UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS   UFR Sciences

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Spécialité : INFORMATIQUE

présentée et soutenue par

Ludovic HENRIO

Équipe d'accueil : OASIS – INRIA Sophia-Antipolis - CNRS - I3S - UNSA

Asynchronous Object Calculus: Confluence and Determinacy

Calcul d'Objet Asynchrone : Confluence et Déterminisme

Thèse dirigée par Denis CAROMEL

Présentée publiquement le vendredi 28 novembre 2003 à 10h30
devant le jury composé de :

Rapporteurs:	Luca CARDELLI	Microsoft research, Cambridge
	Ugo MONTANARI	Universita di Pisa
	Elie NAJM	ENST, Paris
Autres membres du jury:	Gérard BOUDOL	INRIA Sophia Antipolis
	Gilles KAHN	INRIA, Direction générale
Directeur de thèse:	Denis CAROMEL	Université Nice Sophia-Antipolis, IUF
Codirecteur de thèse:	Bernard SERPETTE	INRIA Sophia Antipolis

Table of Contents

Definitions and properties

List of Figures

Remerciements

First, I would like to thank my referees Luca Cardelli, Ugo Montanari and Elie Najm and more generally all the memebers of my jury for their comments. The following of this section will be in french.

Bien sûr je voudrais remercier mon directeur de thèse, Denis Caromel, ainsi que mon co-directeur de thèse, Bernard Serpette, pour toutes les discussions intéressantes, pour leur commentaires, pour leur soutien, mais aussi pour m'avoir parfois permis de mener cette thèse comme je le souhaitais.

Je remercie aussi toute l'équipe OASIS car j'ai pu effectuer cette thèse dans un cadre très agréable. Entre autres, je remercie Christian et Romain pour leur commmentaires et leur patiente quant à la préparation de ma soutenance de thèse mais aussi tous les membres de l'équipe pour leur différentes contributions à cette thèse et plus généralement à toutes ces choses sans lesquelles ce travail n'aurait pas pu être réalisé.

Je tiens aussi a remercier ma famille et mes amis (je préfère ne pas me lancer dans une liste sans fin ici, je ne souhaite pas avoir à me demander comment ordonner tout ceci, et surtout je ne veux pas courir le risque d'oublier quelqu'un). Donc merci a tous ceux qui m'ont soutenu, écouté, aussi bien à propos de mon (difficile) parcours de thèse que concernant tout le reste. Merci à ceux qui ont accepté de relire certaines partie de mes articles sans tout comprendre et parfois, aussi étonnant que cela puisse paraitre, en comprenant! Merci a tous ceux qui m'ont accompagnés lors de sorties sportives (ou non), et surtout a ceux qui sont quand même revenus après la première sortie...

Part I

French Extended Abstract

This part presents a translation in french of the introduction, the conclusion and a informal description of ASP principles described in this thesis.

Cette partie traduit en français l'introduction, la conclusion et décrit informellement les principes d'ASP présentés dans la suite de cette thèse.

Chapter 1

Introduction

Les systèmes à objets distribués deviennent omniprésents. Des *objets communiquants* interagissent à différents niveaux et dans des environnements très variés. Ces objets envoient des messages, appellent des méthodes en fonction de leurs interfaces et reçoivent des requêtes ainsi que des réponses. Ces objets ont un état interne et interagissent par des appels de méthodes. Les appels de méthodes par *RMI* (*Remote Method Invocation*) ont pris de plus en plus d'ampleur sur les plateformes industrielles; ceci est la conséquence de 15 ans de recherche académique. De plus, il semble que cette solution soit à la fois efficace et pratique.

Les appels de méthodes synchrones ne sont pas adaptés aux systèmes actuels pouvant être constitués de milliers de machines communiquant les unes avec les autres. Ainsi, dans un système distribué à grande échelle destiné à faire des calculs, les appels de méthodes devraient systématiquement être asynchrones. Bien sûr, dans un cadre où la plupart du temps est consacré aux communications, les communications asynchrones ne sont plus vraiment nécessaires. Nous sommes habitués à attendre le téléchargement d'une page Web de façon synchrone. De tels appels de méthodes asynchrones sont aussi appelés *requêtes*. Le terme *requête* semble assez adapté car, dans le cas particulier des communications client-serveur, un tel appel de méthode peut être considéré comme une requête adressée au serveur. Bien sûr, dans le cadre général, chaque objet agit au cours de son existence à la fois en temps que client et en temps que serveur.

Afin d'avoir un mécanisme d'appels de méthodes structuré même lorsque ces appels sont asynchrones, nous avons besoin d'un mécanisme capable de retourner un résultat à l'objet ayant envoyé la requête. Ce résultat sera représenté par un *futur*: un futur représente le résultat encore attendu d'un appel de méthode donné. Les futurs s'avèrent être une abstraction très efficace pour des systèmes distribués à grande échelle, garantissant à la fois un faible interdépendance des processus et une programmation très structurée.

