Functional Active Objects: Typing and Formalisation

Ludovic Henrio, Florian Kammüller

▶ **To cite this version:**

Ludovic Henrio, Florian Kammüller. Functional Active Objects: Typing and Formalisation. Foundations of Coordination Languages and Software Architectures (FOCLASA'09), 2010, France. pp.83-101. hal-00485759

HAL Id: hal-00485759
https://hal.science/hal-00485759

Submitted on 21 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Functional Active Objects: Typing and Formalisation

Ludovic Henrio[1]

CNRS – I3S – INRIA, Sophia-Antipolis, France

Florian Kammüller[2]

Technische Universität Berlin, Germany

Abstract

This paper provides a sound foundation for autonomous objects communicating by remote method invocations and futures. As a distributed extension of ς-calculus, we define ASP_{fun}, a calculus of functional objects, behaving autonomously and communicating by a request-reply mechanism: requests are method calls handled asynchronously and futures represent awaited results for requests. This results in a well structured distributed object language enabling a concise representation of asynchronous method invocations. This paper first presents the ASP_{fun} calculus and its semantics. Secondly we provide a type system for ASP_{fun}, which guarantees the "progress" property. Most importantly, ASP_{fun} and its properties have been formalised and proved using the Isabelle theorem prover, and we consider it as a good step toward formalisation of distributed languages.

Keywords: Theorem proving, object calculus, futures, distribution, typing.

1 Introduction

This paper presents a functional active object language, featuring first class futures that has been formalised in the Isabelle/HOL theorem prover. ASP_{fun} is a distributed extension of the ς-calculus [1] where objects are distributed into several activities, and activities are the units of distribution. Communications toward activities are asynchronous (remote) method calls; and futures are identifiers for the result of such asynchronous invocations. A future represents an evaluation-in-progress in a remote activity. We call those futures "first class" because they can be transmitted between activities as any object: several activities may use the same future. The calculus is said to be functional because method update is realised on a copy of the object: there is no side-effect. The calculus is called distributed because it ensures absence of sharing between activities (processes) allowing them

[1] Email: Ludovic.Henrio@inria.fr

[2] Email: flokam@cs.tu-berlin.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to be placed on different machines, and features asynchronous RMI-like communications. From the original actor paradigm [16,2], several languages have been designed. Some languages directly feature actors, distributed active objects (like the ProActive [6] library), or other derived paradigms. The calculus ASP_{fun} provides a simple model for such languages. We can prove strong properties about it and, since the calculus is abstract, our semantics and mechanisation can be a basis for the analysis of related languages. For example, ASP_{fun} enables the correlation with the imperative ASP [7] with respect to communication strategies and static analysability. Typing is a well studied technique [22]; we prove here a classical typing property, progress, in unusual settings, distributed active objects.

The language, its type system and all properties have been completely formalised (`www-sop.inria.fr/oasis/Ludovic.Henrio/misc.html`) and proved in Isabelle/HOL [21]. Our main contributions are:

- A functional active object calculus with futures, and its properties.
- A proposal for a type system for active object languages.
- An investigation on how to provide a type-safe calculus featuring active objects and futures, where typing ensures progress.
- A formalisation of those features in a theorem prover, that will allow further investigations on futures, typing, and active objects paradigms.

Section 2 presents ASP_{fun}, its semantics and properties. Section 3 provides a sound type system for ASP_{fun}. Some details on the formalisation are given in Section 4. Our results are compared with related works in Section 5.

2 ASP_{fun}: A Functional Active Object Calculus

2.1 Syntax

We use three sets of identifiers: the labels of ς-calculus methods (l_i), the activities (α,β,...), and the futures (f_i). Like in ς-calculus, in ASP_{fun}, every term is an object, either given by its definition, or returned by a term evaluation like a method call. The syntax of ASP_{fun} includes *object definition*, *method invocation*, and *method override* inherited from ς-calculus. An object consists of a set of labelled methods. A method is a function with two formal parameters, one represents *self*, i.e., the object in which the method is contained, the other is an actual parameter given at invocation time. Object fields are not defined as they are considered as degenerate methods not using the parameters. A method call is addressed to an object and receives an object as parameter. A method update acts on an object, providing a new value for one method, possibly defining it. ς-calculus terms are identified modulo *renaming of variables*. One of the basic principles of ASP_{fun} is to perform a minimal extension of the syntax of ς-calculus. ASP_{fun} programs only use one additional primitive, *Active*, for creating an active object. The syntax of ASP_{fun} is shown in Table 1; the static syntax (the programs) consists of only underlined constructs; future and active object references are created at runtime.

$s, t ::= \underline{x}$		variable
$\mid \underline{[l_j = \varsigma(x_j, y_j)t_j]^{j \in 1..n}}$	$(\forall j,\ x_j \neq y_j)$	object definition
$\mid \underline{s.l_i(t)}$	$(i \in 1..n)$	method call
$\mid \underline{s.l_i := \varsigma(x, y)t}$	$(i \in 1..n, x \neq y)$	update
$\mid \underline{Active(s)}$		Active object creation
$\mid \alpha$		active object reference
$\mid f_i$		future reference

Table 1
ASP_{fun} syntax

2.2 Informal Distributed Semantics of ASP_{fun}

The semantics of object definition and method invocation is natural: a method invocation reduces to the method body where formal parameters are replaced by actual ones: $[l = \varsigma(x, y)a].l(b)$ reduces to a where x is replaced by $[l = \varsigma(x, y)a]$ and y is replaced by b [1], and a field update returns a new object replacing the original method by the one on the right side of ':='. We focus now on the distributed features of ASP_{fun}.

A *configuration* is a set of activities. Each activity possesses a single active object, which is a ς-calculus term. Activating an object, $Active(s)$, means creating a new activity with the object s to be activated becoming an *active object.*[3] It is immutable. A request sent to an activity is an invocation to the active object; it is processed by the activity. The set of requests processed by an activity is called *request queue* by similarity with the active object model but, here, as the calculus is functional, requests can be treated in an unordered fashion.

