Asynchronous Sequential Processes

Denis Caromel, Ludovic Henrio

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Asynchronous Sequential Processes

Denis Caromel — Ludovic Henrio — Bernard Serpette

N° 4753 – version 2

version initiale Mars 2003 – version révisée Mai 2003

THÈME 2

ISSN 0249-6399

Asynchronous Sequential Processes

Denis Caromel , Ludovic Henrio , Bernard Serpette

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oasis

Rapport de recherche n° 4753 – version 2*— version initiale Mars 2003 — version révisée Mai 2003 57 pages

Abstract: This document presents an object language that allows one to program parallel and distributed applications that behave in a deterministic manner, even if they are distributed over local or wide area networks.

An object calculus, *Asynchronous Sequential Processes*, is proposed and determinism properties are proposed. ASP main characteristics are asynchronous communications and sequential execution within each parallel activity.

Key-words: object calculus, distributed, parallelism, determinism

* This versions includes an index of notations and a new version of rules in table 3 which contained some minor mistakes. Rules are simplified, some mistakes with notations are corrected (especially inside rules), and explanation of rules pages 18-20 is modified consequently. These modifications should improve the readability of technical parts.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Calcul ASP : Asynchronous Sequential Processes

Résumé : Ce document présente un langage objet permettant d'écrire des applications parallèles et distribuées qui se comportent de façon déterministe, quel que soit le réseau sur lequel elles sont distribuées.

Un calcul objet, *Asynchronous Sequential Processes*, est présenté et des propriétés de déterminisme sont proposées. Les principales caractéristiques du calcul ASP sont des communications asynchrones et une exécution séquentielle dans chaque activité

Mots-clés : calcul objet, distribué, parallélisme, déterminisme

Contents

List of Figures

List of Tables

1 Introduction

Seeking determinism for parallel programming, we design a calculus named *ASP*: *Asynchronous Sequential Processes* and formalize determinism properties on this calculus. ASP models an object language with asynchronous communications and sequential execution within each parallel activity.

We start from a purely sequential and classical object calculus (ς_{imp}-calculus) [1] and extend it with a *single* parallel constructor that turns a standard object into an active one, executing in parallel. Automatic synchronization of processes comes from *wait-by-necessity* [5]: a wait automatically occurs upon a strict operation on a communication result not yet available. Both programs (see the included binary tree example, Figure 2), and semantics illustrate the similarities between the sequential and parallel points of view.

The passing of *futures* (results of asynchronous calls) between processes, both as method parameters and as method results is an important feature of our calculus. As futures can proliferate, a strategy must be specified to choose when and how a value should be updated. Therefore, in practice many strategies can be implemented (e.g. eager, lazy): the ASP calculus captures all the possible update strategies, and thus the demonstrated properties are valid for all of them. While communication is asynchronous, a given process is insensitive to the moment when a result comes back. This is a powerful characteristic of the convergence property we exhibit.

On the practical side, the ASP-calculus model is implemented as an *Application Programmer Interface (API)*, ProActive [6], allowing parallel and distributed programming. It is freely available for experiments.

The most valuable property states that the execution of a set of processes is only determined by the order of arrival of requests and asynchronous replies can occur in an arbitrary order without observable consequence. Further, a stronger condition provides determinism in ASP. Note that this work can be related to linearized channels in π-calculus [21] and seems to some extent more general as will be detailed later on. Moreover, the fact that the model described here is implemented proves that our calculus is based on realistic hypothesis.

This paper is organized as follows. Section 2 compares our calculus with other concurrent calculi and their confluence properties. Section 3 presents the sequential part of our calculus, which is based on the ς_{imp}-calculus of [1]. Section 4 informally introduces the ASP calculus and its principles. section 5 gives some examples. Section 6 presents the semantics of ASP and Section 7 presents its main properties. Appendix gives technical details of some proofs and definitions.

2 Related work

The ASP-calculus is based on the untyped imperative object calculus of Abadi and Cardelli (ς_{imp}-calculus of [1]). Our local semantics looks like the one of [11] but we did not find any concurrent object calculus ([12], [15], [27]) with a similar way of communication between asynchronous objects. Thus our calculus seems to introduce new chacteristics especially

in the way of communication with futures which are interesting both theoretically and in practice.

For example in [4] and the more formal calculus and semantics found in [26], the generalised 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 our concepts.

Our calculus can be rewritten in π-calculus ([25], [24]) but this would not help us to prove our confluence property directly. Under certain restrictions (cf. [32], [21]), π-calculus terms can be statically proved to be confluent and such results could be applicable to some ASP terms. But, even if our confluence property is not necessarily statically verifiable it seems much more powerful. The join-calculus [9], [10] would allow us to express mobility and distribution but the difference of semantics between this calculus and ASP would lead us to the same problems.

Proving equivalence between terms can be performed by introducing bisimulation on an object calculus like in [14]. [11] uses CIU equivalence but deals only with static terms and we are interested in dynamic properties like confluence.

Process networks [19] 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.

[31] expressed a programming model ensuring the confluence of programs by analyzing (mainly dynamically) shared memory accesses in order to ensure non-interference. But, it is based on a shared memory mechanism with asynchronous threads and not on possibly distributed programs.

$\pi o \beta \lambda$ [16, 18] is a concurrent object-oriented language. A condition sufficient for returning from a method before the end of its execution is expressed, thus increasing the concurrency without necessarily losing determinacy. $\pi o \beta \lambda$ can be translated in (dialects of) π-calculus ([17]), then [29] and [23] proves the correction of transformations on $\pi o \beta \lambda$ like the one described in [18]. Note that our calculus generalizes the possibility of returning a result before the end of the return by returning a future representing the result as soon as the request is received. Further comparison between ASP and $\pi o \beta \lambda$ will be discussed in conclusion.

From a static point of view, the topology of object dependence graph can be analyzed by a static analysis ([7], [28]) like in [3]. Moreover, [2] expressed 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 but this sub-case is simple to verify (typing).

3 Sequential calculus

3.1 Syntax

We start from an imperative sequential object calculus *à la* Abadi-Cardelli. Note that a few characteristics have been changed between ς_{imp}-calculus and ASP sequential calculus.

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

The abstract syntax of the ASP calculus is the following (l_i are field names, m_j are method names, ς is a binder for method parameters and a location ι is an entry in the store defined below, locations should only appear during the reduction):

$$
\begin{array}{rll}
a, b \in L ::= & x & \text{variable,} \\
& | \, [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j] & \text{object definition,} \\
& | \, a.l_i & \text{field access,} \\
& | \, a.l_i := b & \text{field update,} \\
& | \, a.m_j(b) & \text{method call,} \\
& | \, clone(a) & \text{superficial copy,} \\
& | \, \iota & \text{location (not in source terms).}
\end{array}
$$

As an example, a point object could be defined in the following way:

$Point \triangleq [x = 0,\ y = 0,\ color = [R = 0, G = 0, B = 0; print = \ldots];$
$getX = \varsigma(s,p)s.x,\ setX = \varsigma(s,p)s.x := p, getColor = \varsigma(s,p)s.color, \ldots]$

Note that *let* $x = a$ *in* b and sequence $a; b$ can be easily expressed in our calculus and will be used in the following.

3.2 Semantic structures

Let $locs(a)$ be the set of locations occurring in a and $fv(a)$ the set of variables occurring free in a. The *source terms* (initial expressions) are *closed* terms ($fv(a) = \emptyset$) without any location ($locs(a) = \emptyset$), such terms are called *static terms* ([13]). Locations appear when objects are put in the store.

3.2.1 Substitution

The substitution of b by c in a is written: $a\{\!\{b \leftarrow c\}\!\}$. Substitutions are denoted by $\theta ::= \{\!\{b \leftarrow c\}\!\}$. In method calls (INVOKE), substitution is applied in a classical way on bounded variables : formal parameters are replaced by the locations of the arguments.

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

3.2.2 Store

Reduced objects are objects with all fields reduced (to a location): $o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]$. 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 $dom(\sigma) \cap dom(\sigma') = \emptyset$.

Moreover, 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'):

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

3.2.3 Configuration

Let a *configuration* (a, σ) be a pair (expression, store). We denote by $\vdash (a, \sigma)$ OK a well formed configuration (no free variable and σ defines every useful location):

Definition 1 (Well formed sequential configuration)

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

3.3 Reduction

We define a small step substitution-based operational semantics for our sequential calculus (Table 1). It gives reduction rules for object creation (STOREALLOC), field access (FIELD), method invocation (INVOKE), field update (UPDATE) and shallow clone (CLONE). This semantics is very close to the one defined in [11]. Table 1 applies one rule on the point of reduction represented by the unique occurrence of $\bullet$ in the following *reduction contexts*:

$$\begin{aligned} \mathcal{R} ::= & \bullet \mid [l_0 = \iota_0; .. l_i = \mathcal{R}; l_{i+1} = b_{i+1}; ..; m_j = \varsigma(x_j, y_j) a_j] \\ & \mid \mathcal{R}.m_i \mid \mathcal{R}.m_j(b) \mid \iota.m_j(\mathcal{R}) \mid \mathcal{R}.l_i := b \mid \iota.l := \mathcal{R} \mid clone(\mathcal{R}) \end{aligned}$$

We denote by $\mathcal{R}[a]$ the substitution inside a reduction context: $\mathcal{R}[a] = \mathcal{R}\{\!\{\bullet \leftarrow a\}\!\}$.

To evaluate a source term a, we create an *initial configuration* $(a, \emptyset)$ containing this term and an empty store. Then, this configuration can be evaluated.

It is easy to show that **reduction preserves well-formedness**.

Moreover, a sequential reduction is deterministic up to the choice of freshly allocated locations:

Property 1 (Determinism) $c \rightarrow_S d \wedge c \rightarrow_S d' \Rightarrow d \equiv d'$

4 Parallel calculus

We introduce here a parallel calculus which is based on *activities*. Activities execute instructions concurrently, and interact only through asynchronous method calls. Synchronization is due to wait-by-necessity on the result of an asynchronous method call. Each activity contains a unique active object.

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

$$\frac{\sigma(\iota) = [..; l_i = \iota_i; ....]}{(\mathcal{R}[\iota.l_i],\sigma) \rightarrow_S (\mathcal{R}[\iota_i],\sigma)} \text{ (FIELD)}$$

$$\frac{\sigma(\iota) = [..; m_j = \varsigma(x_j, y_j)a_j; ....]}{(\mathcal{R}[\iota.m_j(\iota')],\sigma) \rightarrow_S (\mathcal{R}[a_j\{\!\{x_j \leftarrow \iota, y_j \leftarrow \iota'\}\!\}],\sigma)} \text{ (INVOKE)}$$

$$\frac{\sigma(\iota) = [..; l_i = \iota_i; ....]}{(\mathcal{R}[\iota.l_i := \iota'],\sigma) \rightarrow_S (\mathcal{R}[\iota],\{\iota \mapsto [\ldots; l_i = \iota'; \ldots]\} + \sigma)} \text{ (UPDATE)}$$

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

Table 1: sequential reduction

4.1 New syntax

We extend the sequential calculus by adding the possibility to create an active object and to serve a request:

$a, b \in L ::= ...$

$\mid Active(a, m_j)$	activates object: deep copy + activity creation m_j is the activity method or $\emptyset$ for FIFO service
$\mid Serve(M)$	Specifies request to serve,
$\mid a \Uparrow f, b$	a with continuation b (not in source terms)

Where M is a list of method labels used to specify which request has to be served.

$$M = m_1, \ldots, m_n$$

4.2 Principles

An *activity* is composed of a set of objects put in a store. Among them one is *active* and every *request* (method call) sent to the activity is actually sent to this object. An activity also contains the pending requests (requests that have been received and should be served later) and the responses to the finished requests (values of the results).

Passive (non active) objects are only referenced by objects belonging to the same activity but any object can reference active objects. The activation of an object ($Active(a, m)$) creates a new activity whom active object is a copy of a.

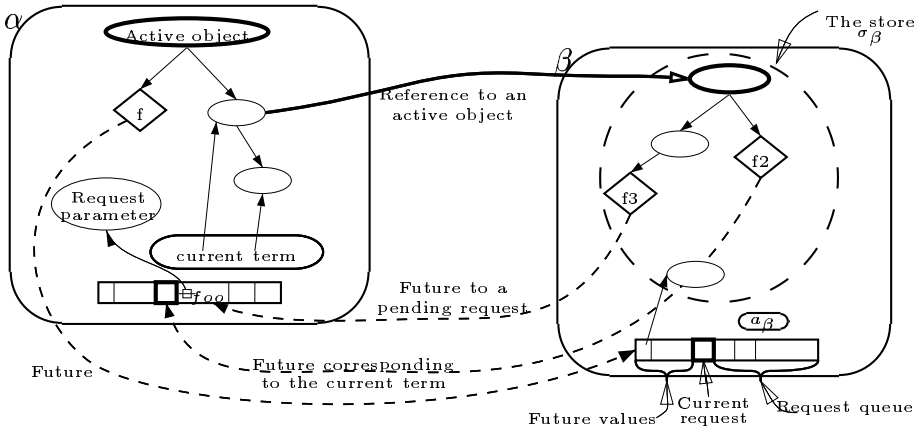


Figure 1: Example of a parallel configuration

For example, with the point object defined in 3.1, $Point.getColor()$ will perform a classical method call with synchronous semantics. In the term
$let\ p = Active(Point, \emptyset)\ in\ let\ col = p.getColor()\ in\ p.setX(2); col.print()$
every method call will be asynchronous. $p.setX(2)$ executes the method in the activity of p and continues the local evaluation in parallel. Execution will be blocked when we try to perform a strict operation on the result of an asynchronous method before the end of its execution. Such blocking states are called *wait-by-necessity*.

4.3 Informal semantics

Figure 1 gives a representation of a configuration consisting of two activities. 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 *request queue* which stores the pending method calls and a *future list* which stores the result of finished requests. A *future* represents the result of a method call to an active object that has not yet been returned.

Figure 1 contains three references to futures (one calculated, one current and one pending). The active objects are bold ellipses; futures references are diamonds; futures values, current future and request queue are merged in the bottom rectangles : calculated futures values are on the left, current future is represented by a bold rectangle and pending requests are on the right. Continuation will not appear in our representation.

The *Active* operator ($Active(a, m_j)$) creates a new activity α with the object a at his root. The object a is copied as well as all its dependencies[1] (deep copy) in a new activity. All subsequent calls to methods of the object activated are considered as remote request

[1] to prevent distant references to passive objects

sending. $AO(\alpha)$ acts as a proxy for the active object of activity α. 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 is called the *service method* as it should specify the order of requests that the activity should serve. If no service method is specified, a *FIFO* service will be performed. That is to say the requests will be served in the order they arrived in the activity. Note that in Figure 1, in case of a FIFO service, the current request (bold square) progresses from left to right in the queue. When the service method terminates, no more request is treated (end of activity).

Communications between activities are due to method calls on active objects and returns of corresponding results. A method call on an active object ($Active(o).foo()$) consists in atomically adding an entry to the *request queue* of callee, and associating a *future* to the response. From a practical point of view, this atomicity is guaranteed by a *rendez-vous* mechanism (the request sender waits for an acknowledgment before continuing its execution). In Figure 1, futures f_2 and f_3 denote pointers to not yet computed requests while f is a future pointing to a value computed by a request sent to β. Futures are generalized references that can be manipulated classically while we do not perform strict operations on the object they represent. Arguments of requests and value of futures are deeply copied when they are transmitted between activities[1]. Active objects are transmitted with a reference semantics.