Contrairement à la plupart des calculs d'objets, nous ne souhaitons pas avoir recours spécifiquement à une notion de *thread* ou de *processus*. Dans tout le discours précédent, nous avons supposé que les calculs étaient effectués par des objets, et donc, nous souhaitons *unifier la notion de thread et d'objet*. Certains objets auront ainsi leur propre processus et seront appelés *objets actifs*. Les objets passifs (non actifs) appartiennent tous à un unique objet actif. On crée de cette façon une *partition* de l'ensemble des objets en différentes *activités*. Chaque activité possède son propre *thread* et contient un unique objet actif capable de traiter des requêtes.

Comme tous les systèmes parallèles, nous avons besoin d'un mécanisme de synchronisation. Nous avons choisi pour cela un mécanisme très naturel à la fois du point de vue du modèle et du programmeur. En effet, une synchronisation naturelle est *automatiquement* effectuée lorsqu'on essaye d'accéder au résultat d'une requête dont la valeur n'a pas encore été retournée. Cette synchronisation dirigée par le flot de données n'a lieu que quand la valeur réelle d'un objet représenté par un futur est nécessaire; Ce mécanisme est appelé *attente par nécessité* [Car93]. Du point de vue théorique, ce mécanisme est naturel car il est impossible d'évaluer une construction nécessitant la valeur réelle d'un objet à partir du futur seul. Du point de vue du programmeur, aucune primitive de synchronisation spécifique n'est nécessaire et seules les synchronisations nécessaires sont effectuées de manière totalement transparente. De plus, l'aspect "flot de données" permet à certaines synchronisations difficiles à prévoir d'avoir lieu automatiquement, en particulier entre deux processus qui ne sont pas directement reliés.

Même si la notion d'*attente par nécessité* a été définie dans [Car93], elle est fortement reliée à l'*évaluation paresseuse* [HM76] (aussi appelée évaluation *à la demande*) et à la notion de futurs dans les langages fonctionnels comme *Multilisp* [Hal85]. En effet, les futurs de Multilisp peuvent être vus comme une certaine adaptation de l'évaluation paresseuse à la programmation parallèle. Bien sûr, les futurs sont peu adaptés à une évaluation des fonctions non-strictes mais ils permettent une évaluation parallèle. Autrement dit, les futurs permettent d'avoir une évaluation *au plus tôt* contrairement à l'évaluation paresseuse. Le fait qu'"une opération qui a besoin de savoir la valeur d'un futur *indéterminé* est *suspendue*" est commun à tous ces mécanismes. Dans notre cas, les futurs permettent d'effectuer des appels asynchrones de façon à la fois puissante et élégante.

La plupart des modèles théoriques considèrent que les communications sont composés de simples messages circulant sur des canaux. Par exemple, le π-calcul et les calculs similaires sont considérés comme bien adaptés à la modélisation des objets, mais aucune notion de méthode ou d'édition de lien dynamique n'est présente dans le modèle. Au contraire, Abadi et Cardelli ont proposé un calcul séquentiel (le ς-calcul [AC96]) beaucoup plus représentatif de la structure des objets et de leur typage. De nombreux langages et calculs concurrents ont été dérivés de celui-ci mais aucun d'eux ne satisfait nos objectifs : principalement l'unification des processus et des objets ainsi que des appels de méthodes et des communications.

Pour résumer, nous souhaitons modéliser les aspects suivants dans un calcul aussi simple et structuré que possible:

- **Orientation objets** : à cause de la structure des objets et de la théorie qui leur est associée.

- **Communications par appels de méthodes** : parce que les appels de méthodes permettent de structurer les communications et sont représentatifs de la programmation orientée-objets.

- **Communications asynchrones** : pour limiter les effets de la latence dans les réseaux et découpler les processus.
- **Futurs** : afin de maintenir une communication structurée (avec des réponses) entre des processus asynchrones.
- **Synchronisation dirigée par le flot de données** : parce que ce mécanisme de synchronisation est naturel, pratique et bien adapté aux futurs.
- **Chaque objet peut devenir un processus** : où, à chaque objet actif correspond un unique processus.

  En fait, habituellement dans les langages orientés-objets un processus est déjà un objet particulier. En Java, la classe `Thread` hérite de la classe `Object` mais un objet de type Thread n'a aucun rôle fonctionnel car il n'a pas de méthode spécifique. Mais ce qui rend notre calcul innovant, c'est que *chaque* objet peut aussi devenir un processus.
- **Politique de service** : afin de pouvoir spécifier l'ordre dans lequel un objet actif doit traiter les requêtes qui lui sont destinées.