Every message sent toward an activity is a method call to the activated object. Such a *remote method invocation* (also called *request*) is asynchronous: the effect of this method call is – both – to create a new request in the request queue of the destination and to replace the original method invocation by a reference to the result of the created request. A reference to a (promised) result is called a *future*. In ASP_{fun}, futures are entities that can be passed to other activities, e.g. as arguments or results of requests; several activities may use the same future. Trying to access the result referenced by a future (e.g. invoking a method on it) is not possible until the future has been received. The current term of any request (even partially evaluated) can be returned at any moment: the current term for the request replaces the corresponding future. This operation is called a *reply*. We chose to allow replies with a partially evaluated term because it fits well with the functional nature of the calculus; but we will see in Section 6 that a more classical semantics returning only requests entirely evaluated is also dead-lock free under reasonable conditions. Future values must be stored forever.

Reduction can occur in any request of any activity. The only restriction is that an object cannot be sent to another activity (e.g. as a request parameter) if this object has free variables. Fortunately, the type system can ensure that any term

[3] In the ς-calculus every term is an object.

typed in an empty environment has no free variable, which is the case for all requests and some of their sub-terms (the ones in evaluation contexts).

It is difficult to give a natural semantics to the update of an active object because this would in general create an additional way of communicating with an active object. However, the functional nature of the calculus (updating an object creates a copy) oriented us toward the following solution: a method update on an active object creates a new activity with the method updated.

Proving determinism for ASP_{fun} is difficult and out of the scope of this paper. Informally, depending on the execution, the set of created activities and the number of requests may vary, but the result of the computation is deterministic.

As a tiny example of the semantics, $\mathit{Active}([l = \varsigma(x,y)[]]).l([])$ first creates an activity with the object $[l = \varsigma(x,y)[]]$, then performs a remote invocation on the method l of this activity (which creates a future), and finally replies, replacing the future by the result of the invocation, $[]$.

Practically, implementing strictly the semantics presented here is not very reasonable, because of the inefficiency of some execution paths. However, we consider this work as a reliable basis for further studies on stateless objects, and as an interesting proposal for a semantics for autonomous services, which in case they are stateless can be implemented such that they never dead-lock.

This model also represents component-like distributed systems interacting by invocation of services: an active object exposes its methods to the external world, and holds references to required external services provided by other active object. Active objects act as the unit of both distribution and composition.

2.3 Small-Step Operational Semantics

The semantics of ASP_{fun} necessitates to define some structures that will be used for the dynamic reduction. First, we define a configuration C as an unordered set of activities: a configuration is a mapping from activity identifiers to activities. Each activity is composed of a request queue (mapping from future identifiers to terms), and an active object (term).

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j\in I_i}\ ,\ t_i]^{i\in 1..p} \quad \text{where } \{I_i\} \text{ are disjoint subsets of } \mathbb{N}$$

As futures are referenced from anywhere, two requests must correspond to two different futures; uniqueness is ensured in this paper by indexing futures over disjoint families. We call *local semantics*, the semantics expressing the execution local to each *activity*, where an activity is the unit of distribution. In [1], the authors present various ς-calculi that only consider objects and their manipulation as primitive; local semantics of ASP_{fun} is just an adaptation of this work. More precisely, local semantics of ASP_{fun} extends ς-calculus, with a second parameter for methods.

Classically we define contexts as expressions with a single hole ($\bullet$). $E[s]$ denotes the term obtained by replacing the single hole by s.

$$\begin{aligned} E ::= {} & \bullet \mid [l_i = \varsigma(x,y)E, l_j = \varsigma(x_j,y_j)t_j^{j\in(1..n)-\{i\}}] \mid E.l_i(t) \mid \\ & s.l_i(E) \mid E.l_i := \varsigma(x,y)s \mid s.l_i := \varsigma(x,y)E \mid \mathit{Active}(E) \end{aligned}$$

For generality of results, we allow the reduction inside binders, like in the general semantics of ς-calculus (Definition 6.2.1 of [1], or even example page 62 showing a reduction inside binders). Then for their "operational semantics" (Section 6.2.4 of [1]) the authors use *reduction contexts* that do not allow reduction inside binders: $F ::= \bullet \mid F.l_i(t) \mid F.l_i := \varsigma(x,y)\, s \mid Active(F)$. Those reduction contexts would avoid using *noFV* requirement in the reductions. We chose to specify the most general semantics – allowing reduction inside binders – thus proving that our properties are valid for both kind of reduction contexts. Properties presented in this paper are also valid for "classical" reduction contexts (replacing E by F), however, we kept the most general formalisation. Reformulating our results for "classical" contexts would be trivial.

For a better integration with the distributed calculus, we choose a small-step semantics ($\rightarrow_\varsigma$) for the ς-calculus. It is composed of the two first rules of Table 2; one invokes a method (using the invoked object as first parameter), the other replaces a method definition, producing a new object with the updated method.

To simplify the reduction rules we let $Q, R ::= (f_{ij} \mapsto s_{ij})^{j \in 1..n_p}$ range over request queues and identify mappings modulo reordering: $\alpha[f_i \mapsto s_i :: Q, b] :: C$ is a configuration containing the activity α which contains a request $f_i \mapsto s_i$, where C is the remainder of the configuration that cannot contain an activity α. Now, $\alpha[Q,s] \in C$ means: α is an activity of C with request queue Q and active object s: $\alpha[Q,s] \in C \Leftrightarrow \exists C'.\, C = \alpha[Q,s] :: C'$. Similarly, $(f_i \mapsto s) \in Q$ stands for: a request of Q associates s to the future f_i. The empty mapping is $\varnothing$; the domain of a mapping is dom; e.g. $\mathrm{dom}(C)$ is the set of activities defined by C. Predicate $noFV(s)$ is true if s has no free variables (the only binder being ς this definition is classical). The parallel reduction $\rightarrow_\parallel$ on configurations is defined in Table 2.