The primitive $Serve$ can appear at any point in source code. Its execution stops the activity until a request matching its arguments is found in the requests queue (a request on one of the method specified as parameter of the $Serve$ primitive). For semantics specification reasons, we introduced the operator $\Uparrow$ which allows us to save the continuation 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 top level activity (no $Serve$ while a request is being served).

When the execution of a request is finished, the corresponding future is associated with the calculated value (*future value*). Then, the execution continues by restoring the continuation of the term that has served the finished request (right part of $\Uparrow$). The *future list* associates futures with their values within the activities that computed them. A future value is called *partial* if its dependencies contain futures references.

A wait-by-necessity occurs when we try to perform a strict operation (e.g. a method call) on a future. This wait-by-necessity can only be removed by *updating* the future i.e. replacing the reference to the future by a copy[1] of the future value (partial or complete).

Note that a field access on an active object is forbidden (it would nearly always be non deterministic) and an activity trying to access a field of an active object is irreversibly stuck (like an access to a non-existing field).

4.4 Strategies

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 strategies. Specifying a strategy would restrict the possible reductions but could simplify and accelerate the reduction. 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 and could, in case of partial results, create a great number of references to the future. At the opposite, a strategy could 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 deadlock configurations. These two strategies have been implemented in ProActive.

5 Examples

5.1 Binary tree

Figure 2 shows an example of a simple parallel binary tree with two methods: *add* and *search*. Each node can be turned into an active object. Lambda expressions, integers and comparisons (Church integers for example), booleans and conditional expressions and methods with many parameters can be easily expressed in our calculus. The definition of classes (*new* method ...) has already been proposed by Abadi and Cardelli in the ς_{imp^-} calculus([1]).

add stores a new key at the right place and creates two empty nodes. Note that, in the concurrent case, nodes are activated as soon as they are created.

search searches a key in the tree and returns the value associated with it or an empty object if the key is not found.

new is the method invoked to create a new node.

We parameterize our example by a factory able to create a sequential (sequential binary tree) or an active (parallel binary tree) node.

$$
\begin{array}{l}
BT \triangleq [new = \varsigma(c)[empty = true, left = [], right = [], key = [], value = [], \\
\qquad search = \varsigma(s,k)(c.search\ s\ k), add = \varsigma(s,k,v)(c.add\ s\ k\ v)], \\
\quad search = \varsigma(c)\lambda s\,k.if\ (s.empty)\ then\ [] \\
\qquad\qquad else\ if\ (s.key == k)\ then\ s.value \\
\qquad\qquad else\ if\ (s.key > k)\ then\ s.left.search(k) \\
\qquad\qquad\quad else\ s.right.search(k), \\
\quad add = \varsigma(c)\lambda s\,k\,v.if\ (s.empty)\ then(s.right := Factory(s); \\
\qquad\qquad\qquad\qquad\qquad\qquad s.left := Factory(s); s.value := v; \\
\qquad\qquad\qquad\qquad\qquad\qquad s.key := k; s.empty := false;\ s) \\
\qquad\qquad else\ if\ (s.key > k)\ then\ s.left.add(k,v) \\
\qquad\qquad else\ if\ (s.key < k)\ then\ s.right.add(k,v) \\
\qquad\qquad else\ s.value := v;\ s \qquad]
\end{array}
$$

where: $Factory(s) \triangleq s.new$ in the sequential case and
$Factory(s) \triangleq Active(s.new)$ for the concurrent binary tree.

Figure 2: Example: a binary tree

In the case of the parallel factory, the following term creates a binary tree, puts in parallel four values in it and searches two in parallel. Then it searches another value and modifies the field b. It always reduces to: $[a = 6, b = 8]$.

$$let\ tree = (BT.new).add(3,4).add(2,3).add(5,6).add(7,8) in$$
$$[a = tree.search(5), b = tree.search(3)].b := tree.search(7)$$

Note that, as soon as a request is delegated to another node, a new one can be handled. Moreover, when the root of the tree is the only node reachable by only one activity, the result of concurrent calls is deterministic. This important determinism property will be detailed in section 7.

5.2 Distributed sieve of Eratosthenes

Let us translate the distributed sieve of Eratosthenes described in [20] in ASP. In [20], the sieve was performed by several processes linked by channels, a process for each prime number, we tried to apply the same methodology and create one activity by prime number. We first considered that the communications comes from the process that performs a *get* on a channel to the one that performs a *put* on the same channel and replace such communication by a call to a request *get* (see Figure 3). *Repeat* performs an infinite loop and will be defined later on:

$$let\ Integer = Active([n = 1; get = \varsigma(s, \_).(s.n := s.n + 1; s.n)], \emptyset)\ in$$
$$let\ Sieve = [parent = [], prime = 0; init = \varsigma(s, par).s.parent := par,$$
$$get = \varsigma(s, \_).let\ n = parent.get()\ in$$
$$if(n \bmod s.prime \neq 0)\ n\ else\ s.get()]\ in$$
$$let\ Sift = [source = Integer;$$
$$act = \varsigma(s, \_).Repeat(let\ n = source.get()\ in$$
$$print(n); Sieve.prime := n;$$
$$s.source := Active(clone(Sieve.init(s.source))))]\ in$$
$$Active(Sift, act)$$

The *Integer* object generates all integers. There is one *Sieve* object for each prime number. It returns the next integer given by its parent that is not divisible by the prime number n. The $Sift$ object represents the main object (we noted $print(n)$ the output of integer n). When a new prime is found, a new *Sieve* in inserted between the $Sift$ and the former last *Sieve*.

Another formulation In the preceding example, every object always replies to a *get* request. Thus, the program will be evaluated sequentially and the pipelining that could be performed on the example of Kahn and MacQueen can not occur here. The following implementation of the sieve allows such pipelining (see Figure 4).

$$let\ Sieve = [N = 0, prime = 0; next = []; put = \varsigma(s, n).s.N := n,$$
$$act = \varsigma(s, \_).Serve(put); Display.put(s.N);$$
$$s.prime := s.N; s.next := Active(s, act);$$
$$Repeat(Serve(put); if\ (s.N \bmod s.prime \neq 0)\ s.next.put(s.N))]\ in$$

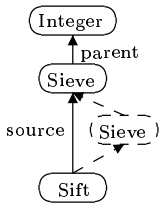


Figure 3: Sieve of Eratosthenes (pull)

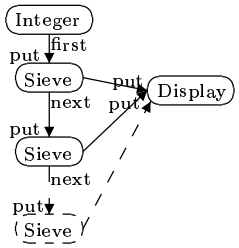


Figure 4: Sieve of Eratosthenes (push)

$$\begin{array}{l} let\ Integer = [n = 1; first = Active(Sieve, act); \\ \qquad act = \varsigma(s, \_).Repeat(s.n := s.n + 1; s.first.put(s.n))]\ in \\ Active(Integer, act) \end{array}$$

where *Display* is an object collecting and printing the prime numbers.

The problem with this example is that every *Sieve* object keeps a reference on the *Display* object. Some conflicts could occur between sending of results to the display. Here, we can consider that determinism is ensured by the fact that as soon as a new *Sieve* is created, the preceding one promise not to use its reference to *Display* any more.

5.3 A bank account server

Let us imagine a bank application with the following characteristics :

- A client activity sends a request to a unique Central Service to get a statement of his account.
- The Central Service dispatches the request to the appropriate activity corresponding to the regional database of the client.
- Further, based on the client device type (browser, PDA, ...) the Central Service requests the formatting of the data (the statement) to the appropriate presentation server. Some advertising could be added.
- The final result has to be sent to the client.

We define here the Central Service object (methods with more than one parameter can be easily defined and will be used in this example) :

$$\begin{array}{l} let\ CentralService\ =\ [\ldots; \\ \quad regionalDatabase = \varsigma(s, ID).\ldots, \\ \quad presentationServer = \varsigma(s, device).\ \ldots, \\ \quad act = \varsigma(s, \_).Repeat(Serve(getStatement)), \\ \quad getStatement = \varsigma(self, ID, device). \\ \qquad let\ state = (self.regionalDatabase(ID)) \\ \qquad\qquad\qquad\qquad .getStatement(ID) \\ \qquad in\quad (self.presentationServer(device)) \\ \qquad\qquad\qquad\qquad .getPresentation(state)\ ] \end{array}$$

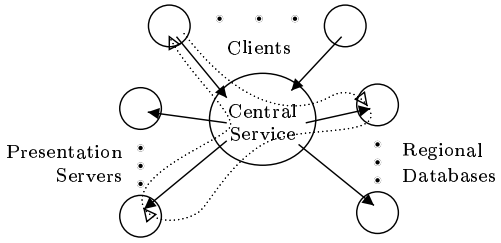


Figure 5: Example : a bank application

The client calls the *getStatement* request on the Central Service, which receives the account number and device type of the client. The Central Service asks for the statement to the appropriate Regional Database and send it to the right Presentation Server before returning the result to the client as a reply to its initial request.

The Regional Databases have to be able to serve the request $getStatement(accountNumber)$. The Presentation
Servers will serve $getPresentation(statement)$ requests. The client will obtain the statement of his account by calling
$CentralServer.getAccount(ID, device)$ where ID is its account number and *device* the kind of device it is using.

Note that at the end of the *getStatement* evaluation, the result to be sent to the client might contain several futures coming from Regional Database and Presentation Server.

6 Parallel semantics

6.1 Structure of parallel activities

We assume that we have three distinct name spaces: activities($\alpha, \beta \in Act$), locations(ι) and futures(f). Note that locations and future identifiers f are local to an activity and a future will be characterized by its identifier f, the source activity α and the destination activity β of the corresponding request ($f_i^{\alpha\to\beta}$). A parallel configuration is a set of activities $P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[\ldots] \| \ldots$ characterized by:

- *current term* a_α to be reduced (the activity). a_α contains the terms corresponding to the treated requests separated by $\Uparrow$. The left part is the current term, the right one is the continuation: future and term corresponding to a request that has been stopped before the end of its execution (because of a *Serve* primitive);
- *active object location* ι_α is the location of the active object of activity α, thus $\sigma_\alpha(\iota_\alpha)$ is the active object of α;
- *store* σ_α contains all objects of the activity α;
- *future values*, a list associating, for each served request, a location ι to its future f: $F_\alpha = \{f \mapsto \iota\}$;

- *request queue* $R_\alpha = \{[m_j; \iota; f_i^{\alpha\to\beta}]\}$, a list of pending requests;
- *current future* f_α, the future associated with the request currently served.

Empty parts of activities will be denoted by $\emptyset$. $\emptyset$ designates an empty list (futures or requests) or an empty current term (no more activity) and an empty current future (when no request is currently treated). Note that locations are local to an activity.

A request can be seen as the "reification" of a method call ([30]). Each request $R ::= [m_j; \iota; f_i^{\alpha\to\beta}]$ consists of

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

We will denote by $R :: r$ for adding a request r at the end of the queue (R) and $r :: R$ for taking the first request (r) at the beginning of the queue. Similarly, $F :: \{f_i \mapsto \iota\}$ adds a new future association to the future values.

In the store, we have:

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

$fut(f_i^{\alpha\to\beta})$ references the future $f_i^{\alpha\to\beta}$ corresponding to a request from activity α to activity β. $AO(\alpha)$ 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. As they are valid across activities, references to active objects and futures are called *generalized references*. 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 $f_i^{\alpha\to\beta}$). A future $f_i^{\alpha\to\beta}$ is characterized by f_i a fresh future identifier (which is local to an activity), α the activity that sends the request and β the activity that receives and handles the request.

6.2 Parallel reduction

We define the infinite loop *Repeat* and the FIFO service that will be used when no service method is specified and serves the requests in the order they arrived:

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

where $\mathcal{M}$ is the set of all method labels. Note that in the following $\mathcal{M}$ could be any set containing all the method labels of the concerned (active) object.

Object activation and terms containing a continuation are added to reduction contexts as follows:

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

6.2.1 Deep copy

The operator $copy(\iota, \sigma)$ creates a store containing the deep copy of $\sigma(\iota)$. The deep copy is the smallest store satisfying the rules of Table 2. The deep copy stops when a generalized reference is encountered. In that case, the new store contains the generalized reference. In Table 2, the first two rules specify which locations should be present in the created store, and the last one means that the codomain is similar in the copied and the original store.

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

Table 2: Deep copy

The following operator adds the part of σ starting at location ι at the location ι' of σ' avoiding collision of locations:

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

6.2.2 Reduction rules

Table 3 describes the reduction rules corresponding to the small step semantics of the parallel calculus. Grayed values are unchanged and unused by reduction rules. We give here a short description of these rules:

LOCAL inside each activity, a local reduction can occur following the rules of Table 1. Note that sequential rules FIELD, INVOKE, UPDATE, CLONE[2] are stuck (wait-by-necessity) when the target location is a generalized reference. Only REQUEST allows to invoke an active object method, and REPLY may transform a future reference into a reachable object (ending a wait-by-necessity).

NEWACT activates an object. A new activity γ is created containing the deep copy of the object $\sigma_\alpha(\iota)$ and empty request queue and future values. A generalized reference to the created activity $AO(\gamma)$ is stored in the source activity α. Other references to ι in α are unchanged (still pointing to a passive object).

m_j specifies the activity (first method executed) and should perform *Serve* instructions. If no method m_j is specified, a FIFO service is performed.

Note that in $Active(Active(\iota, m_j), \emptyset)$, the first target activity is reduced to $\{\iota' \mapsto AO(\gamma)\}$ and acts as a forwarder.

REQUEST sends a new request from activity α to activity β (Figure 6). A new future $f_i^{\alpha\to\beta}$ is created to represent the result of the request, a reference to this future is stored in α. 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 $([m_j; \iota''; f_i^{\alpha\to\beta}])$ is added to the request queue R_β.

[2] cloning future is considered as a strict operation for determinism.

$$\frac{(a,\sigma) \rightarrow_S (a',\sigma')}{\alpha[a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f] \parallel P} \text{ (LOCAL)}$$

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

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

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

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

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

Table 3: Parallel reduction (used or modified values are non-gray)

SERVE serves a new request (Figure 7). The current reduction is stopped and stored as a continuation (future f, expression $\mathcal{R}[[]]$) and the oldest request concerning one of the labels specified in M is treated. The activity is stuck until a matching request is found in the pending request.

ENDSERVICE applies when the current request is finished (current term is a location). It associates, the location of the result to the future f_α. The response is (deep) copied to prevent post-service modification of the value and the current term and current future are obtained from the continuation (Figure 8).