A partir d'une théorie classique dans le domaine des objets, le ς-calcul [AC96], une extension syntaxique assez simple est proposée pour prendre en compte la distribution. Deux primitives simples sont proposées : *Active* et *Serve*. La première transforme un objet en une activité indépendante et potentiellement distante; la deuxième permet à un objet actif d'exécuter (servir) une requête en attente. Une fois activé, un objet devient une entité accessible à distance avec son propre thread : un *objet actif*. En accord avec le raisonnement ci-dessus, nous avons choisi de rendre asynchrones tous les appels de méthodes sur des objets actifs. En ce qui concerne la synchronisation, une *attente par nécessité* interrompt automatiquement l'exécution courante en cas d'opération stricte (comme l'accès à un champ) sur un résultat pas encore disponible (un futur). L'asynchronisme et le découplage des processus est encore augmenté par le fait que les futurs sont des entités de première classe qui peuvent être transmis entre objets (actifs) comme paramètres de méthodes et renvoyés comme résultats.

Le calcul que nous proposons s'intitule *ASP* : *Asynchronous Sequential Processes*, reflétant ainsi une propriété importante : la séquentialité des objets actifs. Les processus en ASP sont formés d'un ensemble d'objets sous le contrôle exclusif d'un objet racine. La théorie que nous proposons permet d'exprimer une condition suffisante pour la confluence des programmes, évitant ainsi au programmeur de considérer un très grand nombre d'entrelacements entre toutes les instructions et les communications. De plus, une propriété de déterminisme identifient certains systèmes comme déterministes. Cette propriété garantit que, quel que soit l'ordre des communications, quel que soit l'ordre de mise à jour des futurs, même en présence de cycles, ces systèmes se comportent de façon déterministe. A part les Process Networks [Kah74, KM77, WWV00], peu de calculs ou langages parallèles assurent des propriétés de déterminisme, et encore moins dans le cas d'objets distribués ayant un état propre et interagissant par des appels de méthodes asynchrones.

Un des objectifs du modèle proposé ici est d'avoir un aspect *pratique*. Ainsi, une API Java open source, ProActive [Pro, CKV98], implémente la théorie proposée dans cette thèse en utilisant une stratégie destinée à dissimuler en partie la latence dans le contexte des réseaux à grande échelle.

Les principales contributions de cette thèse sont :

- La définition formelle d'un calcul d'objets impératif et asynchrone (ASP) avec des futurs et une synchronisation "data-driven".
- Une vision de la programmation parallèle comme une extension simple de la programmation orientée-objet séquentielle; principalement grâce à l'utilisation de l'attente par nécessité, à la transmission transparente de futurs entre processus et leur mise à jour à n'importe quel moment.
- Des conditions suffisantes pour assurer un comportement déterministe des programmes dans ce contexte de programmation très asynchrone.
- Un rapport fort entre l'aspect pratique et théorique, à la fois à travers certaines considérations pratiques évoquées dans cette thèse, et de par le fait que le modèle ASP est implémenté sous la forme d'une librairie Java.

Une présentation courte du calcul ASP et de certaines propriétés et preuves peuvent être trouvée dans les articles suivants : [CHS03, CHS04].

Chapter 2

Description Informelle d'ASP et de ses Propriétés

Dans ce chapitre, nous introduisons un calcul nommé *ASP : Asynchronous Sequential Processes*. ASP modélise un calcul orienté-objet avec des communications asynchrones, des futurs et une exécution séquentielle à l'intérieur de chaque processus en parallèle.

A partir d'un calcul d'objet séquentiel et classique (proche du **imp**ς-calcul [AC96]), ASP est défini par l'ajout de deux constructeurs *Active* et *Serve*. *Active* rend actif un objet standard, une fois activé, cet objet exécutera des opérations en parallèle et servira des requêtes dans l'ordre spécifié par l'opérateur *Serve*. Les appels de méthodes sur les objets actifs sont asynchrones. Le résultat d'un appels asynchrones est représenté par un *futur* jusqu'à ce que le résultat correspondant soit renvoyé. La synchronisation automatique des processus est due à l'attente par nécessité [Car93] : une attente automatique interrompt toute tentative d'opération stricte (comme par exemple un appel de méthode) sur un futur.

Dans le Chapitre 6 nous verrons que de nombreux calculs et langages ont déjà été définis pour modéliser la concurrence. Le nouveau calcul introduit dans cette thèse nous est apparu comme nécessaire pour avoir un modèle structuré de calcul objet impératif avec communications asynchrones et futurs. En effet, aucun des calculs que nous avons étudiés ne présente toutes ces caractéristiques. De plus, ASP reste un calcul avec une syntaxe structurée. Ainsi, la traduction des programmes ASP dans d'autres calculs est possible, mais dans ce cas la structure des programmes ASP serait perdue et la preuve des propriétés exposées dans cette thèse serait probablement très difficile voire impossible. Le Chapitre 15 fournit une comparaison d'ASP avec de nombreux calculs concurrents existants.

