Information and Computation 207 (2009) 459–495

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Asynchronous sequential processes

Denis Caromel, Ludovic Henrio*, Bernard Paul Serpette

CNRS—I3S—UNS—INRIA Sophia-Antipolis, 2004, route des Lucioles, F-06902 Sophia-Antipolis, France

ARTICLE INFO

Article history:
Received 16 December 2004
Revised 14 September 2008
Available online 30 December 2008

Keywords:
Object calculus
Concurrency
Distribution
Parallelism
Object-oriented languages
Components
Determinism
Futures

ABSTRACT

Deterministic behavior for parallel and distributed computation is rather difficult to ensure. To reach that goal, many formal calculi, languages, and techniques with well-defined semantics have been proposed in the past. But none of them focused on an imperative object calculus with asynchronous communications and futures. In this article, an object calculus, *Asynchronous Sequential Processes* (ASP), is defined, with its semantics. We prove also confluence properties for the ASP calculus. ASPs main characteristics are asynchronous communications with futures, and sequential execution within each process. This paper provides a very general and dynamic property ensuring confluence. Further, more specific and static properties are derived. Additionally, we present a formalization of distributed components based on ASP, and show how such components are used to statically ensure determinacy. This paper can also be seen as a formalization of the concept of futures in a distributed object setting.

1. Introduction

Distributed object systems are generally based on remote method calls between objects and rely on a concept of threads rather orthogonal to the object structure. This article formalizes a way of unifying the notion of threads and objects: each object belongs to a single *activity* (we use "activity" rather than "process" for expressing the unit of distribution) and each activity is associated a single thread. Then, activities communicate by *asynchronous method calls* allowing both sender and callee to perform operations between the request sending and its treatment, thus increasing concurrency. *Futures* are introduced to represent awaited results of such asynchronous calls; one of the main contribution of this paper is the *formalization of the notion of futures in the context of distributed objects*.

Confluence properties simplify programming as they avoid having to study every possible interleaving of instructions and communications. They also ease the development of languages and middleware, by authorizing some optimizations or even allowing the implementation of some mechanisms, e.g. [1]. Confluence of various calculi, languages, or programs have been studied in the past, using different techniques. Linear channels in π-calculus [2,3], non interference properties in shared memory systems [4], Process Networks [5] or Jones' technique for creating deterministic concurrency in $\pi o \beta \lambda$ [1] are typical examples. All these works provide a characterization of confluence in a specific domain but none of them deals with a concurrent, imperative, object calculus with asynchronous communications.

The study of confluence has two objectives. First it characterizes deterministic programs, because it is often needed that a program behaves deterministically; one could not imagine a non-deterministic result to a binary or a prime number search. But more importantly we aim at characterizing the possibilities of interference, and characterizing minimally an execution. Such a minimal characterization is useful in several contexts, like for example debugging: a program is easily characterized and reproducible; fault tolerance: the status of an ongoing computation is known in a minimal way, and thus easily stored

* Corresponding author.
E-mail addresses: denis.caromel@inria.fr (D. Caromel), ludovic.henrio@inria.fr (L. Henrio), bernard.serpette@inria.fr (B.P. Serpette).

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.12.004

and re-executed [6]; and program analysis: one would know exactly which interleaving of actions are significant for the program execution.

We design a calculus named *ASP*: *Asynchronous Sequential Processes* and formalize some properties of determinism on this calculus. ASP models an object-oriented language with asynchronous communications and futures, and sequential execution within each parallel process. We start from a purely sequential and classical object calculus (**imp**$_\varsigma$-calculus) [7] and extend it with two parallel constructors: *Active* and *Serve*. *Active* turns a standard object into an active one, executing in parallel and serving requests in the order specified by the *Serve* operator. Parallel composition of activities comes as a consequence of object activation and only exists at runtime. Method calls on active objects are asynchronous: their results are represented by *futures* until the corresponding response is returned. Automatic synchronization of activities comes from *wait-by-necessity* [8]: a wait automatically occurs upon a strict operation (e.g. a method call) on a future.

Innovative aspects of ASP calculus include a formalization of the following features in an object oriented context: futures together with asynchronous method calls, data-driven synchronization, and unification of the notions of processes and objects. Moreover confluence properties seems to be novel in such a distributed objects framework. Our key property states that the *execution is only determined by the order of activities sending requests to a given activity*; asynchronous replies can occur in any order without observable consequence. This work is more general and strongly related to the Process Networks of Kahn [5]. This work also identifies sets of programs that behave deterministically, including a characterization of deterministic components.

On the practical side, the ASP-calculus model is implemented as a Java middleware, ProActive [9], allowing parallel and distributed programming. The properties shown in this paper are useful both in the design of ProActive, e.g., because any strategy for returning a future is equivalent and can be implemented in the middleware, and for the programmer as he only has to ensure that the result will be calculated.

The passing of futures between activities, both as method parameters and as method results is an important feature of ASP. As futures can proliferate, from a practical point of view, a strategy must be specified to choose when and how a request result should be sent back to replace the references to the corresponding futures. The ASP calculus captures all the possible update strategies, and thus properties are valid for all of them. Moreover, *a given activity is insensitive to the moment when a result comes back*. This is a powerful characteristic of the confluence property we exhibit.

This paper is organized as follows. Section 2 presents the sequential part of ASP, which is strongly based on the **imp**$_\varsigma$-calculus. Section 3 informally introduces the ASP calculus, its principles and a few examples. Section 4 defines the semantics of ASP and some intrinsic properties, Section 5 presents its confluence properties, and Section 6 show how to build deterministic components in ASP. Section 7 compares ASP with other concurrent calculi and their confluence properties. An appendix gives some technical details of the proofs and the equivalence relation. This work was partly and briefly presented in [10], more details on proofs and extensions of the calculus can also be found in [11].

2. Sequential calculus

2.1. Syntax

We start from an imperative sequential object calculus strongly inspired from the one of Abadi and Cardelli. The purpose of this calculus is to serve as the sequential core on top of which the parallel calculus will be defined. The only characteristics that have been changed in the ASP sequential calculus, relatively to the **imp**$_\varsigma$-calculus, are the following:

- Because arguments passed to active objects methods will play a particular role, we added a parameter to every method like in [12]. In addition to the self argument of methods, noted x_j and representing the object on which the method is invoked, we add an argument representing a parameter object to be sent to the method, noted y_j.
- We do not include method update in our calculus because we do not find it necessary. It is still possible to express updatable methods in our calculus.
- As in [13], during the reduction, *locations* (reference to objects in a store) can be part of terms. Locations do not appear in user syntax.

In this article, we will distinguish *user syntax/user terms* corresponding to programs and *runtime syntax/runtime terms* that are generated when evaluating programs and will have a different syntax.

The abstract syntax of the ASP calculus is given in Fig. 1, a, b as defined in the figure range over *terms*, $l_i^{i\in 1..n}$ range over field names; $m_j^{j\in 1..m}$ are method names; ς is a binder for method parameters; and a location ι is an entry in the store defined below. *let* $x = a$ *in* b, sequence $a; b$,[1] lambda expressions, and methods with zero or several arguments can be easily expressed in our calculus and are used in the following.

[1] Let $x = a \text{ in } b \triangleq [m = \varsigma(\_, x)b].m(a)$ $\qquad a; b \triangleq [m = \varsigma(\_, \_)b].m(a)$.

$$
\begin{array}{lll}
a, b \in L ::= x & & \text{variable,} \\
& \mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} & \text{object definition,} \\
& \mid a.l_i & \text{field access,} \\
& \mid a.l_i := b & \text{field update,} \\
& \mid a.m_j(b) & \text{method call,} \\
& \mid clone(a) & \text{shallow copy,} \\
& \mid \iota & \text{location (not in user syntax)}
\end{array}
$$

Fig. 1. Sequential syntax.

2.2. Semantic structures

Free and bound variables are defined as usual: ς is the only binder, it binds two variables at the same time. Let $locs(a)$ denote the set of locations appearing in a, and $fv(a)$ the set of variables occurring free in a. *User terms* are *closed* terms without location, where a is closed iff $(fv(a)=\emptyset \wedge locs(a)=\emptyset)$. Locations appear when putting objects in the store.

Substitution. The substitution of b by c in a is written: $a\{\!\{ b \leftarrow c \}\!\}$. Substitutions are denoted by $\theta ::= \{\!\{ b_i \leftarrow c_i \}\!\}^{i\in 1..n}$. In the semantics, substitution is applied in a classical way: it replaces a formal parameter x by the locations of the arguments without replacing inside binders: expressions under $\varsigma(x,z)$ or $\varsigma(z,x)$ are unchanged. Usually, substitution must avoid variable capture, but here variables are always substituted by locations, which avoids this problem.

Let $\equiv$ be the equality modulo renaming of locations (substitution of locations by locations) provided the renaming is injective. In other words, $\equiv$ is the equality modulo "alpha-conversion" of locations.

Store. Reduced objects are objects with all fields reduced to a location:

$$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$$

A *store* σ is a finite map from locations to reduced objects: $\sigma ::= \{\iota_i \mapsto o_i\}$. The domain of σ, $dom(\sigma)$, is the set of locations defined by σ.

Let $\sigma :: \sigma'$ append two stores with disjoint locations. When the domains are not disjoint, $\sigma + \sigma'$ updates the values defined in σ' by those defined in σ. It is defined on $dom(\sigma) \cup dom(\sigma')$ by

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

Note that $\sigma :: \sigma'$ is equal to $\sigma + \sigma'$ but specifies that $dom(\sigma) \cap dom(\sigma') = \emptyset$.

Configuration. Let a sequential *configuration* (a, σ) be a pair (expression, store). We denote by $\vdash (a,\sigma)\text{OK}$ a well-formed configuration, i.e. a configuration that has no free variable and defines every location it uses:

Definition 1 (*Well-formed sequential configuration*)**.**

$$
\vdash (a,\sigma)\text{OK} \Leftrightarrow \begin{cases} locs(a) \subseteq dom(\sigma) \ \wedge \ fv(a) = \emptyset \ \wedge \\ \forall \iota \in dom(\sigma),\ locs(\sigma(\iota)) \subseteq dom(\sigma) \ \wedge \ fv(\sigma(\iota)) = \emptyset \end{cases}
$$

2.3. Reduction

We first define reduction contexts $\mathscr{R}$ as expressions with a unique hole $\bullet$:

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

$\mathscr{R}[a]$ denotes the syntactic replacement of the hole $\bullet$ by a in $\mathscr{R}$:
$\mathscr{R}[a] = \mathscr{R}\{\!\{ \bullet \leftarrow a \}\!\}$. This replacement is however different from classical substitution because it allows variables in a to be captured by the binders in $\mathscr{R}$.

Table 1
Sequential reduction

$$\text{STOREALLOC}\quad \frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o],\sigma) \rightarrow_S (\mathcal{R}[\iota],\{\iota \rightarrow o\} :: \sigma)}$$

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

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

$$\text{UPDATE}\quad \frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j,y_j)a_j]_{j\in 1..m}^{i\in 1..n} \qquad k \in 1..n \qquad o' = [l_i = \iota_i; l_k = \iota'; l_{k'} = \iota_{k'}; m_j = \varsigma(x_j,y_j)a_j]_{j\in 1..m}^{i\in 1..k-1,k'\in k+1..n}}{(\mathcal{R}[\iota.l_k := \iota'],\sigma) \rightarrow_S (\mathcal{R}[\iota],\{\iota \rightarrow o'\} + \sigma)}$$

$$\text{CLONE}\quad \frac{\iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)],\sigma) \rightarrow_S (\mathcal{R}[\iota'],\{\iota' \rightarrow \sigma(\iota)\} :: \sigma)}$$

Table 1 defines a small-step, substitution-based operational semantics for our sequential calculus. It gives reduction rules that are applied at the point represented by the occurrence of $\bullet$ in $\mathscr{R}$; rules are object creation (STOREALLOC), field access (FIELD), method invocation (INVOKE), field update (UPDATE) and shallow clone (CLONE). This semantics is very similar to the one of [13]. To evaluate a user term a, an *initial configuration* $(a, \emptyset)$ is created. It contains the user term a, and the empty store.

Sequential reduction has two interesting properties. First, *reduction preserves well-formedness*. Second, the sequential reduction is deterministic up to the choice of freshly allocated locations:

Property 1 (Determinism)**.** *Let c, d and d′ be three configurations:*

$$c \rightarrow_S d \ \wedge\ c \rightarrow_S d' \ \Rightarrow\ d \equiv d'$$

3. Parallel calculus

This section defines a parallel calculus which is based on *activities*. Each ASP object is either active or passive. There is one active object at the root of each activity. Activities run in parallel, and interact only through asynchronous method calls. Synchronization is due to a data-driven synchronization on the result of an asynchronous method call: wait-by-necessity.

3.1. Principles

An *activity* is a process associated with a set of objects put in a store. Among these objects, one is *active*; every remote method call sent to the activity is actually sent to this object, such remote invocations are called *requests*. An activity also contains the pending requests (requests that have been received and should be served later) and the values of the results for finished requests. *Passive* (i.e., non active) objects are only referenced by objects belonging to the same activity but any object can reference active objects and futures: there is no shared memory except references to active objects and futures. Activities are single threaded, which is crucial for ASP properties.

The activation of an object *Active*(a, m) creates a new activity whose active object is a copy of a. *Serve*$(m_1..m_n)$ performs a blocking service of requests received by the current active object. Unlike many other concurrent calculi based on the ς-calculus, in ASP, the requests are not executed by the thread that performs the method call.

A *future* is an identifier representing the result of a method call to an active object, this allows method invocations on active objects to be asynchronous. Execution will be blocked when we try to perform a strict operation (e.g. accessing a field of the objects) on a future. Such blocking states are called *wait-by-necessity*. When the method is finished, the result is returned, and if the execution was blocked in a wait-by-necessity, it can continue. We call *future reference*, a reference to a future, i.e. a reference to a remote method invocation for which the result has not yet been returned, and *future value*, the result corresponding to a future once it has been computed.

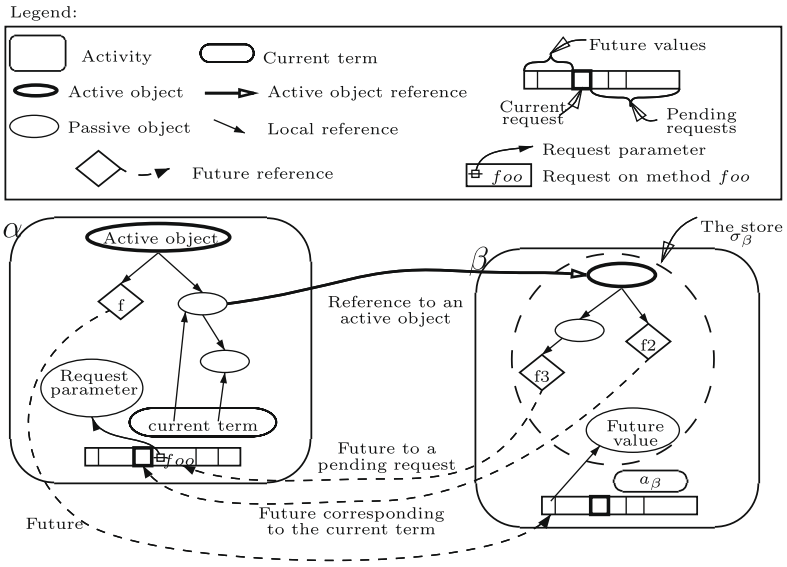

Fig. 2. Example of a parallel configuration.

3.2. User syntax

We extend the sequential calculus by adding the possibility to create active objects and to serve requests:

$a, b \in L ::= \ldots$	sequential calculus term
$\mid Active(a, m_j)$	activates object, m_j is the activity method
$\mid Serve(m_1, .., m_n)^{n>0}$	starts the service of a request,
$\mid a \Uparrow f, b$	a with continuation f, b (not in user syntax)

3.3. Informal semantics

In every activity α, a *current term* a_α represents the current computation. Every activity has its own *store* σ_α which contains one active and many passive objects. It contains also a *pending request queue* which stores the pending method calls and a *future list* which stores the results of finished requests. Fig. 2 gives an illustration of a configuration consisting of two activities. It contains three references to futures (one calculated, one current, and one pending). The active objects are bold ellipses; futures references are diamonds; future values, current future and pending requests are merged in the bottom rectangles: calculated future values are on the left, current futures are represented by a bold rectangle and pending requests are on the right. Continuations will not appear in our representation.

Activities. The *Active* operator $Active(a, m_j)$ creates a new activity α with the object a at his root, a is called the *active* object and acts as the master object of the new activity. The object a is copied with all its dependencies[2] (deep copy) into a new activity. The second argument to the *Active* operator is the name of a method which will be called as soon as the object is activated. This method with no argument is called the *service method* as it should specify the order in which the requests are served. A *FIFO* service serves the requests in the order they arrived in the activity; in that case, in Fig. 2, the current request (bold square) progresses from left to right in the queue. When the service method terminates, no more request is treated. Remote references to the active object of activity α will be denoted by, the generalized reference $AO(\alpha)$ in the runtime syntax.

A field access on an active object reference is irreversibly stuck. However, the fields of an active object can be manipulated by its own activity, and accessed remotely via accessors.

Requests. Communications between activities are due to method calls on active objects and returns of corresponding results. A method call on an active object consists in atomically adding an entry to the *pending requests* of the callee, and associating a *future* to the response (in the ProActive implementation, the request sender waits for an acknowledgment before continuing

[2] To prevent remote references to passive objects.

its execution). Arguments of requests and values of futures are deeply copied when they are transmitted between activities. Active objects are transmitted with a reference semantics.

Serving requests. The primitive *Serve* stops the activity until a request matching its argument is found in the pending requests, i.e. a request on one of the method specified as parameter of the *Serve* primitive. A service can be performed at any time, including while serving another request. $\Uparrow$ is runtime syntax needed to formulate the operational semantics, it is used to save the continuation[3] of the request we are currently serving while we serve another one. Note that with such a mechanism there are several requests being served at the same time, except if *Serve* operations are only performed by the top level activity, i.e., no *Serve* is performed while a request is being served.

When the execution of a request is finished, the corresponding future is associated with the calculated value. Then, the execution restores the continuation that had been stored when the service started.

Futures. Futures are generalized references that represent the result of an asynchronous method call. This result may be unavailable if it is not yet calculated or not yet sent. The result is the *future value*. We call *future update* the operation consisting in sending the calculated value for a future and replacing a reference to the future by this value. Inside each activity, the *future list* maps futures to their values within the activities that computed them. A future value is called *partial* if its dependencies contain future references.

An operation on an object is *strict* if it needs to access the content of the object: field and method access, update, clone. Transmitting an object to another activity is not a strict operation. Futures can be manipulated while we do not perform strict operations on them. A *wait-by-necessity* occurs when a strict operation occurs on a future: the activity is blocked until the future is updated. In Fig. 2, futures f_2 and f_3 denote pointers to not yet computed requests while f points to a future value computed but not yet updated.