REPLY updates a (total or partial) future value (Figure 9). It replaces a reference to a future by its value. 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. Moreover some operations (e.g. INVOKE) need the real object value of

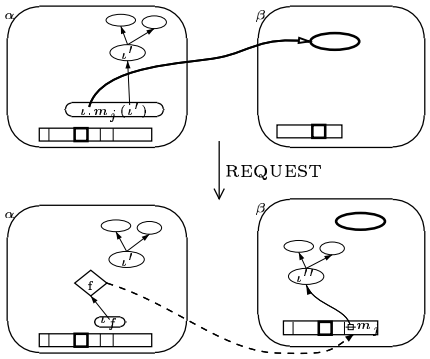


Figure 6: REQUEST

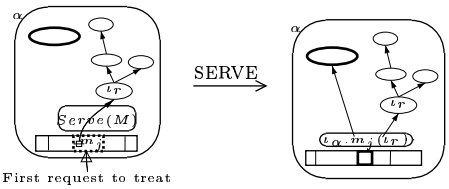


Figure 7: SERVE

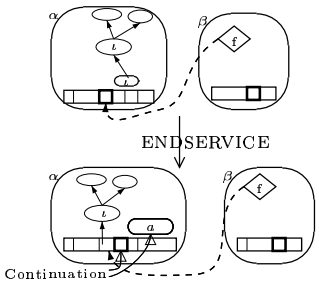


Figure 8: ENDSERVICE

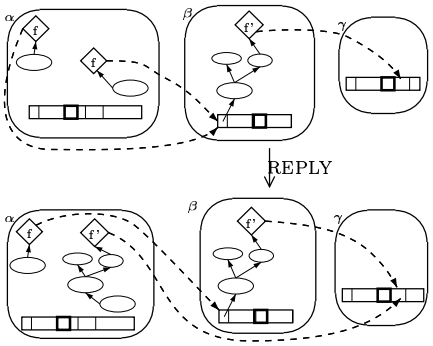


Figure 9: REPLY

some of their operands. Such operations may lead to wait-by-necessity, which can only be resolved by the update of the future value. Note that a future $f_i^{\gamma \to \beta}$ can be updated in an activity different from the origin of the request ($\alpha \neq \gamma$) because of the capability to transmit futures (e.g. as method call parameters). After an update, a future cannot be removed from the future values because the future might have proliferated in other activities; reference counting could be used to perform garbage collection of futures ([22], [8]).

An *initial configuration* consists of a single activity, called *main activity*, containing only a current term $\mu[a; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset]$. This activity will never receive a request, it can only communicate by sending requests or receiving replies.

6.3 Well-formedness

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

$$ActiveRefs(\alpha) = \{\beta | \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = AO(\beta)\},$$

$$FutureRefs(\alpha) = \{f_i^{\beta\to\gamma} | \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = fut(f_i^{\beta\to\gamma})\}$$

Definition 2 (Futures list) *Let $FL(\gamma)$ be the list of futures that have been calculated, the current futures and futures corresponding to pending requests. It is depicted by the rectangles of Figure 1.*

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

A parallel configuration is *well formed* if all local configurations are well formed (in the sense of definition 1), every referenced activity exists, and every future reference points to a future that has been or will be calculated (in the absence of dead or live locks):

Definition 3 (Well-formedness)

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

It is easy to show that **parallel reduction preserves well-formedness**:

Property 2 (Well-formed parallel reduction)

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

7 Properties and confluence

In this Section, we start with a property on topology inside an activity, then we introduce a notion of compatibility between configuration and an equivalence relation. We conclude with different confluence and determinism properties. In the following, α_P denotes the activity α of configuration P.

7.1 Futures and parameters isolation

The following property states that the value of a future and request parameters are situated in isolated parts of the store.

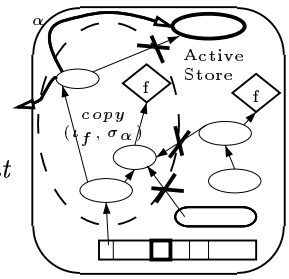
This invariant is proved by checking it on each reduction rule. The part of σ_α that does not belong to the preceding partition may be freely garbage collected. The only modifications allowed on the futures and parameters partitions is the update of a calculated future value.

Property 3 (Store partitioning) *Let*

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

At any stage of computation, each activity has the following invariant ($\bigoplus$ is the disjoint union):

$$\sigma_\alpha \supseteq ActiveStore(\alpha) \bigoplus_{\{f \mapsto \iota_f\} \in F_\alpha} copy(\iota_f, \sigma_\alpha) \bigoplus_{[m_j; \iota_r; f] \in R_\alpha} copy(\iota_r, \sigma_\alpha)$$



7.2 Compatibility

In this section, we introduce notations that will be useful for establishing confluence of terms in 7.4. The compatibility property is due to the fact that the order of evaluation is entirely defined by the order of request sending. More precisely, we prove that the order of activities sending requests to a given one is sufficient. Note that this means that futures updates and imperative aspects of our calculus do not act upon confluence.

First, let us precise the choice of fresh futures $f_i^{\alpha \to \beta}$. We consider now that the future identifier f_i is the name of the invoked method indexed by the number of requests that have already been received by β. Thus if the 4^{th} request received by β comes from γ and concerns method foo, its future identifier will be $foo_4^{\gamma \to \beta}$. In the following f will denote a method label. We denote by $(RSL_\alpha)_n$ the n^{th} element of the Request Sender List of activity α.

Definition 4 (Request Sender List) *The request sender list is the list of request senders in the order the requests have been received and indexed by the invoked method ($FL(\alpha)$ has been defined in 6.3):*

$$(RSL_\alpha)_n = \beta^f \text{ if } f_n^{\beta \to \alpha} \in FL(\alpha)$$

Note that this list is obtained from futures associated to served requests, current requests and pending requests. Moreover for each i between 0 and n (if $n+1$ requests have been received by α) $(RSL_\alpha)_i$is well defined (corresponds to a unique future).

For a FIFO service then the order of requests can not be changed when they are served. 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). Indeed if $f_n^{\beta \to \alpha}$ is the current future then $f_0^{\delta \to \alpha} \dots f_{n-1}^{\gamma \to \alpha}$ correspond to calculated futures and $f_{n+1}^{\delta \to \alpha} \dots$ correspond to pending requests.

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

$$\alpha_1{}^{f_1} \dots \alpha_n{}^{f_n} \trianglelefteq \alpha'_1{}^{f'_1} \dots \alpha'_m{}^{f'_m} \Leftrightarrow \begin{cases} n \leq m \\ \forall i \in [1..n], \alpha_i = \alpha'_i \end{cases}$$

Definition 6 (RSL compatibility: $RSL_\alpha \bowtie RSL_\beta$) *Two RSLs are compatible if they have a least upper bound or equivalently if one is prefix of the other*

$$RSL_\alpha \bowtie RSL_\beta \Leftrightarrow RSL_\alpha \sqcup RSL_\beta \text{ exists } \Leftrightarrow (RSL_\alpha \trianglelefteq RSL_\beta \vee RSL_\beta \trianglelefteq RSL_\alpha)$$

Let $RSL_\alpha\big|_M$ represent the restriction of the RSL list on the set of labels M ($(\alpha^{f_0} :: \beta^{f_1} :: \gamma^{f_2})\big|_{f_0,f_2} = \alpha^{f_0} :: \gamma^{f_2}$).

Let $\mathcal{M}_{\alpha_P}$ be a static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P[3].

Two configurations are said to be compatible if all the restriction of their RSL that can be served are compatible (have a least upper bound):

Definition 7 (configuration compatibility: $P \bowtie Q$)

$$P \bowtie Q \Leftrightarrow \forall \alpha \in P \cap Q, \forall M \in \mathcal{M}_{\alpha_P} \cup \mathcal{M}_{\alpha_Q}, RSL_{\alpha_P}\big|_M \sqcup RSL_{\alpha_Q}\big|_M exists$$

If $Serve$ operations are only performed by top level activity (no $Serve$ while a request is being served, all the "usable" restriction of the RSL are unchanged during service and it is sufficient to consider the the future values in the order they have been calculated.

7.3 Equivalence modulo replies

P and Q are said to be *equivalent modulo future replies* ($P \equiv_F Q$) if they are identical modulo renaming of activity, locations and futures already computed, and the update of some futures. The last point is equivalent to considering references to futures already calculated as equivalent to local reference to the part of store which is the (deep copy of the) future value. Or, a future is equivalent to the future after update or more formally, a future is equivalent to a part of store if this part of store is equivalent to the store which is the (deep copy of the) future value (provided the updated part does not overlap with the remaining of the store).[4]

This equivalence is also useful in case of cycle of futures because we can obtain configurations that will never converge but behave identically (Figure 10).

Note that this equivalence is decidable and that all definitions given (informally) here are equivalent (for example $P \xrightarrow{\text{REPLY}} P' \Rightarrow P \equiv_F P'$).

Let $T \in \{\text{LOCAL}, \text{NEWACT}, \text{REQUEST}, \text{SERVE}, \text{ENDSERVICE}, \text{REPLY}\}$ be any parallel reduction.

Property 4 (Equivalence and reduction)

$$P \xrightarrow{T} Q \wedge P \equiv_F P' \Rightarrow \begin{cases} \textit{if } T = \text{REPLY } \textit{then } Q \equiv_F P' \\ \textit{else } \exists Q',\ P' \xrightarrow{\text{REPLY}^*} \xrightarrow{T} Q' \wedge Q' \equiv_F Q \end{cases}$$

This important property states that if one can apply a reduction rule on a configuration then, after several REPLY, a reduction using the same rule can be applied on any equivalent configuration. The proof consists in verifying that the application of a given rule on equivalent terms preserves equivalence (cf. 8.1 for details).

[3] it can be specfied by a type system. Thus we have $P \xrightarrow{*} Q \Rightarrow \mathcal{M}_{\alpha_P} = \mathcal{M}_{\alpha_Q}$

[4] Note that the equivalence of request must be adequate to the RSL compatibility. Indeed, the equivalence on pending request must allow them to be reordered provided the compatibility of RSLs is maintained : requests that can not interfere (because they can not be served by the same $Serve$ primitive) can be safely exchanged. Modulo this permutation, equivalent configurations must be composed of equivalent pending requests in the same order

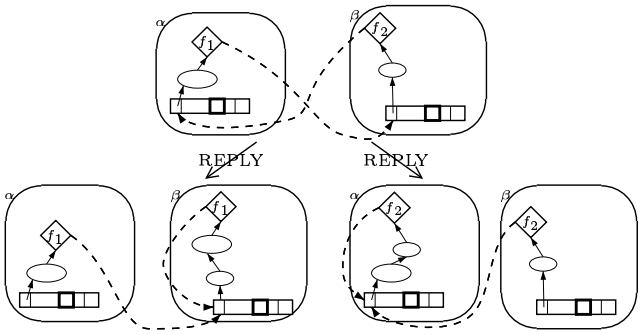


Figure 10: Updates in a cycle of futures

This property suggests to define the reduction $\overset{T}{\Longrightarrow}$ which is $\longrightarrow$ preceded by some applications of the REPLY rule.

Definition 8 (Generalized Parallel Reduction)

$$\overset{T}{\Longrightarrow} \quad = \quad \overset{\text{REPLY}^*}{\longrightarrow} \overset{T}{\longrightarrow} \textit{ if } T \neq \text{REPLY } \textit{and } \overset{\text{REPLY}^*}{\longrightarrow} \textit{ if } T = \text{REPLY}$$

Then property 4 can be rewritten in the following way:

Property 5 (Equivalence and generalized parallel reduction)

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

In the following, we present sufficient conditions for confluence of ASP configurations. We suppose now that equivalent activities, futures and locations have the same identifier in equivalent configurations (i.e. if $P \equiv_F Q$, the activity corresponding to α_P in the configuration Q is α_Q). In practice this could be solved by applying the appropriate renaming on activities before manipulating them.

7.4 Confluence

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

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

$$P_1 \curlyvee P_2 \Leftrightarrow \exists R_1, R_2, \begin{cases} P_1 \overset{*}{\longrightarrow} R_1 \\ P_2 \overset{*}{\longrightarrow} R_2 \\ R_1 \equiv_F R_2 \end{cases}$$

The principles of confluence property can be summarized by the fact that the only source of non-determinism is the application of two REQUEST rules on the same destination activity; the order of updates of futures does not have any influence on the reduction of a term. The

only constrain to the moment where REPLY must occur is a wait-by-necessity on a future. Note that even if this property is natural it allows a lot of asynchronism and proves that the mechanism of futures is adequate and powerful. Moreover, in the following we will generalize this property as for non-FIFO service, the order of requests does not matter if they can not be involved in the same *Serve* primitive

The next property states that if, from a given term, we obtain two compatible configurations, then these configurations are confluent. Thus there are two sequences that reduce the two configurations to a common one (modulo equivalence)

Property 6 (Confluence)

$$\begin{cases} P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \\ Q_1 \bowtie Q_2 \end{cases} \Longrightarrow Q_1 \curlyvee Q_2$$

The proof of this property is rather long but the key idea is that if two configurations are compatible then there is a way to perform missing sending of requests in the right order (see 8.2 for more details). Thus the configurations can be reduced to a common one (modulo future replies equivalence). Note that, if we consider two 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. In other words $Q_1 \bowtie Q_2$ is a necessary condition for Q_1 and Q_2 to be confluent. Of course, another equivalence relation could be found for which compatibility between terms would not be necessary for confluence.

Note that, if we replaced the primitive $Serve(M)$ by a primitive allowing to serve a request coming from a given activity $Serve(\alpha)$ then our calculus would be deterministic. This aspect is not developed in this paper but could be very interesting if the order of activities sending a request to a given one was known. Note that such a calculus would be more similar to process networks where *get* are performed on a given channel and a channel only have one source process.

Note also that, even if the moment when future values are updated is not specified in the semantics, this does not act upon determinacy. This characteristic prooves the interest of our calculus and its properties.

The next section identifies a set of terms that behave deterministically.

7.5 Deterministic Objects Networks

Serve are blocking primitives and if at any time, two activities cannot send concurrently a request on a given method (or set of method labels M that appears in a $Serve(M)$) of the same activity then there is not any conflict between .

We introduce here a definition of Deterministic Objects Networks (DON).

Definition 10 (DON) *A term P is in a Deterministic Objects Networks form ($DON(P)$) if : for all Q such that $P \xrightarrow{*} Q$, Q verifies:*

$$\forall \alpha \in Q,\ \forall M \in \mathcal{M}_{\alpha_P}, \exists^1 \beta \in Q, \exists m \in M,\ a_\beta = \mathcal{R}[\iota.m(\ldots)] \wedge \iota \mapsto AO(\alpha) \in \sigma_\beta$$

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

A term is a deterministic objects network if 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.

The preceding DON definition is dynamic but could easily be approximated by statically determining the set of active objects that can send a request on method m of activity α. Note that this means that we first have to statically decide whether an object is active or not (by static analysis). Note also that, we need a static approximation of reachable configurations Q.

For example the Sieve of Eratosthenes examples verify $DON(P)$. But if the first one is easy to verify statically (this can be seen on the Figure 3) the second example seems much more difficult (in Figure 3 many objects have a reference to *Display*). This comes from the fact that, all sieve objects keep a reference to the Display object. Dynamically, at each time a single one will be able to send a request on *Display* object.

From the definition of DON we can conclude easily that DON terms always reduces to compatible configurations :

Property 7 (DON and compatibility)

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

Indeed, RSL compatibility comes from the fact that $DON(P)$ implies that two activities can not be able to send requests that can interfere to the same third activity (unicity of β for all terms obtained from α). Moreover, we can easily prove that reduction of DON terms always leads to the same RSLs, for all orders of request sending, we always serve the requests in the same order.