2.1 Principes

ASP est basé sur la notion d'activités. Chaque objet ASP est soit *actif* soit *passif*. La racine de chaque activité est un objet actif. Les activités exécutent des instructions de manière parallèle et concurrente et n'interagissent que par des appels de méthodes. Les appels de méthodes adressés à des objets actifs sont tous *asynchrones*. C'est l'attente par nécessité sur le résultat d'un qui permet de synchroniser les différentes activités (synchronisation data-driven).

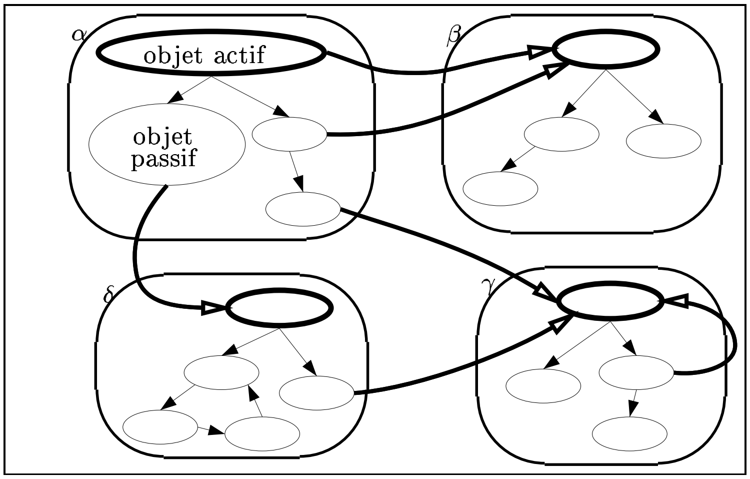


Figure 2.1: Un exemple simple de topologie des objets et des activités

De façon simplifiée, une *activité* est un processus unique (thread d'exécution) ainsi qu'un ensemble d'objets mis dans un *store*. Parmi ces objets, un seul est actif, et toutes les requêtes (appels de méthode) envoyées à l'activité sont en fait adressées à cet objet. Une activité contient en plus les requêtes en attente (qui ont été reçues et seront servies plus tard), et les réponses des requêtes traitées (valeurs des résultats). Les objets *passifs* (c'est à dire non-actifs) ne sont référencés que par des objets internes à l'activité mais peuvent référencer des objets actifs appartenant à d'autres activités. En d'autres termes, les activités ne partagent pas de mémoire et les seules références généralisées sont les objets actifs et les futurs (décrits ci-dessous). La Figure 2.1 donne un exemple de topologie d'objets dans une configuration formée de quatre activités. Les objets sont représentés par des ellipses, les références par des flèches et les objets actifs par des ellipses en gras. Sur cette figure, l'absence de partage de mémoire est représentée par le fait que les seules flèches d'une activité à une autre sont des flèches en gras pointant vers des objets représentés par des ellipses en gras (référence vers un objet actif).

Les appels de méthodes asynchrones respectent les principes suivants:

- Quand un objet envoie une requête à une activité (appel de méthode sur un objet actif), celle-ci est stockée dans une queue et un futur lui est associé. Un *futur* représente le résultat d'une méthode qui n'a pas encore été retourné. De telles requêtes sont appelées *requêtes en attente* (*pending requests*).

- Plus tard, ces requêtes seront *servies* (c'est à dire prises dans la queue des requêtes pour être évaluées), elles deviennent ainsi des *requêtes courantes* (*current requests*). Une fois que ces requêtes sont traitées, une valeur est associée à leur résultat, et la correspondance entre le futur associé à cette requête et la valeur du résultat calculé est stockée dans une *liste de valeurs de futurs* (*future values list*).

- Encore plus tard, les références distantes au futur pourront être mises à jour (*future update*), c'est à dire que la référence au futur sera remplacée par la valeur du résultat associé.

L'activation d'un objet ($Active(a, m)$) crée une nouvelle activité ayant pour objet actif une copie (profonde) de l'objet a. Après l'activation d'un objet, tous les appels adressés à celui-ci sont asynchrones. C'est à dire que l'exécution du programme continue après l'appel jusqu'à ce qu'une opération stricte sur le résultat d'un appel asynchrone soit rencontré avant que ce résultat ne soit mis à jour. Ces états bloquants sont appelés *attente par nécessité*.

$Serve(M)$ effectue un service bloquant des requêtes en attente. M spécifie la liste des méthodes (identifiées par leur nom) devant être servies. La première méthode qui convient est servie.

Contrairement à la plupart des calculs basés sur le ς-calcul, en ASP, les requêtes sont exécutées par le processus associé à la destination de la requête et non par le processus qui envoie la requête.

2.2 Syntaxe

2.2.1 Termes Sources

La syntaxe d'ASP est la suivante, elle est composée d'un calcul d'objets séquentiels "classique" à la Abadi-Cardelli auquel on ajoute deux opérateurs propres au calcul parallèle. Ces deux derniers opérateurs permettent respectivement de créer un objet actif et de servir une nouvelle requête. On notera l_i les champs des objets et m_j les noms de méthode. ς est un lieur pour les paramètres de méthodes qui sont au nombre de deux, le premier représente l'objet appelé (self), le deuxième est un paramètre transmis à la méthode. L'existence de ces deux paramètre est justifiée par la sémantique attachée au passage de paramètre entre objets actifs. La mise à jour de méthode ne nous a pas semblé nécessaire mais elle peut être simulée dans notre calcul séquentiel.