3.4. Future update strategies

As futures can be safely manipulated and transmitted between activities, references to futures can proliferate. Different strategies can be implemented for returning the value of a future. In our semantics, every reference to a future can be replaced by the future value (partial or complete) at any time. Thus, we capture all the possible future update strategies. Specifying a strategy would restrict the possible reductions but could simplify and optimize the execution. For example, an eager strategy would send a result as soon as its value has been calculated; the future list would become useless but this would necessitate to contact every activity containing a reference to the future at once. The opposite strategy would consist in returning only complete results and forbidding the usage of futures as parameter of method call; such a strategy leads to many stuck and even deadlocked configurations. These two strategies have been implemented in ProActive.

3.5. Example: sieve of Eratosthenes

Remember the Sieve of Eratosthenes finds prime numbers by applying successive filters: when a prime number n is found, a new filter is created such that every number that is divisible by n will be rejected by the filter. If all filters for prime numbers strictly smaller than n have been created and n is accepted by all these filters then n is prime. This section translates in ASP the distributed Sieve of Eratosthenes described in [14] for Process Networks. In Process Networks, the sieve was performed by several processes linked by *channels* (with operations *get* and *put*), a process for each prime number. Each process gets numbers from the process associated to the previous prime number, and pushes values to the one associated with the next prime. We applied the same methodology and created one activity per prime number found. We first considered that the communications comes from the activity that performs a *get* on a *channel* to the one that performs a *put* on the same *channel* and replace such Process Networks communication by a call to a request *get* (see Fig. 4). *Repeat* performs an infinite loop and will be defined later on. Fig. 3 defines a "pull" Sieve of Eratosthenes in ASP, depicted in Fig. 4.

The *Integer* object generates all integers. There is one *Sieve* object for each prime number. It returns the next integer given by its parent sieve that is not divisible by the prime number n. The *Sift* object manages the whole computation. We supposed *Display* is an object to which the list of primes have to be sent. When a new prime is found, a new *Sieve* is inserted between the *Sift* and the former last *Sieve*. Note that the only strict operations on integers n are the ones performed by the *Display* object and MOD. In this example, every object always replies to a *get* request: prime numbers are pulled one after the other one by the sift object. Thus, the program will be evaluated sequentially and the pipelining that could be performed on the example of Kahn and MacQueen cannot occur here. The following implementation of the sieve allows such pipelining; see Fig. 5.

A push version. Fig. 6 defines a "push" Sieve of Eratosthenes in ASP, depicted in Fig. 5. Integers are put successively into a pipeline of sieve objects which finally send the next prime to a *Display* objects.

[3] Here, continuation stands for the state of the interrupted service, that is to say the instructions still to be executed to finish the service.

$$
\begin{array}{l}
let\ Integer = Active([n = 1; get = \varsigma(s, \_)(s.n := s.n + 1; s.n)], \emptyset)\ in \\
let\ Sieve = [parent = [], prime = 0; init{=}\varsigma(s, par)s.parent := par, \\
\qquad get = \varsigma(s, \_)\ let\ n{=}parent.get()\ in \\
\qquad\qquad if(n \bmod s.prime \neq 0)\ then\ n\ else\ s.get()]\ in \\
let\ Sift = [source = Integer; \\
\qquad act{=}\varsigma(s, \_)Repeat(let\ n = source.get()\ in \\
\qquad\qquad Display.send(n);\ Sieve.prime := n; \\
\qquad\qquad s.source := Active(Sieve.init(s.source)))]\ in \\
Active(Sift, act)
\end{array}
$$

Fig. 3. Example: Sieve of Eratosthenes (pull).

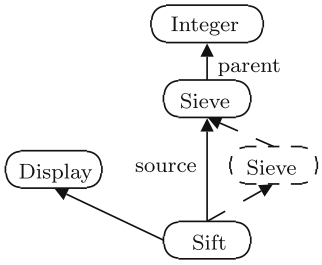

Fig. 4. Sieve of Eratosthenes (pull).

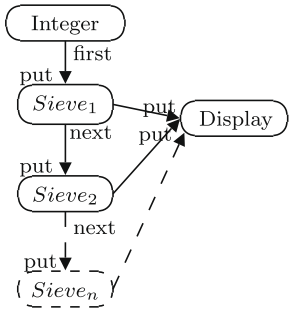

Fig. 5. Sieve of Eratosthenes (push).

4. Parallel semantics and properties

4.1. Structure of parallel activities

We assume that we have three distinct name spaces: activities ($\alpha, \beta \in Act$), locations (ι) and futures (f, f_i)[4]. Activities have unique names, and locations are local to an activity. A future will be characterized by its identifier f_i, the source activity α and the destination activity β of the corresponding request: $f_i^{\alpha\to\beta}$. Future identifiers f_i must be chosen such that $f_i^{\gamma\to\alpha}$ is unique. For example, the future identifiers can be unique for each destination activity.

A parallel configuration is a set of activities:

$$P, Q ::= \alpha[a; \sigma; \iota; F; R; f] \parallel \beta[\ldots] \parallel \ldots$$

We will denote by $\alpha \in P$ the fact that an activity named α belongs to the configuration P. Each activity $\alpha[a; \sigma; \iota; F; R; f]$ is characterized by:

- a *current term* $a = b \Uparrow f_i^{\gamma\to\alpha}, c$ to be reduced. a contains the terms corresponding to the different requests being treated, services are separated by $\Uparrow$. The left part b is the term currently evaluated, the right one $f_i^{\gamma\to\alpha}, c$ is the continuation:

[4] There are already name spaces for the fields, methods and variables identifiers which appear in the user terms and are not created dynamically.

$$
\begin{array}{l}
let\ Sieve = [N=0, prime = 0; next=[]; put = \varsigma(s,n)\ s.N := n,\\
\quad act = \varsigma(s,\_)\ Serve(put); Display.send(s.N);\\
\qquad s.prime := s.N; s.next := Active(s, act);\\
\qquad Repeat(Serve(put); if\ (s.N \bmod s.prime \neq 0)\ then\ s.next.put(s.N))]\ in\\
let\ Integer = [n = 1; first = Active(Sieve, act);\\
\qquad act = \varsigma(s,\_)\ Repeat(s.n := s.n + 1; s.first.put(s.n))]\ in\\
\ Active(Integer, act)
\end{array}
$$

Fig. 6. Example: sieve of Eratosthenes (push).

respectively the future and current term of a service that has been stopped before the end of its execution, because a new *Serve* instruction was encountered during this request service. When the current service will be finished, the current term c will be restored, and its result associated to the future $f_i^{\gamma\to\alpha}$. c can also contain continuations;

- a *store* σ containing all objects of the activity α;
- an *active object location* ι, the location of the active object of activity α. $\sigma(\iota)$ is the active object of α which will handle requests;
- a mapping *future values* associating, for each served request, a location ι to its future f_i: $F = \{f_i \mapsto \iota\}$. In fact, the value of future f_i is the part of store that has ι for root (the deep copy of $\sigma(\iota)$);
- a list of *pending requests* $R = \{[m_j; \iota; f_i^{\gamma\to\alpha}]\}$;
- a *current future* f, the future associated with the request currently served; more precisely, if the current term is $a \Uparrow f_i^{\gamma\to\alpha}, b$ then f will be the future associated with the value computed by a.

Empty parts of activities will be denoted by $\emptyset$: $\emptyset$ can be an empty list (futures or requests), an empty current term (no activity), or an empty current future (when no request is currently treated). When necessary we will denote by σ_α the store of activity α and similarly for the other components of the activities. Moreover, M denotes a finite and non-empty set of method labels.

$$M = \{m_1, \ldots, m_n\}^{\,n>0}$$

A request can be seen as the "reification" of a method call [15]. Each request $r ::= [m_j; \iota; f_i^{\gamma\to\alpha}]$ consists of

- the name of the *target method*: m_j,
- the location of the *argument* passed to the request: ι,
- the *future* identifier which will be associated to the result: $f_i^{\gamma\to\alpha}$.

We will denote by :: the concatenation of request queues. Consequently, $R :: r$ adds a request r at the end of the request queue R. and $R' :: r :: R$ matches a queue containing the request r. Similarly, $F :: \{f_i \mapsto \iota\}$ adds a new future association to the set of future values.

In the store, one has:

$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j) a_j]_{j\in 1..m}^{i\in 1..n}$	reduced object
$\mid AO(\alpha)$	active object reference
$\mid fut(f_i^{\alpha\to\beta})$	future reference (proxy)

$fut(f_i^{\alpha\to\beta})$ references the future $f_i^{\alpha\to\beta}$. $AO(\alpha)$ references the active object in activity α. $AO(\alpha)$ and $fut(f_i^{\alpha\to\beta})$ act as "proxy" to a remote activity or to a future object. Note that, when a reference to a future appears in an activity, the activity that may know the corresponding value can easily be contacted because it is encoded in the future reference: β in $f_i^{\alpha\to\beta}$.

The runtime syntax guarantees that there are no shared references in ASP except futures and active objects; moreover active objects are only accessible through asynchronous method calls and future values are immutable. Such a structure is crucial to ensure ASP properties.

4.2. Parallel reduction

First, object activation and continuations are added to reduction contexts:

$$\mathscr{R} ::= \ldots \mid Active(\mathscr{R}) \mid \mathscr{R} \Uparrow f, a$$

4.2.1. Deep copy

We define here operators on the store that will be used in the semantics. The operator $copy(\iota, \sigma)$ creates a store containing the deep copy of $\sigma(\iota)$. $copy(\iota, \sigma)$ is the smallest store satisfying the rules of Table 2. In Table 2 the first two rules specify the domain of the deep copy: recursively, all locations referenced by ι; and the last one states that the codomain is the same in

Table 2
Deep copy

$$\iota \in dom(copy(\iota,\sigma))$$

$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow locs(\sigma(\iota')) \subseteq dom(copy(\iota,\sigma))$$

$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow copy(\iota,\sigma)(\iota') = \sigma(\iota')$$

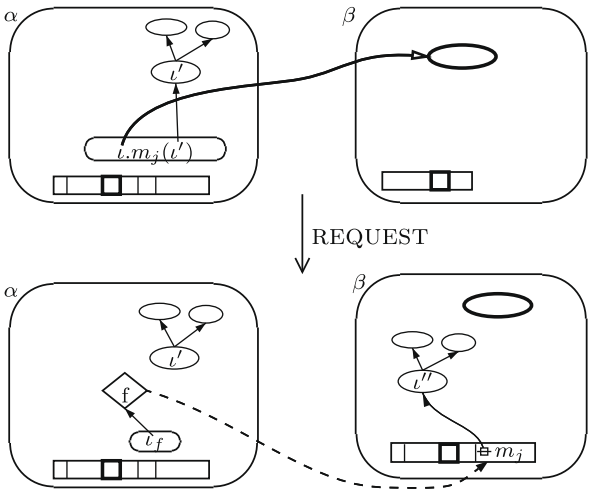

Fig. 7. REQUEST.

the copied and the original store. The deep copy of a future or an active object reference is the reference itself: deep copy does not follow global references. The part of store formed by the deep copy is independent: $\vdash (\iota,\sigma)\text{OK} \Rightarrow \vdash (\iota, copy(\iota,\sigma))\text{OK}$.

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

$$Merge(\iota,\sigma,\sigma') = \sigma'\theta + \sigma$$
$$\text{where } \theta = \{\!\{\, \iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \setminus \{\iota\}, \iota'' \text{ fresh} \,\}\!\}$$

The following operator adds the part of σ reachable from the location ι at the location ι' of σ' avoiding collision of locations; only ι' can be updated:

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

4.2.2. Reduction rules

Table 3 describes the reduction rules corresponding to the small step semantics of the parallel calculus. Note that LOCAL, NEWSERVICE and ENDSERVICE are local rules involving a single activity. Here is a short description of these rules:

LOCAL inside each activity, a local reduction can occur following the rules of Table 1. Only one sequential rule needs a slight modification: cloning a future is considered as a strict operation in order to ensure determinism: the CLONE rule applied to a future is stuck. To summarize, rules FIELD, INVOKE, UPDATE, CLONE are stuck when the target location is a future reference. However, REPLY may transform a future reference into a reachable object. FIELD, UPDATE, and CLONE are stuck when the target location is activity reference, but REQUEST allows to invoke an active object method.

NEWACT activates an object. A new activity γ is created containing the deep copy of the object $\sigma(\iota)$ and empty pending requests and future values. A reference to the created activity $AO(\gamma)$ is created in α. Remark that other references to ι in α are still pointing to the passive object. m_j specifies the method run initially by the active object. It is a method with no argument that is executed upon object activation and should perform *Serve* operations.

REQUEST sends a new request from activity α to activity β. Fig. 7 illustrates the application of the REQUEST rule, using the same variables as in the rule. A new future $f_i^{\alpha\to\beta}$ is created to represent the result of the request. α stores a reference to this future

Table 3
Parallel Reduction (values neither used nor modified are grayed)

$$\text{LOCAL}\quad \frac{(a,\sigma)\rightarrow_S (a',\sigma') \qquad \rightarrow_S \text{does not clone a future}}{\alpha[a;\sigma;\iota;F;R;f]\parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f]\parallel P}$$

$$\text{NEWACT}\quad \frac{\gamma \text{ fresh activity} \qquad \iota' \notin dom(\sigma) \qquad \sigma' = \{\iota' \mapsto AO(\gamma)\}::\sigma \qquad \sigma_\gamma = copy(\iota_2,\sigma)}{\alpha[\mathcal{R}[Active(\iota_2,m_j)];\sigma;\iota;F;R;f]\parallel P \longrightarrow \alpha[\mathcal{R}[\iota'];\sigma';\iota;F;R;f]\parallel \gamma[\iota_2.m_j();\sigma_\gamma;\iota_2;\emptyset;\emptyset;\emptyset]\parallel P}$$

$$\text{REQUEST}\quad \frac{\begin{array}{c}\sigma_\alpha(\iota)=AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\rightarrow\beta} \text{ new future} \qquad \iota_f \notin dom(\sigma_\alpha)\\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\ \iota'\ ;\ \sigma_\beta,\ \iota'') \qquad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\rightarrow\beta})\}::\sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\parallel P \longrightarrow\\ \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\parallel \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta::[m_j;\iota'';f_i^{\alpha\rightarrow\beta}];f_\beta]\parallel P\end{array}}$$

$$\text{SERVE}\quad \frac{m_j\in M \qquad \forall m\in M,\ m\notin R}{\alpha[\mathcal{R}[Serve(M)];\sigma;\iota;F;R::[m_j;\iota_r;f']::R';f]\parallel P \longrightarrow \alpha[\iota.m_j(\iota_r)\Uparrow f,\mathcal{R}[[]];\sigma;\iota;F;R::R';f']\parallel P}$$

$$\text{ENDSERVICE}\quad \frac{\sigma' = Copy\&Merge(\sigma,\ \iota\ ;\ \sigma,\ \iota')}{\alpha[\iota\Uparrow f',a;\sigma;\iota;F;R;f]\parallel P \longrightarrow \alpha[a;\sigma';\iota;F::\{f\mapsto\iota'\};R;f']\parallel P}$$

$$\text{REPLY}\quad \frac{\sigma_\alpha(\iota)=fut(f_i^{\gamma\rightarrow\beta}) \qquad F_\beta(f_i^{\gamma\rightarrow\beta})=\iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta,\ \iota_f\ ;\ \sigma_\alpha,\ \iota)}{\begin{array}{c}\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\parallel P \longrightarrow\\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\parallel P\end{array}}$$

and can continue its execution. A request containing the name of the method, the location of a deep copy of the argument stored in σ_β, and the associated future is added to the end of the pending requests R_β: $[m_j;\iota'';f_i^{\alpha\rightarrow\beta}]$. A particular rule is necessary when an asynchronous method call is performed on the caller activity itself, i.e., when $\alpha=\beta$. This is different from a sequential call on the active object:

$$\text{REQUEST } \alpha=\beta\quad \frac{\begin{array}{c}\sigma_\alpha(\iota)=AO(\alpha)\iota'' \qquad \notin dom(\sigma_\alpha) \qquad f_i^{\alpha\rightarrow\alpha} \text{ new future} \qquad \iota_f\notin dom(\sigma_\alpha)\\ \iota''\neq\iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\alpha,\ \iota'\ ;\ \{\iota_f\mapsto fut(f_i^{\alpha\rightarrow\alpha})\}::\sigma_\alpha,\ \iota'')\end{array}}{\alpha[\mathscr{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|Q \longrightarrow \alpha[\mathscr{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha::[m_j;\iota'';f_i^{\alpha\rightarrow\alpha}];f_\alpha]\|Q}$$

SERVE serves a new request (Fig. 8). The reduction of the current term is stopped and stored as a continuation: future f, expression $\mathscr{R}[[]]$, and the first request on one of the labels specified in M is treated. To ensure ASP properties, it is crucial that only the oldest request matching a criterion can be served. Indeed, treating the last request matching M would be highly dependent on the interleaving between communications and local reductions. The activity is actually stuck until a matching request is found in the pending request queue.

ENDSERVICE associates the result of the request that has just been treated to the current future f. It applies when the current request is finished (i.e. current term is a location). The result is deep copied to prevent post-service modification of the value. The new current term and current future are obtained from the continuation (Fig. 9).

REPLY updates a future value (Fig. 10). It applies when the value associated to a future has been calculated and replaces a reference to a future by the part of store associated with it, i.e., by the deep copy of the location associated to $f_i^{\gamma\rightarrow\beta}$). Deliberately, we do not specify when this rule should be applied. It is only required that an activity contains a reference to a future, and another one has calculated the corresponding result. However, wait-by-necessity can only be resolved by the update of the future value, which constraints the moment when this rule can be applied. In general, a future $f_i^{\gamma\rightarrow\beta}$ also needs to be updated in an activity different from the origin of the request ($\alpha\neq\gamma$) because of the capacity to transmit futures, e.g.

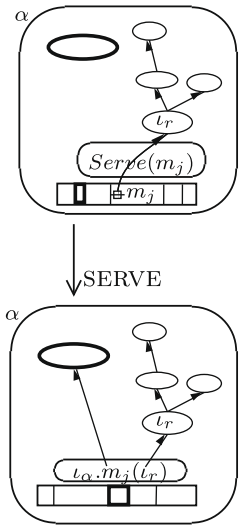

Fig. 8. SERVE.

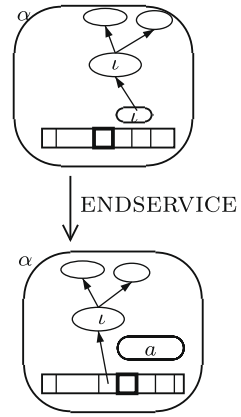

Fig. 9. ENDSERVICE.

as method call parameters. A future can also be updated in the same activity it has been calculated, when $\alpha = \beta$, leading to the following particular case for the rule REPLY:

$$\text{REPLY } \alpha = \beta \quad \frac{\sigma_\alpha(\iota) = \mathit{fut}(f_i^{\gamma\to\alpha}) \qquad F_\alpha(f_i^{\gamma\to\alpha}) = \iota_f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\alpha,\ \iota_f\ ;\ \sigma_\alpha,\ \iota)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P}$$

After an update, a future value must be kept stored because the future might have proliferated in other activities; however garbage collection techniques could be applied to futures [16,17].

