) \in \mathcal{G}(P)$, which means:

$$P \xrightarrow{*} Q \wedge a_{\alpha_Q} = \mathcal{R}[\iota.foo(\iota')] \wedge \sigma_{\alpha_Q}(\iota) = AO(\beta) \Rightarrow (\dot{\alpha}, \dot{\beta}, foo) \in \mathcal{G}(P)$$

Finally, let us characterize $\mathcal{M}_{\dot{\beta}_P}$ by

$$\forall \beta,\ M \in \mathcal{M}_\beta \Rightarrow M \in \mathcal{M}_{\dot{\beta}}$$

Then, the following property is an approximation of DON terms:

Definition 6 (Static DON) *Suppose the approximation of the set of activities is such that two activities cannot be merged:* $\forall \alpha, \neg Part(\dot{\alpha})$. *A program P is a Static Deterministic Object Network* $SDON(P)$ *if for all methods that can be sent at any time from two different activities toward a given destination, those methods cannot interfere:*

$$SDON(P) \Leftrightarrow \left(\begin{array}{l} (\dot{\alpha}, \dot{\beta}, m_1) \in \mathcal{G}(P) \\ (\dot{\alpha}', \dot{\beta}, m_2) \in \mathcal{G}(P) \\ \dot{\alpha} \neq \dot{\alpha}' \end{array} \right\} \Rightarrow \forall M \in \mathcal{M}_{\dot{\beta}_P}, \{m_1, m_2\} \not\subseteq M \Bigg)$$

Theorem 2 (SDON determinism). *SDON terms behave deterministically.*

Proof : It is sufficient to prove that $SDON(P) \Rightarrow DON(P)$, or that $\neg DON(P) \Rightarrow \neg SDON(P)$. Suppose P is not a DON, then it may send in the future two concurrent requests, and thus there is an activity β of a configuration Q such that $P \xrightarrow{*} Q$ and:

$$\exists M \in \mathcal{M}_{\beta_P}, \exists \alpha \neq \alpha', \exists m_1, m_2 \in M, \begin{cases} a_\alpha = \mathcal{R}[\iota.m_1(\iota')] \wedge \sigma_\alpha(\iota) = AO(\beta) \\ a_{\alpha'} = \mathcal{R}[\iota_2.m_2(\iota_2')] \wedge \sigma_{\alpha'}(\iota_2) = AO(\beta) \end{cases}$$

Then, as $\mathcal{G}(P)$ is an approximated call graph:

$$\exists M \in \mathcal{M}_{\beta_P}, \exists \alpha \neq \alpha', \exists m_1, m_2 \in M_{\beta_P} (\dot{\alpha}, \dot{\beta}, m_1) \in \mathcal{G}(P) \ \wedge \ (\dot{\alpha}', \dot{\beta}, m_2) \in \mathcal{G}(P)$$

and, as $\forall \alpha, \neg Part(\dot{\alpha})$, and by definition of $\mathcal{M}_{\dot{\beta}_P}$:

$$\exists \dot{\alpha} \neq \dot{\alpha}' \wedge (\dot{\alpha}, \dot{\beta}, m_1) \in \mathcal{G}(P) \ \wedge \ (\dot{\alpha}', \dot{\beta}, m_2) \in \mathcal{G}(P) \ \wedge \ m_1, m_2 \in M \wedge M \in M_{\dot{\beta}_P}$$

Finally, P is not a SDON. □

Of course, not every DON is a SDON, but SDON can be considered as the best approximation of DON that does not require control flow analysis.

6.2 Deterministic Components

We define a deterministic assemblage of components based on the fact that PCs provide an abstraction for activities and thus the SDON definition can be entirely expressed in terms of specifications of PCs and connections of interfaces. Indeed, suppose that for any two methods of the same SI cannot interfere, then a component system is deterministic if each SI can be accessed by a single activity (that is by a single component). Then, ensuring that only one CI is finally plugged to each SI is sufficient to ensure confluence.

As each PC can be considered as an abstraction of an activity, for each PC, we denote $\mathcal{M}_{PC}$ is the potential service of the activity defined by PC_{Act}.