Classically, the substitution $s\{x \leftarrow t\}$ is capture avoiding (renaming is performed to avoid free variables in t to be captured by binders in s), whereas the replacement of $\bullet$ by a term in a context is not.

- LOCAL performs a local reduction inside an activity: one step of the reduction $\rightarrow_\varsigma$ is performed on one request.
- ACTIVE creates an activity, the term s passed as argument to the Active primitive is the active object. The newly created activity receives a fresh activity identifier γ. Initially, the new activity has an empty request queue, and γ replaces the activation instruction: $Active(s)$, thus allowing future invocations to this activity.
- REQUEST sends a request from the activity α to the activity β, with $\alpha \neq \beta$. A new request is created at the destination, invoking the method l on the active object (t'); a fresh future f_k is associated to this request, and replaces the invocation on the sender side. Freshness is defined classically: f_k is fresh in C if $\forall\, \alpha[Q,t] \in C$, $f_k \notin \mathrm{dom}(Q)$.
- SELF-REQUEST is the REQUEST rule when the destination is the sender, $\alpha = \beta$.
- REPLY updates a future: it picks the request calculating a value for the future f_k, and sends the current value of this request (s) to an activity that refers to the future. The request may be partially evaluated. Necessarily $noFV(s)$ holds. The structure of the rule avoids introducing a separate rule for $\alpha = \beta$.

CALL
$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i(t)\right] \to_\varsigma E\left[s_i\{x_i \leftarrow [l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}, y_i \leftarrow t\}\right]}$$

UPDATE
$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i := \varsigma(x, y)t\right] \to_\varsigma E\left[[l_i = \varsigma(x, y)t, l_j = \varsigma(x_j, y_j)s_j^{j\in(1..n)-\{i\}}]\right]}$$

LOCAL
$$\frac{s \to_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \to_\| \alpha[f_i \mapsto s' :: Q, t] :: C}$$

ACTIVE
$$\frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad noFV(s)}{\alpha[f_i \mapsto E[Active(s)] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

REQUEST
$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\left[f_i \mapsto E[\beta.l(s)] :: Q, t\right] :: \beta[R, t'] :: C \to_\| \alpha\left[f_i \mapsto E[f_k] :: Q, t\right] :: \beta\left[f_k \mapsto t'.l(s) :: R, t'\right] :: C}$$

SELF-REQUEST
$$\frac{f_k \text{ fresh} \qquad noFV(s)}{\alpha\left[f_i \mapsto E[\alpha.l(s)] :: Q, t\right] :: C \to_\| \alpha\left[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t\right] :: C}$$

REPLY
$$\frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

UPDATE-AO
$$\frac{\gamma \notin \mathrm{dom}(C) \cup \{\alpha\} \qquad noFV(\varsigma(x, y)s) \qquad \beta[R, t'] \in \alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x, y)s] :: C}$$

Table 2
$\mathrm{ASP}_{\mathrm{fun}}$ semantics

- UPDATE-AO updates a method of an activity $\beta[R, t']$. It creates a new activity whose active object performs a (local) update on t': $t'.l := \varsigma(x, y)s$. It requires that the new method definition for l has no free variable.

The requirement $noFV(s)$ for the communicated terms is necessary. Indeed, communicating a term with free variables would cause variables to escape the scope of their binder. Giving an alternative semantics to communicated free variables without entailing shared memory is out of the scope of this paper [24].

2.4 An Example

The following example illustrates some of the advantages of futures for the implementation of services. Here a customer wants to make a hotel reservation. He uses a broker for this service. The example further illustrates how futures can be em-

ployed to provide some confidentiality. The broker does not need to give away his data base of hotel references: he can instead return just a reference to the result of his negotiations; the booking reference. We omit the actual search of the broker in his database and instead hardwire the solution always to "hotel". Also the internal administration of hotel is omitted. The example shows the flow of control given by the requests, replies, and the passing on of the futures: the booking reference can flow directly to the customer, possibly without passing by the broker. This shows that futures allow the implementation of efficient communication flows.

$$
\begin{array}{lll}
 & \text{customer}[f_0 \mapsto \text{broker.book}(date, limit), t] & \\
 & \parallel \text{broker}[\varnothing, [\text{book} = \varsigma(x, (date, limit))\text{hotel.room}(date), \ldots]] & \\
 & \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x, date)\text{bookingreference}, \ldots]] & \\
\rightarrow_{\parallel}^{*} & \text{customer}[f_0 \mapsto f_1, t] & \text{(REQUEST, LOCAL)} \\
 & \parallel \text{broker}[f_1 \mapsto \text{hotel.room}(date), \ldots] & \\
 & \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x, date)\text{bookingreference}, \ldots]] & \\
\rightarrow_{\parallel}^{*} & \text{customer}[f_0 \mapsto f_1, t]] & \text{(REQUEST, LOCAL)} \\
 & \parallel \text{broker}[f_1 \mapsto f_2, \ldots] & \\
 & \parallel \text{hotel}[f_2 \mapsto \text{bookingreference}, \ldots] & \\
\rightarrow_{\parallel}^{*} & \text{customer}[f_0 \mapsto \text{bookingreference}, t] & \text{(REPLY}^{*}\text{)} \\
 & \parallel \text{broker}[f_1 \mapsto f_2, \ldots] & \\
 & \parallel \text{hotel}[f_2 \mapsto \text{bookingreference}, \ldots] &
\end{array}
$$

2.5 Properties of ASP_{fun}

This section presents two major properties of ASP_{fun}: the semantics is well-formed; and reduction does not create cycles of futures and activity references.

2.5.1 Well-formed Configuration

To show correctness of the semantics, we define a *well-formed configuration* as referencing only existing activities and futures; then we prove that reduction preserves well-formedness.