Thus the set of DON terms is a deterministic sub-calculus of ASP:

Property 8 (DON determinism)

$$\begin{cases} DON(P) \\ P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \end{cases} \Longrightarrow\ Q_1 \curlyvee Q_2$$

As explained before, the examples of sieve of Eratosthenes are DON and thus their execution is deterministic.

We shown here that we can easily identify a sub-calculus (DON terms) of ASP that is deterministic. Some terms can be identified as DON statically. This is a generalization of Process Networks because we can wait on several requests at the same time.

7.6 Case of FIFO service

Let us now consider the case where each activity performs a FIFO service. The *request flow graph* is the graph where nodes are activities and there is an edge between two activities if one activity sends requests to another one ($\alpha \rightarrow_R \beta$ if α sends a request to β).

It is easy to prove that if the request flow graph is a tree then for each α, RSL_α contains occurrences of at most one activity. Then for all Q and R such that $P \xrightarrow{*} Q \wedge P \xrightarrow{*} R$, $Q \bowtie R$ so we can conclude:

Property 9 (Tree request flow graph)

If the request flow graph forms a set of trees then the reduction is deterministic.

This property proves the determinism of the binary tree of Figure 2. What is important here, is to see that the parts of reduction where request flow graph forms a tree are deterministic. In order to prove that a term is confluent, we only have study determinism on moments where request flow graph is not a tree. For example, consider a program that first, creates and communicates over a set of activities forming a tree, performs a global synchronization step and finally communicates over another tree. Such a program is confluent.

8 Proofs

8.1 Equivalence modulo futures

Let $Act(P)$ be the set of activities defined in P.

8.1.1 Renaming

We introduce a set Θ of renaming of activities and futures from configuration P to configuration Q:

$$\begin{aligned}\Theta &::= (\theta_{act}, \theta_{fut})\\ \theta_{act} &::= \{\!\{\alpha_1 \to \alpha'_1 \ldots\}\!\};\\ \theta_{fut} &::= \{\!\{f_i^{\beta\to\alpha} \to f_i'^{\beta'\to\alpha'}, \ldots\}\!\};\end{aligned}$$

where $\alpha \in P$ and $\alpha' \in Q$ and θ_{act} is a bijection from $Act(P)$ to $Act(Q)$, θ_{fut} is a bijection from F_{α_P} to $F_{\alpha'_Q}$ for any $\alpha \to \alpha' \in \theta_{act}$.

If the activity names are chosen in a deterministic manner, we have $\theta_{act} = Id$

We must ensure the following additional constraints :

$$f_i^{\beta\to\alpha} \to f_i'^{\beta'\to\alpha'} \in \theta_{fut} \Rightarrow \begin{cases} \alpha \to \alpha' \in \theta_{act} \\ \beta \to \beta' \in \theta_{act} \end{cases}$$

8.1.2 Reordering requests

The equivalence relation must be defined modulo the reordering of some requests. Indeed two requests can be exchanged if they concern different methods which can not interfere. That is to say if there is no service concerning both method labels.

Let φ_{α_i} be a permutation on the request queue of α'_i compatible with the RSL equivalence:

$$\forall M \in \mathcal{M}_{\alpha_P},\ R_{\alpha_P}\big|_M \equiv_F R_{\alpha_Q}\big|_M$$

Where $R_\alpha\big|_M$ is the restriction of the request queue of α to the list of requests for which method labels belong to M.

In other words φ_{α_i} only permutes requests on methods that cannot be selected concurrently by the same $Serve(M)$.

8.1.3 Future updates

The equivalence modulo futures consists in considering the reference to calculated futures like local reference to deep copy of the value of the future. In other words, future references can be followed as if they were local references, with a deep copy when two futures references concerns the same future.

The following example (Figure 11) illustrates a simple update of future value. Of course the two configuration are equivalent.

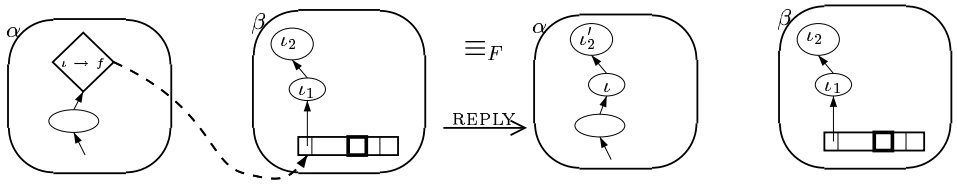


Figure 11: Simple example of future Equivalence

We say that two configurations are equivalent modulo future updates if they are the same modulo renaming of activity, locations and futures, and the update of some already calculated futures.

The condition to update some futures can be summarized simply by considering references to futures as identical to local reference to the part of store which is the (deep copy of the) future value. Or a future is equivalent to the future after update or more formally, a future is equivalent to a part of store if this part of store is equivalent to the store which is the (deep copy of the) future value (provided the updated part of the store does not overlap with the remaining of the store).

Following references and sub-terms Let us formalize the idea that "we follow the references". We want to follow future references as local ones. We introduce $\overset{\alpha}{\mapsto}_L$.

First, L represents the list of references or part of expression that must be followed. Table 4 describes some of these rules We denote by $\overset{\alpha}{\mapsto}_{L_1.L_2}$ the concatenation $\overset{\alpha}{\mapsto}_{L_1}\overset{\alpha}{\mapsto}_{L_2}$.

$\iota \overset{\alpha}{\mapsto}_{ref} \sigma(\iota)$	$[\ldots l_i = a \ldots] \overset{\alpha}{\mapsto}_{l_i} a$	$[\ldots m_j = \varsigma(s,x)a \ldots] \overset{\alpha}{\mapsto}_{m_j} a^a$	$a.l_i := b \overset{\alpha}{\mapsto}_{Update1(l_i)} a$	
$a.l_i := b \overset{\alpha}{\mapsto}_{Update2(l_i)} b$		$\ldots$		

[a] It is easy to add renaming of bounded variables here

Table 4: Paths definition

Let $\overset{\alpha*}{\mapsto}_{L_2}$ be the preceding relation where we can follow futures if necessary (we can have $n = 0$):

$$a \overset{\alpha*}{\mapsto}_{L_0\ldots L_n} b \Leftrightarrow a \overset{\alpha}{\mapsto}_{L_0} \iota_1 \wedge \sigma_\alpha(\iota_1) = fut(f_i^{\gamma\to\beta_1}) \wedge F_\beta(f_i^{\gamma\to\beta_1}) = \iota'_1 \wedge \iota'_1 \overset{\beta_1}{\mapsto}_{L_1} \iota_1 \wedge \ldots \wedge \iota'_n \overset{\beta_n}{\mapsto}_{L_n} b$$

This definition consist in, first, following a path inside an activity α, then, follow a future reference from α to β_1 and continuing the path in β_1 etc... note that when we follow a future reference, two local (ref) and a future reference are in fact considered as identical to a single local reference.

For example, in figure 11 the three arrows of the left configurations that are around the future reference (dashed arrow) are considered as equivalent with a single arrow on the right configuration.

Note that:

Lemma 1

$$a \overset{\alpha}{\mapsto}_L b \Rightarrow a \overset{\alpha*}{\mapsto}_L b$$

Lemma 2

$$a \overset{\alpha*}{\mapsto}_L b \wedge a \overset{\alpha*}{\mapsto}_L b' \Rightarrow b = b' \vee b = fut(f_i^{\gamma\rightarrow\beta}) \vee b' = fut(f_i^{\gamma\rightarrow\beta})$$

Here , the two particular cases ($b = fut(f_i^{\gamma\rightarrow\beta}) \vee b' = fut(f_i^{\gamma\rightarrow\beta})$) are due to the fact that when we arrive at a future reference we do not necessarily follow this reference.

Equivalence definition

Definition 11 (Equivalence $P \equiv_F Q$) *Let $R = \varphi(Q\Theta)$[5]. Then*

$$\begin{aligned} P \equiv_F Q \quad \Leftrightarrow \quad & \forall \alpha, \exists a, \alpha_P \overset{\alpha*}{\mapsto}_L a \Leftrightarrow \exists a', \alpha_R \overset{\alpha*}{\mapsto}_L a' \\ & \wedge\ \alpha_P \overset{\alpha}{\mapsto}_L a \wedge \alpha_P \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, l_0, a' \begin{cases} L = L_0.L_1 \wedge L' = L_0.L_1' \wedge \alpha_R \overset{\alpha*}{\mapsto}_{L_0} c \\ c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \\ & \wedge\ \alpha_R \overset{\alpha}{\mapsto}_L a \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, l_0, a' \begin{cases} L = L_0.L_1 \wedge L' = L_0.L_1' \wedge \alpha_P \overset{\alpha*}{\mapsto}_{L_0} c \\ c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \end{aligned}$$

The first condition expresses both the equivalence inside an activity and by following futures and the two last conditions[6] express the correctness of aliasing (alias must be the same in both configurations). Note that, in the two last conditions the existence of a' such that $\alpha_P \overset{\alpha*}{\mapsto}_L a'$ is ensured by the first condition.

Note that the above equivalence is not precise in the sense that if P and Q contain non equivalent garbage collectable terms then P and Q may be considered as equivalent by definition 11. But this will have no consequence on the subsequent reductions.

Property 10 (equivalence relation) $\equiv_F$ *is an equivalence relation*

We will need to have equivalence of sub-terms. For the first definition this equivalence can be defined straightforwardly (even if it involves a notion of renaming of activity which is global). In fact in both cases *sub-terms are equivalent if they are part of equivalent expressions.* Which will be expressed by the fact that $P \equiv_\alpha^\Theta Q \wedge a \equiv_\alpha^\Theta a$ or equivalently:

[5] remind this means renaming each activity, each future and generalized reference and permuting in each activity α_i by using φ_{α_i}

[6] named alias conditions in the following

Definition 12 (Equivalence of sub-terms)

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

This implies that:

Lemma 3 (sub-term equivalence)

$$\begin{aligned} a \equiv_F a' \quad \Rightarrow \quad & a \overset{\alpha*}{\mapsto}_{L'} b \Leftrightarrow \exists a',\, a' \overset{\alpha*}{\mapsto}_{L'} b' \\ & \wedge\ a \overset{\alpha}{\mapsto}_L b \wedge a \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \begin{cases} L = L_0.L_1 \wedge L' = L_0.L_1' \wedge a' \overset{\alpha*}{\mapsto}_{L_0} c \\ c \overset{\gamma}{\mapsto}_{L_1} b' \wedge c \overset{\gamma}{\mapsto}_{L_1'} b' \end{cases} \\ & \wedge\ a' \overset{\alpha}{\mapsto}_L b \wedge a' \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \begin{cases} L = L_0.L_1 \wedge L' = L_0.L_1' \wedge a \overset{\alpha*}{\mapsto}_{L_0} c \\ c \overset{\gamma}{\mapsto}_{L_1} b' \wedge c \overset{\gamma}{\mapsto}_{L_1'} b' \end{cases} \end{aligned}$$

In the following proofs, we will not detail in general the arguments related to renamings. That is to say, we will always suppose that, when $P \equiv Q$, P and Q use the same locations futures and activity names (or more precisely, renaming of futures activities and locations have already been applied). We will focus on parts of proof related to updates of futures. In other words, the alpha conversion part of proofs is considered as straightforward. For example, we will always consider that when we choose a fresh location in Q we can choose the location that we want (for example the same as in P) which is always possible modulo alpha conversion in P and Q.

8.1.4 properties of $\equiv_F$

Consider the case where we add a new entry in the store and reference it from the same places (L_i is any set of paths inside α). Adding equivalent sub-terms at the same place in two equivalent configurations produce equivalent configurations:

Lemma 4 ($\equiv_F$ and store append)

$$\begin{cases} P \equiv_F Q \\ a \equiv_F a' \\ P' = P \ \textit{except}\ \sigma_{\alpha_{P'}} = \sigma_{\alpha_P} :: \{\!\{\iota \to a\}\!\}\ \textit{and}\ \alpha_{P'} \overset{\alpha}{\mapsto}_{L_i} \iota \\ Q' = Q \ \textit{except}\ \sigma_{\alpha_{Q'}} = \sigma_{\alpha_Q} :: \{\!\{\iota' \to a'\}\!\}\ \textit{and}\ \alpha_{Q'} \overset{\alpha}{\mapsto}_{L_i} \iota' \end{cases} \Rightarrow P' \equiv_F Q' \wedge \iota \equiv_F \iota'$$

Note that ι and ι' are fresh locations.

Proof : For parts of activities inside a, $(R' = \varphi(Q'\theta_{act}\theta_{fut}))$ $\alpha_{P'} \overset{\alpha}{\mapsto}_{L_i} \iota \overset{\alpha*}{\mapsto}_{L'} b$, we have $\alpha_{R'} \overset{\alpha}{\mapsto}_{L_i} \iota'$ by hypothesis and $\iota' \overset{\alpha*}{\mapsto}_{L'} b'$ by definition of $a \equiv_F a'$ (lemma 3). This proves

$$\forall \alpha,\, \exists a,\, \alpha_{P'} \overset{\alpha*}{\mapsto}_L a \Rightarrow \exists a',\, \alpha_{R'} \overset{\alpha*}{\mapsto}_L a'$$

The opposite implication is similar (symmetric).

For alias conditions, note that for all L_i, $\alpha_{P'} \overset{\alpha}{\mapsto}_{L_i} \iota \wedge \alpha_{P'} \overset{\alpha}{\mapsto}_{L_j} \iota$ but by hypothesis, $\alpha_{Q'} \overset{\alpha}{\mapsto}_{L_i} \iota' \wedge \alpha_{Q'} \overset{\alpha}{\mapsto}_{L_i} \iota'$. The bijectivity inside a and a' is also ensured by lemma 3. □

In the same way we have (when we update an entry instead of creating a new one):

Lemma 5 ($\equiv_F$ and store update)

$$\begin{cases} P \equiv_F Q \\ a \equiv_F a' \quad \wedge \quad \iota \equiv_F \iota' \\ P' = P \ \mathit{except}\ \sigma_{\alpha'_P} = \{\iota \to a\} + \sigma_{\alpha_P} \\ Q' = P \ \mathit{except}\ \sigma_{\alpha'_Q} = \{\iota' \to a'\} + \sigma_{\alpha_Q} \end{cases} \Rightarrow P' \equiv_F Q'$$

Note that $\iota \equiv_F \iota'$ is important because ι and ι' are already in P and Q.

Proof : Similar to the preceding one. Note that: $\iota \equiv_F \iota' \Rightarrow \left(\alpha_P \overset{\alpha*}{\mapsto}_{L_i} \iota, \Leftrightarrow \alpha_Q \overset{\alpha*}{\mapsto}_{L_i} \iota'\right)$.
□

Lemma 6 ($\equiv_F$ and substitution)

$$\begin{cases} P \equiv_F Q \\ \iota \equiv_F \iota' \end{cases} \Rightarrow a\{\!\{x \leftarrow \iota\}\!\} \equiv_F a\{\!\{x \leftarrow \iota'\}\!\}$$

The proof is straightforward.

Recall that:

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

Lemma 7 (Another definition of deep copy)

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

As a direct consequence:

$$\iota \equiv_F \iota' \Rightarrow \forall \iota_1 \in \mathit{copy}(\iota, \sigma_{\beta_P}), \exists \iota'_1 \in \mathit{copy}(\iota', \sigma_{\beta_Q}),\ \iota_1 \equiv_F \iota'_1$$

and

The following property states that adding equivalent stores to equivalent configurations produces equivalent configurations.