Definition 7 (Deterministic Primitive Component (DPC)) *A primitive component* $PC = {}_{Name} < \{SI_i\}^{i \in 1..k}, \{CI_j\}^{j \in 1..l} >$ *is a DPC if its activity* ${}_{Name_{Act}} < a, srv, \varphi_S, \varphi_C >$ *associates its server interfaces to disjoint subsets of the served methods of the embedded active object; and such that two interfering requests necessarily belong to the same SI:*

$$\forall M \in \mathcal{M}_{PC}, \forall m_1, m_2 \in M \ (m_1 \in \varphi_S(SI_i) \wedge m_2 \in \varphi_S(SI_j)) \Rightarrow i = j$$

Definition 8 (Deterministic Composite Component (DCC)) *A DCC is a composite component built by connecting deterministic components.*

$$DC ::= DCC \mid DCP$$

$$DCC ::= \mathit{Name} \ll DC_1, \ldots, DC_m; \varepsilon_S; \psi; \varepsilon_C \gg$$

Where each SI is only used once, either bound or exported:

$$\psi, \varepsilon_C \text{ and } \varepsilon_S \text{ are injective} \quad \wedge \quad codom(\psi) \cap codom(\varepsilon_S) = \emptyset$$

Non-Deterministic Connections Figure 9 illustrates the non-deterministic bindings between components, corresponding to restrictions expressed in Definition 8. The condition of Definition 8 that prevents the composition from being determinate is written above each sub-figure.

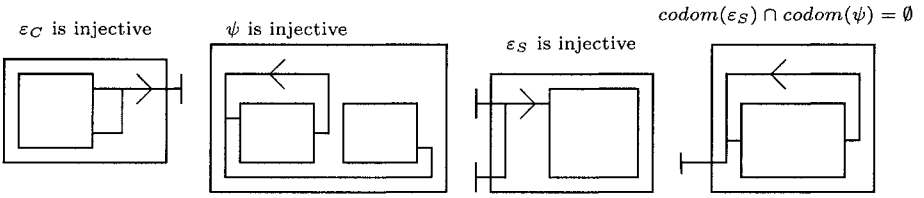


Fig. 9. Non-deterministic bindings between components

A DCC assemblage verifies the SDON property because each DPC statically identifies an activity; and the absence of sharing of SIs ensures that two activities cannot send concurrent requests on the same SI. Finally, the definition for DPC ensures that two requests on different SIs are not interfering.

Theorem 3 (DCC determinism). *DCC components behave deterministically.*

This theorem relies on the fact that composite components only forward requests if necessary, that is to say a request sent by a PC will be directly or indirectly transmitted to the PC that is finally plugged to the concerned interface, according to the Ψ_{CC} function defined in Section 5.1. In other words, neither the content nor the order of requests on a given binding is modified by the composite components involved in the communication.

Let us formally prove Theorem 3 in the case of the first translational semantics. In the case of the compositional semantics, more intermediate activities are created but each of them still verifies the SDON property.

Proof : Let $\Phi = \Phi_{CC}$ for CC a DCC. A DCC is only composed of injective ψ, ε_C and ε_S functions, codomains of ψ and ε_S are disjoint, and domain of ψ and domain of ε_C are disjoint, thus *the Φ function is injective.* In this translation, there is a bijection between the set of deployed activities and the set of

PCs (statically defined), thus we can consider PCs as the abstract domain for activities. This abstraction does not merge activities: $\forall PC, \neg Part(PC)$. We denote $comp(SI)$ the PC such that $SI \in Exported(PC)$ and similarly $comp(CI)$ the PC such that $CI \in Imported(PC)$. An approximation of $\mathcal{G}(P)$ becomes:

$$\{(PC, PC', m) | CI \in dom(\Phi) \wedge PC = comp(CI) \wedge PC' = comp(\Phi(CI)) \\ \wedge PC'_{Act} =< a, s, \varphi_S, \varphi_C > \wedge m \in \varphi_S(\Phi(CI))\}$$

And thus the SDON property is verified (with $PC'_{Act} =< a, s, \varphi_S, \varphi_C >$):

$$\left.\begin{array}{l}(PC, PC', m_1) \in \mathcal{G}(P) \\ (PC_2, PC', m_2) \in \mathcal{G}(P) \\ PC \neq PC_2\end{array}\right\} \Rightarrow \forall k \in 1,2,\ m_k \in \varphi_S(SI_k) \wedge PC' = comp(SI_k) \wedge SI_1 \neq SI_2 \\ \Rightarrow \forall M \in \mathcal{M}_{PC'},\ \{m_1, m_2\} \not\subseteq M$$

Indeed, $m_1, m_2 \in M$ and $M \in \mathcal{M}_{PC'}$ would imply $SI_1 = SI_2$ because PC' is a DPC. Finally a DCC behaves deterministically when deployed with the first translational semantics. □

DCC assemblage allows to statically ensure deterministic behavior of components, only based on the following requirements.