We can define the infinite loop *Repeat* and the FIFO service that serves the requests in the order they arrived:

$$\mathit{Repeat}(a) \triangleq [\mathit{repeat} = \varsigma(x)a; x.\mathit{repeat}()].\mathit{repeat}()$$

$$\mathit{FifoService} \triangleq \mathit{Repeat}(\mathit{Serve}(\mathcal{M}))$$

where $\mathcal{M}$ is the set of all the method labels of the concerned active object.

An *initial configuration* consists of a single activity, with the user program a as current term: $\mu[a; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset]$. This activity never receives any request, it communicates by sending requests, creating activities, or receiving replies.

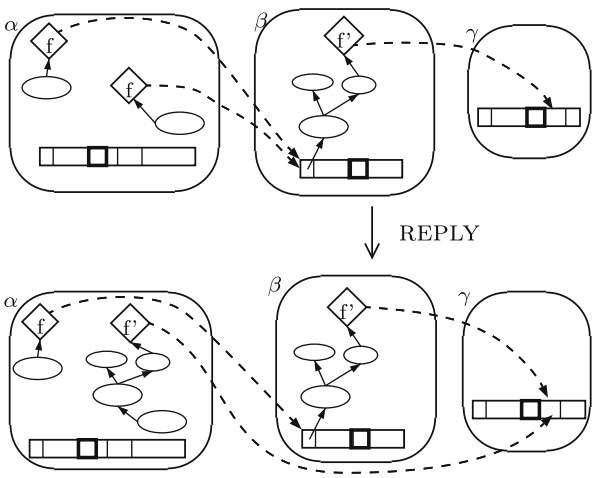

Fig. 10. REPLY.

4.3. Well-formedness

Let $FL(\gamma)$ be the list of futures that have been calculated, the current futures (i.e., f_γ and those in the continuation of the current term), and futures corresponding to pending requests. It is depicted by the rectangles at the bottom of the activity in Fig. 2.

Definition 2 (*Future list*)**.**

$$FL(\gamma) = \{f_i^{\beta\to\gamma} \mid \{f_i^{\beta\to\gamma} \mapsto \iota\} \in F_\gamma\} :: \{f_\gamma\} :: \mathcal{F}(a_\gamma) :: \{f_i^{\beta\to\gamma} \mid [m_j, \iota, f_i^{\beta\to\gamma}] \in R_\gamma\}$$
$$\text{where} \begin{cases} \mathcal{F}(a \Uparrow f, b) = f :: \mathcal{F}(b) \\ \mathcal{F}(a) = \emptyset \qquad\qquad \text{if } a \neq a' \Uparrow f, b \end{cases}$$

Let $ActiveRefs(\alpha)$ and $FutureRefs(\alpha)$ be respectively the set of active objects and the set of futures referenced in α:

$$ActiveRefs(\alpha) = \left\{\beta \mid \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = AO(\beta)\right\},$$
$$FutureRefs(\alpha) = \left\{f_i^{\beta\to\gamma} \mid \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = fut(f_i^{\beta\to\gamma})\right\}$$

Definition 3 (*Well-formedness*)**.** A parallel configuration is *well-formed* if all local configurations are well-formed, according to Definition 1; every referenced activity belongs to the configuration; and every future reference points to a future that either has been calculated, or has a corresponding entry in a pending request queue:

$$\vdash P\text{OK} \Leftrightarrow \forall \alpha \in P, \begin{cases} \vdash (a_\alpha, \sigma_\alpha)\text{OK} \ \wedge \ \vdash (\iota_\alpha, \sigma_\alpha)\text{OK} \\ \beta \in ActiveRefs(\alpha) \Rightarrow \beta \in P \\ f_i^{\beta\to\gamma} \in FutureRefs(\alpha) \Rightarrow f_i^{\beta\to\gamma} \in FL(\gamma) \end{cases}$$

Property 2 (Correct reduction)**.** *Well-formedness is preserved by parallel reduction:*

$$\vdash P\text{OK} \wedge P \longrightarrow P' \implies \vdash P'\text{OK}$$

4.4. Future and parameter isolation

The following property states that the value of each future and each request parameter are situated in isolated parts of the store. Fig. 11 illustrates the isolation of a future value on the left and a request parameter on the right.

Property 3 (Store partitioning)**.** *Let*

$$ActiveStore(\alpha) = copy(\iota_\alpha, \sigma_\alpha) \cup \bigcup_{\iota \in locs(a_\alpha)} copy(\iota, \sigma_\alpha),$$

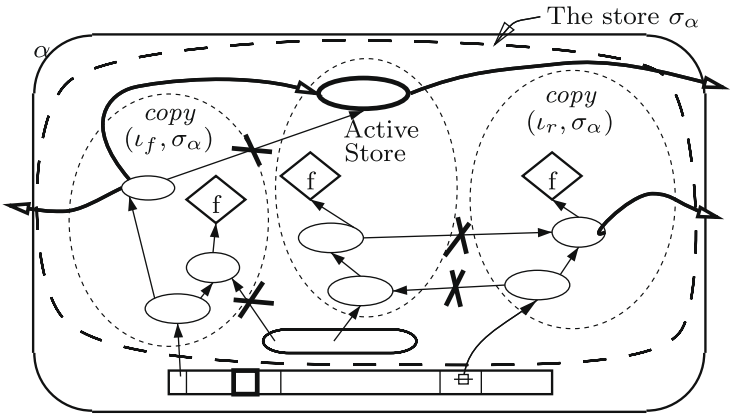

Fig. 11. Store partitioning: future value, active store, request parameter.

At any stage of computation, each activity has the following invariant:

$$\sigma_\alpha \supseteq \left(\mathit{ActiveStore}(\alpha) \oplus \bigoplus_{\{f \mapsto \iota_f\} \in F_\alpha} \mathit{copy}(\iota_f, \sigma_\alpha) \oplus \bigoplus_{[l_j; \iota_r; f] \in R_\alpha} \mathit{copy}(\iota_r, \sigma_\alpha) \right)$$

where $\oplus$ is the disjoint union.

This invariant is proved by checking it on each reduction rule. The part of σ_α that is outside the above partition may be freely garbage collected. The futures and parameters partitions are immutable, except for future updates.

5. Confluence and determinism

This section introduces a notion of compatibility and an equivalence relation between configurations. Then, we present several confluence properties. Our objective is to identify what creates concurrency in ASP. ASP combines concurrent activities with limited concurrency in order to simplify the reasoning about programs. The confluence property presented here provides a minimal condition ensuring that two executions of the same program produce the same result. Here, *two configurations are said to be confluent if they can lead to the same result.* Appendices formalize the notion of equivalence modulo future updates and proves the crucial theorems presented in this section.

5.1. Definitions and hypothesis

First, let $\overset{*}{\longrightarrow}$ denote the reflexive transitive closure of $\longrightarrow$; and α_P denote the activity α of configuration P.

We suppose that the names of *freshly allocated activities are chosen deterministically*: the first activity created by α will have the same identifier for all the possible executions. Non-deterministic choice of fresh activity names would require to rename activities when defining the equivalence of two configurations. We would have to apply also the renaming to the definition of compatibility between request sender list (Definition 6), leading to more complex notations.

We also specify future identifiers, $f_i^{\alpha \to \beta}$. To simplify notations, f_i is now the name of the method invoked, indexed by its arrival number in β: if the 4^{th} request received by β comes from γ and concerns method *foo*, then its future is $\mathit{foo}_4^{\gamma \to \beta}$. Consequently, in the following, f denotes a method label.

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

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

This set is called *potential services*, and can be either specified by the programmer and checked, or statically inferred. Note that $P \overset{*}{\longrightarrow} Q \Rightarrow \mathcal{M}_{\alpha_Q} \subseteq \mathcal{M}_{\alpha_P}$. For example, consider a program P containing two *Serve* primitives, one serving requests on m_1 or m_2 ($\mathit{Serve}(m_1, m_2)$), and another serving requests on m_3 ($\mathit{Serve}(m_3)$). Potential services would be: $\mathcal{M}_{\alpha_P} = \{(m_1, m_2), (m_3)\}$.

Two requests on methods m_1 and m_2 are said to be *interfering* in α for 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: $\exists M \in \mathscr{M}_{\alpha_P}, \{m_1, m_2\} \subseteq M$.

5.2. Configuration compatibility

We introduce now concepts that will be useful for establishing confluence properties in Section 5.4. Informally, two configurations are compatible if, for each activity γ present in both, the list of senders of requests received by γ in one configuration is the prefix of the same list in the other configuration. Moreover, two non-interfering requests can safely be exchanged. This compatibility relation will be a criterion for confluence and will be used to show that the order of evaluation is entirely defined by the order of request sending.

Definition 4 (*Request Sender List*)**.** The request sender list (RSL) of α is the list of the identifiers of the activities that have sent requests to α. These identifiers are ordered by the *request reception order*. Each element is indexed by the name of the invoked method. The i^{th} element of $RSL(\alpha)$ is inferred from the i^{th} future computed by the activity as follows:

$$(RSL(\alpha))_i = \beta^f \text{ if } f_i^{\beta\to\alpha} \in FL(\alpha)$$

$FL(\alpha)$ has been defined in Section 4.3. The RSL list is obtained from *futures* associated to *served requests*, *current requests*, and *pending requests*. This list is ordered by the request *arrival* order, and thus, some entries corresponding to served requests can appear after some current or pending requests. *Serve* operations can be performed while another request is being served; then the relation between RSL order and FL order is complex. However, if only the service method performs *Serve* operations, then all the restrictions to potential services of the RSL and of the FL are in the same order. For a FIFO service the order of requests is not changed from the moment they are received. Thus the RSL is directly obtained from the concatenation of the future values in the order they have been calculated, the unique current future, and the pending requests in the order they arrived. If $f_n^{\beta\to\alpha}$ is the current future then $f_1^{\delta\to\alpha} \dots f_{n-1}^{\gamma\to\alpha}$ correspond to the calculated futures and $f_{n+1}^{\delta\to\alpha} \dots$ correspond to the pending requests.

Definition 5 (*RSL comparison* $\trianglelefteq$)**.** RSLs are ordered by the prefix order on activities:

$$\alpha_1^{f_1} \dots \alpha_n^{f_n} \trianglelefteq \alpha_1'^{f_1'} \dots \alpha_m'^{f_m'} \;\Leftrightarrow\; n \le m \;\wedge\; \forall i \in [1..n], \alpha_i = \alpha_i'$$

Definition 6 (*RSL compatibility:* $RSL(\alpha_P) \bowtie RSL(a_Q)$)**.** Two RSLs are compatible if one is prefix of the other:

$$RSL(\alpha_P) \bowtie RSL(a_Q) \Leftrightarrow \big(RSL(\alpha_P) \trianglelefteq RSL(\alpha_Q) \vee RSL(\alpha_P) \trianglelefteq RSL(\alpha_Q)\big)$$

An equivalent criteria for RSL compatibility is that two RSLs are compatible if they have a least upper bound: i.e. if $RSL(\alpha_P) \sqcup RSL(\alpha_Q)$ exists.

Let $RSL(\alpha)|_M$ represent the *restriction* of the $RSL(\alpha)$ list to the set of labels M. For instance $(\alpha^{f_0} :: \beta^{f_1} :: \gamma^{f_2})|_{\{f_0,f_2\}} = \alpha^{f_0} :: \gamma^{f_2}$. Two configurations are said to be compatible if all the restrictions of their RSL that can be served are compatible. We suppose that configurations to be compared derive from the same ancestor P_0. Thus there is P_0 such that $P_0 \xrightarrow{*} P$ and $P_0 \xrightarrow{*} Q$ and then the compatibility of P and Q is defined by:

Definition 7 (*Configuration compatibility:* $P \bowtie Q$)**.** If P_0 is an initial configuration such that $P_0 \xrightarrow{*} P$ and $P_0 \xrightarrow{*} Q$

$$P \bowtie Q \;\Leftrightarrow\; \forall \alpha \in P \cap Q, \forall M \in \mathcal{M}_{\alpha_{P_0}}, RSL(\alpha_P)|_M \bowtie RSL(\alpha_Q)|_M$$

Two configurations P and Q are compatible if for every activity α present in both, the RSL of α in P and the RSL of α in Q are compatible. We will show that two compatible configurations are confluent; this ensures that the *execution is fully determined by the arrival order of the request senders*. In fact, the behavior of each activity is determined by the requests it received, including their parameters; but, considering the whole configuration, the order of request senders is sufficient: it determines uniquely the request parameters.

5.3. Equivalence modulo future updates

We now define an equivalence relation that is insensitive to future updates. First, let $\equiv_R$ be an equivalence relation on pending requests allowing them to be reordered provided the compatibility of RSLs is maintained: requests that can not interfere, i.e. that cannot be served by the same *Serve* primitive, can be safely exchanged.

Equivalence modulo future updates, denoted by $\equiv_F$, is an extension of $\equiv_R$ allowing the update of some calculated futures. Informally, if one can reach an object by following a *path* from the root of the activity α, then the same path can be followed in the same activity of the equivalent term and leads to an equivalent object. Paths express the accesses to parts of terms,

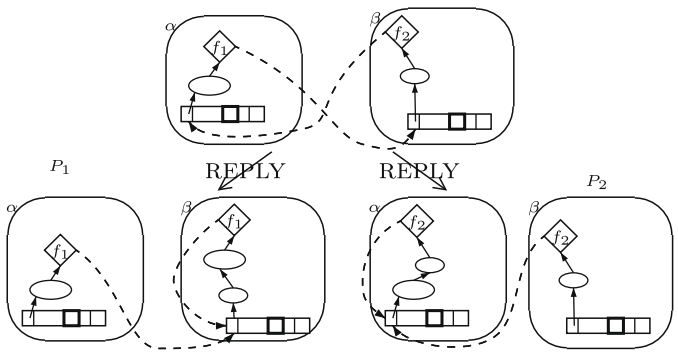

Fig. 12. Cycle of futures: though different, bottom configurations behave the same.

fields, locations inside the current activity store, ... and are insensitive to the dereferencing of futures. Equivalence modulo future updates consists in considering references to futures already calculated as equivalent to local references to the part of store which is the (deep copy of the) future value. For $\equiv_F$, a future is equivalent to a part of store if this part of store is the deep copy of the future value.

The definition of equivalence modulo future updates is formalized in Appendix 8, Definition 22. We focus now on a few sufficient conditions for equivalence modulo future update. First, two configurations only differing by some future updates are equivalent:

$$P \xrightarrow{\text{REPLY}} P' \Rightarrow P \equiv_F P'$$

We also have the following sufficient condition, which is more general:

$$P_1 \xrightarrow{\text{REPLY}} P' \ \wedge \ P_2 \xrightarrow{\text{REPLY}} P' \quad \Rightarrow \quad P_1 \equiv_F P_2$$

But this condition is still not necessary, for example the equivalence modulo future updates deals with cycles of futures which cannot be identified thanks to the preceding condition. Configurations with cycles can lead to executions that will never converge but behave identically, see Fig. 12 for an example.

We exhibit now some properties of $\equiv_F$. Let T be any reduction defined in Table 3: $T \in \{\text{LOCAL}, \text{NEWACT}, \text{REQUEST}, \text{SERVE}, \text{ENDSERVICE}, \text{REPLY}\}$. $\xrightarrow{T}$ represents the application of the rule T and $\xrightarrow{T}$ the reflexive transitive closure of $\xrightarrow{T}$. Then let us denote by $\overset{T}{\Longrightarrow}$ the reduction $\xrightarrow{T}$ preceded by some applications of the REPLY rule.

Definition 8 (*Parallel Reduction with Future Updates*)**.**

$$\overset{T}{\Longrightarrow} \ = \ \xrightarrow{\text{REPLY}^*} \xrightarrow{T} \text{ if } T \neq \text{REPLY} \text{ and } \xrightarrow{\text{REPLY}^*} \text{ if } T = \text{REPLY}$$

Informally, this reduction achieves as many replies as necessary, and then applies another transition.

Property 4 (Equivalence and Reduction)**.** *If one can apply a reduction rule on a configuration then, after several* REPLY*, the same rule can be applied on any equivalent configuration, and an equivalent configuration is obtained.*

$$P \xrightarrow{T} Q \ \wedge \ P \equiv_F P' \ \Rightarrow \ \exists Q', P' \overset{T}{\Longrightarrow} Q' \ \wedge \ Q' \equiv_F Q$$

The following corollary states that one can actually apply several REPLY before the reduction $P \xrightarrow{T} Q$ without any consequence on the Property 4:

Property 5 (Equivalence and Generalized Parallel Reduction)**.**

$$P \overset{T}{\Longrightarrow} Q \ \wedge \ P \equiv_F P' \ \Rightarrow \ \exists Q', P' \overset{T}{\Longrightarrow} Q' \ \wedge \ Q' \equiv_F Q$$

These properties show a slight similarity with properties of weak bisimulation: two equivalent configurations can perform the same reduction and become equivalent, REPLY being a non-observable transition.

5.4. Partial confluence

We now present the main theorem of this paper, it states that compatible configurations are confluent.

Definition 9 (*Confluent Configurations:* $P_1 \curlyvee P_2$)**.**

Two configurations are *confluent* if they can be reduced to equivalent configurations.

$$P_1 \curlyvee P_2 \;\Leftrightarrow\; \exists R_1, R_2,\; P_1 \xrightarrow{*} R_1 \;\wedge\; P_2 \xrightarrow{*} R_2 \;\wedge\; R_1 \equiv_F R_2$$

Theorem 1 (Partial Confluence)**.** *If, from a given term, we obtain two compatible configurations, then these configurations are confluent.*

$$P \xrightarrow{*} Q_1 \;\wedge\; P \xrightarrow{*} Q_2 \;\wedge\; Q_1 \bowtie Q_2 \;\Rightarrow\; Q_1 \curlyvee Q_2$$

The content of the confluence theorem can be summarized by: non-determinism can only originate from the invocation of two interfering REQUEST rules on the same destination activity. The order of updates of futures never has any influence on the reduction of a term; the only constrain to the moment when a REPLY must occur is a wait-by-necessity on that future. Even if this property is natural, it allows a lot of asynchrony and proves that the mechanism of futures is rather powerful, even though they are single assignments variables and the calculus is imperative. Furthermore, the order of requests does not matter if they cannot be involved in the same *Serve* primitive, thus some requests on different methods do not interfere; they can be safely exchanged. On the contrary, consider two different requests R_1 and R_2 on the *same method* of a given destination activity; if in Q_1, R_1 is before R_2, and in Q_2, R_2 is before R_1; then the configurations obtained from Q_1 and Q_2 will never be equivalent regarding $\equiv_F$.

The proof of Theorem 1 is presented in Appendix A.6; the key idea is that from two compatible configurations, one can send missing requests in the right order, and the configurations can be reduced to a common one modulo $\equiv_F$.

Note that if, instead of *Serve*(M), the service primitive was serving the oldest request coming from a given activity *Serve*(α), then the resulting calculus would be fully deterministic. Indeed, the request would always be served in the same order. Such a calculus would be similar to Process Networks where *get* operations are performed on a given channel and a channel have a single source process. The next sections identify a set of terms that behave deterministically.