Definition 2.1 [Well-formed configuration] A configuration C is well-formed, denoted $wf(C)$, if and only if for all α, f_i, s, Q, and t each of the following holds:

$$\alpha[Q, E[\beta]] \in C \ \vee \ \alpha[f_i \mapsto E[\beta] :: Q, t] \in C \ \Rightarrow \ \beta \in \text{dom}(C)$$

$$\alpha[Q, E[f_k]] \in C \ \vee \ \alpha[f_i \mapsto E[f_k] :: Q, t] \in C \ \Rightarrow \ \exists \gamma, R, t'. \gamma[R, t'] \in C \wedge f_k \in \text{dom}(R)$$

We have shown that, starting from a well-formed configuration, the reduction shown in Table 2 always reaches a well-formed configuration.

Property 1 (Reduction preserves well-formedness) $(s \rightarrow_{\parallel} t \wedge wf(s)) \ \Rightarrow \ wf(t)$

This can be considered as a correctness property for the semantics of ASP_{fun}: no ill-formed configuration can be created by the reduction.

2.5.2 Absence of Cycles

Informally, ASP_{fun} avoids blocking method invocations because a not fully evaluated future can be returned to the caller at any time. The natural question arises whether there is the possibility for live-locks: a cycle of communications (here, a cycle of replies in fact) in which no real progress is made apart from the actual exchange of communication. However, we can show that, given a configuration with no cycle, any possible configuration that may be derived from there has no cycle either. By cycles we mean cycles of activity references and futures.

We say that an activity or a future knows another one if it holds a reference to it. An activity holds a reference if it has this reference inside its active object. A future holds a reference if the request computing this future contains this reference. Table 3 shows the rules defining the $knows_C$ relationship for a configuration C together with the *nocycle* property. It is necessary to interleave references to futures and activities in the definition of $knows_C$ because, for example, a reference from an active object becomes a reference from a future when a REQUEST rule is evaluated.

$$\frac{\alpha[Q, E[\beta]] \in C}{\alpha \ knows_C \ \beta} \qquad \frac{\alpha[f_i \mapsto E[\beta] :: Q, t] \in C}{f_i \ knows_C \ \beta} \qquad \frac{\alpha[Q, E[f_k]] \in C}{\alpha \ knows_C \ f_k}$$

$$\frac{\alpha[f_i \mapsto E[f_k] :: Q, t] \in C}{f_i \ knows_C \ f_k} \qquad nocycle(c) \Leftrightarrow \neg\exists r.\, r \ knows_C^+ \ r$$

Table 3
The *nocycle* property

We proved that the reduction defined in Table 2 maintains the absence of cycles for a well-formed configuration.

Theorem 2.2 *Reduction does not create cycles:*

$$nocycle(C) \ \wedge \ wf(C) \ \wedge \ C \rightarrow_{\|} C' \quad \Rightarrow \quad nocycle(C')$$

The proof is by rule analysis. The theorem also relies on the fact that domains of request queues are disjoint, which is enforced by the definition of a configuration in ASP_{fun}. Absence of cycles ensures that there are no live-locks related to the distributed aspects of ASP_{fun}, i.e. no infinite cycle of replies. Live-locks that can exist in ASP_{fun} are inherited from ς-calculus: they are either infinite loops inside a ς-calculus term, or infinite sequences of method calls (distributed or not).

Absence of cycle is a limit to the expressiveness of the language (no cross-references), but this restriction is inherited from the functional nature of the language. Indeed, functional languages have no references, whereas active objects and futures create some kind of references; preventing cycles and modification is necessary to still consider ASP_{fun} as a functional language.

2.6 Initial Configuration

This section shows how a reasonable initial configuration can be built from a program, i.e. a term without future or active object reference (as defined in Section 2.1). In most distributed languages, programmers do not write configurations, but usual programs invoking some distribution or concurrency primitives (in ASP_{fun} *Active* is the only such primitive). This is reflected by the ASP_{fun} syntax given in Section 2.1. A "program" is a term a given by this static syntax (it has no future or active object reference). In order to be evaluated, this program must be placed in an initial configuration. The initial configuration has a single activity single request consisting of the user program:

$$\mathit{initConf}(s_0) = \alpha[f_0 \mapsto s_0, []]$$

where s_0 is a static term without free variables as defined in Section 2.1. Note, that this configuration is necessarily well-formed, and the activity α will never be accessible. Consequently, any reachable configuration is well-formed. We can also immediately see that the initial configuration has no cycles, and using Theorem 2.2, we can show that any reachable configuration has no cycles.

Property 2 *Any configuration reachable from an initial configuration is well-formed and has no cycles ($\rightarrow_{\parallel}^{*}$ is the reflexive transitive closure of $\rightarrow_{\parallel}$).*

$$\mathit{initConf}(s_0) \rightarrow_{\parallel}^{*} C \quad \Rightarrow \quad \mathit{wf}(C) \ \wedge \ \mathit{nocycle}(C)$$

3 Typing Active Objects

This section provides a type system for ASP_{fun}. This first involves typing the *Active* primitive, but also type-checking an ASP_{fun} configuration. After the classical subject-reduction property, we show that the type system ensures *type uniqueness*, and *well-formedness* of configurations, and more importantly *progress*. We will see that typing ensures that no method can be invoked on a term that is unable to handle it; the semantics ensures that no invocation or update on a future or an activity can be indefinitely blocked.

3.1 A Local Type system

We first adapt the simple type system that Abadi and Cardelli devised as $\mathbf{Ob_1}$ in [1]. Object types are of the form $[l_i : B_i \rightarrow D_i]^{i \in 1..n}$. The syntax of ASP_{fun} is extended by adding type information on both variables under the binder ($\varsigma(x, y)$ becomes $\varsigma(x{:}A, y{:}B)$). As highlighted in [1], adding type information on the binders ensures type uniqueness.