Lemma 8 ($\equiv_F$ and store merge)

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

Proof :

$$\begin{aligned} \mathit{Copy\&Merge}(\sigma_{\beta_P}, \iota_0\ ;\ \sigma_{\alpha_P}, \iota) &= \mathit{Merge}(\iota, \sigma_{\alpha_P},\ \mathit{copy}(\iota_0, \sigma_{\beta_P})\{\!\{\iota_0 \leftarrow \iota\}\!\}) \\ &= \mathit{copy}(\iota_0, \sigma_{\beta_P})\theta_0 + \sigma_{\alpha_P} \end{aligned}$$

From lemma 7, we have:

$$a \in copy(\iota_0, \sigma_{\beta_P}) \Leftrightarrow \exists L,\ \iota_0 \overset{\beta}{\mapsto}_L a$$

and moreover

Lemma 9 (Copy and Merge) *If* $P' = P$ *except* $\sigma_{\alpha'_P} = Copy\&Merge(\sigma_{\beta_P}, \iota_0\ ;\ \sigma_{\alpha_P}, \iota)$

$$\iota_0 \overset{\beta_P}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}}{\mapsto}_L a'$$

Thus for all $a \in copy(\iota_0, \sigma_{\beta_P})$

$$\iota_0 \overset{\beta_P}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}}{\mapsto}_L a\theta_0$$

From this lemma and the same for configuration Q, the proof is straightforward.

Informally, if a location is in the merged sub-store of $\alpha_{P'}$ then it comes from the sub-store $copy(\iota_0, \sigma_{\beta_P})$ of P and it is equivalent to a location inside $copy(\iota'_0, \sigma_{\beta_Q})$ (because $\iota_0 \equiv_F \iota'_0$) which produces a location in the merged sub-store of $\alpha_{Q'}$. We conclude by $\iota \equiv_F \iota' \Rightarrow \left(\iota \overset{\alpha_{P'}}{\mapsto}_L a \Leftrightarrow \iota' \overset{\alpha_{Q'}}{\mapsto}_L a'\right)$ and finally P' and Q' are equivalent. The proof of alias conditions follow the same kind of reasoning.

□

8.1.5 REPLY and $\equiv_F$

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

Property 11 (REPLY and $\equiv_F$)

$$P \overset{\text{REPLY}}{\longrightarrow} P' \Rightarrow P \equiv_F P'$$

Proof : We only have to prove that the updated store is equivalent with the old one. The remaining of the activities are unchanged.

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

We have

$$\sigma'_\alpha = Merge(\iota, \sigma_\alpha,\ copy(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota\}\!\}) = copy(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota\}\!\}\theta + \sigma_\alpha = copy(\iota_f, \sigma_\beta)\theta_0 + \sigma_\alpha$$

Let ι' be in the updated part of the store of α: $\iota \overset{\alpha}{\mapsto}_L \iota'$. Then $\iota' = \iota_0\theta_0$ where $\iota_0 \in copy(\iota_f, \sigma_\beta)$ and $\iota_f \overset{\beta}{\mapsto}_L \iota_0$ (lemma 9).

Let L_0 such that $\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota$. Thus we have $\alpha_P \overset{\alpha*}{\mapsto}_{L_0.L} \iota_0$ (or more precisely $\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \wedge F_\beta(f_i^{\gamma\to\beta}) = \iota_f \wedge \iota_f \overset{\beta}{\mapsto}_L \iota_0$) if and only if $\alpha_{P'} \overset{\alpha}{\mapsto}_{L_0.L} \iota'$

For the alias conditions , note that:

If $\alpha_{P'} \overset{\alpha}{\mapsto}_{L_0.L} \iota_0$ and $\alpha_{P'} \overset{\alpha}{\mapsto}_{L'} \iota_0$ then $L' = L_0.L''$ by definition of the Merge operator (the only common location between original and merged store is ι) and thus, because the deep copy creates a part of store similar to the original one (lemma 9),

$$\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \wedge F_\beta(f_i^{\gamma\to\beta}) = \iota_f \wedge \begin{cases} \iota_f \overset{\beta}{\mapsto}_L \iota_0 \\ \iota_f \overset{\beta}{\mapsto}_{L''} \iota_0 \end{cases}$$

□

Note that this proof justifies the definition of $\overset{\alpha*}{\mapsto}_{L_0.L}$ and of the equivalence modulo futures.

8.1.6 Equivalence modulo futures and reduction

The objective here is to prove that if a reduction can be made on a configuration then the same one can be made on an equivalent configuration. This is a very important property. The proof is decomposed in two parts. First, we may need to apply several REPLY rules to be able to perform the same reduction on the two terms. Indeed one of the configurations can be waiting the value of the future by necessity because the future updated or not in the two configurations are not the same (definition of equivalence). The second part consist in verifying that the application of the same reduction rule on equivalent terms leads to equivalent terms. Note the similarity with properties of bisimulation.

In the following, let T be any parallel reduction rule: T range over $\{$LOCAL, NEWACT, REQUEST, SERVE, ENDSERVICE, REPLY$\}$. $\overset{T}{\longrightarrow}$ denotes the application of a parallel rule named T (cf Table 3).

$\equiv_F$ verifies the important following property:

Property 12 ($\equiv_F$ and reduction(1))

$$P \equiv_F P' \wedge P \overset{T}{\longrightarrow} Q \Rightarrow \begin{cases} \textit{if } T = \text{REPLY } \textit{then } Q \equiv_F P' \\ \textit{else } \exists Q',\ P' \overset{\text{REPLY}^*}{\longrightarrow} \overset{T}{\longrightarrow} Q' \wedge Q' \equiv_F Q \end{cases}$$

This property suggests to define the new reduction $\overset{T}{\Longrightarrow}$:

Let $\overset{T}{\Longrightarrow}$ be the reduction $\overset{T}{\longrightarrow}$ preceded by some applications of the REPLY rule if the rule of $\overset{T}{\longrightarrow}$ is not REPLY and any (possibly 0) number of application of the REPLY if the rule is REPLY. More precisely:

$$\overset{T}{\Longrightarrow} \quad = \quad \begin{array}{ll} \overset{\text{REPLY}^*}{\longrightarrow} \overset{T}{\longrightarrow} & \text{if } T \neq \text{REPLY} \\ \overset{\text{REPLY}^*}{\longrightarrow} & \text{if } T = \text{REPLY} \end{array}$$

Note that if the applied rule is REPLY, $\overset{T}{\Longrightarrow}$ may do nothing. That is necessary for example to simulate the update of a future on an (equivalent) configuration where this future has already been updated.

$\equiv_F$ verifies the important following property:

Property 13 ($\equiv_F$ and reduction(2))

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

The properties 12 and 13 are equivalent. The following proof is valid for both.

Proof : If we cannot apply the same reduction than $P \longrightarrow Q$ (same rule on the same activities ...) on P', we apply $\overset{\text{REPLY}}{\longrightarrow}$ enough times to be able to apply the reduction $P' \overset{\text{REPLY}^*}{\longrightarrow} P''$. It is straightforward to check that if two configurations are equivalent, the same reduction can be applied on the two configuration except if one of them is stuck. Stuck configurations can occur 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 leads to an error.
- Access to a future: if in an activity of P' we have $a = \ldots \iota' \ldots$ and $\sigma'_\alpha(\iota') = fut(f_i^{\gamma\to\beta})$ then in P, $a = \ldots \iota \ldots$ where $\iota = \iota'\theta_\alpha$ and $\sigma_\alpha(\iota)$ is not a future. The future equivalence ensures that $f \in F_{\beta'_P}$. Then $P' \overset{\text{REPLY}^*}{\longrightarrow} P''$ where $P'' \equiv_F P$ (property 11) and in P'' $\sigma'_\alpha(\iota')$ is not a future reference. Then the same reduction can be applies on P'' and P. Actually the REPLY rule needs to be applied:
  - 0 time if the object to be accessed is not a future,
  - 1 time if it is actually a future whose value is not a future
  - n times if it is a future whose future value is n times itself a future reference.

Note that the reduction that occur in P cannot access an object inside a future that has not been updated in P' because $P \equiv_F P' \Rightarrow a_{\alpha_P} \equiv_\alpha a_{\alpha'_P} \Rightarrow \forall \iota \in a_{\alpha_P}, \iota\theta_\alpha \in a_{\alpha'_P}$

Now, we have to verify that if $P'' \equiv_F P$ and we apply the same reduction rule on P and P'' on equivalent activity(ies) we obtain equivalent configurations:

$$\begin{cases} P \overset{T}{\longrightarrow} Q \\ P'' \overset{T}{\longrightarrow} Q' \Rightarrow Q' \equiv_F Q \\ P'' \equiv_F P \end{cases}$$

This is obtained by a (long) case study. The different cases depend of the reduction applied and the rule applied to prove the equivalence. In the following we will only focus on the cases where one of the location concerned by the reduction points to a future in P and is an object in P''. Other cases (several futures or no future) can be trivially obtained. Of course, we will (implicitly) use the fact that if two terms are equivalent, they have the same form.

LOCAL

STOREALLOC Direct from lemma 4.

FIELD

$$\frac{\sigma(\iota) = [..; l_i = \iota_i; ....]}{(\mathcal{R}[\iota.l_i], \sigma) \rightarrow_S (\mathcal{R}[\iota_i], \sigma)} \text{ (FIELD)}$$

If $\iota.l_i \equiv_\alpha \iota_2.l_i$ then $\iota \equiv_\alpha \iota_2$ and $\iota_{i1} \equiv_\alpha \iota_{i2}$. Thus $P'' \equiv_F Q$

INVOKE Straightforward: Note that the two method bodies must be equivalent and the two arguments too. The final equivalence comes from lemma 6.

UPDATE Direct from lemma 5.

CLONE Note that you cannot clone a future. Other cases are trivial.

NEWACT The only interesting case is the presence of futures in the newly created activity. Lemma 8 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 By hypothesis (modulo renaming), we can choose the same name for future in P and Q, the same location for the copy of the argument. Lemma 8 can be applied to manage with futures that can be present in the deep copy of the request parameters.

The rest of the proof is straightforward. For example the equivalence of requests is established by taking the same location and future names and $[m_j; \iota; f_i^{\alpha\rightarrow\beta}] \equiv_\beta [m'_j; \iota; f_i^{\alpha\rightarrow\beta}]$ comes from $m_j = m'_j$ because $a_\alpha \equiv_\alpha a'_\alpha$

SERVE The equivalence between the two request lists implies that the served requests are equivalent which is sufficient to conclude. Note that the fact that, in the equivalence definition, the reordering φ of requests must be compatible with the RSL equivalence is essential here. More precisely:

$$P'' \equiv_F P \Rightarrow \forall M \in \mathcal{M}_{\alpha_P}, R_{\alpha_P}\big|_M \equiv_F R_{\alpha_{P''}}\big|_M$$

And thus the SERVE will serve equivalent requests.

ENDSERVICE The equivalence between futures lists is straightforward. The proof is based on the application of the lemma 8.

REPLY In this case we have $P' \equiv_F Q$ ($P' \Longrightarrow P'$). Note that here, we may be unable to apply directly the same rule on the two equivalent terms for three reasons:

- either we may need several future updates to have the reference on which we want to update the concerned future (we need several futures updates to apply the same rule),
- or the future has already been updated in the equivalent term (no REPLY rule is applied),

- or there was a cycle of futures references and the order of futures updated was different (in Figure 10 we have either only references to future f_1 or to future f_2)

Note that most of this proof is simplified by the important lemma 8. □

Corollary 1

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

Proof : $P \stackrel{T}{\Longrightarrow} Q$ can be decomposed in $P \stackrel{\text{REPLY}^*}{\longrightarrow} P_1 \stackrel{T}{\longrightarrow} Q$ (or $P \stackrel{\text{REPLY}^*}{\longrightarrow} Q$ if $T = \text{REPLY}$). □

Corollary 2

$$P \equiv_F P' \wedge P \stackrel{T}{\longrightarrow} Q \Rightarrow \exists P'', Q', \begin{cases} P'' \equiv_F P' \\ P'' \stackrel{T}{\longrightarrow} Q' \\ Q' \equiv_F Q \end{cases}$$

Proof : Direct from propositions 13 and 11 □

8.1.7 Another formulation

We formalize here another definition of equivalence between terms based on renamings and prove its equivalence with the preceding one.

Let us extend the preceding renaming (of activities and futures) with a renaming on locations inside an activity θ_{α_i} and between two activities $\theta_{\alpha_i \to \alpha'_j, \iota'}$. Remind that θ_{act} is a bijection from $Act(P)$ to $Act(Q)$, θ_{fut} is a bijection from F_{α_P} to $F_{\alpha' Q}$, φ_{α_i} is a permutation on the request queue of α'_I compatible with the RSL equivalence.

$$\begin{aligned}
\Theta &::= (\theta_{act}, \theta_{fut}, \theta_{\alpha_1}, \ldots \theta_{\alpha_i \to \alpha'_i, \iota}, \ldots, \varphi_{\alpha_1} \ldots) \\
\theta_{act} &::= \{\!\{\alpha_1 \to \alpha'_1 \ldots\}\!\}; \\
\theta_{fut} &::= \{\!\{f_i^{\beta \to \alpha} \to f_i'^{\beta' \to \alpha'}, \ldots\}\!\}; \\
\forall \alpha_i \to \alpha'_i \in \theta_{act},\ \theta_{\alpha_i} &::= \{\!\{\iota_1 \to \iota'_1 \ldots\}\!\};\ \text{where } \iota_1 \in locs(\alpha_i), \iota'_1 \in locs(\alpha'_i) \\
\alpha_i \in P, \alpha'_j \in Q,\ \theta_{\alpha_i \to \alpha'_j, \iota'} &::= \{\!\{\iota_1 \to \iota'_1, \ldots\}\!\} \text{ where } \iota_1 \in locs(\alpha_i),\ \iota', \iota'_1 \in locs(\alpha'_j) \\
\alpha_i \in P, \alpha'_j \in Q,\ \theta_{\alpha_i \leftarrow \alpha'_j, \iota} &::= \{\!\{\iota_1 \to \iota'_1, \ldots\}\!\} \text{ where } \iota, \iota_1 \in locs(\alpha_i),\ \iota'_1 \in locs(\alpha'_j)
\end{aligned}$$

The two last renaming of locations allow to express future updates:

In the two last lines ι and ι' represent the location of the updated future. To prove that a future $f_i^{\beta \to \alpha}$in P and its update in the location ι' of the activity γ of Q are equivalent, we must provide a renaming $\theta_{\alpha \to \gamma', \iota'}$.