5.5. Deterministic Object Networks

If two different activities never send concurrently to the same target activity a request on a given method (or set of method labels M that appears in a *Serve*(M)), then no conflict can appear and the reduction is confluent. To formalize this principle, we define a deterministic object network (DON) as a program that, at any time, for each set of label M on which α can perform a *Serve* primitive, only one activity can send a request on methods of M.

Definition 10 (*DON*)**.** A configuration P is a Deterministic Object Network if it verifies the property *DON*(P):

$$DON(P) \Leftrightarrow \left(P \xrightarrow{*} Q \Rightarrow \forall \alpha \in Q,\; \forall M \in \mathcal{M}_{\alpha p}, \exists^1 \beta \in Q, \exists m \in M,\; a_\beta = \mathscr{R}[\iota.m(\ldots)] \;\wedge\; \sigma_\beta(\iota) = AO(\alpha) \right)$$

where $\exists^1$ means "there is at most one".

DON(P) implies that two activities cannot be able to send requests that can interfere to the same third activity, and thus RSLs are compatible.

Property 6 (DON and compatibility)**.** *DON terms always reduce to compatible configurations:*

$$\mathrm{DON}(P) \;\wedge\; P \xrightarrow{*} Q_1 \;\wedge\; P \xrightarrow{*} Q_2 \;\Rightarrow\; Q_1 \bowtie Q_2$$

As compatibility implies confluence, the set of DON terms is a deterministic subset of ASP terms that cannot be identified purely syntactically.

Theorem 2 (DON determinism)**.** *DON terms ensure the Church-Rosser property:*

$$DON(P) \;\wedge\; P \xrightarrow{*} Q_1 \;\wedge\; P \xrightarrow{*} Q_2 \;\Rightarrow\; Q_1 \curlyvee Q_2$$

5.6. Toward a static approximation of DON terms

The DON definition is dynamic. However, it could be approximated by statically determining the set of active objects that can send a request on method m of activity α. Thus, a static approximation of the set of activities and of the remote method invocations performed on those activities is necessary. Static analysis has been studied in the literature. Our objective is not to detail a static analysis of ASP because it would be very similar to existing works; we rather explain how such an analysis could be used to statically ensure determinism. Additionally, Section 6 will show how component-oriented programming can be used to statically ensure determinism. Suppose one has a static approximation of activities: $\dot{\alpha}$, $\dot{\beta}$, ...

Definition 11 (*Approximated call graph*)**.** An *approximated call graph* $\mathcal{G}(P)$ relates activities by the method they accept. We write $\dot{\alpha} \xrightarrow{foo} \dot{\beta} \in \mathcal{G}(P)$ if a request on the method *foo* may be sent from $\dot{\alpha}$ to $\dot{\beta}$. More formally:

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

We define a *faithful* approximation as an approximation that does not merge two dynamic activities as a single static one.

Definition 12. A static approximation is *faithful* for an activity α, written $F(\alpha)$, if whenever $\dot{\alpha} = \dot{\gamma}$ then $\alpha = \gamma$. An approximated call graph is faithful if for all $\dot{\alpha} \xrightarrow{foo} \dot{\beta} \in \mathcal{G}(P)$, it is the case that $F(\beta)$.

Two real activities can be merged into a single abstract one but not the opposite. Then, the following property is an approximation of DON terms:

Definition 13 (*Static DON*)**.** Suppose $\mathcal{G}(P)$ is faithful. Let the potential service be approximated such that $\mathscr{M}_{\beta_P} \subseteq \mathscr{M}_{\dot{\beta}_P}$. Then a program P is a Static Deterministic Object Network, $SDON(P)$, if the following condition holds:

$$\dot{\alpha} \neq \dot{\alpha'} \ \wedge \ \dot{\alpha} \xrightarrow{m_1} \dot{\beta} \in \mathcal{G}(P) \ \wedge \ \dot{\alpha'} \xrightarrow{m_2} \dot{\beta} \in \mathcal{G}(P) \Rightarrow \forall M \in \mathcal{M}_{\dot{\beta}_P}, \{m_1, m_2\} \not\subseteq M$$

Theorem 3 (DON determinism)**.**

$$SDON(P) \ \wedge \ P \xrightarrow{*} Q_1 \ \wedge \ P \xrightarrow{*} Q_2 \ \Rightarrow \ Q_1 \curlyvee Q_2$$

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

$$\exists M \in \mathcal{M}_{\beta_P}, \exists \alpha \neq \alpha', \exists m_1, m_2 \in M, \begin{cases} a_\alpha = \mathscr{R}[\iota.m_1(\iota')] \wedge \sigma_\alpha(\iota) = AO(\beta) \\ a_{\alpha'} = \mathscr{R}[\iota_2.m_2(\iota'_2)] \wedge \sigma_{\alpha'}(\iota_2) = AO(\beta) \end{cases}$$

Then, by definition of $\mathcal{G}(P)$ $\dot{\alpha} \xrightarrow{m_1} \dot{\beta} \in \mathcal{G}(P)$ and similarly for m_2. As $\mathcal{G}(P)$ is faithful, $\dot{\alpha} \neq \dot{\alpha'}$. Finally P is not a SDON because: $\dot{\alpha} \neq \dot{\alpha'} \wedge \dot{\alpha} \xrightarrow{m_1} \dot{\beta} \in \mathcal{G}(P) \ \wedge \ \dot{\alpha'} \xrightarrow{m_2} \dot{\beta} \in \mathcal{G}(P) \ \wedge \ m_1, m_2 \in M \ \wedge \ M \in \mathcal{M}_{\dot{\beta}_P}$ □

5.7. Tree topology determinism

This section can be seen as a simple application of SDON that has the advantage to be valid even in the highly interleaving case of FIFO services. The *request flow graph* $\mathcal{R}$ is the graph where nodes are activities and there is an edge between two activities if one activity can send requests to another one:

$$\dot{\alpha} \to \dot{\beta} \in \mathcal{R} \Leftrightarrow \exists foo. \dot{\alpha} \xrightarrow{foo} \dot{\beta} \in \mathcal{G}(P)$$

If the request flow graph is a tree then the term verifies the SDON definition.

Property 7 (Tree Request Flow Graph)**.** *If the request flow graph $\mathcal{R}$ forms a set of trees then the reduction is deterministic.*

The next property is a direct application of DON determinism. It is both more dynamic and thus more precise considering control flow; and only sensitive to object topology and thus less precise considering the global references really used. The *activity dependence graph* $\mathcal{A}$ is a graph where nodes are activities, and there is an edge between two activities if the store of an activity has a reference to another activity: $(\dot{\alpha}, \dot{\beta}) \in \mathcal{A}$ if $\exists \iota, \sigma_{\dot{\alpha}}(\iota) = AO(\dot{\beta})$.

Property 8 (Tree Topology)**.** *If at every step of reduction the activity dependence graph $\mathcal{A}$ forms a set of trees then the execution is deterministic.*

Of course, Properties 7 and 8 can be combined and one can remove from the activity dependence graph, every edge on which no communication is performed. FIFO is, to some extent, the worst case with respect to determinism, as any change to the order of request reception will lead to non-determinism.

5.8. Channels in ASP

DON terms are ASP programs behaving deterministically; to some extent, these programs are related to Process Networks [14]. To show this, we define channels in ASP and use them to relate Process Networks with DON terms.

Definition 14 (*Channels*)**.** Let channels be pairs (M,α) where M is a set of method labels and α a destination activity. Let $\mathcal{S}$ be a set of channels such that, for all $(M_1,\alpha),(M_2,\alpha)\in\mathcal{S}$:

$$M_1 \neq M_2 \;\Rightarrow\; \forall m_1 \in M_1 . m_2 \in M_2 .\;\; \nexists M \in \mathcal{M}_{\alpha_P}, \{m_1,m_2\} \subseteq M$$

For each program P, there are many valid sets of channels forming a lattice. At the top, the less precise repartition has one channel for each activity $(\cup_{M\in\mathcal{M}_{\alpha_P}} M,\alpha)$. In the case where one only serves one method at a time, the most precise set of channels is pairs (method label, activity).

This notion of channel relates DON to the Process Networks: a program where each channel has only one source activity is clearly a DON program; and in Process Networks, each channel has only one sender process. ASP can be seen a generalization of Process Networks because one can wait on several requests simultaneously, and futures act as hidden channels insensitive to reply order.

5.9. A deterministic example

We show below how the theorems presented in this section can be applied to the examples of the sieve of Eratosthenes, presented in Section 3.5.

The pull version (Fig. 3) is deterministic because, as shown in Fig. 4, its activities dependence graph always forms a tree, and is a SDON. To be precise, during the creation of a new activity for a new *Sieve*, during a small time, the topology is not a tree because both *Sift* and the activated *Sieve* have a reference to the former first *Sieve*, see Fig. 4. However, following the remark of Section 5.7, it has no consequence because when the topology is not a tree, *Sift* does not use this link to communicate. This example shows that tree topology of activities can easily be used to prove determinacy.

We now focus on the push example because, there, every Sieve activity always keeps a reference to the Display (Fig. 5), and determinism can only be verified by a more dynamic property: this program is not a SDON. Indeed what ensures determinacy is the fact that as soon as another *Sieve* is created, the former one no longer sends results to the *Display*. As soon as a new *Sieve* is created, the preceding one no longer uses its reference to *Display*. We call $Sieve_n$ the name of the n^{th} created sieve activity.

First, the Sieve of Eratosthenes is deterministic because its RSLs are the same for every execution, and thus are *compatible*. More precisely, in every execution the RSLs are of the form:

$$\begin{aligned} RSL(Sieve_n) &= Sieve_{n-1} :: Sieve_{n-1} :: Sieve_{n-1} :: \ldots \\ RSL(Display) &= Sieve_1 :: Sieve_2 :: Sieve_3 :: \ldots :: Sieve_n \end{aligned}$$

We can also show that the Sieve of Eratosthenes is DON. For the Display activity which is the only that is accessed by several others, consider the activities that can send a request do Display. The *Display.put* request can only be invoked from the last created Sieve, between the first *put* request served and the next sieve creation:

$$a_{Sieve_k} = \mathscr{R}[\iota.put(\ldots)] \wedge \sigma_\beta(\iota) = AO(Display) \Rightarrow Sieve_k \text{is the last sieve}$$

Consequently, at most one activity can send a request to the Display at each moment, and finally the push version of sieve is a *DON*. The static analysis necessary for verifying DON is simpler than inferring exactly all the possible RSLs as shown above. However, direct and automatic static verification of DON is still difficult to implement.

In the sieve example all the activities always serve the only request (put or get) that can be sent to them. Like FIFO service, this case is the worst one with respect to determinism: SDON determinacy is equivalent to tree determinacy. The interest of SDON compared to tree determinacy lies in its ability to generalize determinacy properties to activities selectively serving different requests coming from different activities. Fig. 13 shows a configuration that is a SDON but do not verify the tree determinacy, it corresponds to the term:

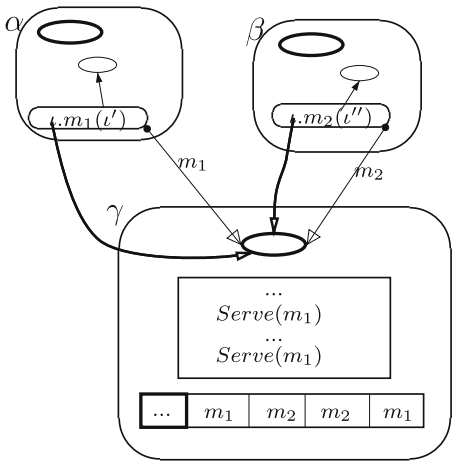

Fig. 13. A deterministic configuration according to SDON.

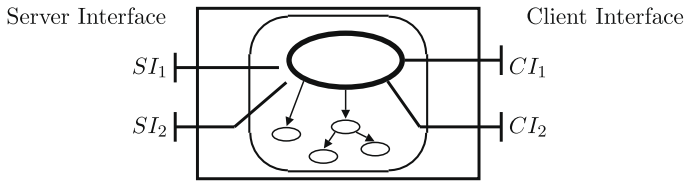

Fig. 14. A primitive component.

let gamma = *Active*([...*Serve*(m_1)...*Serve*(m_2)]) *in*
let alpha = *Active*([...*gamma*.m_1...]) *in*
let beta = *Active*([...*gamma*.m_2...]) *in* ...

6. Deterministic components

This section demonstrates how to build hierarchical, distributed, and deterministic components upon ASP calculus. The component approach presented here is realized by asynchronous components communicating by asynchronous method calls. We show how components can be used to ensure the DON property. Indeed, components provide an abstraction for the topology of active objects: the component structure is an abstraction of the activity structure. The SDON property can be ensured by analysis of the component structure.

6.1. Primitive components

Primitive components are the base of the component assembly, they encode the business code, and have provided (server) and required (client) interfaces.

Definition 15 (*Primitive Component—Fig. 14*)**.** A *primitive component* is characterized by a component name Name, together with names for its *Server Interfaces* (SI), and for its *Client Interfaces* (CI).

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

We define $Exported(PC) = \{SI_i\}^{i\in 1..k}$, and $Imported(PC) = \{CI_j\}^{j\in 1..l}$.

Primitive Component Activity: To give functionalities to a primitive component, we attach it an ASP term a corresponding to an object to be activated; the service method *srv* 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.

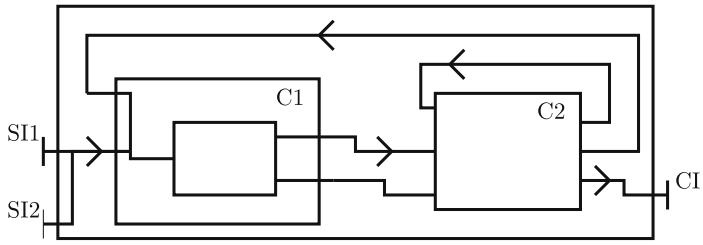

Fig. 15. A composite component.

$\mathscr{M}$ range over the set of method labels, and $\mathscr{L}$ over the set of field labels.

$$PC_{Act} ::= \text{Name}_{Act} < a, srv, \varphi_S, \varphi_C > \quad \text{where} \begin{cases} \varphi_S : Exported(PC) \to \wp(\mathscr{M}) \\ \varphi_C : Imported(PC) \to \mathscr{L} \end{cases} \text{ are total functions}$$

6.2. Hierarchical components

From primitive components, *composite components* can be built by interconnecting components and exporting some SIs and CIs. To simplify notations, we suppose that each interface of a primitive component has a unique name (qualified names could be used for disambiguation).

Definition 16 (*Composite Component*)**.** A composite component is a set of components $(C_1, \ldots, C_n)$ exporting some server interfaces (ε_S), some client interfaces (ε_C), and connecting some client and server interfaces (defining a partial binding ψ):

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

where a component C_i is either a primitive or a composite one:

$$C ::= PC \mid CC$$

And each CC definition (Name $\ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg$) verifies that a given CI can only be connected once, and only the interfaces of the direct sub-components can be plugged or exported:

$$\varepsilon_S \subseteq \bigcup_{sc \in C_1 \ldots C_m} Exported(sc) \qquad \text{is also denoted } Exported(CC)$$
$$\psi : \bigcup_{sc \in C_1 \ldots C_m} Imported(sc) \to \bigcup_{sc \in C_1, \ldots C_m} Exported(sc) \qquad \text{is a partial function}$$
$$\varepsilon_C \subseteq \bigcup_{sc \in C_1 \ldots C_m} Imported(sc) \qquad \text{is also denoted } Imported(CC)$$
$$\text{Such that } dom(\psi) \cap \varepsilon_C = \emptyset$$

A composite component forwards the requests it receives to its sub-components, and forwarding the requests emitted by its subcomponents. According to the bindings requests can be sent from the external world to sub-components, from sub-components to the external world, or directly between two sub-components. Fig. 15 shows a composite component composed of two components with all kinds of allowed bindings; arrows show the flow of message forwarding. [18] gives a translational semantics for ASP components.

Definition 17 (*Closed Component*)**.** A component C is closed if it does not export or import any interface:

$$Imported(C) = \emptyset \ \wedge \ Exported(C) = \emptyset$$

Definition 18 (*Complete Component*)**.** A primitive component is always complete. A composite component Name $\ll C_1, \ldots, C_m; \varepsilon_S; \psi; \varepsilon_C \gg$ is *complete* if it is formed of complete components and all the interfaces of its inner components are plugged or exported:

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

Non-complete component can lead to deadlocks or requests without destination, because some of their subcomponents are not completely connected. However we chose to allow non-complete components to be defined.

6.3. Determinism and components

We now use SDON definition to build deterministic components. For this, we define the deterministic assemblage of components based on the fact that primitive components provide an abstraction for activities. Then, the SDON definition can be ensured by the specification of deterministic primitive components and connections of interfaces. We first require that for any two methods belonging to a given SI, those methods cannot interfere. If additionally, each SI can be accessed by a single primitive component. Then, ensuring that only one CI is plugged to each SI avoids interfering requests and ensures confluence. As each primitive component *PC* is an abstraction of an activity, we denote $\mathcal{M}_{PC}$ the potential services associated to the activity defined by *PC*.

Definition 19 (*Deterministic Primitive Component (DPC)*)**.** A primitive component $PC = \text{Name} < \{SI_i\}^{i\in 1..k}, \{CI_j\}^{j\in 1..l} >$ is a DPC if its activity $\text{Name}_{Act} < a, srv, \varphi_S, \varphi_C >$ 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:

$$\begin{cases} \forall i, i' \in 1..k,\ i \neq i' \Rightarrow \varphi_S(SI_i) \cap \varphi_S(SI_{i'}) = \emptyset \\ \forall M \in \mathcal{M}_{PC},\ \exists i \in 1..k,\ M \subseteq \varphi_S(SI_i) \end{cases}$$

Definition 20 (*Deterministic Composite Component (DCC)*)**.** A DCC is a composite component built by connecting deterministic components.

$$\begin{aligned} DC &::= DCC \mid DCP \\ DCC &::= \text{Name} \ll DC_1, \ldots, DC_m; \varepsilon_S; \psi; \varepsilon_C \gg \end{aligned}$$

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

$$\psi \text{is an injective partial function} \quad \wedge \quad codom(\psi) \cap \varepsilon_S = \emptyset$$

A DCC assemblage verifies the SDON property because each DPC statically identifies an activity; and the absence of sharing of SIs ensure that two activities cannot send concurrent requests on the same SI.

Theorem 4 (DCC determinism)**.** *DCC components are deterministic.*

This is due to the simple forwarder role of composite components: a request sent by a PC will be (indirectly) transmitted to another PC that is connected to it. Neither the content nor the order of requests on a given binding is modified by the composite components. To summarize, DCCs statically ensure deterministic behavior of component, based on the following requirements:

- Potential services can be statically determined or are specified and checked dynamically (every served set have 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 [7], on ASP source terms.
- Requests follow bindings, they are neither modified nor reordered.
- 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 component semantics.

7. Related works

7.1. General formalisms

The ASP-calculus is based on the untyped imperative object calculus of Abadi and Cardelli (**imp**$_\varsigma$-calculus of [7]). ASP local semantics looks like the one described in [13] but we did not find any concurrent object calculus [19,20,21] with a similar way of communication between asynchronous objects. Thus the ASP calculus seems to introduce new characteristics, especially asynchronous communications with futures which are interesting both in theory and in practice (e.g. hiding latency in wide area networks).