$a, b \in L ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$	définition d'un objet,
$\mid a.l_i$	accès à un champ,
$\mid a.l_i := b$	mise à jour d'un champ,
$\mid a.m_j(b)$	appel de méthode,
$\mid clone(a)$	copie superficielle,
$\mid Active(a, m_j)$	active l'objet a, copie profonde + création de l'activité m_j doit définir l'activité de l'objet ou $\emptyset$ pour un service FIFO
$\mid Serve(m_j^{j\in 1..k})$	primitive de service : Sert la plus ancienne requête portant sur une méthode de la liste $m_j^{j\in 1..k}$.

2.2.2 Structures Dynamiques

Une configuration parallèle est formée d'un ensemble d'activités, chaque activité contient plusieurs champs qui seront définis informellement dans la suite de ce chapitre. Un définition formelle de la sémantique d'ASP est faite dans le Chapitre 10.

$$P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\ldots] \| \ldots$$

Un *store* est une fonction partielle associant un objet à une location.

$$\sigma ::= \{\iota_i \mapsto o_i\}$$

Où o_i est un objet réduit ou une référence globale.

$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1...n}_{j\in 1..m}$	Objet réduit
$\mid AO(\alpha)$	Référence à un objet actif
$\mid fut(f_i^{\alpha\to\beta})$	Référence vers un futur

$[l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1...n}_{j\in 1..m}$ est un objet où tous les champs sont réduits à une référence à l'intérieur du store. $fut(f_i^{\alpha\to\beta})$ référence un futur $f_i^{\alpha\to\beta}$ correspondant à une requête de l'activité α vers l'activité β. $AO(\alpha)$ référence l'objet actif de l'activité α.

2.3 Sémantique Informelle

Dans chaque activité α, un *terme courant* a_α représente le calcul en cours. Chaque activité a son propre *store* σ_α. Le *store* de l'activité α contient un objet actif et plusieurs objets passifs. Une activité contient aussi une liste de *requêtes en attente* (*queue des requêtes*) qui contient les appels de méthodes en attente sur l'objet actif ; et une *liste de futurs* qui contient le résultat des requêtes terminées.