Of course, each renaming must be bijective and

- for each $\alpha'_i = \alpha_i \theta_{act}$ the sets $codom(\theta_{\alpha_i})$, $(codom(\theta_{\beta \to \alpha'_i})_{\beta \in P}$ are disjuncts;
- for each α_i the sets $dom(\theta_{\alpha_i})$, $(dom(\theta_{\alpha_i \leftarrow \beta'})_{\beta' \in Q}$ are disjuncts;

We define the following equivalence relation:

Definition 13 (Equivalence modulo futures(2))
Suppose we want to prove $P \equiv_F Q$, we must find $\Theta := (\theta_{act}, \theta_{fut}, \theta_{\alpha'_1}, \ldots)$ such that $\equiv_F$ is the largest equivalence relation closed backward under the rules of table 5 (or more precisely table 6) (we note $\alpha' = \alpha\theta_{act}$). x is either α or $\alpha \to \beta', \iota'$ or $\alpha \leftarrow \beta', \iota$.

$$\frac{a \equiv b}{a \equiv_\alpha b} \qquad \frac{\sigma_{\alpha_P}(\iota) \equiv_{\alpha\to\beta,\iota'} \sigma_{\beta_Q}(\iota\theta_{\alpha\to\beta,\iota'})}{\iota \equiv_{\alpha\to\beta,\iota'} \iota\theta_{\alpha\to\beta,\iota'}} \qquad \frac{\begin{array}{c}a \text{ is in the location } \iota' \text{ of the activity } \gamma \text{ of } Q \\ F_{\beta_P}(f_i^{\alpha\to\beta}) = \iota \qquad \iota \equiv_{\beta\to\gamma,\iota'} \iota'\end{array}}{fut(f_i^{\alpha\to\beta}) \equiv_x a}$$

Table 5: equivalence (short)

and all the trivial induction rules corresponding to operators in the syntax.

$\equiv_\alpha$ denotes the equivalence between terms that appear in α and α'. $\equiv_{\alpha\to\beta',\iota'}$ denotes the equivalence between terms contained in a future calculated in the activity α of P and its updated value in the location ι' of the activity β' of Q. $\equiv_x$ denotes any equivalence relation (either inside an activity or between activities if x is of the form $\alpha \leftarrow \beta', \iota$ or $\alpha \to \beta', \iota'$).

$\theta_{\alpha\to\beta',\iota'}$ means that the future calculated in the activity α of P has been updated in the location ι' of the activity β' of Q. The renaming $\theta_{\alpha\to\beta',\iota'}$ must be applied to locations of α in order to obtain locations (corresponding to a deep copy) in β'.

Symmetrically $\theta_{\alpha\leftarrow\beta',\iota}$ is useful when a future in Q has been updated in the activity α of P, at the location ι.

This definition is coinductive because, we may need to use the fact that $\iota \equiv \iota'$ to prove that $\iota \equiv \iota'$[7]. The figure 13 shows an example of configuration where such kind of definition is necessary

Important remarks:

- $\theta_{\alpha\to\alpha',\iota}$ is different from θ_α and is useful for the update of a future in the same activity.
- "a is in the location ι' of the activity γ of Q" could be (easily) expressed more formally but we would have to use even more complicated rules and notations;
- The equivalence between activities should verify $\alpha' = \alpha\theta_{act}$.(considered as implicit)
- the unaccessible parts of the store are not taken into account and can be garbage collected in both definition of equivalence.

[7]note that this is not surprising as if we try to define

$$fut(f) \equiv_x fut(f\theta_{fut}) \qquad AO(\alpha) \equiv_x AO(\alpha') \qquad \emptyset \equiv_x \emptyset \qquad \frac{\sigma_{\alpha_P}(\iota) \equiv_\alpha \sigma_{\alpha'_Q}(\iota\theta_\alpha)}{\iota \equiv_\alpha \iota\theta_\alpha}$$

$$\frac{\sigma_{\alpha_P}(\iota_1) \equiv_{\alpha\leftarrow\beta,\iota_0} \sigma_{\beta_Q}(\iota_1\theta_{\alpha\leftarrow\beta,\iota})}{\iota \equiv_{\alpha\leftarrow\beta,\iota_0} \iota\theta_{\alpha\leftarrow\beta,\iota}} \qquad \frac{\begin{array}{c}b \text{ is in the location } \iota \text{ of the activity } \gamma \text{ of } P \\ F_{\beta_Q}(f_i^{\alpha\to\beta}) = \iota' \qquad \iota \equiv_{\gamma\leftarrow\beta,\iota} \iota'\end{array}}{b \equiv_x fut(f_i^{\alpha\to\beta})}$$

$$\frac{\sigma_{\alpha_P}(\iota) \equiv_{\alpha\to\beta,\iota'_0} \sigma_{\beta_Q}(\iota\theta_{\alpha\to\beta,\iota'})}{\iota \equiv_{\alpha\to\beta,\iota'_0} \iota\theta_{\alpha\to\beta,\iota'}} \qquad \frac{\begin{array}{c}a \text{ is in the location } \iota' \text{ of the activity } \gamma \text{ of } Q \\ F_{\beta_P}(f_i^{\alpha\to\beta}) = \iota \qquad \iota \equiv_{\beta\to\gamma,\iota'} \iota'\end{array}}{fut(f_i^{\alpha\to\beta}) \equiv_x a}$$

$$\frac{\iota \equiv_\alpha \iota' \qquad R_{\alpha_P} \equiv_\alpha R_{\alpha'_Q}}{[m_j, \iota, f] :: R_{\alpha_P} \equiv_\alpha [m_j, \iota', f\theta_{fut}] :: R_{\alpha'_Q}} \qquad \frac{\forall f \in F_{\alpha_P},\ F_{\alpha_P}(f) \equiv_\alpha F_{\alpha'_Q}(f\theta_{fut})}{F_{\alpha_P} \equiv_\alpha F_{\alpha'_Q}}$$

$$\frac{\begin{array}{c}a_{\alpha_P} \equiv_\alpha a_{\alpha'_Q} \qquad \iota_{\alpha_P} \equiv_\alpha \iota_{\alpha'_Q} \qquad F_{\alpha_P} \equiv_\alpha F_{\alpha'_Q} \\ R_{\alpha_P} \equiv_\alpha \varphi(R_{\alpha'_Q}) \qquad f_{\alpha'_Q} = f_{\alpha_P}\theta_{fut}\end{array}}{\alpha[a_{\alpha_P}; \sigma_{\alpha_P}; \iota_{\alpha_P}; F_{\alpha_P}; R_{\alpha_P}; f_{\alpha_P}] \equiv_\alpha \alpha'[a_{\alpha'_Q}; \sigma_{\alpha'_Q}; \iota_{\alpha'_Q}; F_{\alpha'_Q}; R_{\alpha'_Q}; f_{\alpha'_Q}]}$$

$$\frac{\exists\Theta = (\theta_{act}, \theta_{fut}, \theta_{\alpha_1}, \ldots) \qquad \alpha \in P \Leftrightarrow \alpha \in Q\theta_{act} \qquad \forall\alpha \in P, \alpha[\ldots] \equiv_\alpha \alpha'[\ldots]}{P \equiv_F Q}$$

Table 6: Equivalence

8.1.8 Equivalence of the two definition

Let us now prove that the two definitions are equivalent. Let $\equiv_{F2}$ be the second formulation of equivalence

Proof : We only give here some details of the proof. The whole proof is longer but based on the principles given below

$\equiv_{F2} \Rightarrow \equiv_F$ If $P \equiv_{F2} Q$, then for all α we have $\alpha_P[\ldots] \equiv_x \alpha_R[\ldots]$. We will not worry about renaming of activities and futures. By recurrence, we will prove that:

$$\left(\forall L,\ \alpha_P \overset{\alpha*}{\mapsto}_L b \Rightarrow \alpha_Q \overset{\alpha*}{\mapsto}_L b' \wedge b \equiv_x b'\right) \Rightarrow \left(\alpha_P \overset{\alpha*}{\mapsto}_{L.l} c \Rightarrow \alpha_Q \overset{\alpha*}{\mapsto}_{L.l} c' \wedge c \equiv_x c'\right)$$

It is easy to verify that $\alpha_P[\ldots] \equiv_\alpha \alpha_R[\ldots]$ and $\alpha_P \overset{\alpha*}{\mapsto}_\emptyset \alpha_P$

Most cases are simple case analysis. For example, if $l = m_j$, then b is an object and b' too (because $b \equiv_x b'$). We use

$$\frac{\ldots \qquad c \equiv_x c' \qquad \ldots}{b = [\ldots, m_j = \varsigma(s,x)c, \ldots] \equiv_x [\ldots, m_j = \varsigma(s,x)c', \ldots] = b'}$$

then we have $\alpha_P \overset{\alpha*}{\mapsto}_{L.m_j} c \Rightarrow \alpha_Q \overset{\alpha*}{\mapsto}_{L.m_j} c' \wedge c \equiv_x c'$

Note that the case where we have a future that is only updated in one configuration need only to be considered in the case $l = ref$[8]. In that case a simple store access by an access to the calculated (but not updated) future.

We detail here the case where $l = ref$:

- future reference: if:

$$\frac{\begin{array}{c} c \text{ is in the location } \iota \text{ of the activity } \gamma \text{ of } P \\ F_{\beta_Q}(f_i^{\delta\to\beta}) = \iota' \qquad \iota \equiv_{\beta\leftarrow\gamma,\iota} \iota' \end{array}}{c \equiv_x fut(f_i^{\delta\to\beta})}$$

  and then $\alpha_P \overset{\alpha*}{\mapsto}_L \iota \overset{\gamma}{\mapsto}_{ref} c$ and, suppose the considered $fut(f_i^{\delta\to\beta})$ is in the location ι_0 of the activity γ_0 in Q. We have
  $\alpha_Q \overset{\alpha*}{\mapsto}_L \iota_0 \wedge \sigma_{\gamma_0}(\iota_0) = fut(f_i^{\delta\to\beta}) \wedge F_{\beta_Q}(f_i^{\delta\to\beta}) = \iota' \wedge \iota' \overset{\gamma_0}{\mapsto}_{ref} \sigma(i')$ Thus $\alpha_Q \overset{\alpha*}{\mapsto}_{L.ref} \sigma_{\beta_Q}(i')$.

  We conclude easily because $\iota \equiv_{\beta\leftarrow\gamma,\iota} \iota'$ is ensured by proving $c = \sigma_\gamma(\iota) \equiv_{\beta\leftarrow\gamma,\iota} \sigma_{\beta_Q}(i')$

- if $x = \alpha \leftarrow \beta, \iota_0$

$$\frac{c = \sigma_{\alpha_P}(\iota_1) \equiv_{\alpha\leftarrow\beta,\iota_0} \sigma_{\beta_Q}(\iota_1\theta_{\alpha\leftarrow\beta,\iota}) = c'}{b = \iota \equiv_{\alpha\leftarrow\beta,\iota_0} \iota\theta_{\alpha\leftarrow\beta,\iota} = b'}$$

  This proves that $b \overset{\alpha*}{\mapsto}_{ref} c$, $b \overset{\alpha*}{\mapsto}_{ref} c'$ and $c \equiv_x c'$ and concludes for equivalence.

All the cases that are not detailed before involve the same kind of arguments.

The bijectivity of renaming ensures the two last conditions of definition 11. This is proved by showing that if one condition is not verified then a renaming is not bijective:

Note that we can prove moreover that if $\alpha_P \overset{\alpha}{\mapsto}_L a$ and $\alpha_P \overset{\alpha}{\mapsto}_{L'} a$ then there is ι such that $\alpha_P \overset{\alpha}{\mapsto}_L \iota$ and $\alpha_P \overset{\alpha}{\mapsto}_{L'} \iota$

Suppose for example:

$$\begin{cases} \alpha_P \overset{\alpha}{\mapsto}_L \iota \wedge \alpha_P \overset{\alpha}{\mapsto}_{L'} \iota_2 \wedge \iota \neq \iota_2 \\ \alpha_R \overset{\alpha}{\mapsto}_L \iota' \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} \iota'_2 \end{cases} \wedge \iota' = \iota'_2$$

[8] in fact it could be either considered in $l = ref$ or in all other cases

then one would need to have : $\theta_\alpha = \{\!\{\iota \to \iota', \iota_2 \to \iota' \ldots\}\!\}$ which is not injective.

The bijectivity of the renamings for futures ($\theta_{\alpha_i \leftarrow \alpha'_j, \iota}$) corresponds to the case where $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$ follows futures references.

$\equiv_F \Rightarrow \equiv_{F2}$ The idea is to start from an activity α and follow arrows to determine renaming. Recurrence cases are trivial. We detail only store access and futures:

Note that all the non garbage collectable objects in α are obtained from α by following $\overset{\alpha}{\mapsto}_L$. Also note that for garbage collectable terms, they are taken into account by non of both equivalence relations.

Thus we suppose that P and Q are equivalent according to definition 11.

If a location in α_P is accessible from α_P then there is L such that $\alpha_P \overset{\alpha}{\mapsto}_L \iota$. Definition 11 ensures $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$. We can specify the different renamings from this. Two cases are possible:

- either $\alpha_R \overset{\alpha}{\mapsto}_L \iota'$ then $\{\!\{\iota \to \iota'\}\!\} \in \theta_\alpha$
- or $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$ and ι' is in the activity γ of R then $\{\!\{\iota \to \iota'\}\!\} \in \theta_{\alpha \leftarrow \gamma, \iota_0}$ where ι_0 is the location in α_P corresponding to the last reference to future. More precisely we decompose: $\alpha_R \overset{\alpha*}{\mapsto}_{L_0} \iota'_0 \overset{\gamma}{\mapsto}_{L'}$, then we have $\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota_0$ (because $\alpha_R \overset{\alpha*}{\mapsto}_{L_0} \iota'_0 \Rightarrow \alpha_P \overset{\alpha*}{\mapsto}_{L_0} \iota_0$ and $\iota_0 \in \alpha_P$ because L_0 is a prefix of L)

$\{\!\{\iota \to \iota'\}\!\} \in \theta_{\alpha \to \beta, \iota_0}$ is obtained from the cases where $\alpha_P \overset{\alpha*}{\mapsto}_L \iota$ and $\alpha_R \overset{\alpha}{\mapsto}_L \iota'$. As futures are not garbage collected, these three cases are sufficient to determine renamings.

From these specifications, it is easy to prove $P \equiv_{F2} Q$ by recurrence.

Note that bijectivity conditions on renamings are ensured by the two last conditions of definition 11.

For example, if we had θ_α non bijective then we would have $\{\!\{\iota \to \iota'\}\!\} \in \theta_\alpha$ and $\{\!\{\iota_2 \to \iota'\}\!\} \in \theta_\alpha$ and thus $\alpha_P \overset{\alpha}{\mapsto}_L \iota \wedge \alpha_P \overset{\alpha}{\mapsto}_L \iota_2 \wedge \alpha_R \overset{\alpha}{\mapsto}_L \iota' \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} \iota'$ which is contradictory with definition 11. In the same way we ensure that above construction never requires $\{\!\{\iota \to \iota'\}\!\} \in \theta_\alpha \wedge \{\!\{\iota \to \iota_2\}\!\} \in \theta_\alpha$ if $\iota' \neq \iota_2$. □

8.1.9 Decidability of $\equiv_F$

Property 14 *$\equiv_F$ is decidable.*

Proof : To prove that $\equiv_F$ is decidable, first note that the set of renaming Θ that could prove $P \equiv_F Q$ is finite. Indeed the set of activities of P and Q is finite and the set of locations and of futures too. We conclude easily that the set of possibly valid θ_{act}, θ_{fut}, and θ_{α_i} is finite. In the same way, the number of $\theta_{\alpha_i \to \alpha'_j, \iota'}$ is bounded by the square of the number of activities multiplied by the number of location in activity α'_j. For each $\theta_{\alpha_i \to \alpha'_j, \iota'}$, the set of valid such renamings can be bounded too because the set of locations is finite.