Proving equivalence between terms can be performed by introducing bisimulation on an object calculus like in [22]. We decided to express an equivalence relation specific to ASP but some aspects of our equivalence are close to bisimulation techniques. CIU (Closed Instance of Use) equivalence introduced in [22] deals only with static terms. In order to capture the intrinsic properties of the calculus, we were first interested in dynamic properties like confluence, thus CIU equivalence is inadequate to our problem.

7.2. Concurrent calculi and languages

Obliq and Øjeblik. Obliq [23] is a language based on the ς-calculus that expresses both parallelism and mobility. It is based on threads communicating with a shared memory. Like in ASP, calling a method on a remote object leads to a remote execution of the method but this execution is performed by the original thread (or more precisely the original thread is blocked). Moreover, for a non-serialized object, many threads can manipulate the same object. Whereas in ASP, the notion of executing thread is linked to the activity and thus every object is "serialized" but a remote invocation does not stop the current thread because of the futures mechanism. Finally, in ASP wait-by-necessity is sufficient for synchronization and no specific notion of thread is necessary. Øjeblik [12] is a subset of Obliq, expressive enough for allowing the expression of any Obliq program. A formal semantics has been defined for Øjeblik. The main results on Øjeblik concerns migration. The generalized references for all mutable objects, the presence of threads and the principle of serialization (with mutexes) make the Obliq and Øjeblik languages very different from ASP. In fact there is no way of ensuring a confluence property similar to ours on Obliq.

More generally, Gordon and Hankin [19], and Jeffrey [20] also introduce parallel calculi based on threads and shared memory which are, for the same reasons as Øjeblik inadequate to our case.

π-calculus. π-calculus is based on communications over channels. Pierce and Turner introduced a language derived from π-calculus PICT to implement object-based programming and synchronization based on channels in [24].

Our calculus could be rewritten in the π-calculus [25,26] but this would not help us to prove our confluence property directly. Indeed, in π-calculus, synchronization is based on channels. On the contrary, ASP relies on data-driven synchronization over an imperative object calculus, and thus its semantics is different from π-calculus. Indeed, while the synchronization in ASP is implicit, the π-calculus impose to explicit synchronization channels; the same constraint in ASP would require one to know the first point where the value of a future is needed, and communication for the reception would be explicit, this is both complicated and undecidable in the general case. Thus, it seems impossible to express in the π-calculus the distinction between asynchronous requests and sequential method call that exists in ASP, because it is impossible to distinguish them statically. For example a systematic access to objects through an additional channel that is only filled when the future value is calculated (in the case of a future), cannot be encoded directly.

Under certain restrictions: linear and linearized channels [2,3] π-calculus terms can be statically proved to be confluent and such results could be applicable to some ASP terms. Our objective here is to have a very general confluence property, even if it is in general not statically verifiable. Then several static approximations of this properties can be performed. Still, our confluence property is closed to linearized channels. Indeed, using the channels defined in Section 5.8, programs that verify the DON property would be considered in π-calculus as communicating over linearized channels and would behave deterministically. ASP can be considered as richer because updates of response along non-linearized channels are allowed. Moreover, channels in ASP are more flexible than in π-calculus because several methods can be in the same channel, and one can wait for a request on any subset of these method. We can perform a *Serve* on a part of a channel without losing determinacy.

Process Networks. Process Networks of Kahn et al. [5,14,27] are explicitly based on the notion of *channels* between processes, performing *put* and *get* operations on them. One channel can link at most one source process and many destinations. Destinations do not split the channel output, but each one reads every value put in the channel. The reading on a channel is performed by a blocking *get* primitive. The order of reading on channels is fixed by the source program. Process Networks are deterministic. Process Networks provide confluent parallel processes but require that the order of service is predefined and two processes cannot send data on the same channel, which is more restrictive and less concurrent than ASP.

The ASP channel view introduced in this paper can also be compared to Process Networks channels. Like in the π-calculus case, ASP channels seem more flexible and our property more general especially by the fact that future updates can occur at any time: return channels do not have to verify any constraint and *Serve* can be performed on a part of a "channel".

The $\pi o\beta\lambda$ language. $\pi o\beta\lambda$ [1,28] is a concurrent object-oriented language. A sufficient condition is given for increasing the concurrency without losing determinacy. For program verifying this condition, one can return results from a method before the end of its execution. Then the execution of the method continues in parallel with the caller thread. This sufficient condition is expressed by an equivalence between original and transformed program. $\pi o\beta\lambda$ can be translated to (dialects of) the π-calculus (e.g. [29]). From such a translation, Sangiorgi [30], and Liu and Walker [31,32] proved the correctness of transformations on $\pi o\beta\lambda$ described in [28].

In $\pi o\beta\lambda$., a caller always waits for the method result (synchronous method call), which can be returned before the end of the called method. In ASP, method calls are asynchronous thus more instructions can be executed in parallel: the futures mechanism allows one to continue the execution in the calling activity without having the result of the remote call. A simple extension to ASP could provide a way to assign a value to a future before the end of the execution of a method. Note that in $\pi o\beta\lambda$ this characteristic is the source of parallelism whereas in ASP this would simply allow an earlier future update; in ASP the source of parallelism is the object activation and the systematic asynchronous method calls between activities. The condition given in [28], stating that the result of a method is not modified after being returned, has its counterpart in ASP

by a deep copy of the result (Property 3: Store partitioning). Similarly, the *unique reference* condition from the same work is reflected in ASP with the constraints on objects topology (no remote reference to passive objects).

Actors. The active object concept is rather closed to, and was somehow inspired by, the notion of *actors* [33,34]. Both rely on asynchronous communications, but actors are rather functional, while ASP is in an imperative and object-oriented setting. While actors are interacting by asynchronous *message passing*, ASP is based on asynchronous *method calls*, which remain strongly typed and structured, with future-based synchronizations and without explicit continuations. To some extent, ASP future semantics accounts for the capacity to achieve confluence and determinism in an imperative setting. Finally, the bisimulation techniques used by Agha et al. in [33,34] would have been inadequate to obtain the main result presented in the current paper: a strong, somehow intrinsic, but dynamic property on processes interacting by asynchronous communications.

To some extent, this article develops the idea introduced in [34,33] that "The behavior of a component is locally determined by its initial state and the history of its interactions". However, we chose to take into account a more global history in order to be only sensitive on the order of the message senders instead of the complete history of messages. In ASP, the history of an activity interactions is uniquely determined by the RSLs of all activities.

Other languages and calculi. Futures have first been introduced in Multilisp and ABCL [35]. More recently, programming paradigms similar to ASP have been developed in different contexts, among them one can distinguish $\lambda(fut)$, Creol, and AmbientTalk. The join-calculus is also a key calculus that could be compared to ASP. We detail those related works below

Halstead defined Multilisp [36], a language with shared memory with *futures*. But the combination of shared memory and side effects prevents Multilisp from being determinate. In some way, this paper adapts the notion of futures in a distributed asynchronous object calculus.

$\lambda(fut)$ is a concurrent lambda calculus with futures, with non determinism primitives (cells and handles). In [37], the authors define a semantics for this calculus, and two type systems. This paper provide a very good formalization of futures, with explicit creation point, for lambda calculus; much in the same spirit as in Multilisp. This formalization is better suited for multithreading, whereas ASP is particularly adapted to distribution. Alice [38] is an ML-like language that can be considered as an implementation of $\lambda(fut)$.

Ref. [39] provides a language with futures that features "uniform multi-active objects": roughly each method invocation is asynchronous because each object is active; each object has several current threads, but only one is active at each moment. However, their futures are also explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties on such multi-active objects with futures. This work is also a formalization of the Creol language [40].

AmbientTalk [41] is a language closed to ProActive and ASP. The main difference is that, in AmbientTalk, *invocations on futures are asynchronous*. This changes much the flow of execution, in a way that may be difficult to grasp for the programmer. A variant of ASP could be used as a model of AmbientTalk, providing interesting confluence properties.

The join-calculus [42,43] is a calculus with mobility and distribution. Synchronization in join-calculus is based on filtering patterns over channels. The differences between channel synchronization and data-driven synchronization described for the π-calculus also make the join-calculus inadequate for expressing a semantics closed to the one of ASP.

7.3. Static analysis

We presented here dynamic properties ensuring confluence, as well as a first step toward static approximation of these properties. It is beyond the scope of this paper to design a static analysis of a (distributed) object language, we provide below some related works that would be useful in order to design a convenient static analysis. The set of active objects and their topology could be approximated statically, using classical static analysis techniques [44,45,46].

A simpler methodology like balloon types [47] could also be useful. Balloon types [47] express a way of restricting the object topology by typing. The balloon types topology is a sub-case of the topology that is sufficient for confluence of ASP programs and it is simple to verify (typing). Indeed, if we applied a balloon types methodology, it would create an objects topology where references between activities form a tree and these reference only link active objects. In ASP, passive objects can reference active ones. Thus the topology of objects ensured by [47] is sufficient but not necessary.

In [48], the topology of object graphs has been analyzed by a static analysis [46] to parallelize programs. It uses a variant of the *Tree Determinism* property.

8. Conclusion

We proposed a calculus modeling asynchronous communications with futures in object systems, and exhibited confluence properties. Such properties simplify programming as they prevent the programmer from having to study every possible interleaving of instructions and messages to understand the behavior of a given program. More than determinism properties, this paper also clearly identifies the interferences that can be source of non-determinism. Furthermore, data-driven synchronization alleviates the programmer from the study of synchronization, and thus, it is a very convenient way of programming.

The ASP calculus is based on asynchronous *activities* processing *requests* and responding by means of *futures*. When an activity has sent a request, it can perform other operations while the result value is not needed and the result to come is represented by a future. Such futures are first class entities that can be passed as parameter and results. We also extended ASP into a distributed and hierarchical component model.

ASP ensures a confluence property on compatible terms: two configurations with compatible RSLs (Request Sender List) are confluent. To summarize, the execution is only determined by the ordered list of activity sending requests to a given one. What makes ASP properties powerful is the insensitivity to the moment when results of requests are obtained. Consequently, an equivalence relation was introduced to consider equivalent a term before and after the REPLY reduction. We defined a sufficient condition for determinacy: *Deterministic Object Networks* (DON) terms behave deterministically. We proved that every program communicating over a tree (Property 7: Tree request flow graph) behaves deterministically, even in the case of a FIFO service. All these properties illustrate the fact that the future mechanism in an imperative language seems rather powerful and convenient. Components can provide a way to statically identify active objects, i.e. distributed entities: *components are the unit of distribution and concurrency*. This allows the definition of deterministic composition of components, that are easy to identify statically.

To summarize, confluence properties result from several factors. First, *futures are immutable* entities, which ensures that their value is the same whatever moment they are accessed. Second, *activities are mono-threaded*, which, together with the *absence of shared memory* between activities, prevents concurrent accesses inside each activity. Consequently, concurrency is restricted to requests simultaneously sent from different activities to the same destination.

In the proposed framework, deep copy is necessary to ensure the ASP properties. A study of shared memory ensuring the same properties (e.g., "share on read") is beyond the scope of this study and may be difficult to implement efficiently. However, some of the properties proved in this paper can be used to lower the cost of sending this deep copy along the network, for example:

- We could send a future instead of the deep copy of the parameter when calling a request, and update this future only if the request parameter is really needed thus using the wait-by-necessity mechanism.
- request sending could be performed in parallel with other operations. The only requirement is to ensure causal ordering of requests, for example by adding a FIFO sending queue on the sender side.

Both solutions still require a deep copy of the request parameter to be performed locally at the request sending point.

As a perspective, more precise static approximations of DON programs are still to be investigated. It would also be interesting to specify $\mathcal{M}_{\alpha_P}$ statically.

Appendix A. Technical details on the equivalence modulo futures

This appendix formalizes the equivalence modulo future updates and proves its main properties. In the proofs, we will not detail cases of REPLY and REQUEST where $\alpha = \beta$, indeed these are considered as special cases and can be deduced from the study of the general cases of REPLY and REQUEST. We only detail here the most crucial proof, i.e. the proof related to the relationship between equivalence and reduction (Property 4).

A.1. Renaming

Remember that we consider for confluence properties that activity identifiers are chosen deterministically and thus renaming of activity identifiers is not necessary; consequently, we only define renaming on future identifiers. Θ is a renaming of futures, i.e. an alpha-conversion of futures:

$$\Theta ::= \{\!\{ f_i^{\beta\to\alpha} \leftarrow f_i{}'^{\beta\to\alpha}, \dots \}\!\};$$

A.2. Reordering requests ($R_1 \equiv_R R_2$)

The equivalence relation is defined modulo the reordering of some requests. Two requests can be exchanged if they concern methods which cannot interfere; i.e., if there is no service concerning both method labels.

Two request queues are equivalent if all their restrictions on requests that can interfere in the same $Serve(M)$ are equivalent. In other words, for every set M of labels belonging to a $Serve(M)$ primitive of α the list of requests that can be captured by $Serve(M)$ is equivalent in both configurations (α_P and α_Q). Moreover, in this article, we only compare terms coming from the same initial configuration P_0; as $\mathcal{M}_{\alpha_{P_0}}$ is a static approximation, the potential services of two compared configurations are the same. R_1 is a correct reordering of the request queue R_2 if and only if $R_1 \equiv_R R_2$ where $\equiv_R$ is defined in Table A.1. The first rule expresses the fact that two requests can be exchanged if they do not interfere. The other two rules express reflexivity and transitivity.

Table A.1
Reordering requests

$$\frac{\{M \in \mathcal{M}_{\alpha_{P_0}} | m_1 \in M\} \cap \{M \in \mathcal{M}_{\alpha_{P_0}} | m_2 \in M\} = \emptyset}{R_1 :: [m_1; \iota_1; f_1] :: [m_2; \iota_2; f_2] :: R_2 \equiv_R R_1 :: [m_2; \iota_2; f_2] :: [m_1; \iota_1; f_1] :: R_2}$$

$$R \equiv_R R \qquad \frac{R \equiv_R R_1 \qquad R_1 \equiv_R R'}{R \equiv_R R'}$$

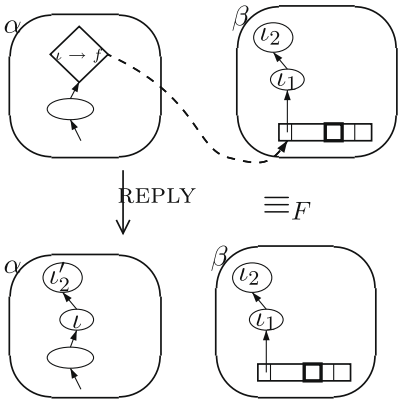

Fig. A.1. Simple example of future equivalence.

Table A.2
Path definition

$$[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \overset{\alpha}{\mapsto}_{m_j(s',x')} a_j\{\!\{x_j \leftarrow s', y_j \leftarrow x'\}\!\}$$

$$a.l_i(b) \overset{\alpha}{\mapsto}_{Inv2(l_i)} b \qquad a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow f} a \qquad \frac{R_\alpha \equiv_R R'}{\alpha_p \overset{\alpha}{\mapsto}_{rq} R'} \qquad [m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs\ arg\ \iota}$$

A.3. Future updates

The equivalence modulo future updates consists in considering the reference to calculated futures like local reference to a deep copy of the value of the future. In other words, future references can be followed as if they were local references to a deep copy. Thus, when two futures references concerns the same future, they are not considered as aliases. Fig. A.1 illustrates an update of a future value. The two configurations are equivalent.

A.3.1. Following references and sub-terms

We formalize below the idea that "future references can be followed as if they were local references." We first define the relation $\overset{\alpha}{\mapsto}_L$ expressing paths inside an activity. Then $\overset{\alpha^*}{\mapsto}_L$ expresses paths that can follow future references.

We first define $a \overset{\alpha}{\mapsto}_x b$. x indicate the path inside an activity that has to be followed for going from the term a to term b. Table A.2 shows a few examples of primitive paths. Then paths are defined inductively concatenating primitive paths in a straightforward manner . denotes paths concatenation (e.g. $a \overset{\alpha}{\mapsto}_{L_1.L_2} b \Leftrightarrow \exists c,\ a \overset{\alpha}{\mapsto}_{L_1} c \overset{\alpha}{\mapsto}_{L_2} b$).

The only particular details in the definition of paths are the following. Bound variables are renamed inside the rule that enters a method body in order to avoid considering alpha conversion of formal parameters at a higher level. Starting with an activity identifier, a path begins with an access to the current term a, active object location ι, futures values F, current future f, or pending requests. Equivalence of pending requests uses $\equiv_R$ (Table A.1). Note that there is no b such that $AO(\beta) \overset{\alpha}{\mapsto}_x b$ or $fut(f^{\gamma\to\beta}) \overset{\alpha}{\mapsto}_x b$.

Let $\overset{\alpha^*}{\mapsto}_L$ be the preceding relation where one can follow futures if necessary and thus cover other activities than α.

Definition 21 ($a \overset{\alpha^*}{\mapsto}_L b$)**.**

$$a \overset{\alpha^*}{\mapsto}_{L_0 \ldots L_n} b \Leftrightarrow (n = 0 \wedge a \overset{\alpha}{\mapsto}_{L_0} b) \quad \vee \exists \iota_i, f_i, \beta_i, \gamma_i^{\,i \le n} \begin{cases} a \overset{\alpha}{\mapsto}_{L_0} \mathit{fut}(f_1^{\gamma_1 \to \beta_1}) \wedge F_{\beta_1}(f_1^{\gamma_1 \to \beta_1}) = \iota_1 \wedge \\ \sigma_{\beta_1}(\iota_1) \overset{\beta_1}{\mapsto}_{L_1} \mathit{fut}(f_2^{\gamma_2 \to \beta_2}) \wedge F_{\beta_2}(f_2^{\gamma_2 \to \beta_2}) = \iota_2 \\ \wedge \ldots \wedge \\ \sigma_{\beta_n}(\iota_n) \overset{\beta_n}{\mapsto}_{L_n} b \end{cases}$$

This definition first follows a path inside an activity α, then follows a future reference from α to β_1, and continues the path in β_1 etc. Note that when one follows a future reference, two local (*ref*) and one future references are in fact considered as identical to a single local reference. In other words, the following of local and future references from $\mathit{fut}(f_i^{\gamma_i \to \beta_1})$ in α to $\sigma_{\beta_1}(\iota_1)$ in β_1 is not taken into account in the path L. For example, in Fig. A.1 the three arrows of the first configurations that are around the future reference (the future reference plus the arrow before and after) are considered as equivalent with a single arrow on the second configuration.

Furthermore, following a path from a given term leads to the same expression except if the destination of the path is a future reference:

Lemma 1 (Uniqueness of path destination)**.**

$$a \overset{\alpha^*}{\mapsto}_L b \wedge a \overset{\alpha^*}{\mapsto}_L b' \Rightarrow b = b' \vee \exists f_n, \iota_n, \beta_n, \gamma, \delta, \begin{cases} (\sigma_{\beta_n}(\iota_n) = \mathit{fut}(f_n^{\gamma \to \delta}) \vee F_\delta(f_n^{\gamma \to \delta}) = \iota_n) \\ \wedge (b = \iota_n \vee b' = \iota_n) \end{cases}$$

The particular case (when $b \neq b'$) occurs when the destination of the path is a future reference. Thus, the path does not necessarily follow this reference. For example, if $b = \iota_n$ in β_n where $\sigma_{\beta_n} = \mathit{fut}(f_n^{\gamma \to \delta})$ then one can have $b' = \iota_f$ where ι_f is the location of future $\mathit{fut}(f_n^{\gamma \to \delta})$ in β.