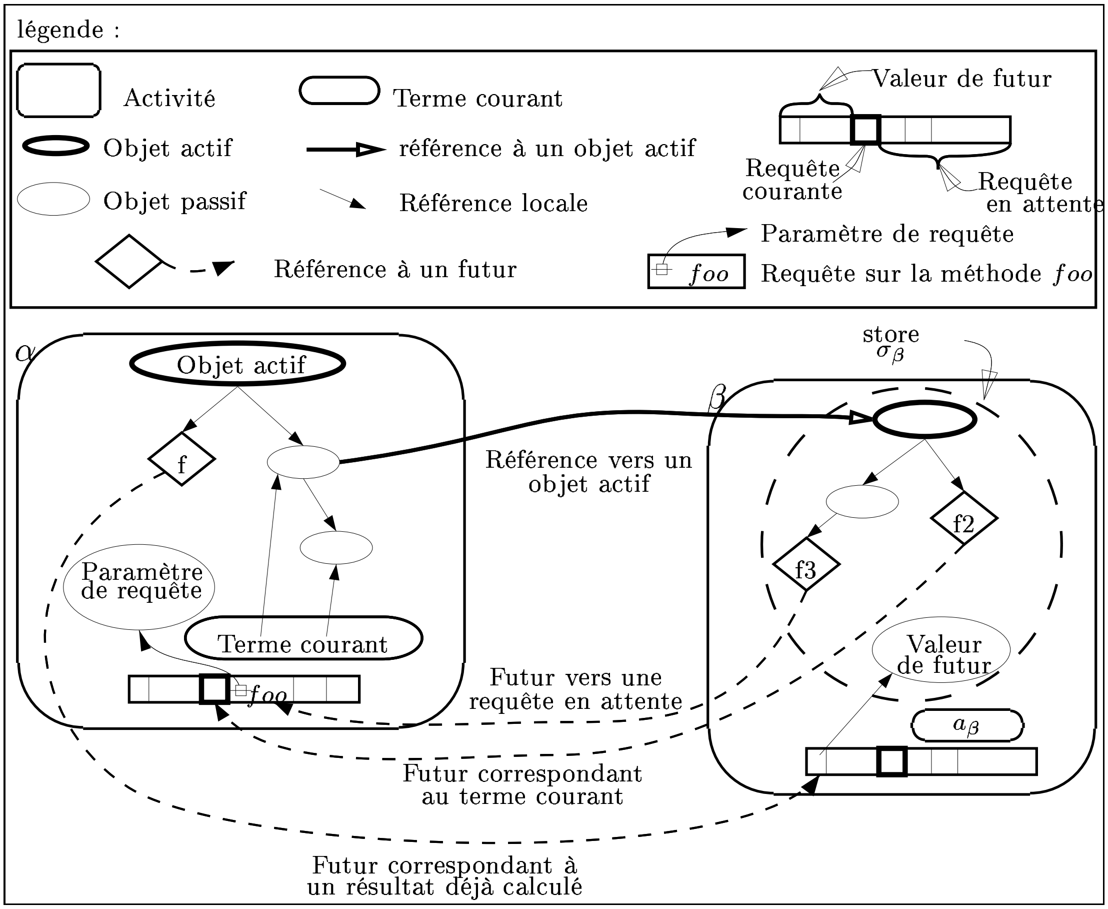


Figure 2.2: Exemple de configuration parallèle

La Figure 2.2 représente une configuration formée de deux activités. Elle contient trois références à des futurs (un calculé, un en cours et un correspondant à une requête en attente). Les objets actifs sont représentés par des ellipses en gras; les références vers des futurs sont des losanges; les valeurs des futurs, le futur courant et les requêtes en attente sont rassemblées dans les rectangles en bas des activités : les futurs calculés sont à gauche, le futur courant est représenté par un rectangle en gras et les requêtes en attente sont à droite. Les continuations n'apparaissent pas dans les schémas.

2.3.1 Les Activités

L'opérateur *Active* (*Active*(a, m_j)) crée une nouvelle activité α ayant à sa racine une copie de l'objet a. Une copie profonde est effectuée[1], c'est à dire que tous les objets référencés directement ou indirectement par a seront recopiés dans la nouvelle activité excepté les objets futurs et actifs pour lesquels seule la référence sera recopiée. Le second argument de l'opérateur *Active* est le nom de la méthode qui sera appelée dés

[1] Cela évite d'avoir des références distantes à des objets passifs.

que le nouvel objet actif sera créé. Cette méthode est appelée *méthode de service* car elle devrait spécifier l'ordre dans lequel l'activité doit servir les requêtes. Si aucune méthode de service n'est spécifiée, un service FIFO sera effectué. C'est à dire que les requêtes seront servies dans l'ordre dans lequel elles arrivent dans l'activité.

Dans la Figure 8.2 et dans le cas d'un service FIFO, la requête courante (rectangle en gras) progresse de la gauche vers la droite. Lorsque la méthode de service termine, plus aucune requête n'est traitée (il n'y a plus d'activité). Les références distantes à l'objet actif de l'activité α seront notées $AO(\alpha)$. $AO(\alpha)$ peut être considéré comme un *proxy* vers l'objet actif de l'activité α.

2.3.2 Les Requêtes

Les communications entre activités se font par des appels de méthodes sur des objets actifs. Un appel de méthode sur un objet actif ($Active(o, \emptyset).foo()$) consiste, de façon atomique, à ajouter une entrée dans la queue des requêtes de l'appelé et à associer un futur à la réponse. D'un point de vue pratique, cette atomicité est garantie par un mécanisme de *rendez-vous* (l'émetteur de la requête attend un accusé de réception avant de continuer son exécution). Les arguments d'une requête ainsi que les valeurs des futurs sont transmis entre activités par copie profonde[1]. Les objets actifs sont transmis par référence.

La Figure 2.3 montre un exemple d'appel de requête depuis l'activité α vers l'activité β. On remarquera qu'un futur a été créé, que le paramètre a été passé par copie profonde et que la nouvelle requête a été mise à la fin de la queue des requêtes.

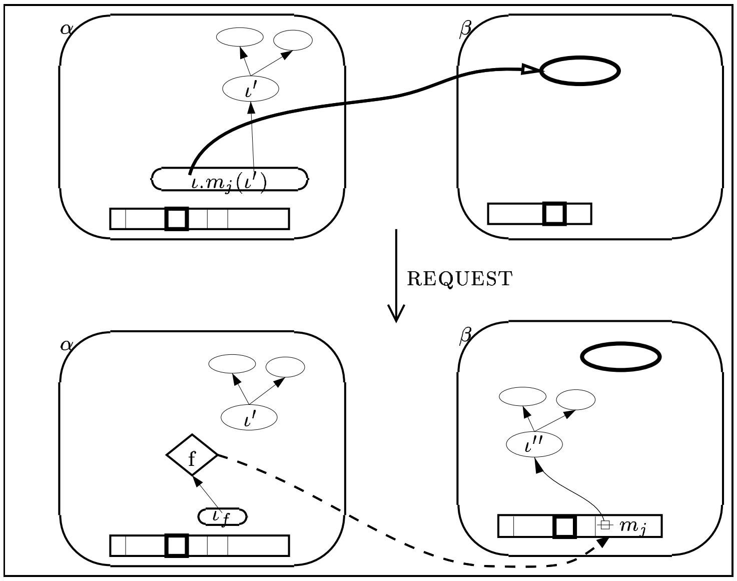


Figure 2.3: Envoi de requête

2.3.3 Les Futurs

Une opération portant sur un objet est dite *stricte* si elle a besoin de la valeur réelle de l'objet, c'est à dire si elle a besoin d'accéder au contenu de l'objet : les seules opérations strictes sont les accès aux champs, les appels de méthodes, les mises à jour de champs et la copie (clone). Par exemple, la transmission d'objet comme paramètre de méthode ou le stockage dans le champ d'un autre objet ne sont pas des opérations strictes.