Note that here, the set of such apparently valid renamings is really huge but the real number of renamings that should be considered is much smaller. In practice the renamings should be created during the equivalence proof.

Now, we only have to prove that verifying that whether Θ allows to prove $P \equiv_F Q$ or not is decidable. This is easy to show if we consider that, starting from the inductive rules at the activities level, we prove the equivalence by going deeper into the activity store. The algorithm verifying whether Θ proves $P \equiv_F Q$ or not can be summarized by inductively verifying that the rules of Table 6 are verified using Θ. finiteness of the verification is ensured by marking the rules already verified or more simply the locations already visited. (Note that if we mark locations already visited we have to verify, when arriving at two marked locations that they are equivalent according to Θ).

Note that we did not mentioned here the constraints of bijectivity and other constraints on domain and codomain of renaimings of Θ which are still important. □

8.1.10 Examples

We detail below a list of examples of equivalent terms. The verification of equivalence consist in simply following the arrows on the diagram (trivial). We give some details on the renaming that have to be used in the case of the second equivalence relation.

In the following case: We must stake

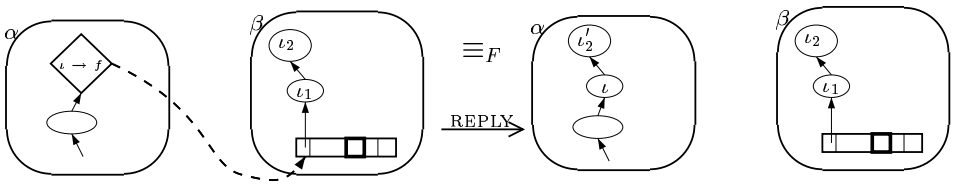


Figure 12: Simple example of future Equivalence

$$\theta_{\beta\to\alpha,\iota} = \{\!\{\iota_1 \to \iota, \iota_2 \to \iota'_2\}\!\}$$

The Figure 14 illustrates the case where there is a cycle of future references. The proof of equivalence is not detailed here, but is based on the renamings:

$$\begin{aligned}
\theta_{\alpha\to\alpha,\iota_2} &= \{\!\{\iota_1 \to \iota_2, \iota_2 \to \iota_3\}\!\} \\
\theta_{\alpha\to\alpha,\iota_3} &= \{\!\{\iota_1 \to \iota_3, \iota_2 \to \iota_4\}\!\} \\
\theta_{\alpha\to\alpha,\iota_4} &= \{\!\{\iota_1 \to \iota_4, \iota_2 \to \iota_5\}\!\}
\end{aligned}$$

The figure 15 illustrates the importance of the bijectivity properties: in Q' every renaming is bijective but we would need to have $\theta_{\beta\to\alpha,\iota} = \{\!\{\iota_1 \to \iota, \iota_2 \to \iota'_2\}\!\}$ and $\theta_{\beta\to\alpha,\iota'} = \{\!\{\iota_1 \to \iota', \iota_2 \to \iota'_2\}\!\}$ which would be contradictory with $codom(\theta_{\beta\to\alpha,\iota})$ and $codom(\theta_{\beta\to\alpha,\iota'})$ disjuncts.

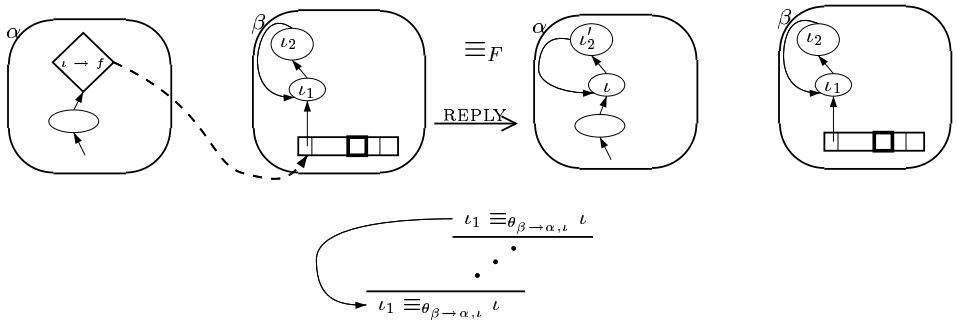


Figure 13: Example of "cyclic" proof

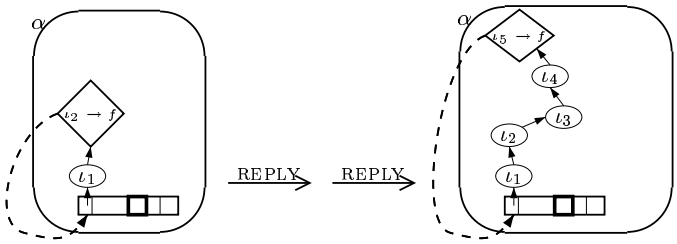


Figure 14: Equivalence in case of cycle of futures

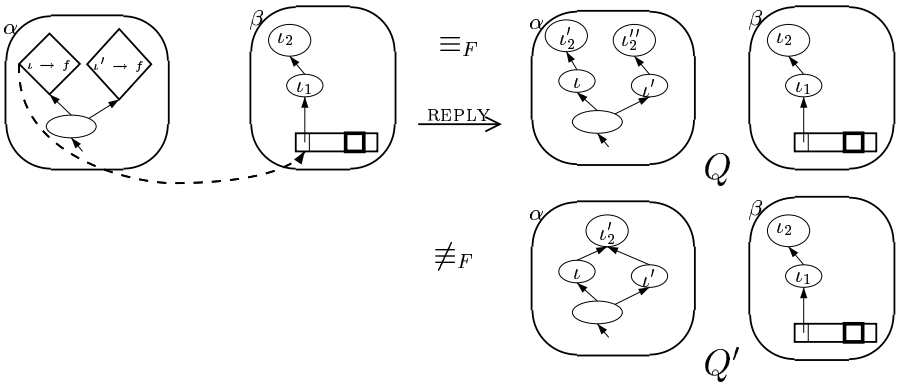


Figure 15: Another example

8.2 Proving Confluence: Diamond Property

8.2.1 Context

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

Definition 14 (confluent configurations: $P_1 \curlyvee P_2$)

$$P_1 \curlyvee P_2 \Leftrightarrow \exists R_1, R_2, \begin{cases} P_1 \xrightarrow{*} R_1 \\ P_2 \xrightarrow{*} R_2 \\ R_1 \equiv_F R_2 \end{cases}$$

This subsection aims at proving the following confluence property:

Property 15 (Confluence)

$$\begin{cases} P_0 \xrightarrow{*} Q \\ P_0 \xrightarrow{*} Q' \\ Q \bowtie Q' \end{cases} \Longrightarrow Q \curlyvee Q'$$

Let P_0 be an initial configuration. Let us consider two configurations 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.

We introduce the set of configurations smaller than this upper bound. $\mathcal{Q}(Q, Q')$ be the set of configurations obtained from P_0 and compatible with Q and Q':

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

8.2.2 Local confluence

We prove the following property by a long case study

Property 16 (diamond property) *Let P be a configuration obtained from P_0: $P_0 \xrightarrow{*} P$*

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

Let us introduce the following lemma:

Lemma 10

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

Proof : It is easy to verify that if a rule (different from REPLY) can be applied on P then it can be applied on $P'$and the reasoning of property 13 suffices to conclude. □

Proof :(of the property)

This proof is a (long) case study on the conflict between rules. We will not detail the cases where one of the applied rule is REPLY. These cases can be verified but are not useful for the proof of the property 17.

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. The following cases are labeled with the two rules in conflict.

- If the concerned rules are different, the activities (α, β) will be indexed by the corresponding rule (e.g. α_{REQUEST} is the activity α of the REQUESTrule : the source activity of the request)
- if the rules are the same, the activities will be indexed by 1 and 2.

Lemmas

Lemma 11 (Extensibility of local reduction)

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

Lemma 12 (*copy* and locations)

$$Refs(copy(\iota,\sigma)) \subseteq Refs(\sigma)$$

Lemma 13 (Multiple copies)

$$copy(\iota,\sigma) + copy(\iota',\sigma') = copy(\iota', copy(\iota,\sigma) + \sigma') \qquad \text{if } \iota \in Refs(copy(\iota',\sigma'))$$

Lemma 14 (Copy and store append)

$$\sigma' + Copy\&Merge(\sigma_\alpha,\iota\ ;\ \sigma_\beta,\iota') \equiv_F Copy\&Merge(\sigma_\alpha,\iota\ ;\ \sigma' + \sigma_\beta,\iota') \qquad \text{if } \iota' \notin Refs(\sigma')$$

Lemma 15 (independent stores)

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

In the following, we suppose that we can choose any location, future or activity names when we need a fresh one. This is justified by the fact that reduction is not sensible (modulo equivalence) to activity futures and locations names.

local vs. parallel reduction

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

LOCAL/NEWACT No conflict : $\alpha_{\text{LOCAL}} = \alpha_{\text{NEWACT}}$ impossible because $\mathcal{R}[Active(\iota)]$ cannot be reduced locally. In the following such cases will not be detailed.

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

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

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

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

We 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 $\sigma_{\beta 2} = \sigma_\beta :: \sigma$. And thus lemma 11 allows to perform the local reduction on the extended store:

$(a_\beta, \sigma_\beta) \rightarrow_S (a_{\beta 1}, \sigma_{\beta 1}) \Rightarrow (a_\beta, \sigma :: \sigma_\beta) \rightarrow_S (a'_{\beta 2}, \sigma'_{\beta 2} :: \sigma)$
where $(a'_{\beta 2}, \sigma'_{\beta 2}) \equiv (a_{\beta 1}, \sigma_{\beta 1})$ and $P_2 = \alpha[\mathcal{R}[\iota_f]; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_{\beta 2}; \iota_\beta; F_\beta; R_{\beta 2}; f_\beta] \| Q$
$\longrightarrow \alpha[\mathcal{R}[\iota_f]; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a'_{\beta 2}; \sigma'_{\beta 2}; \iota_\beta; F_\beta; R_{\beta 2}; f_\beta] = P'_2 \| Q$

$\sigma_{\beta 1}$ is obtained by some updates on σ_β: $\sigma_{\beta 1} = \sigma_0 + \sigma_\beta$. Lemma 14 is used for adding the request to the store obtained by local reduction. We can apply the request rule to P_1 (let $\sigma'_{\beta 1}$ be the new store :
$\sigma'_{\beta 1} = Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \sigma_{\beta 1}, \iota'') \equiv_F \sigma_0 + Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \sigma_\beta, \iota''))$ and obtain a configuration equivalent to P'_2 (lemma 15):
$(a'_{\beta 2}; \sigma'_{\beta 2}) \equiv (a_{\beta 1}, \sigma_{\beta 1} :: \sigma) = (a_{\beta 1}, (\sigma_0 + \sigma_\beta) :: \sigma) \equiv_F (a_{\beta 1}, \sigma_0 + (\sigma_\beta :: \sigma)) \equiv_F (a_{\beta 1}, \sigma'_{\beta 1})$

LOCAL/ENDSERVICE We have:

$$\text{in ENDSERVICE, } \sigma'_\alpha = \sigma_1 :: \sigma_\alpha \text{ where } Refs(\sigma_1) \cap Refs(\sigma_\alpha) = \emptyset$$

And thus lemma 11 is sufficient to conclude.

LOCAL/SERVE no conflict.

creating an activity

NEWACT/NEWACT No conflict. We may only need top rename activities.

NEWACT/REQUEST We only have 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.

NEWACT/ENDSERVICE No conflict.

NEWACT/SERVE No conflict.

Localized operations (SERVE, ENDSERVICE)

- **SERVE:**

SERVE/SERVE No conflict.

NEWSERVICE/SERVE If $\alpha_{\text{SERVE}} = \beta_{\text{REQUEST}}$ Informally, if we 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 because SERVEtakes the *first* request matching M. Note that the fact that the first request is taken is essential to ensure confluence.

- **ENDSERVICE:**

ENDSERVICE/ENDSERVICE No conflict.

REQUEST/ENDSERVICE There can only be a conflict when $\alpha_{\text{ENDSERVICE}} = \beta_{\text{REQUEST}} = \beta$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\to\beta} \text{ new future} \qquad \iota_f \notin dom(\sigma_\alpha) \\ \sigma_{\beta 1} = Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \sigma_\beta, \iota'') = \sigma + \sigma_\beta \qquad \sigma_{\alpha 1} = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| Q \\ \longrightarrow \alpha[\mathcal{R}[\iota_f]; \sigma_{\alpha 1}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \| \beta[a_\beta; \sigma_{\beta 1}; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha\to\beta}]; f_\beta] \| Q = P_1'\end{array}} \text{(REQUEST)}$$

$$\frac{\begin{array}{c}\iota' \notin dom(\sigma_\beta) \qquad F_\beta' = F_\beta :: \{f_\beta \mapsto \iota'\} \\ \sigma_{\beta 2} = Copy\&Merge(\sigma_\beta, \iota \ ; \ \sigma_\beta, \iota') = \sigma' + \sigma_\beta\end{array}}{\beta[\iota \Uparrow f_i^{\delta\to\beta}, a; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \| P \longrightarrow \beta[a; \sigma_{\beta 2}; \iota_\beta; F_\beta'; R_\beta; f_i^{\delta\to\beta}] \| P = P_2'} \text{(ENDSERVICE)}$$

The conflict only concerns the store. But the the merges that are performed on the store are independent ($\iota'' \notin dom(\sigma_\beta)$). We can suppose that these two operations create disjuncts set of locations. Then we can perform the missing two rules on the configurations P_1' and P_2'. We obtain configurations with stores $\sigma'_{\beta 2} \equiv_F \sigma + \sigma_{\beta 2}\ \sigma' + \sigma_{\beta 1} \equiv_F \sigma'_{\beta 1}$and The crucial point of the proof uses lemma 15 to prove: $\sigma'_{\beta 2} \equiv_F \sigma + \sigma_{\beta 2} = \sigma + \sigma' + \sigma_\beta = \sigma' + \sigma + \sigma_\beta = \sigma' + \sigma_{\beta 1} \equiv_F \sigma'_{\beta 1}$

ENDSERVICE/SERVE No conflict.

Concurrent request sending: REQUEST/REQUEST $\underline{\alpha_1 = \beta_2}$ or $\underline{\beta_1 = \alpha_2}$ same kind of arguments as in the case LOCAL/REQUEST.

$\underline{\alpha_1 = \alpha_2}$No conflict.

$\underline{\beta_1 = \beta_2}$ Impossible because $P_1, P_2 \in \mathcal{Q}$ so, if $\beta_1 = \beta_2$ then the two requests come from the same activity (RSL compatibility) $\alpha_1 = \alpha_2$ and there is no conflict. □

8.2.3 Extension

In this subsection we extend the local diamond property presented before to obtain a real diamond property which will allow us to conclude about confluence of our calculus.

First, we introduce the following lemma:

Lemma 16

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

Proof : Trivial because for any $\alpha \in P$, $P \equiv_F P' \Rightarrow RSL_{\alpha_P} = RSL_{\alpha'_P}$. □

Property 17 (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 : If one of the $\Longrightarrow$ applies only REPLY rules then we can conclude immediately by corollary 13.