A.3.2. Equivalence definition

Below is a formal definition of equivalence modulo future updates. The first condition (1) expresses the equivalence both inside an activity and by following futures and the last two conditions (2, 3) express the correctness of aliasing (alias must be the same in both configurations). These two last conditions will be named *alias conditions* in the following. Note that in the alias conditions the existence of a' and a'' such that $\alpha_P \overset{\alpha^*}{\mapsto}_L a'$ and $\alpha_P \overset{\alpha^*}{\mapsto}_{L'} a''$ is already ensured by the first condition. Consequently, the alias conditions ensure that $a' = a''$ and a' and a'' are "correctly aliased".

Definition 22 (*Equivalence modulo future updates:* $P \equiv_F Q$)**.**

$$P \equiv_F Q \;\Leftrightarrow\; \forall \alpha \text{s.t.} \alpha \in P \vee \alpha \in R, \forall L \left(\exists a, \alpha_P \overset{\alpha^*}{\mapsto}_L a \Leftrightarrow \exists a', \alpha_R \overset{\alpha^*}{\mapsto}_L a' \right) \tag{1}$$

$$\wedge\, \forall L, L', a, \left(\alpha_P \overset{\alpha}{\mapsto}_L a \wedge \alpha_P \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L_0', a', L_1, L_1', \gamma \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset \\ \alpha_R \overset{\alpha^*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} a' \;\wedge\; \alpha_R \overset{\alpha^*}{\mapsto}_{L_0'} c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \right) \tag{2}$$

$$\wedge\, \forall L, L', a, \left(\alpha_R \overset{\alpha}{\mapsto}_L a \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L_0', a', L_1, L_1', \gamma \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset \\ \alpha_P \overset{\alpha^*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} a' \;\wedge\; \alpha_P \overset{\alpha^*}{\mapsto}_{L_0'} c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \right) \tag{3}$$

where $R = Q\Theta$ and Θ is a renaming of future identifiers.

The alias conditions can be expressed as follows: "if two paths lead to a common term (e.g., the same location) then in the equivalent configuration these paths also lead to a common term." On the left of the implication, the paths are local to an activity. Actually two references to a future leads to the same future value, and are aliased; while the two updated futures will be two different deep copies of the future value; consequently, ensuring correctness of aliases is sufficient on local paths. The paths on the right of the implication can follow the future references but the *last* alias must be local to an activity. This condition ensures that the last pair of paths (L_1 and L_1') will still be aliased when the future values is updated. If some aliases appear before the last one (if $L_0 \neq L_0'$), then alias conditions also have to be verified with L_0 and L_0': $\alpha_R \overset{\gamma}{\mapsto}_{L_0} c \wedge \alpha_R \overset{\gamma}{\mapsto}_{L_0'} c$ must correspond to aliased objects in α_P. Note that alias condition is trivially true when $L = L'$.

The roles of the different paths in the alias conditions are also illustrated in Fig. A.2. One can verify that the alias of paths L and L' in the bottom configuration is simulated by two aliases in the first one. Note that the last alias is local to an activity.

Property 9 (Equivalence relation)**.** $\equiv_F$ *is an equivalence relation.*

In the following equivalence of sub-terms will be needed: *sub-terms are equivalent if they are part of equivalent expressions*.

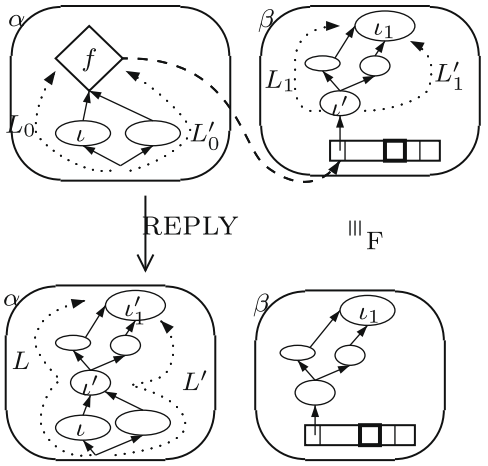

Fig. A.2. The principle of the alias conditions.

Definition 23 (*Equivalence of sub-terms*)**.**

$$a \equiv_F a' \Leftrightarrow \exists L,\ a \in \alpha_P \wedge a' \in \alpha_Q \wedge P \equiv_F Q \wedge \left(\alpha_P \overset{\alpha^*}{\mapsto}_L a, \Leftrightarrow \alpha_Q \overset{\alpha^*}{\mapsto}_L a'\right)$$

This definition means that equivalent sub-terms will be implicitly related to the configurations to which they belong. Consequently, writing $a \equiv_F a'$ will suppose that there are two configurations P and Q such that $P \equiv_F Q$. The definition of equivalence modulo future updates on configurations has the following consequences on the sub-terms.

Lemma 2 (Sub-term equivalence)**.**

$$\begin{aligned} a \equiv_F a' \quad \Rightarrow \quad & \forall L \left(\exists b,\ a \overset{\alpha^*}{\mapsto}_{L'} b \Leftrightarrow \exists b',\ a' \overset{\alpha^*}{\mapsto}_{L'} b'\right) \\ & \wedge \forall L, L', b,\ a \overset{\alpha}{\mapsto}_L b \wedge a \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \exists c, L_0, L'_0, b', L_1, L'_1, \gamma \begin{cases} L = L_0.L_1 \wedge L' = L'_0.L'_1 \wedge L_1 \neq \emptyset \\ a' \overset{\alpha^*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} b' \ \wedge\ a' \overset{\alpha^*}{\mapsto}_{L'_0} c \overset{\gamma}{\mapsto}_{L'_1} b' \end{cases} \\ & \wedge \forall L, L', b, a' \overset{\alpha}{\mapsto}_L b \wedge a' \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \exists c, L_0, b', L_1, L'_1, \gamma \begin{cases} L = L_0.L_1 \wedge L' = L'_0.L'_1 \wedge L_1 \neq \emptyset \\ a \overset{\alpha^*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} b' \ \wedge\ a \overset{\alpha^*}{\mapsto}_{L'_0} c \overset{\gamma}{\mapsto}_{L'_1} b' \end{cases} \end{aligned}$$

A.4. Properties of $\equiv_F$

This section details properties of $\equiv_F$ that are useful in order to prove Property 4 and finally confluence. The following property is a consequence of Tables A.1, A.2, and Definition 22:

Property 10 (Equivalence and compatibility)**.**

$$P \equiv_F Q \Rightarrow P \bowtie Q$$

Consider now the case where a new entry is added to the store of two equivalent terms and is referenced from the same place in both terms. Adding equivalent sub-terms at the same place in two equivalent configurations produces equivalent configurations:

Lemma 3 ($\equiv_F$ and store update)**.**

$$\begin{cases} P \equiv_F Q \quad \wedge \quad a \equiv_F a' \quad \wedge \quad \vdash P' \text{ok} \wedge \vdash Q' \text{ok} \\ \iota \in dom(\sigma_{\alpha_P}) \wedge \iota' \in dom(\sigma_{\alpha_Q}) \quad \wedge \quad \iota \equiv_F \iota' \\ P' = P \text{ except } \sigma_{\alpha_{P'}} = \{\iota \to a\} + \sigma_{\alpha_P} \\ Q' = Q \text{ except } \sigma_{\alpha_{Q'}} = \{\iota' \to a'\} + \sigma_{\alpha_Q} \end{cases} \Rightarrow P' \equiv_F Q'$$

The condition "$\vdash P'\text{OK} \wedge \vdash Q'\text{OK}$" ensures that local or generalized references inside a and a' are already defined in P and Q. Furthermore, $\iota \equiv_F \iota'$ is a necessary condition because ι and ι' are already in P and Q and thus must be reached by the same path. An equivalent version consists in replacing the condition $\iota \in dom(\sigma_{\alpha_P}) \wedge \iota' \in dom(\sigma_{\alpha_Q}) \wedge \iota \equiv_F \iota'$ by: $\exists L, \alpha_P \overset{\alpha}{\mapsto}_L \iota \wedge \alpha_Q \overset{\alpha}{\mapsto}_L \iota$.

Lemma 4 ($\equiv_F$ and substitution)**.**

$$\iota \equiv_F \iota' \Rightarrow a \{\!\{ x \leftarrow \iota \}\!\} \equiv_F a \{\!\{ x \leftarrow \iota' \}\!\}$$

This lemma proves the soundness of $\equiv_F$ with respect to the substitution applied in the INVOKE rule.

Deep copy is defined by: $Copy\&Merge(\sigma, \iota; \sigma', \iota') \triangleq Merge(\iota', \sigma', copy(\iota,\sigma) \{\!\{\iota \leftarrow \iota'\}\!\})$. This can be reformulated using the notion of paths as follows.

Lemma 5 (A characterization of deep copy)**.**

$$a \in copy(\iota, \sigma_\beta) \Leftrightarrow \exists L, \iota \overset{\beta}{\mapsto}_L a$$

This characterization only describes the domain of the deep copy; the following requirement is still necessary to specify that the codomain of the copied store is identical to the original one: $\iota' \in dom(copy(\iota, \sigma)) \Rightarrow copy(\iota, \sigma)(\iota') = \sigma(\iota')$. The following lemma is a consequence of the preceding properties.

Lemma 6 (Copy and merge)**.**
If $P' = P$ *except* $\sigma_{\alpha_{P'}} = Copy\&Merge(\sigma_{\beta_P}, \iota_0 ; \sigma_{\alpha_P}, \iota)$ *then*

$$\begin{cases} \iota_0 \overset{\beta_P}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}}{\mapsto}_L a' \\ \iota_0 \overset{\beta_P{}^*}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}{}^*}{\mapsto}_L a' \end{cases}$$

Lemma 6 states that the part of the store that is deeply copied verifies Lemma 5 and thus paths starting from the destination of the deep copy in $\alpha_{P'}$ are the same as paths starting from the source location in β_P. The following property states that adding equivalent deep copies to equivalent configurations produces equivalent configurations.

Lemma 7 ($\equiv_F$ and store merge)**.**

$$\begin{cases} P \equiv_F Q \wedge \iota \in \alpha_P \wedge \iota' \in \alpha_Q \wedge \iota_0 \in \beta_P \wedge \iota'_0 \in \beta_Q \\ a \equiv_F a' \wedge \iota \equiv_F \iota' \wedge \iota_0 \equiv_F \iota'_0 \\ P' = P \text{ except } \sigma_{\alpha_{P'}} = Copy\&Merge(\sigma_{\beta_P}, \iota_0 ; \sigma_{\alpha_P}, \iota) \\ Q' = Q \text{ except } \sigma_{\alpha_{Q'}} = Copy\&Merge(\sigma_{\beta_Q}, \iota'_0 ; \sigma_{\alpha_Q}, \iota') \end{cases} \Rightarrow P' \equiv_F Q'$$

A.5. Sufficient conditions for equivalence

The following properties relate the formal definition of $\equiv_F$ with the intuitive one saying that two configurations are equivalent modulo future updates if they differ only by the update of some calculated futures.

Property 11 (REPLY and $\equiv_F$)**.**

$$P \xrightarrow{\text{REPLY}} P' \Rightarrow P \equiv_F P'$$

This property is proved by checking that the updated store is equivalent with the old one. More precisely, we have the following sufficient condition for equivalence modulo future updates:

Property 12 (Sufficient condition for equivalence)**.**

$$\begin{cases} P_1 \xrightarrow{\text{REPLY}} P' \\ P_2 \xrightarrow{\text{REPLY}} P' \end{cases} \Rightarrow P_1 \equiv_F P_2 \qquad \begin{cases} P \xrightarrow{\text{REPLY}} P_1 \\ P \xrightarrow{\text{REPLY}} P_2 \end{cases} \Rightarrow P_1 \equiv_F P_2$$

These assertions are easily proved by transitivity of $\equiv_F$. Recall this condition is not a necessary condition for equivalence modulo future updates as it does not deal with mutual references between futures as shown in Fig. 12.

A.6. Equivalence modulo future updates and reduction

We prove here that if a reduction can be made on a configuration then the same one can be made on an equivalent configuration. This was expressed by Property 4 of Section 5.3 recalled below:

$$P \equiv_F P' \wedge P \xrightarrow{T} Q \Rightarrow \exists Q', P' \xRightarrow{T} Q' \wedge Q' \equiv_F Q$$

The proof consists of two parts. First, one may need to apply several REPLY rules before performing the same reduction on the two terms. Indeed one of the configurations may require a future update to perform the reduction that is feasible on the equivalent term: by definition of equivalence modulo future updates, some futures may be updated in a configuration and calculated but not updated in an equivalent configuration. The second part shows that the same reduction rule applied on two equivalent terms leads to equivalent terms.

Proof. Note that, from a given source configuration a reduction is uniquely specified by the name of the applied rule and the names of the different activities concerned, except for REPLY where the future identifier is necessary.

If necessary, $\xrightarrow{\text{REPLY}}$ is applied enough times on P' ($P' \xrightarrow{\text{REPLY}^*} P''$) to be able to apply the same reduction as $P \longrightarrow Q$ (same rule on the same activities) on P'. Indeed, for each awaited future reference, since P can perform the reduction, the future has already been calculated in P, and $P \equiv_F P'$ implies that the future has also been calculated in P'. Thus P' only needs to update it.

More precisely, it is straightforward to check that if two configurations are equivalent, the same reduction can be applied on the two configurations except if one of them is stuck. P' n be stuck in two situations:

- In the case of a forbidden access to an object (e.g., field access on an active object or non-existing field or method) by the definition of equivalence, the reduction on the two equivalent terms should lead to the same error. This is impossible because P can be reduced.
- In the case of an access to a future (wait-by-necessity): if in an activity of P' one has $a_{\alpha_{P'}} = \ldots \iota' \ldots$ and $\sigma_{\alpha_{P'}}(\iota') = \mathit{fut}(f_i^{\gamma \to \beta})$ and the operation performed on ι' is strict; additionally in P, $a_{\alpha_P} = \ldots \iota \ldots$ and $\sigma_{\alpha_P}(\iota)$ is not a future. The future equivalence ensures that $f_i^{\gamma \to \beta} \in F_{\beta_{P'}}$. Then it is possible to update $f_i^{\gamma \to \beta}$ in P': $P' \xrightarrow{\text{REPLY}} P_1$. If in P_1 $\sigma_{\alpha_{P'_1}}(\iota') = \mathit{fut}(f_j^{\gamma \to \beta})$ then, another time, we update the future f_j. After a finite number of updates, we obtain P'' such that $P' \xrightarrow{\text{REPLY}^*} P''$ and $\sigma_{\alpha_{P''}}(\iota')$ is not a future reference. Indeed, if the number of updates was infinite, then neither P' nor P could be reduced, which would contradict the hypothesis.

  Then $P' \xrightarrow{\text{REPLY}^*} P''$ where $P'' \equiv_F P$ (Property 11) and in P'' $\sigma_{\alpha_{P''}}(\iota')$ is not a future reference. Then the same reduction can be applied on P'' and P. Actually the REPLY rule needs to be applied:

  – 0 times if the object to be accessed is not a future,
  – 1 time if it is actually a future whose value is not a future,
  – several times if it is a future whose future value is itself a future reference.

Note that only the objects required by the reduction T must be updated.

Now, one has to verify that if $P'' \equiv_F P$ and the same reduction rule is applied on P and P'', one obtains equivalent configurations:

$$P \xrightarrow{T} Q \ \wedge \ P'' \xrightarrow{T} Q' \ \wedge \ P'' \equiv_F P \ \Rightarrow \ Q' \equiv_F Q$$

where both applications of the rule T are the same (same application points).

The proof consists in a (long) case analysis detailed below. The different cases depend on the reduction applied and the rules applied to prove the equivalence. In the following the proofs will focus only on the cases where one of the locations involved in the reduction points to a future in P and is an object in P'' (updated future). Other cases (several futures or no future) can be trivially obtained. Of course, we will use the fact that if two terms are equivalent, they have the same form (such arguments are only detailed for the FIELD rule). In the following, no details about the renaming of futures and locations are given: one could easily prove a first step that deals with renaming:

$$P \xrightarrow{T} Q \ \wedge \ P'' \xrightarrow{T} Q' \ \wedge \ P'' \equiv P \ \Rightarrow \ Q' \equiv Q$$

LOCAL One should consider cases depending on the local rule applied:

STOREALLOC Consequence of Lemma 3.

FIELD

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \qquad k \in 1..n}{(\mathscr{R}[\iota.l_k], \sigma) \rightarrow_S (\mathscr{R}[\iota_{k1}], \sigma)}$$

Because of the equivalence between current terms of P and P', one has:

$$a_P = \iota.l_k \equiv_F \iota_2.l_k = a_{P''}$$

then $\iota \equiv_F \iota_2$ and $\iota_{k1} \equiv_F \iota_{k2}$ (where ι_{k2} is the location of field l_k in P'') because $\iota \overset{\alpha_P}{\mapsto}_{l_k} \iota_{k1}$ and $\iota_2 \overset{\alpha_{P''}}{\mapsto}_{l_k} \iota_{k2}$. Thus $P'' \equiv_F Q$.

INVOKE Straightforward: note that the two method bodies must be equivalent and the arguments too. The final equivalence comes from Lemma 4.

UPDATE Direct from Lemma 3 and the equivalence of the involved terms.

CLONE Cloning of futures is forbidden by ASP semantics. Other cases are trivial. This case justifies the fact that cloning a future is considered as a strict operation: the future update consists in a deep copy of the value whereas the *clone* operator performs a shallow clone. Performing a CLONE and then a REPLY rule creates two deep copies of the future value. On the contrary performing a REPLY before a CLONE reduction creates only one deep copy with two shallow copies of the first object of the future value.

NEWACT

$$\frac{\begin{array}{c}\gamma \text{ fresh activity} \\ \iota' \notin dom(\sigma) \qquad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \qquad \sigma_\gamma = copy(\iota_2, \sigma)\end{array}}{\alpha[\mathscr{R}[Active(\iota_2, m_j)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathscr{R}[\iota']; \sigma'; \iota; F; R; f] \parallel \gamma[\iota_2.m_j(); \sigma_\gamma; \iota_2; \emptyset; \emptyset; \emptyset] \parallel P}$$

The only interesting case is the presence of futures in the newly created activity. In this case, Lemma 7 is sufficient to conclude. Indeed in NEWACT, $\sigma_\gamma = copy(\iota'', \sigma_\alpha)$ could be written $\sigma_\gamma = Copy\&Merge(\sigma_\alpha, \iota''\ ;\ \emptyset, \iota'')$.

REQUEST

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\rightarrow\beta} \text{ new future} \qquad \iota_f \notin dom(\sigma_\alpha) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota'\ ;\ \sigma_\beta, \iota'') \qquad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\rightarrow\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{l}\alpha[\mathscr{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[\mathscr{R}[\iota_f]; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha\rightarrow\beta}]; f_\beta] \parallel P\end{array}}$$

Modulo renaming, one can choose the same name for the created future in P and Q, and the same location for the copy of the argument. Lemma 7 can be applied to manage futures that can be present in the deep copy of the requests parameters.