Un *futur* est une référence généralisée qui peut être manipulée classiquement tant que l'on n'effectue pas d'opération stricte sur l'objet qu'il représente. Dans la Figure 8.2, les futurs f_2 et f_3 pointent vers des résultats de requêtes qui n'ont pas encore été calculés, alors que f est un futur dont la valeur associée a été calculée par l'activité β. C'est le résultat d'une requête adressée à β.

Une *attente par nécessité* a lieu automatiquement lorsqu'une opération stricte porte sur un futur qui n'a pas encore été mis à jour. Cette attente par nécessité ne peut être arrêtée qu'en *mettant à jour le futur* c'est à dire en remplaçant la référence vers le futur par une copie[1] de la valeur calculée pour celui-ci.

La Figure 2.4 montre la mise à jour d'un futur dans l'activité α.

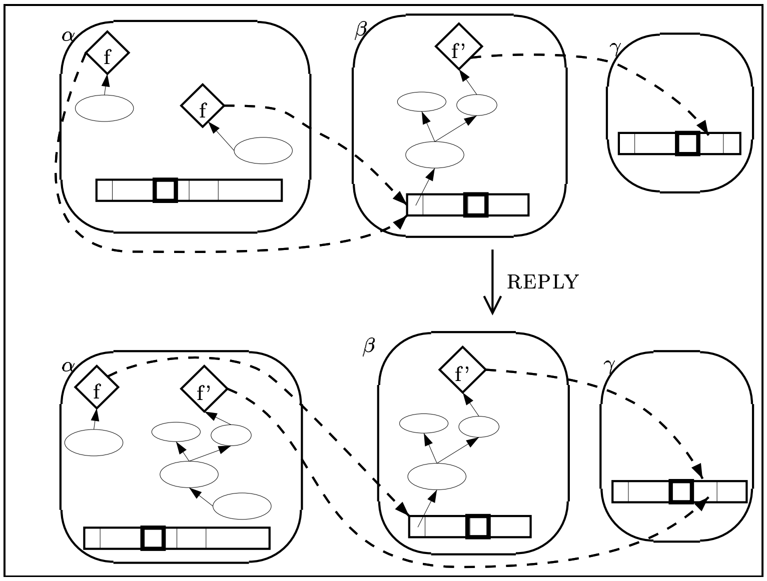


Figure 2.4: Mise à jour de futur

2.3.4 Le Service des Requêtes

La primitive *Serve* peut apparaître à n'importe quel endroit du code source. Son exécution interrompt l'activité jusqu'à ce qu'une requête correspondant à ses arguments soit trouvée dans la queue des requêtes (une requête portant sur l'une des méthodes dont le nom est spécifié comme paramètre de la primitive *Serve*). Pour des raisons de spécifications sémantiques, nous introduisons un opérateur supplémentaire ⇑ permettant de stocker la continuation de la requête que nous étions en train de servir avant de choisir d'en servir une autre. Il est important de noter que, avec un tel mécanisme, plusieurs requêtes peuvent être en cours de traitement à chaque instant sauf si les instructions *Serve* ne sont effectués que par l'activité principale (aucun *Serve* pendant qu'une autre requête est en cours de traitement).

Quand l'exécution d'une requête est terminée, la valeur calculée est associée au futur correspondant. Puis, l'exécution continue en restituant la continuation qui avait été stockée au moment du service de cette requête. Le terme qui avait provoqué le service de la requête terminée continue son exécution (il devient le terme courant). La ***liste des futurs*** associe à chaque futur calculé sa valeur, à l'intérieur de l'activité qui a effectué le calcul. On dit qu'une valeur de futur est ***partielle*** si elle fait référence (directement ou indirectement) à un futur.

Il est important de noter que l'accès à un champ ou la modification d'un champ d'un objet actif est interdite et une activité tentant d'accéder à un champ d'un objet actif est bloquée de manière irréversible.