Else:

$$P_1 \Longrightarrow Q_1$$

then $P_1 \overset{\text{REPLY}^*}{\longrightarrow} P_1' \longrightarrow Q_1$
in the same way $P_2 \overset{\text{REPLY}^*}{\longrightarrow} P_2' \longrightarrow Q_2$
where $P_1' \equiv_F P_2'$. And by corollary 2:

$$\exists P_1'',\ P_1' \overset{\text{REPLY}^*}{\longrightarrow} P_1'' \wedge P_1'' \longrightarrow Q_2' \wedge Q_2' \equiv_F Q_2$$

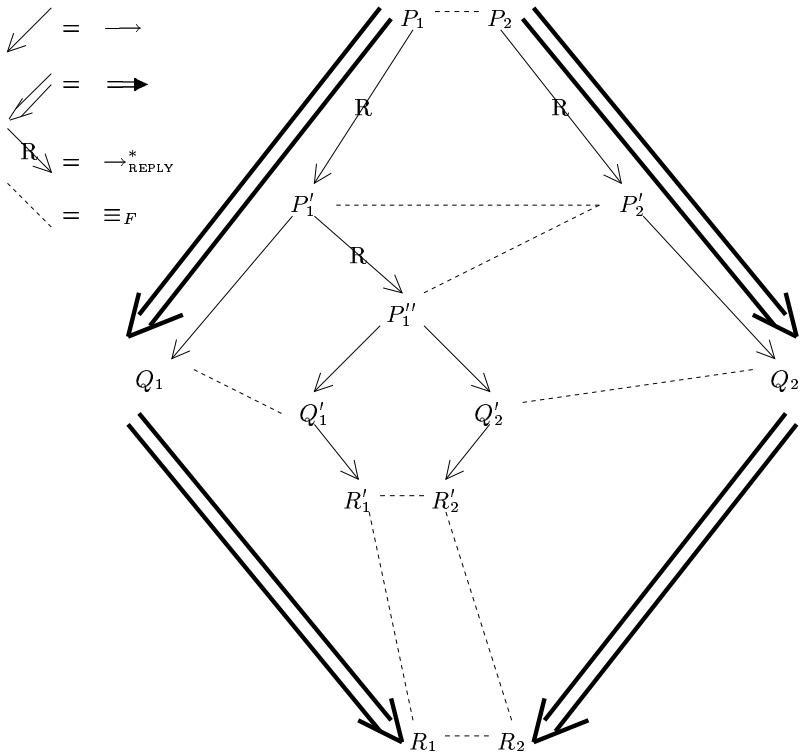


Figure 16: The Diamond property proof

Moreover by lemma 10:

$$\begin{cases} P'_1 \overset{\text{REPLY}^*}{\longrightarrow} P''_1 \\ P'_1 \longrightarrow Q_1 \end{cases} \Rightarrow \begin{cases} P''_1 \longrightarrow Q'_1 \\ Q'_1 \equiv_F Q_1 \end{cases}.$$

Then we use diamond property 16 (we use lemma 16 to prove $Q'_1, Q'_2 \in \mathcal{Q}$):

$$\begin{cases} P''_1 \longrightarrow Q'_1 \\ P''_1 \longrightarrow Q'_2 \\ Q'_1, Q'_2 \in \mathcal{Q} \end{cases} \Longrightarrow Q'_1 \equiv_F Q'_2 \vee \exists R_1, R_2, \begin{cases} Q'_1 \longrightarrow R'_1 \\ Q'_2 \longrightarrow R'_2 \\ R'_1 \equiv_F R'_2 \\ R'_1, R'_2 \in \mathcal{Q} \end{cases}.$$

We conclude by using property 13:

$$Q'_1 \longrightarrow R'_1 \wedge Q'_1 \equiv_F Q_1 \Rightarrow Q_1 \Longrightarrow R_1 \wedge R'_1 \equiv_F R_1$$

$$Q'_2 \longrightarrow R'_2 \wedge Q'_2 \equiv_F Q_2 \Rightarrow Q_2 \Longrightarrow R_2 \wedge R'_2 \equiv_F R_2$$

thus proving (remark $R_1 \equiv_F R_2$ and $R_1, R_2 \in \mathcal{Q}$ come trivially):

$$\begin{cases} Q_1 \Longrightarrow R_1 \\ Q_2 \Longrightarrow R_2 \\ R_1 \equiv_F R_2 \\ R_1, R_2 \in \mathcal{Q} \end{cases}$$

□

9 Conclusion

In this paper we presented a calculus modeling asynchronous communications in object systems, and exhibited confluence properties.

The ASP calculus is based on asynchronous *activities* processing *requests* and responding by mean of *futures*: after sending a request, the execution continues until the result of this request is needed; the result is represented by a future until its value is calculated and updated. The semantics captures strategies that range from eager future update (as soon as the service is over) to on-demand result transmission (only when the value of a future is needed). In ProActive, a Java API implementing the ASP model, two different strategies for future update are implemented. But these two strategies do not need to store future and thus futures do not need to be garbage collected, and network latency can be overlapped with computation.

The most interesting property we proved is a sufficient condition to ensure confluence of processes treating requests in FIFO order (Property 15: RSL-based Confluence). We proved that the execution of a set of activities is only determined by the ordered list of activities sending requests (Request Senders List), as perceived locally by each activity. For a given activity, the RSL is the chronological sequence of source activities of received requests; only *activity names* are needed, target methods and parameters are unnecessary. If the RSLs of two configurations have a Least Upper Bound (LUB) then the two configurations can be reduced to the same third one.

Note that what provides determinism in our calculus is the balancing between asynchronous execution and synchronisations due to rendez-vous mechanism and wait-by necessity (data-flow synchronisation). Of course the topology of objects stating that we can only access an activity through its active object is important too. Note also that what makes our properties poweful is that it is insensitive to order of replies : the determinism is only determined by the RSL in each activity. In other word an execution is uniquely determined by the order of activities sending request to each active object.

It seems possible to generalize (weaker conditions) or to find complementary properties to ensure confluence. A promising perspective seems to lie in an ASP extension to deal with non-FIFO activities. More specifically, we plan to extend the current work on configurations that behave like Process Networks [19].

Comparison with related calculi

In $\pi o\beta\lambda$[18] caller always wait for the method result (synchronous method call), which can be returned before the end of the called method. 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. In $\pi o\beta\lambda$, this characteristic is the source of parallelism whereas in our case 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 [18], stating that the result of a method is not modified after being returned, is ensured in ASP by a deep copy of the result (Property 3: Store partitioning). Similarly, the *unique reference* condition from the same work is balanced in ASP with the constraints on objects topology.

The preconditions for RSL-based confluence (Property 15) are both simple and dynamic. In fact, one just has to ensure that, at any time a unique activity can make a request on a given one. This can be compared to linear (or linearized) channels of π-calculus [21] but we proved that the concurrency between returns of results does not lead to non-determinism and π-calculus linearity conditions are static(typing). Determinism can be proved in configurations where a general static property would not be sufficient to conclude. For example, a synchronization mechanism could dynamically enforce an order of request sending.

References

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

[2] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 32–59. Springer-Verlag, New York, NY, 1997.

[3] Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of java programs. In S. F. Smith and C. L. Talcott, editors, *4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 141–161. Kluwer Academic Publishers, 2000.

[4] Luca Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 286–297, New York, NY, 1995.

[5] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[6] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998. Proactive available at www.inria.fr/oasis/proactive.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando (Florida, USA), June 1994. ACM. SIGPLAN Notices, 29(6).

[8] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC)*, Rhodes Island, August 2001.

[9] C. Fournet, G. Gonthier, JJ. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag, Berlin.

[10] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[11] Gordon, Hankin, and Lassen. Compilation and equivalence of imperative objects. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 17, 1997.

[12] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*. Elsevier ENTCS, 1998.

[13] Andrew D. Gordon, Paul D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings FST+TCS'97*, LNCS. Springer-Verlag, December 1997.

[14] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 386–395, St. Petersburg Beach, Florida, 21–24 January 1996.

[15] Alan Jeffrey. A distributed object calculus. In *ACM SIGPLAN Workshop Foundations of Object Oriented Languages*, 2000.

[16] Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, 1992. UMCS-92-12-1.

[17] Cliff B. Jones. Process-algebraic foundations for an object-based design notation. Technical report, University of Manchester, 1993. UMCS-93-10-1.

[18] Cliff B. Jones and S.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, and Hans-Jörg Schek, editors, *Object-Orientation with Parallelism and Persistence*, chapter 1, pages 1–22. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.

[19] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[20] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Information Processing 77: Proc. IFIP Congress*, pages 993–998. North-Holland, 1977.

[21] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of POPL '96*, pages 358–371. ACM, January 1996.

[22] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[23] Xinxin Liu and David Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *LNCS*, pages 217–231. Springer, 1995.

[24] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[25] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[26] Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. Aliasing models for mobile objects. *Information and Computation*, 175(1):3–33, 2002.

[27] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 1–20. Springer-Verlag, 1992.

[28] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Lecture Notes in Computer Science*, 915:651–??, 1995.

[29] 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.

[30] Brian Cantwell Smith. Reflection and semantics in lisp. In *POPL'84 : conference record of the Annual ACM symposium on Principles of Programming Languages*, pages 23–35, 1984.

[31] Guy L. Steele, Jr. Making asynchronous parallelism safe for the world. In ACM, editor, *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA*, pages 218–231, New York, NY, USA, 1990. ACM Press.

[32] Martin Steffen and Uwe Nestmann. Typing confluence. Interner Bericht IMMD7-xx/95, Informatik VII, Universität Erlangen-Nürnberg, 1995.

A Notations

Concepts

Active object	Root object of an activity	10
Activity	A process made of a single active object and a set of passive objects	10
Wait-by-necessity	Blocking of execution upon a strict operation on a future: $\alpha[\mathcal{R}[\iota \ldots], \sigma_\alpha \ldots] \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma \to \beta})$	11
Service Method	Method started upon activation: m_j in $Active(a, m_j)$	12
Request	Asynchronous remote method call	10
Future	Represents the result of a request before the response is sent back	12
Future value	Value associated to a future $f_i^{\alpha \to \beta}$ $copy(\iota, \sigma)$ where $\{f_i^{\alpha \to \beta} \mapsto \iota\} \in F_\alpha$	12
Computed future	A future which has a value associated: $f_i^{\alpha \to \beta}$ where $f_i^{\alpha \to \beta} \in dom(F_\beta)$	21
Not (yet) updated future	Reference to a computed future	23
Partial future value	Future value containing references to futures	12
Closed term	Term without free variable ($fv(a) = \emptyset$)	8
Source term	Closed term without location ($fv(a) = \emptyset \wedge locs(a) = \emptyset$)	8
Reduced object	Object with all fields reduced (to a location): $o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]$	8

Syntax: ASP source terms

$[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]$	Object definition	8
$a.l_i$	Field access	8
$a.l_i := b$	Field update	8
$a.m_j(b)$	Method call	8
$clone(a)$	Superficial copy	8
$Active(a, m_j)$	Object activation	10
$Serve(M)$	Request service	10
M	list of method labels	10

ASP intermediate terms and semantics structures

ι	Location	8
(a, σ)	Sequential configuration	9
$a \Uparrow f, b$	a with continuation b, f is the future associated with the configuration	10

α, β	Activity: $\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha]$ [current term, store, active object, future values, pending requests, current future]	10
P, Q	Configuration	16
$AO(\alpha)$	Generalized reference to the activity α	12
$f_i^{\alpha\to\beta}$	Future identifier	16
$fut(f_i^{\alpha\to\beta})$	Generalized reference to the future $f_i^{\alpha\to\beta}$	17
$r = [m_j; \iota; f_i^{\alpha\to\beta}]$	Request: Remote method call	10
$R_\alpha = \{[m_j; \iota; f_i^{\alpha\to\beta}]\}$	Pending requests: a queue of requests	17

General Notations

$\{a \mapsto b\}$	Association/finite mapping	8
$\theta ::= \{\!\{b \leftarrow c\}\!\}$	Substitution	8
$\xrightarrow{*}$	Transitive closure of any reduction $\rightarrow$	23
$\bigoplus$	Disjoint union	22
$L\big\vert_M$	Restriction of (RSL) list L on labels belonging to M	23
L_n	n^{th} element of the list L	22
$\sqcup$	Least upper bound	22
$\exists^1$	There is at most one	26

Stores

σ	Store: finite map from locations to objects (reduced or generalized reference) $\sigma ::= \{\iota_i \mapsto o_i\}$	8
$dom(\sigma)$	set of locations defined by σ	8
$\sigma :: \sigma'$	Append of disjoint stores	9
$\sigma + \sigma'$	Updates the values defined in σ' by those defined in σ $(\sigma + \sigma')(\iota) = \begin{cases} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{cases}$	9
$Merge(\iota, \sigma, \sigma')$	Store Merge: mergesindependently σ and σ' except for ι which is taken from σ'	9
$copy(\iota, \sigma)$	Deep copy of $\sigma(\iota)$	18
$Copy\&Merge(\sigma, \iota\ ;\ \sigma', \iota')$	Appends in $\sigma'(\iota')$ a deep copy of $\sigma(\iota)$	18

Semantics

$\mathcal{R}$	Reduction context	9,17
$\mathcal{R}[a]$	Substitution inside a reduction context	9
$\rightarrow_S$	Sequential reduction	10

$\longrightarrow$	Parallel reduction	19
$\xrightarrow{T}$	Parallel reduction where T is the applied rule	23
$\Longrightarrow$	Parallel Reduction with future updates: Parallel reduction preceded by some reply rules	24
$\overset{T}{\Longrightarrow}$	Parallel Reduction with future updates where T is the applied rule: $\xrightarrow{\text{REPLY}^*}\xrightarrow{T}$ if $T \neq$ REPLY and $\xrightarrow{\text{REPLY}^*}$ if $T =$ REPLY	24
$FL(\alpha)$	Futures list	21
RSL_α	Request Sender List of α: $(RSL_\alpha)_n = \beta^f$ if $f_n^{\beta\to\alpha} \in FL(\alpha)$	22
$\trianglelefteq$	RSL comparison (prefix order on activities)	22
$\mathcal{M}_{\alpha_P}$	Static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P: $P \xrightarrow{*} Q \wedge Q = \alpha[\mathcal{R}[Serve(M)],\ldots] \parallel \ldots \Rightarrow M \in \mathcal{M}_{\alpha_P}$	23
$ActiveRefs(\alpha)$	Set of active objects referenced by α: $\{\beta \mid \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = AO(\beta)\}$	20
$FutureRefs(\alpha)$	Set of futures referenced by α: $\{f_i^{\beta\to\gamma} \mid \exists \iota \in dom(\sigma_\alpha),\ \sigma_\alpha(\iota) = fut(f_i^{\beta\to\gamma})\}$	21

Equivalences

$\equiv$	equality modulo renaming (alpha-conversion) of locations	8
$\equiv_F$	Equivalence modulo future replies/updates	23

Properties

$\vdash P$ OK	Well formed configuration	21
$RSL_\alpha \bowtie RSL_\beta$	RSL compatibility	22
$P \bowtie Q$	Configuration compatibility	23
$P_1 \curlyvee P_2$	Configuration confluence: $\exists R_1, R_2, P_1 \xrightarrow{*} R_1 \wedge P_2 \xrightarrow{*} R_2 \wedge R_1 \equiv_F R_2$	24
$DON(P)$	Deterministic Object Network	25

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399