Table 4 defines the typing of local ASP_{fun} terms as presented in 2.1. It is strongly inspired by the typing of $\mathbf{Ob_1}$. A, B, and D range over types. The variable T represents a *type environment* containing type assumptions for variables, and is identified modulo reordering. Its extension by a new assumption stating that the variable x has type A is denoted by $x : A :: T$. We now authorise :: to update a mapping entry: $(x : A) :: T$ associates the type A to x, even if an entry for x

$$\textsc{Val}\ x \quad x:A :: T \vdash x:A$$

$$\textsc{Type Object}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1..n} \qquad \forall i \in 1..n,\ x_i:A :: y_i:B_i :: T \vdash t_i : D_i}{T \vdash [l_i = \varsigma(x_i : A, y_i : B_i)t_i]^{i\in 1..n} : A}$$

$$\textsc{Type Call}\quad \frac{T \vdash s : [l_i : B_i{\to}D_i]^{\ i\in 1..n} \qquad j \in 1..n \qquad T \vdash t : B_j}{T \vdash s.l_j(t) : D_j}$$

$$\textsc{Type Update}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1..n} \qquad T \vdash s : A \qquad j \in 1..n \qquad x:A :: y:B :: T \vdash t : D_j}{T \vdash s.l_j := \varsigma(x : A, y : B)t : A}$$

Table 4
Typing the local calculus

$$\textsc{Type Active}\quad \frac{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \mathit{Active}(a) : A}$$

$$\textsc{Type Activity Reference}\quad \frac{\beta \in \mathrm{dom}(\Gamma_{act})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \beta : \Gamma_{act}(\beta)}$$

$$\textsc{Type Future Reference}\quad \frac{f_k \in \mathrm{dom}(\Gamma_{fut})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash f_k : \Gamma_{fut}(f_k)}$$

$$\textsc{Type Configuration}\quad \frac{\mathrm{dom}(\Gamma_{act}) = \mathrm{dom}(C) \qquad \mathrm{dom}(\Gamma_{fut}) = \bigcup\{\mathrm{dom}(Q) \mid \exists\ \alpha, a.\ \alpha[Q, a] \in C\} \qquad \forall\, \alpha[Q, a] \in C.\ \begin{cases} \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash a : \Gamma_{act}(\alpha) & \wedge \\ \forall\, f_i \in \mathrm{dom}(Q).\ \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash Q(f_i) : \Gamma_{fut}(f_i) \end{cases}}{\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle}$$

Table 5
Typing configurations

existed in T. The first rule of Table 4 accesses the type environment. Type Object describes how an object's type is checked from its constituents: an object of type $[l_i : B_i{\to}D_i]^{i\in i..n}$ is formed from bodies t_i of types B_i using self parameter x_i of type A, and additional parameter y_i of type B_i. When a method l_j is invoked on an object s of type $[l_i : B_i{\to}D_i]^{i\in i..n}$ the result $s.l_j(b)$ has type D_j provided s has type B_j (Type Call). A method update requires that the updated object has the same type as self in the new method (Type Update).

In [1], additional rules ensure that the typing environment is well-formed. We simplified it here by defining environment as a mapping. Also, a rule for correct formation of object types is introduced in [1], mainly ensuring that there is no infinitely nested object type. This last assumption has been omitted here as it did not seem necessary and, indeed, the properties shown below have been mechanically proved without any additional assumptions on type formation.

3.2 A Type System for ASP$_{fun}$

The type system for ASP$_{\text{fun}}$ is based on an inductive typing relation on ASP$_{\text{fun}}$ terms; it is defined in Table 5. From local typing (Table 4), in addition to types of variables, we need to refer to types for futures and activities. Thus, we add a pair of parameters $\langle \Gamma_{act}, \Gamma_{fut} \rangle$ in the assumptions of a typing statement, we write $\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash x : A$ instead of $T \vdash x : A$. These parameters consist of a mapping Γ_{act} from activities to the type of their active object, and another one Γ_{fut} from future identifiers to the type of the corresponding request value. Thus, we first adorn each rule of Table 4 with those two additional parameters.

Then, we add to these rules the three first rules of Table 5 that define the local typing of ASP$_{\text{fun}}$. These rules allow the typing of references to activities and futures, and define typing of the *Active* primitive: the type of an activated object is the type of the object.

The last rule of Table 5 incorporates into a configuration the local typing assertions. This rule states that a configuration C has the configuration type $\langle \Gamma_{act}, \Gamma_{fut} \rangle$ if the following conditions hold.

- The same activity names are defined in C and in Γ_{act};
- the same future references are defined in the activities of C and in Γ_{fut};
- for each activity of C, its active object has the type defined in Γ_{act};
- and each request has the type defined in Γ_{fut} for the corresponding future.

Similarities can be found between typing of activity or future references, and reference types [22]. A closer work seems to be the typing rules for futures [20].

3.3 Basic Properties of the Type System

Let us start by a couple of simple properties of the typing system. First, type-uniqueness existing for $\mathbf{Ob_1}$ is also verified by our type system.

Property 3 (Unique Type) *Each expression in ASP$_{fun}$ has a unique type.*

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A \wedge \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A' \Longrightarrow A = A'$$

Well-typed configurations are well-formed. Indeed, if an activity or a future is referenced in the configuration, it must have a type, and thus be defined in Γ_{act} or Γ_{fut}, and also in the configuration.

Property 4 (Typing ensures well-formedness) $\vdash C : A \Rightarrow wf(C)$

3.4 Subject Reduction

Subject reduction ensures that reduction preserves the typing relation. Therefore, it is often also called *preservation*. We prove subject reduction of ASP$_{\text{fun}}$ with respect to the type system given in the previous section.

We prove first the subject reduction property for the local reduction:

Property 5 (Local Subject Reduction)

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t : A \ \wedge \ t \rightarrow_{\varsigma} t' \Rightarrow \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t' : A$$

Then, we prove subject reduction for the full typing relation of configurations.

Theorem 3.1 (Subject Reduction)

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ C \rightarrow_{\|} C' \Rightarrow \exists \ \Gamma'_{act}, \Gamma'_{fut}. \ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$$

where $\Gamma_{act} \subseteq \Gamma'_{act}$, *and* $\Gamma_{fut} \subseteq \Gamma'_{fut}$.