The rest of the proof is straightforward. For example, the equivalence of requests $([m_j; \iota; f_i^{\alpha\rightarrow\beta}] \equiv_F [m'_j; \iota; f_i^{\alpha\rightarrow\beta}])$ is ensured because we take the same location, the same future name, and $m_j = m'_j$ because $a_{\alpha_P} \equiv_F a_{\alpha_{P'}}$.

SERVE Informally, the equivalence between the two request lists implies that the served requests are equivalent, which is sufficient to conclude. The fact that the equivalence definition is defined modulo a reordering of requests is essential here; more precisely: $P'' \equiv_F P \Rightarrow \forall M \in \mathcal{M}_{\alpha_P}, R_{\alpha_P}|_M \equiv_F R_{\alpha_{P''}}|_M$. Thus the first request of $R_{\alpha_P}|_M$ will be equivalent modulo future updates in both configurations. Consequently, SERVE will serve equivalent requests.

ENDSERVICE The equivalence between future lists is straightforward. The proof is based on the application of Lemma 7.

REPLY In this case $P' \equiv_F Q$ and $P' \Longrightarrow'$. Thus $Q = P' = P''$ is sufficient.

Note that most of this proof has been simplified by Lemma 7. □

Corollary 5 (page 473) is a direct consequence of the property shown above. More precisely, if $T =$ REPLY then the proof is straightforward. Else $P \overset{T}{\Longrightarrow} Q$ can be decomposed in $P \xrightarrow{\text{REPLY}^*} P_1 \xrightarrow{T} Q$. The conclusion comes from the application of the preceding property to P_1; $P_1 \equiv_F P'$ because $\equiv_F$ is transitive.

Appendix B. Details on the confluence theorem proof

This appendix proves the confluence theorem of Section 5. After some notations and preliminary lemmas, we focus on a local confluence property; finally Section B.3 generalizes local confluence and proves Theorem 1, recalled below:

$$P \xrightarrow{*} Q_1 \ \wedge\ P \xrightarrow{*} Q_2 \ \wedge\ Q_1 \bowtie Q_2 \ \Rightarrow\ Q_1 \curlyvee Q_2$$

Let P_0 be an initial configuration. Let us consider two configurations Q and Q' obtained from the same initial one: $P_0 \xrightarrow{*} Q$, $P_0 \xrightarrow{*} Q'$. Let us suppose that the two configurations are compatible: $Q \bowtie Q'$ that is to say their RSLs have a least upper bound. Let $\mathcal{Q}(Q, Q')$ be the set of configurations obtained from P_0 and having requests sender lists smaller than the ones of Q or Q':

$$\mathcal{Q}(Q,Q') = \{R | P_0 \xrightarrow{*} R \ \wedge \ \forall \alpha \in R,\ RSL(\alpha_R)\big|_M \trianglelefteq (RSL(\alpha_Q)\big|_M \sqcup RSL(\alpha_{Q'})\big|_M)\}$$

Note that $Q, Q' \in \mathcal{Q}$ and also for all intermediate configuration between P and Q (P' such that $P \xrightarrow{*} P' \xrightarrow{*} Q$), one has $P' \in \mathcal{Q}$. To prove confluence, we must reduce terms Q and Q' to a common one by performing the missing reductions from Q and Q'. The terms derived from Q and Q' will be constrained to stay inside $\mathcal{Q}$ to ensure that a common term will be reached.

B.1. Lemmas

We consider configurations where all the futures and the active objects of σ, σ_0, and σ' are well defined (it is necessary for $\equiv_F$ to be well defined). That is to say, all stores and expressions are parts of a well-formed parallel configuration.

The following lemmas gives properties on the appending and merging of stores.

Lemma 8 (Independent stores)**.**

$$dom(\sigma_1) \cap dom(\sigma_2) = \emptyset \Rightarrow \begin{cases} \sigma_1 :: \sigma_2 = \sigma_2 :: \sigma_1 \\ \sigma_1 + \sigma_2 = \sigma_2 + \sigma_1 \\ \sigma_1 + (\sigma_2 :: \sigma) = \sigma_2 :: (\sigma_1 + \sigma) \end{cases}$$

Lemma 9. *Local reduction can be extended with an independent store:*

$$(a, \sigma) \rightarrow_S (a', \sigma') \Rightarrow (a, \sigma :: \sigma_0) \rightarrow_S (a'', \sigma'' :: \sigma_0) \text{ where } (a'', \sigma'') \equiv_F (a', \sigma')$$

Lemma 10 (Multiple copies)**.** *Several store copies compose as follows:*

$$\iota \in dom(copy(\iota', \sigma')) \ \Rightarrow \ \big(copy(\iota, \sigma) + copy(\iota', \sigma') = copy(\iota', copy(\iota, \sigma) + \sigma')\big)$$

Additionally adding an independent store can be done either before or after merging two other stores, provided locations created during the merge are chosen conveniently. That is due to the fact that configurations can be identified modulo renaming of locations:

Corollary 1 (*Copy and store update*)**.** *If $\iota' \notin dom(\sigma')$ then there is a way of choosing locations allocated by Copy&Merge$(\sigma_1, \iota\,;\, \sigma_2, \iota')$ such that:*

$$\sigma' + \mathit{Copy\&Merge}(\sigma_1, \iota\,;\, \sigma_2, \iota') \equiv_F \mathit{Copy\&Merge}(\sigma_1, \iota\,;\, \sigma' + \sigma_2, \iota')$$

B.2. Local confluence

This section presents and proves what we call *local confluence*, that is a classical confluence property starting from a given term and performing two concurrent reductions; it is necessary to establish confluence properties of Section 5. This property is strongly based on the definition of compatibility between configurations ($\bowtie$) via the use of the set $\mathcal{Q}$.

Property 13 (Diamond property)**.**
Let P be a configuration obtained from P_0: $P_0 \xrightarrow{*} P$

$$\begin{cases} P \xrightarrow{T_1} P_1 \\ P \xrightarrow{T_2} P_2 \\ P, P_1, P_2 \in \mathcal{Q}(Q, Q') \end{cases} \Longrightarrow P_1 \equiv_F P_2 \ \vee \ \exists P'_1, P'_2, \begin{cases} P_1 \xrightarrow{T_2} P'_1 \\ P_2 \xrightarrow{T_1} P'_2 \\ P'_1 \equiv_F P'_2 \\ P'_1, P'_2 \in \mathcal{Q}(Q, Q') \end{cases}$$

Proof. This proof is a (long) case study on the conflict (concurrency) between rules. Cases where one of the applied rules is REPLY will not be detailed. These cases can be verified but are not useful for the proof of the Property 14. Indeed, for REPLY rule we only need to use Property 4.

This analysis is only interesting when there is a real conflict between two rules. That is to say at least a component of one activity can be read or modified by two rules.

Recall we can suppose that one can choose any location or future name when one needs a fresh one. The fact that activities are chosen deterministically avoids the problem of renaming activities and ensures that the name of an activity will be the same for two application of the same NEWACT rule.

In the following, if the conflicting rules are different, the activities (α, β) will be indexed by the corresponding rule (e.g. α_{REQUEST} is the activity α of the REQUEST rule: the source activity of the request). If the rules are the same, the activities will be indexed by 1 and 2. The proof can be divided into four parts. Except for concurrent request sending part, the fact that $P'_1, P'_2 \in \mathcal{Q}(Q, Q')$ is straightforward as the RSL of P'_1 and P'_2 are either the ones of P_1 or the ones of P_2.

Local vs. Parallel Reduction

LOCAL/LOCAL. Obvious consequence of the determinism of local reduction.

LOCAL/NEWACT $\alpha_{\text{LOCAL}} = \alpha_{\text{NEWACT}}$ impossible because $\mathscr{R}[Active(\iota, m)]$ cannot be reduced locally. No conflict.

LOCAL/REQUEST $\alpha_{\text{LOCAL}} = \alpha_{\text{REQUEST}}$ impossible (this would correspond to a method call which would be both local and distant).

$\alpha_{\text{LOCAL}} = \beta_{\text{REQUEST}}$ let $\alpha = \alpha_{\text{REQUEST}}$ and $\beta = \alpha_{\text{LOCAL}} = \beta_{\text{REQUEST}}$

$$\text{LOCAL}\quad \frac{(a_\beta, \sigma_\beta) \to_S (a_{\beta 1}, \sigma_{\beta 1})}{\beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| Q \longrightarrow \beta[a_{\beta 1}; \sigma_{\beta 1}; \iota_\beta; F_\beta; R_\beta; f_\beta] \| Q = P_1}$$

$$\text{REQUEST}\quad \frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \\ \iota'' \notin dom(\sigma_\beta) \quad f_i^{\alpha\to\beta} \text{new future} \quad \iota_f \notin dom(\sigma_\alpha) \quad \sigma_{\beta 2} = Copy\&Merge(\sigma_\alpha, \iota' ; \sigma_\beta, \iota'') \quad \sigma_{\alpha 2} = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathscr{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| Q \longrightarrow \\ \alpha[\mathscr{R}[\iota_f]; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_{\beta 2}; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha\to\beta}]; f_\beta] \| Q = P_2\end{array}}$$

Let us first perform the local reduction on P_2. One can suppose (up to renaming) that the locations added to σ_β by the two rules are disjuncts. The deep copy of the argument of the request is added in an independent store; thus $\sigma_{\beta 2} = \sigma :: \sigma_\beta$. Thus Lemma 9 allows us to perform the local reduction on the extended store:

$$(a_\beta, \sigma_\beta) \to_S (a_{\beta 1}, \sigma_{\beta 1}) \Rightarrow (a_\beta, \sigma :: \sigma_\beta) \to_S (a'_{\beta 2}, \sigma :: \sigma'_{\beta 2})$$

where $(a'_{\beta 2}, \sigma'_{\beta 2}) \equiv (a_{\beta 1}, \sigma_{\beta 1})$ and $(a'_{\beta 2}, \sigma :: \sigma'_{\beta 2}) \equiv (a_{\beta 1}, \sigma :: \sigma_{\beta 1})$. Finally:

$$\begin{array}{l} P_2 = \alpha[\mathscr{R}[\iota_f]; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_{\beta 2}; \iota_\beta; F_\beta; R_{\beta 2}; f_\beta] \| Q \\ \qquad \longrightarrow \alpha[\mathscr{R}[\iota_f]; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a'_{\beta 2}; \sigma'_{\beta 2} :: \sigma; \iota_\beta; F_\beta; R_{\beta 2}; f_\beta] \| Q = P'_2 \end{array}$$

Now we will focus on the application of the request rule to P_1 and consider that $\sigma_{\beta 1}$ is obtained by some updates on σ_β (indeed every action of a local rule on the store can be written as a store update): $\sigma_{\beta 1} = \sigma_0 + \sigma_\beta$.

Corollary 1 is used for adding the request to the store obtained by local reduction. One can apply the request rule to P_1. Let $\sigma'_{\beta 1}$ be the new store:

$$\begin{array}{ll} \sigma'_{\beta 1} = Copy\&Merge(\sigma_\alpha, \iota' ; \sigma_{\beta 1}, \iota'') & \equiv_F \sigma_0 + Copy\&Merge(\sigma_\alpha, \iota' ; \sigma_\beta, \iota'') \\ & \equiv_F \sigma_0 + \sigma_{\beta 2} \end{array} \tag{4}$$

and obtain a configuration equivalent to P'_2 by Lemma 8. More precisely:

$$\begin{array}{lll} (a'_{\beta 2}; \sigma :: \sigma'_{\beta 2}) & \equiv_F (a_{\beta 1}, \sigma :: \sigma_{\beta 1}) \equiv_F (a_{\beta 1}, \sigma :: (\sigma_0 + \sigma_\beta)) & \\ & \equiv_F (a_{\beta 1}, \sigma_0 + (\sigma :: \sigma_\beta)) \equiv_F (a_{\beta 1}, \sigma'_{\beta 1}) & \text{by}(4) \end{array}$$

LOCAL/ENDSERVICE, *and* LOCAL/SERVE No conflict.

Creating an Activity

NEWACT/NEWACT, NEWACT/ENDSERVICE, *and* NEWACT/SERVE No conflict.

NEWACT/REQUEST One only has to prove that (if $\alpha_{\text{NEWACT}} = \beta_{\text{REQUEST}}$) creating a new activity does not interfere with receiving a request. This is similar to the case LOCAL/REQUEST.

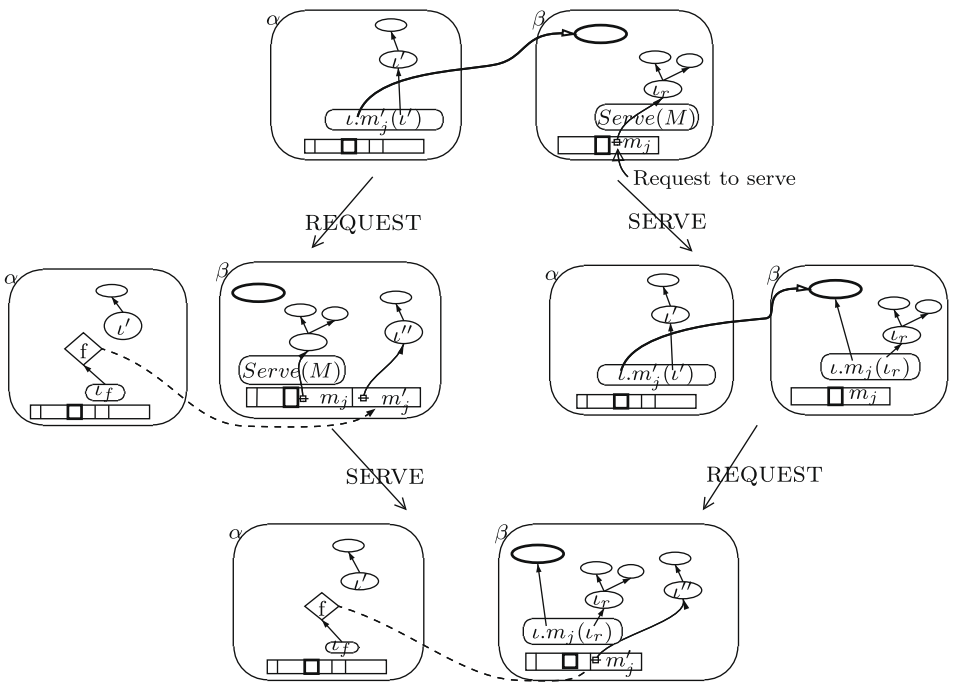

Fig. B.1. SERVE/REQUEST.

*Localized Operations (*SERVE, ENDSERVICE*)*
SERVE/SERVE, ENDSERVICE/SERVE, *and eos*/ENDSERVICE No conflict

SERVE/REQUEST The only conflicting case occurs when $\alpha_{\text{SERVE}} = \beta_{\text{REQUEST}}$. Informally, if one can perform a *serve*(M) on P then there is a request matching the labels of M in the request queue; so adding a new request to the request queue will not change the served one because SERVE takes the *first* request matching M. Fig. B.1 illustrates this case (we consider that this figure is sufficiently explicit to avoid us giving the technical details of the proof), we suppose in this figure that $m_j \in M$. Note that the fact that the first request matching a pattern is taken is essential to ensure confluence.

REQUEST/ENDSERVICE Conflict is only possible when $\alpha_{\text{ENDSERVICE}} = \beta_{\text{REQUEST}} = \beta$, the principle of the proof concerning this case is shown in Fig. B.2.

REQUEST

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\to\beta}\text{newfuture} \qquad \iota_f \notin dom(\sigma_\alpha)\\ \sigma_{\beta 1} = Copy\&Merge(\sigma_\alpha, \iota' ;\ \sigma_\beta, \iota'') = \sigma + \sigma_\beta \qquad \sigma_{\alpha 1} = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathscr{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| Q\\ \longrightarrow \alpha[\mathscr{R}[\iota_f]; \sigma_{\alpha 1}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_{\beta 1}; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha\to\beta}]; f_\beta] \| Q = P'_1\end{array}}$$

ENDSERVICE

$$\frac{\begin{array}{c}\iota' \notin dom(\sigma_\beta)\\ F'_\beta = F_\beta :: \{f_\beta \mapsto \iota'\} \qquad \sigma_{\beta 2} = Copy\&Merge(\sigma_\beta, \iota ;\ \sigma_\beta, \iota') = \sigma' + \sigma_\beta\end{array}}{\beta[\iota \Uparrow f_i^{\delta\to\beta}, a; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| P \longrightarrow \beta[a; \sigma_{\beta 2}; \iota_\beta; F'_\beta; R_\beta; f_i^{\delta\to\beta}] \| P = P'_2}$$

The conflict only concerns the store. But the merges that are performed on the store are independent ($\iota'' \notin dom(\sigma_\beta)$); and we can suppose that these two operations create disjuncts sets of locations. Then one can perform the missing rule on each configuration: P'_1 and P'_2. A configuration with the following stores is obtained (modulo renaming, the same stores updates as in the first two rules can be performed): $\sigma'_{\beta 2} \equiv_F \sigma + \sigma_{\beta 2}$, and $\sigma' + \sigma_{\beta 1} \equiv_F \sigma'_{\beta 1}$. Then, the proof relies on the following equality (using Lemma 8):

$$\sigma'_{\beta 2} \quad \equiv_F \sigma + \sigma_{\beta 2} = \sigma + \sigma' + \sigma_\beta = \sigma' + \sigma + \sigma_\beta = \sigma' + \sigma_{\beta 1} \equiv_F \sigma'_{\beta 1}$$

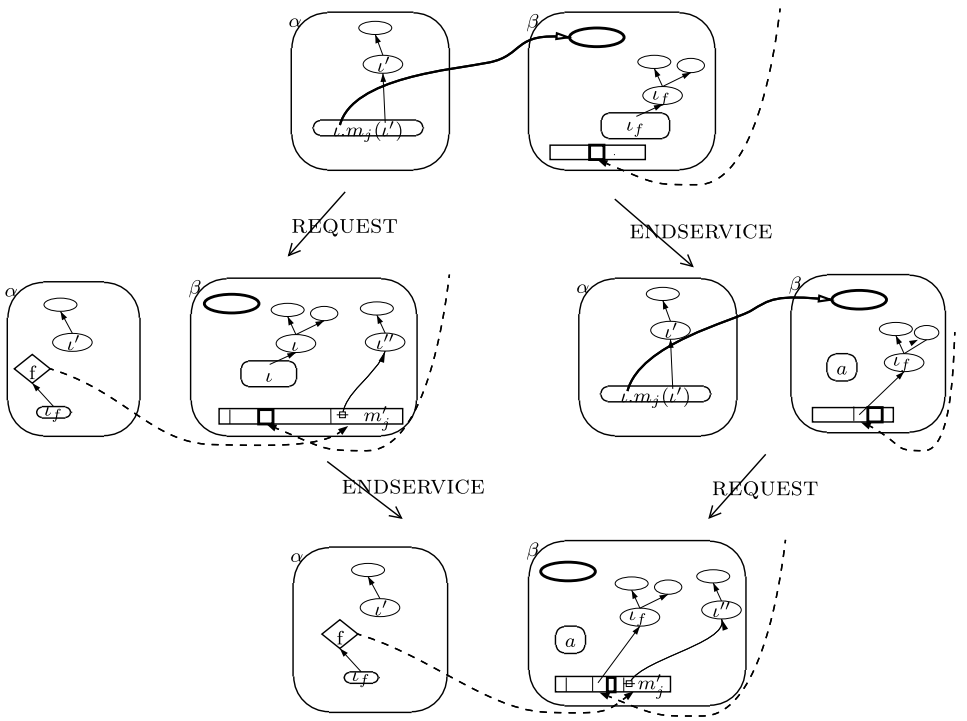

Fig. B.2. ENDSERVICE/REQUEST.