- Potential services can be statically determined, or are statically specified (every served set has been declared as a potential service).
- SI interfaces are respected: they only receive requests on the methods they define; this could be checked by typing techniques [2] on ASP source terms.
- Requests follow bindings and are not modified while following these bindings.
- There is a bijection between primitive components and functional activities.

The two first requirements correspond to static analysis or specification; whereas the two last ones must be guaranteed by the components semantics which is the case for both translational semantics of Section 5.

We have shown in [9] that every Process Network can be translated into a (deterministic) ASP term, which can then be fit into a deterministic assemblage of components. Such a bijection between process networks and DCCs will finally provide a large number of DCCs.

7 Conclusion

This article defines a *hierarchical* component calculus that provides a very convenient abstraction of activities and method calls. This abstraction allows static verification of *determinism* properties. Our component model is aimed at *distribution*, featuring *asynchronous remote method invocations*, and *futures* as generalized references passing through components. Primitive components are defined as a set of *Server Interfaces* (SI) and *client interfaces* (CI), together with an ASP term for the primitive component content. Intuitively, each SI corresponds to a set of methods, each CI to a field. Composite components are recursively made of primitives and other composites, with a partial binding between SIs and CIs, and some SIs and CIs exported.

Primitive deterministic components are defined by imposing that each set of interfering requests belongs to the same server interface. A deterministic composite (DCC) avoids potential interferences by imposing at most a single binding towards a server interface. For DCC, both translational semantics lead to configurations that respect the SDON properties, hence their deterministic nature. This results mainly relies on the fact that primitive components provide an abstraction of activities, and interfaces provide an abstraction of potential services.

One might have noticed the absence of any notion of location or machine, in contrast to calculus such as Ambient [8]. Because of the ASP calculus properties, an activity and further a component, can be placed "*anywhere*" without any semantic consequence. A given hierarchical component can be entirely mapped on a single machine, within the same address space, or fully distributed over the network, each inner component being located alone on its own machine. Abstracting activities by components is also convenient for distribution; allowing to map each primitive to a single location and to span composites over several machines.

Two translational semantics for the component model are proposed. The second translation allows to envision an even more interesting perspective: deterministic component reconfiguration. As components and bindings are achieved by ASP active objects, one can imagine to apply the general deterministic property (DON) to reconfiguration phase and to design coherent reconfigurations.

References

1. Icse '79: Proceedings of the 4th international conference on software engineering, 1979. Chairman-F. L. Bauer and Chairman-Leon G. Stucki and Chairman-M. M. Lehman.
2. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
3. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
4. Mark Astley and Gul A. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Proceedings of the ACM SIGSOFT 6th International Symposium on Foundations of Software Engineering (FSE)*, 1998.
5. Françoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3–7 November*, LNCS. Springer Verlag, Berlin, Heidelberg, 2003.
6. Philippe Bidinger and Jean-Bernard Stefani. The kell calculus: operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France, 2003.
7. Eric Bruneton, Thierry Coupaye, Matthieu Leclerc, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica

Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*. Springer, 2004.

8. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, pages 140–155.
9. Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag New York, Inc., 2005. To appear.
10. Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.
11. Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998. ProActive available at `http://www.inria.fr/oasis/proactive`.
12. Bruneton E., Coupaye T., and 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.
13. Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.
14. Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6(1), 1999.
15. Andrew D. Gordon, Paul D. Hankin, and Sren B. Lassen. Compilation and equivalence of imperative objects. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 17:74–87, 1997.
16. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.
17. Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, 1974.
18. Uwe Nestmann and Martin Steffen. Typing confluence. In Stefania Gnesi and Diego Latella, editors, *Proceedings of FMICS'97*, pages 77–101. Consiglio Nazionale Ricerche di Pisa, 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.
19. Thomas Parks and David Roberts. Distributed Process Networks in Java. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, April 2003.
20. Alan Schmitt and Jean-Bernard Stefani. The kell calculus: A family of higher-order distributed process calculi. *Lecture Notes in Computer Science*, 3267, Feb 2005.