Note that activities and futures may be created by the reduction and thus the typing environment may have to be extended.

3.5 Progress and Absence of Dead-locks

Finally, we can prove progress for well-typed configurations. Progress states that any expression of the language is either a value, or can be reduced. In ASP_{fun}, we prove progress for each request of a configuration. A term is a value, i.e. a totally evaluated term, if it is either an object (like in [1]) or an activity reference.

$$\mathit{isvalue}(s) \Leftrightarrow \exists l_i, \, t_i, \, A. \, s = [l_i = \varsigma(x_i : A, y_i : B) t_i]^{i \in 1..n} \vee \exists \alpha, \, s = \alpha$$

The type system is useful for ensuring that every accessed method exists on the invoked object. In fact, local typing ensures progress of local reduction. Typing for configurations extends the typing relation to distributed objects, ensuring for example that a method invocation on a future will be possible once the result is returned. Absence of dead-locks for the distributed semantics is only ensured by the functional nature of ASP_{fun}, by the absence of loops, and by the particular semantics of the calculus. This can be simply formulated as follows:

Property 6 $\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \wedge \alpha[f_i \mapsto s :: Q, t] \in C \Rightarrow \mathit{isvalue}(s) \vee \exists C' \, . \, C \rightarrow_{\|} C'$

More precisely, we prove that the term s in the theorem above can be reduced. Unfortunately, as already shown in [1], ς-calculus does not ensure that a reduced term is different from the source one, but this is an issue related to the local reduction which is not the concern of this paper. We proved that on the distributed side, the term really always progresses and that no reduction loop is induced by the distributed features of ASP_{fun}. We can reformulate the preceding theorem:

Theorem 3.2 (Progress)

$\mathit{nocycle}(C) \wedge \vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \ \Rightarrow \mathit{isvalue}(s) \vee \exists \, C' \, . \, C \rightarrow_{\|} C'$

where C' *can be chosen to verify:* $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$.

By proving progress we also show that ASP_{fun} is dead-lock free: as any term that is not already a value must progress, this ensures the absence of dead-lock.

As configurations reachable from the initial configurations have no cycle, a variant of the progress theorem can be stated by replacing the *nocycle* hypothesis, by the reachability from a well-typed initial configuration:

Property 7 (Progress from initial configuration)

$initConf(a) \rightarrow_{\|}^{*} C \wedge \varnothing \vdash s : A \wedge \alpha[f_i \mapsto s :: Q, t] \in C \Rightarrow isvalue(s) \vee \exists C'. C \rightarrow_{\|} C'$
where C' *can be chosen to verify:* $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$.

4 Formalisation in Isabelle/HOL

Isabelle/HOL offers a classical higher order logic as a basis for the modelling of application logics. Inductive definitions and datatype definitions can be defined in a way close to programming language syntax and semantics. We adapted the semantics for the ς-calculus defined in [15] in order to use reduction contexts; we proved that both models are equivalent. As our purpose is to demonstrate the modelling of typing and distribution concerns, the operational semantics is simply based on DeBruijn indices. We are currently investigating the use of locally nameless techniques [3] instead.

4.1 Crucial Aspects of the Proofs

This section details some of the parts of the formalisation that seem the most important to us, it gives proof sketches and is not much coupled Isabelle/HOL.

Finiteness When considering language semantics we often implicitly assume finiteness of programs and configurations. In fact, the implicit assumption is worth mentioning: for programs it grants induction over the recursive datatype of ς-terms, and for configurations, it permits the assumption that there are always fresh activity and future names available. Our formalisation relies on this assumption. We show that initial configurations and configurations reduced from them are all finite.

Absence of Cycles Proving the absence of cycles (Theorem 2.2) required us several steps, we first defined a datatype for future or activity reference, and then specified the $knows_C$ and $knows_C^+$ relations defined in Section 2.5.2. In order to handle the proofs, we refine the $knows_C^+$ relation by remembering the list of intermediate activities: $r\ knows_{C(L)}^+\ r'$ iff $r\ knows_C^+\ r'$, passing by the references in L.

We first prove lemmas relating cycles, $knows_C^+$ and paths. E.g., if $r\ knows_{C(L)}^+\ r'$, and C' is obtained from C by just modifying the request corresponding to f_k; then $r\ knows_{C'(L)}^+\ r'$ provided $f_k \notin L$, and $f_k \neq r$.Consequently, it is sufficient to prove that no cycle is created by the modified activities and requests. We also show that when $r\ knows_{C(L)}^+\ r'$, L can be chosen to include neither r nor r'. The remainder of the proof is a long case study on the reduction rules that uses lemmas presented above, well formedness of the initial configuration, and shows that if there is a cycle in the obtained configuration, there was necessarily one in the original configuration.

Proving Progress Proving progress relies on a long case analysis on the reduction rule. We focus first on one crucial argument: how can the absence of free variable be ensured in order to communicate an object between two activities. Each request can be typed in an empty environment (for variables); thus it does not have any free variable, and thus each sub-term of a request that is not under a binder has no free variable. We prove that one can reduce at least the part of the request under the evaluation context F. Indeed, in F the term in the position of the hole has no

free variables: $\langle \Gamma_{act}, \Gamma_{fut} \rangle, \emptyset \vdash F[a] : A \ \Rightarrow \ noFV(a)$

Considering the other arguments of the proof, the absence of cycles ensures that an application of a REPLY rule cannot return a future value which is the future itself, in which case the configuration would be reducible to itself. This ensures that no live-lock exists because of the distributed semantics. Of course, the proof also massively uses the fact that well-typed configurations are well-formed.