Concurrent Request Sending: REQUEST/REQUEST

$\alpha_1 = \beta_2$ **or** $\beta_1 = \alpha_2$ Same kind of arguments as in the case LOCAL/REQUEST with $\alpha_{\text{LOCAL}} = \beta_{\text{REQUEST}}$.

$\alpha_1 = \alpha_2$ No conflict.

$\beta_1 = \beta_2$ As $P_1, P_2 \in \mathcal{Q}$, if $\beta_1 = \beta_2$ then (RSL compatibility) either the two requests come from the same activity, $\alpha_1 = \alpha_2$ and there is no conflict or the two requests m_1 and m_2 cannot interfere: $\nexists M \in \mathcal{M}_{\alpha_{P_0}}$ such that $\{m_1, m_2\} \subseteq M$. In that second case, adding requests in any order leads to equivalent configurations due to the equivalence rule for pending requests. Let P'_1 be the configuration obtained from P_1 by sending the missing request from α_2; and P'_2 the configuration obtained from P_2 by sending the missing request from α_1. The crucial point is to show that the RSL are compatible, let $m_1 \in M_1$ and $m_2 \in M_2$ (M_1 and M_2 are not unique but belong to disjoint sets):

$$RSL(\alpha_{P'_1})|_{M_1} = RSL(\alpha_{P_1})|_{M_1} = RSL(\alpha_{P'_2})|_{M_1} \trianglelefteq RSL(\alpha_Q)|_{M_1} \sqcup RSL(\alpha_{Q'})|_{M_1}$$

$$RSL(\alpha_{P'_1})|_{M_2} = RSL(\alpha_{P_2})|_{M_2} = RSL(\alpha_{P'_2})|_{M_2} \trianglelefteq RSL(\alpha_Q)|_{M_2} \sqcup RSL(\alpha_{Q'})|_{M_2}$$

And finally, $P'_1, P'_2 \in \mathcal{Q}(Q, Q')$, and RSLs are compatible.

The proof of $P'_1 \equiv_F P'_2$ applies the same arguments to pending requests instead of RSLs. □

B.3. Extension

This section extends the local diamond property presented before to obtain a general diamond property which will allow us to conclude about the confluence of ASP calculus.

Lemma 11 ($\equiv_F$ and $\mathcal{Q}(Q, Q')$)**.**
If P is in $\mathcal{Q}$ and P is equivalent modulo future updates to P′ then P′ is in $\mathcal{Q}$:

$$P \equiv_F P' \wedge P \in \mathcal{Q}(Q, Q') \wedge P_0 \xrightarrow{*} P' \Rightarrow P' \in \mathcal{Q}(Q, Q')$$

Lemma 12 (REPLY vs. other reduction)**.** *Let $\xrightarrow{\neg\text{REPLY}}$ consist in applying any reduction except the* REPLY *rule.*

$$P \xrightarrow{\neg\text{REPLY}} R \wedge P \xrightarrow{\text{REPLY}} P' \Rightarrow P' \xrightarrow{\neg\text{REPLY}} R' \wedge R' \equiv_F R$$

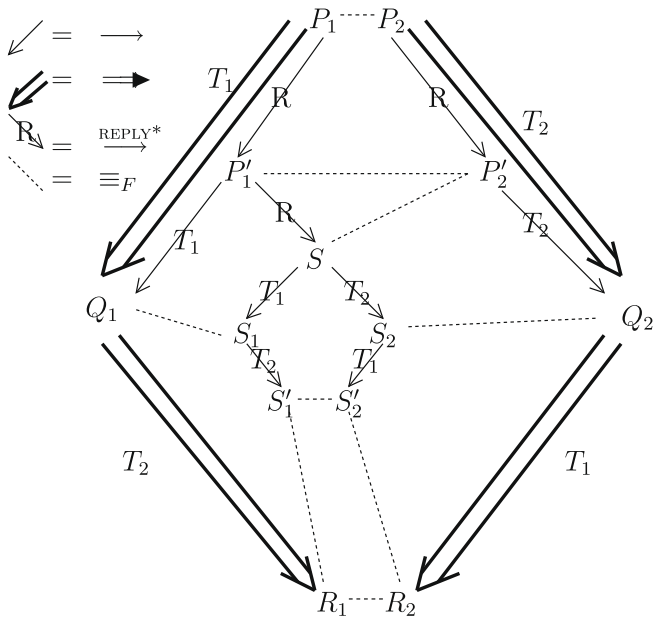

Fig. B.3. The diamond property (Property 14) proof.

Then, global confluence is a consequence of local confluence. The following property is a generalized local confluence:

Property 14 (Diamond property with $\equiv_F$)**.**

$$\begin{cases} P_1 \overset{T_1}{\Longrightarrow} Q_1 \\ P_2 \overset{T_2}{\Longrightarrow} Q_2 \\ Q_1, Q_2 \in \mathcal{Q}(Q, Q') \\ P_1 \equiv_F P_2 \end{cases} \Longrightarrow Q_1 \equiv_F Q_2 \vee \exists R_1, R_2, \begin{cases} Q_1 \overset{T_2}{\Longrightarrow} R_1 \\ Q_2 \overset{T_1}{\Longrightarrow} R_2 \\ R_1 \equiv_F R_2 \\ R_1, R_2 \in \mathcal{Q}(Q, Q') \end{cases}$$

Proof. Fig. B.3 illustrates the proof detailed in the following.

If one of the $\Longrightarrow$ on the left of the implication consists only in some REPLY rules then one can conclude immediately by Property 4 and Property 11. Else, both T_1 and T_2 are reduction rules different from REPLY and can be decomposed: there is P'_1 such that: $P_1 \xrightarrow{\text{REPLY}^*} P'_1 \xrightarrow{T_1} Q_1$. Note that one could have $P_1 = P'_1$. In the same way, there is P'_2 such that: $P_2 \xrightarrow{\text{REPLY}^*} P'_2 \xrightarrow{T_2} Q_2$. By Property 9, $\equiv_F$ is transitive and then $P'_1 \equiv_F P'_2$. By Property 4: $\exists S_2,\ P'_1 \overset{T_2}{\Longrightarrow} S_2 \wedge S_2 \equiv_F Q_2$ Thus there is a configuration S such that:

$$P'_1 \xrightarrow{\text{REPLY}^*} S \wedge S \xrightarrow{T_2} S_2 \wedge S_2 \equiv_F Q_2$$

Moreover, by Lemma 12:

$$\begin{cases} P'_1 \xrightarrow{\text{REPLY}^*} S \\ P'_1 \xrightarrow{T_1} Q_1 \end{cases} \Rightarrow \begin{cases} S \xrightarrow{T_1} S_1 \\ S_1 \equiv_F Q_1 \end{cases}$$

Then, using diamond Property 13 (Lemma 11 ensures $S_1, S_2 \in \mathcal{Q}$):

$$\begin{cases} S \xrightarrow{T_1} S_1 \\ S \xrightarrow{T_2} S_2 \\ S_1, S_2 \in \mathcal{Q} \end{cases} \Longrightarrow S_1 \equiv_F S_2 \vee \exists R_1, R_2, \begin{cases} S_1 \xrightarrow{T_2} S'_1 \\ S_2 \xrightarrow{T_1} S'_2 \\ S'_1 \equiv_F S'_2 \ \wedge\ S'_1, S'_2 \in \mathcal{Q} \end{cases}$$

Finally, using Property 4:

$$S_1 \xrightarrow{T_2} S_1' \wedge S_1 \equiv_F Q_1 \Rightarrow Q_1 \stackrel{T_2}{\Longrightarrow} R_1 \wedge S_1' \equiv_F R_1$$

$$S_2 \xrightarrow{T_1} S_2' \wedge S_2 \equiv_F Q_2 \Rightarrow Q_2 \stackrel{T_1}{\Longrightarrow} R_2 \wedge S_2' \equiv_F R_2$$

Note that $R_1 \equiv_F R_2$ and $R_1, R_2 \in \mathcal{Q}$ are obtained trivially and finally:

$$\begin{cases} Q_1 \stackrel{R}{\Longrightarrow} 1 & \wedge & Q_2 \stackrel{R}{\Longrightarrow} 2 \\ R_1 \equiv_F R_2 & \wedge & R_1, R_2 \in \mathcal{Q} \end{cases} \quad \square$$

Proving confluence from Property 14 is a classical result. It only relies on the fact that all intermediate configurations between P_0 and Q and between P_0 and Q' belong to $\mathcal{Q}$.

References

[1] Cliff B. Jones, An object-based design method for concurrent programs, Technical Report, University of Manchester, 1992.
[2] Uwe Nestmann, Martin Steffen, Typing confluence, in: Stefania Gnesi, Diego Latella (Eds.), FMICS '97: Second International ERCIM Workshop on Formal Methods in Industrial-Critical Systems, pp. 77–101. Consiglio Nazionale Ricerche di Pisa, Italy, July 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.
[3] Naoki Kobayashi, Benjamin C. Pierce, David N. Turner, Linearity and the pi-calculus, Proceedings of POPL '96, ACM, 1996, pp. 358–371.
[4] Guy L. Steele Jr., Making asynchronous parallelism safe for the world, in: POPL'90, Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, January 17–19, 1990, San Francisco, CA, ACM Press, New York, 1990, pp. 218–231.
[5] Gilles Kahn, The semantics of a simple language for parallel programming, in: J.L. Rosenfeld (Ed.), Information Processing '74: Proceedings of the IFIP Congress, North-Holland, New York, 1974.
[6] F. Baude, D. Caromel, C. DelbT, L. Henrio, Promised messages: recovering from inconsistent global states, in: ACM SIGOPS Conference Principles and Practice of Parallel Programming (PPoPP), Poster., 2007.
[7] Martín Abadi, Luca Cardelli, A Theory of Objects, Springer-Verlag, New York, 1996.
[8] Denis Caromel, Towards a method of object-oriented concurrent programming, Communications of the ACM 36 (9) (1993) 90–102.
[9] Denis Caromel, Wilfried Klauser, Julien Vayssière, Towards seamless computing and metacomputing in Java, Concurrency: Practice and Experience 10 (11–13) (1998) 1043–1061., ProActive available at http://www.inria.fr/oasis/proactive.
[10] Denis Caromel, Ludovic Henrio, Bernard Paul Serpette, Asynchronous and deterministic objects, Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press, 2004, pp. 123–134.
[11] Denis Caromel, Ludovic Henrio, A Theory of Distributed Objects, Springer-Verlag, Inc., New York, 2005.
[12] Uwe Nestmann, Hans Hüttel, Josva Kleist, Massimo Merro, Aliasing models for mobile objects, Information and Computation 175 (1) (2002) 3–33.
[13] Andrew D. Gordon, Paul D. Hankin, Søren B. Lassen, Compilation and equivalence of imperative objects, in: S. Ramesh, G. Sivakumar (Eds.), FSTTCS, Lecture Notes in Computer Science, vol. 1346, Springer, 1997, pp. 74–87.
[14] Gilles Kahn, David MacQueen, Coroutines and networks of parallel processes, in: B. Gilchrist (Ed.), Information Processing '77: Proceedings of IFIP Congress, North-Holland, Amsterdam, 1977, pp. 993–998.
[15] Brian Cantwell Smith, Reflection and semantics in Lisp, in: Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, January 15–18, 1984, ACM Press, New York, pp. 23–35.
[16] Bernard Lang, Christian Queinnec, José Piquer, Garbage collecting the world, in: Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, ACM SIGPLAN Notices, January 1992, pp. 39–50.
[17] Fabrice Le Fessant, Detecting distributed cycles of garbage in large-scale systems, in: Conference on Principles of Distributed Computing (PODC), Rhode Island, August 2001.
[18] Denis Caromel, Ludovic Henrio, Asynchonous distributed components: concurrency and determinacy, in: Proceedings of the IFIP International Conference on Theoretical Computer Science 2006 (IFIP TCS'06), Santiago, Chile, Springer Science, August 2006 (19th IFIP World Computer Congress).
[19] Andrew D. Gordon, Paul D. Hankin, A concurrent object calculus: reduction and typing, Proceedings HLCL'98, vol. 16, Elsevier ENTCS, Amsterdam, 1998
[20] Alan Jeffrey, A distributed object calculus, in: ACM SIGPLAN Workshop Foundations of Object Oriented Languages, 2000.
[21] Oscar Nierstrasz, Towards an object calculus, in: M. Tokoro, O. Nierstrasz, P. Wegner (Eds.), Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing, LNCS, vol. 612, Springer-Verlag, 1992, pp. 1–20.
[22] Andrew D. Gordon, Gareth D. Rees, Bisimilarity for a first-order calculus of objects with subtyping, in: Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96) [49], pp. 386–395.
[23] Luca Cardelli, A language with distributed scope, in: Conference Record of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95), San Francisco, January 22–25, 1995, ACM SIGACT-SIGPLAN, ACM Press, New York, pp. 286–297.
[24] Benjamin C. Pierce, David N. Turner, Concurrent objects in a process calculus, in: Takayasu Ito, Akinori Yonezawa (Eds.), Proceedings of Theory and Practice of Parallel Programming (TPPP'94), Sendai, Japan, LNCS, Springer-Verlag, Berlin, Heidelberg, 1995, pp. 187–215.
[25] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, part I/II, Journal of Information and Computation 100 (1992) 1–77.
[26] Robin Milner, The polyadic π-calculus: a tutorial, in: Friedrich L. Bauer, Wilfried Brauer, Helmut Schwichtenberg (Eds.), Logic and Algebra of Specification, Series F. NATO ASI, vol. 94, Springer-Verlag, Berlin, Heidelberg, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.
[27] Daren L. Webb, Andrew L. Wendelborn, Julien Vayssière, A study of computational reconfiguration in a Process Network, in: Proceedings of the 7th Workshop on Integrated Data Environments Australia (IDEA'7), February 2000.
[28] Cliff B. Jones, Steve J. Hodges, Non-interference properties of a concurrent object-based language: proofs based on an operational semantics, in: Burkhard Freitag, Cliff B. Jones, Christian Lengauer, Hans-Jörg Schek (Eds.), Object-Orientation with Parallelism and Persistence, Kluwer Academic Publishers, Dordrecht, 1996, pp. 1–22 (Chapter 1).
[29] Cliff B. Jones, Process-algebraic foundations for an object-based design notation, Technical Report, University of Manchester, 1993 (UMCS-93-10-1).
[30] Davide Sangiorgi, The typed π-calculus at work: a proof of Jones's parallelisation theorem on concurrent objects, Theory and Practice of Object-Oriented Systems 5 (1) (1999) (An early version was included in the Informal Proceedings of FOOL 4, January 1997).
[31] Xinxin Liu, David Walker, Confluence of processes and systems of objects, in: Peter D. Mosses, Mogens Nielsen, Michael I. Schwarzbach (Eds.), TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, LNCS, vol. 915, Springer-Verlag, Berlin, Heidelberg, 1995, pp. 217–231.

[32] Xinxin Liu, David Walker, Partial confluence of processes and systems of objects, Theoretical Computer Science 206 (1–2) (1998) 127–162.
[33] Gul Agha, Ian A. Mason, Scott F. Smith, Carolyn L. Talcott, Towards a theory of actor computation (extended abstract), in: W.R. Cleaveland (Ed.), CONCUR'92: Proceedings of the Third International Conference on Concurrency Theory, Springer-Verlag, Berlin, Heidelberg, 1992, pp. 565–579.
[34] Gul Agha, Ian A. Mason, Scott F. Smith, Carolyn L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1) (1997) 1–72.
[35] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, Yasuaki Honda, Modelling and programming in an object-oriented concurrent language ABCL/1, in: A. Yonezawa, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, Cambridge, MA, 1987, pp. 55–89.
[36] Robert H. Halstead Jr., Multilisp: a language for concurrent symbolic computation, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (4) (1985) 501–538.
[37] J. Niehren, J. Schwinghammer, G. Smolka, A concurrent lambda calculus with futures, Theoretical Computer Science 364 (3) (2006) 338–356.
[38] Joachim Niehren, David Sabel, Manfred Schmidt-Schau, Jan Schwinghammer, Observational semantics for a concurrent lambda calculus with reference cells and futures, in: Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII), Electronic Notes in Theoretical Computer Science, vol. 173, New Orleans, April 2, 2007, pp. 313–337
[39] F.S. de Boer, D. Clarke, E. Broch Johnsen, A complete guide to the future, in: ESOP, 2007, pp. 316–330.
[40] Einar Broch Johnsen, Olaf Owe, Ingrid Chieh Yu, Creol: a type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (1–2) (2006) 23–66.
[41] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter, Ambient-oriented programming in ambienttalk, in: D. Thomas (Ed.), ECOOP, Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 230–254.
[42] Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, Didier Remy, A calculus of mobile agents, in: U. Montanari, V. Sassone (Eds.), Proceedings of the 7th International Conference on Concurrency Theory (CONCUR), LNCS, vol. 1119, Springer-Verlag, Berlin, Heidelberg, August 1996, pp. 406–421.
[43] Cédric Fournet, Georges Gonthier, The reflexive CHAM and the join-calculus, in: Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96) [49], pp. 372–385.
[44] Alain Deutsch, Interprocedural may-alias analysis for pointers: beyond *k*-limiting, in: PLDI'94 Conference on Programming Language Design and Implementation, Orlando, Florida, vol. 29, no. 6, ACM SIGPLAN Notices, June 1994, pp. 230–241.
[45] Alain Deutsch, Semantic models and abstract interpretation techniques for inductive data structures and pointers, in: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 21–23, 1995, pp. 226–229.
[46] Mooly Sagiv, Thomas Reps, Susan Horwitz, Precise interprocedural dataflow analysis with applications to constant propagation, Theoretical Computer Science 167 (1–2) (1996) 131–170.
[47] Paulo Sergio Almeida, Balloon types: controlling sharing of state in data types, in: Mehmet Akşit, Satoshi Matsuoka (Eds.), ECOOP'97—Object-Oriented Programming 11th European Conference, Jyvskyl, Finland, vol. 1241, Springer-Verlag, New York, 1997, pp. 32–59.
[48] Isabelle Attali, Denis Caromel, Romain Guider, A step toward automatic distribution of Java programs, in: S.F. Smith, C.L. Talcott (Eds.), FIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Kluwer Academic Publishers, Dordrecht, 2000, pp. 141–161.
[49] ACM SIGACT-SIGPLAN, Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96), St. Petersburg, Florida, January 21–24, ACM Press, 1996.