4.2 Summary of the Formal Model

The formalisation of functional ASP is constructed as an extension of the base Isabelle/HOL theory for the ς-calculus [15]. The term type of the ς-calculus is represented by an Isabelle/HOL datatype definition called `dB`. In this datatype definition, objects are represented as finite maps `Obj (Label` $\Rightarrow_f$`dB) type`. We formalised finite maps in the first argument of `Obj` using the abstract concept of axiomatic type classes. It is crucial to have finite maps as a basic Isabelle/HOL type to be able to employ the recursive datatype construction here. The second argument of the constructor `Obj` is a type annotation. The type of configurations relies on partial functions expressed by the type constructor $\Rightarrow\!\!\!\!\!+$.

```
futmap = FutureRef ⇸ dB
configuration = ActivityRef ⇸ (futmap × dB)
```

The parallel semantics of ASP_{fun} is given as an inductive relation over this type of configurations encoding the reduction relation $\rightarrow_{\|}$ (see Table 2).

In our model we developed a simple mechanisation of a reduction context using datatype and an operator to "fill" the "hole" enabling a fairly natural notation of `E↑t` for $E[t]$ (remember this substitution is not "capture avoiding" contrarily to the variable substitution). We can illustrate the benefits of our context concept most directly by the representation of the configuration rule for requests.

```
request: ⟦ ∀ A ∈ dom C. fn ∉ dom(C .A A); C A' = Some(m',a');
           m'(fk) = Some(E↑(Call(ActRef B) li p); C B = Some(mb, a);A'≠ B; noFV p ⟧
    ⟹ C →|| C (A' ↦ (m' (fk ↦ E↑(FutRef(fn))), a')) (B ↦ (mb (fn ↦ (Call a li p)), a))
```

The theorem `progress_ASP_reach` below is a particular instance of the progress theorem employing the previous results that all reachable configurations are finite and have no cycles; it corresponds to Property 7.

```
theorem progress_ASP_reach:
 ⟦reachable C; ⊢C: CT; A∈ dom C; fi∈C.RA; ¬(isvalue  C.FA<fi>)⟧
    ⟹ ∃C'. (C →|| C') ∧ ( C'.FA<fi>≠C.FA<fi>∨ C.FA<fi> →ς C.FA<fi>)
```

Experience The entire development takes around 14000 lines of code. The most difficult and crucial step is certainly the definition of the right model for the calculus, its semantics, but also for the additional constructs used in intermediate lemmas. Even if the length and form of the proof is far from optimal, the development for formalising such a theory is really consequent; and it becomes difficult to keep a proof minimal and well-structured when it grows to several thousands of lines in length. Handling simplification steps in such a complex and rich theory becomes tricky. Additionally, making modular proofs for subject reduction and equivalent properties is difficult in a theorem prover because useful lemmas are tightly coupled with the numerous and complex hypotheses involved by the case analysis.

5 Related Works and Positioning

Actors [16] are widely used for distributed autonomous entities, and their interactions by messages. They are rather functional entities but their behaviour can be changed dynamically, giving them a state.

Agents and Artifacts with simpA, concentrating on the higher-level of modelling concurrent agent based systems, also presents a calculus [23]. Based on FeatherweightJava its agent concept resembles ASP_{fun}'s activities but the calculus has no type system and proofs. ASP_{fun} may be used to provide formal support.

Obliq [5] is based on the ς-calculus; it expresses both parallelism and mobility. It relies on threads communicating with a shared memory. Like in ASP_{fun}, calling a method on a remote object leads to a remote execution but this execution is performed by the original thread. Øjeblik, e.g. [4], a subset of Obliq, equally differs from ASP_{fun} by thread execution. The authors investigate safety of *surrogation* meaning that objects should behave the same independent of migration.

Jeffrey's distributed object calculus [17] is based on Gordon and Hankin's concurrent object calculus [13] extended with explicit locations. The main objective is to avoid configurations where one object at one location is being accessed by another. A type system enforces these restrictions. Because migrating objects can carry remote calls, in order to ensure subject-reduction, Jeffrey introduces serialisable objects, which are non-imperative. Compared to our calculus the most decisive difference is that *activities abstract away the notion of location*, and are remotely accessible thanks to a request queue. The concept of futures somehow explicitly supports mobility and serialisation.

Futures have been studied several times in the programming languages literature, originally appearing in Multilisp [14] and ABCL [25].

$\lambda(fut)$ is a concurrent lambda calculus with futures. It features non determinism primitives (cells and handles). In [20], the authors define a semantics for this calculus, and two type systems. They use futures with explicit creation point in the context of λ-calculus; much in the same spirit as in Multilisp. Alice [19] is an ML-like language that can be considered as an implementation of $\lambda(fut)$.

In [9], the authors provide a language with futures that features "uniform multiactive objects": roughly each method invocation is asynchronous because each object is active. Thus, compared to ASP_{fun}, the calculus has no *Active* primitive. Each object has several current threads, but only one is active at each moment. Their futures are also explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties. This work also formalises the Creol language [18].

ASP [8] is an imperative distributed object calculus; it is based on the $\varsigma_{\mathbf{imp}}$-calculus [1]. It features asynchronous method calls and transparent first class futures, no instruction deals directly with futures. Activities in ASP are monothreaded: one request is served at each moment, and a primitive can be used to select the request to serve. Some confluence properties for ASP have been studied in [8,7]. ProActive [6] is an implementation of the ASP calculus.

More recently, in [11], the authors suggest a communication model, called *Am-*

bientTalk, based on an actor-like language, and adapted to loosely coupled small devices communicating over an ad-hoc network. The communication model is quite similar to the ASP calculus, but with queues for message sending, handlers invoked asynchronously, and automatic asynchronous calls on futures. The resulting programming model is slightly different from ASP and ASP_{fun} because there is no blocking synchronisation in AmbientTalk. In AmbientTalk, the flow of control might be difficult to understand for complex applications, because one can never ensure that a future has been returned at a precise point of the program. AmbientTalk should be dead-lock free, but unfortunately as no formalisation of the language has been proposed to our knowledge, this has not been formally proved. Our framework could be relatively easily adapted to prove the absence of dead-locks in AmbientTalk by transferring our progress property.

Futures have been formalised in several settings, generally functional-based [20,9,12]; those developments rely on explicit creation of futures by thread creation primitives, in a concurrent but not distributed setting. They are getting more and more used in real life languages; for example, explicitly created futures are also featured by the `java.util.concurrency` library. ASP's [7,8] particularities are: distribution, *absence of shared memory*, and *transparent futures*, i.e. futures created *transparently* upon a remote method invocation.

This paper presented a *distributed evaluation* of the *functional* ς-calculus, using *transparent* futures and active objects. It can also be seen as a study of the functional fragment of ASP. That is why we consider this calculus as complementary to the preceding ones. Futures can be passed around between different locations in a much transparent way; thanks to its functional nature and its type-system, this calculus ensures progress. Progress for active objects means that evaluation cannot lead to dead-locks. ASP_{fun} is called "functional" because objects are immutable. In ASP_{fun}, activities are organised in an actor-like manner, that is why we consider our language as a form of "functional actors", or "functional active objects". The main novelty of ASP_{fun} is that it is simple enough to allow for a mechanised specification and mechanised proofs of typing properties.

6 Conclusion

We presented a functional calculus for communicating objects and its type system. This work can be seen both as a distributed version of ς-calculus and as an investigation on the functional fragment of ASP. It has been entirely formalised and proved in the Isabelle theorem prover. The functional nature of ASP_{fun} should make it influence directly stateless distributed systems like skeleton programming [10]. Our approach could be extended to study frameworks where most of the services are stateless, and the state-full operations can be isolated (access to a database), e.g. workflows and SOA. Our formalisation in a theorem prover should also impact other developments in the domain of semantics for distributed languages.

A calculus of communicating objects The calculus is an extension of ς-calculus with only the minimal concepts for defining active objects and futures. Syntactically the extension only requires one new primitive: *Active* creates a new activity from a

term. The absence of side-effects and the guarantee of progress make the program easy to reason about and easy to parallelise. ASP_{fun} is distributed in the same sense as ASP: it enables parallel evaluation of activities while being oblivious about the concrete locations in which the execution of these activities takes place. The actual deployment is not part of the programming language, and should be provided by an application deployer rather than by the application programmer.

Well-formed terms and absence of cycle We proved that ASP_{fun} semantics is correct: no reference to non-existing activities or futures can be created by the reduction. Also, no cycle of future or activity references can be created. Thus, starting from an initial configuration, we always reach a well-formed configuration without cycle.

A type system for functional active objects We extended the simple type system for ς-calculus: *Active* returns an object of the same type as its parameter; activities are typed like their active objects; and futures are typed like the request calculating their value. The type system ensures progress and preservation. Preservation states that the types are not changed during execution. Progress states that a program does not get stuck. In ASP_{fun}, this is due to the following facts:

- The type system plus the subject reduction property ensure that all method calls will access an existing method.
- Well-formedness ensures that all accessed activities and futures exist.
- Absence of cycles prevents cycles of mutually waiting synchronisations.
- As partially evaluated futures can be replied, any chosen request can be reduced.
- All operations are defined for both local and active objects avoiding "syntactical" dead-locks like updating a method of an activity.
- Terms under *evaluation* contexts can be safely communicated between activities.

Can we find a better progress property? Let us analyse the limitations of the progress property. First, though a reduction is possible, the reduced term can sometimes be identical to the original one. The absence of cycle ensures that such a situation can only occur in the local semantics. This is inherent to the ς-calculus and is out of the scope of this paper. Second, the reduction can occur in any chosen request but not at any chosen place. Indeed objects can only be sent between activities if they do not have free variables that otherwise would escape their binder.

Alternative semantics It is possible to restrict the REPLY rule to only return completely evaluated futures. Then, if one picks a request, there is no more any guarantee that it can evolve, but the absence of cycle ensures that *some request in the configuration can always be reduced.* Some intermediate reductions have to be added to guarantee the progress property. Finally, *returning only completely evaluated futures leads to a more efficient semantics, and still ensures a form of progress.*

References

[1] M. Abadi and L. Cardelli. "A Theory of Objects". Springer, New York, 1996.

[2] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] B. Aydemir, A, Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *Princ. of Programming Languages, POPL'08*, ACM, 2008.

[4] S. Briais and U. Nestmann. Mobile objects "must" move safely. *Formal Methods for Open Object-Based Distributed Systems, FMOODS'02.* Kluwer Academic Publishers, 2002.

[5] L. Cardelli. A language with distributed scope. In *POPL 1995*. ACM.

[6] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[7] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. *Princ. of Programming Languages, POPL'04*, pages 123–134. ACM Press, 2004.

[8] D. Caromel and L. Henrio. "A Theory of Distributed Objects". Springer, 2005.

[9] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. *Eur. Symposium on Programming, ESOP'07*. LNCS **4421**:316–330, Springer, 2007.

[10] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.

[11] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. *ECOOP'06*. LNCS **4067**, Springer.

[12] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[13] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. *Proceedings FST+TCS'97*, LNCS. Springer, 1997.

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

[15] L. Henrio and F. Kammüller. A mechanized model of the theory of objects. *Formal Methods for Open Object-Based Distributed Systems,* . LNCS **4468** Springer, 2007.

[16] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. IJCAI, 1973.

[17] A. Jeffrey. A distributed object calculus. *ACM SIGPLAN Workshop on Foundations of Object Oriented Languages, FOOL'00*, 2000.

[18] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2), 2006.

[19] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Conference on Mathematical Foundations of Programming Semantics*, ENTCS, 2007.

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

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – a proof assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[22] B. C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

[23] A. Ricci, M. Viroli, and M. Cimadamore. Prototyping Concurrent Systems with Agents and Artifacts: Framework and Core Calculus. *Foundations of Coordination Languages and Software Architectures, FOCLASA'07.* ENTCS, Elsevier 2007.

[24] A. Schmitt. Safe dynamic binding in the Join calculus. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Proceedings of IFIP TCS 2002*, Kluwer.

[25] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. *Object-Oriented Concurrent Programming*. MIT Press, 1987.