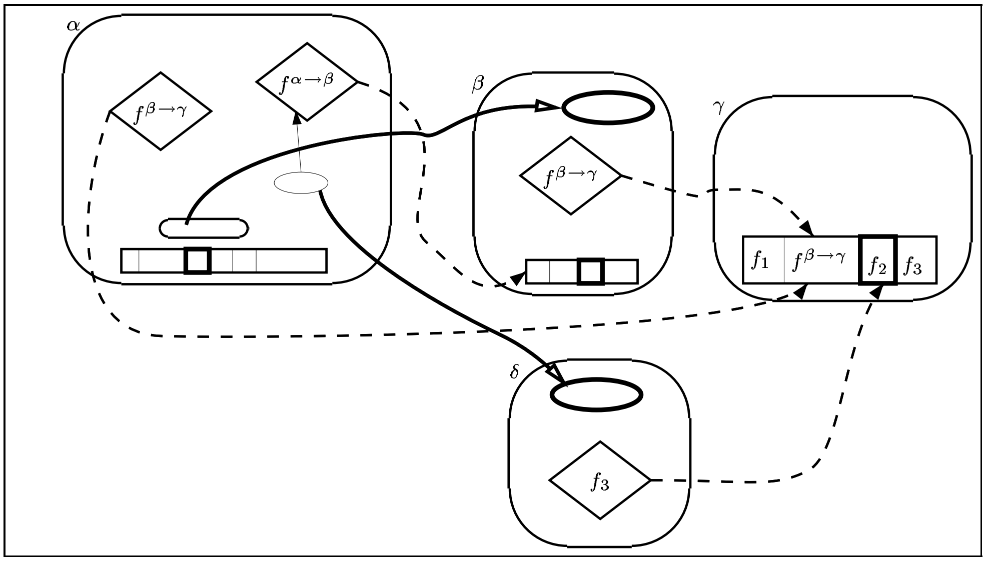


Figure 2.5: Autre exemple de configuration

2.4 Propriétés

Cette section donne un très bref aperçu informel des propriétés d'ASP.

2.4.1 Topologie

La topologie des objets en ASP suit quelques propriétés importantes. Tout d'abord, il n'y a pas de mémoire partagée entre activités ; c'est à dire que les seules références généralisées d'ASP sont les futurs et les objets actifs. Cette propriété est directement garantie par la syntaxe du calcul. De plus, tout au long de l'exécution, chacun des paramètres des requêtes et chacun des futurs est situé dans une partie isolée du *store*. Ceci permet de garantir que l'exécution courante ne va pas modifier la valeur des futurs déjà calculés ou des paramètres de requêtes encore en attente.

2.4.2 Confluence et Déterminisme

Les propriétés de confluence évitent au programmeur d'étudier l'entrelacement des instructions et des communications. De nombreux travaux assurent la confluence de calculs, langages ou programmes. Les canaux linéaires en π-calcul [NS97, KPT96], les propriétés de non-interférence dans les systèmes à mémoire partagée [Ste90], les Process Networks [Kah74] ou les techniques de Jones pour créer des programmes concurrents sont des exemples typiques. Mais aucune de ces approches ne concerne un langage d'objets concurrent et impératif avec des communications asynchrones.

La principale propriété présentée dans cette thèse montre que l'exécution d'un ensemble d'activités est déterminée de façon unique par l'ordre des activités ayant envoyé des requêtes à un destinataire donné. Les réponses asynchrones peuvent avoir lieu dans n'importe quel ordre sans aucune conséquence observable. Ce travail semble être, d'une certaine façon, fortement relié et plus général que les canaux linéarisés en π-calcul [KPT96] et les Process Networks [Kah74]. Une spécification d'un ensemble de termes se comportant de façon déterministe est donné en Section 11.8. Cet ensemble de termes peut être vu comme une généralisation des Process Networks. Enfin, une caractérisation plus simple des programmes déterministes est donnée en Section 11.9.

Chapter 3

Conclusion

Dans cette thèse, nous avons proposé un calcul modélisant les communications asynchrones dans les langages orientés objets, et montré des propriétés de confluence. De telles propriétés simplifient la programmation car elles évitent au programmeur d'avoir à étudier tous les entrelacements possibles entre messages et instructions pour comprendre le comportement d'un programme.

Nous avons présenté un calcul nouveau nommé ASP et prouvé des conditions suffisantes pour la confluence des programmes écrits dans ce langage. Notre objectif était de fournir des propriétés le plus générales possibles. A partir de ces propriétés dynamiques, des propriétés plus statiques et plus faciles à vérifier peuvent être déduites. De plus, les propriétés présentées dans cette étude, ont déjà de grandes conséquences pratiques, entre autres en ce qui concerne ProActive.

Le calcul ASP est basé sur la notion d'*activités* s'exécutant de façon *asynchrones*, traitant des *requêtes* et répondant par le biais de *futurs*. Dans chaque activité, l'exécution est séquentielle et il existe une bijection entre activités et objets actifs.

En ce qui concerne l'asynchronisme et la synchronisation, dès qu'un processus a envoyé une requête, il peut poursuivre son exécution (de manière asynchrone) tant que le résultat de la requête n'est pas nécessaire. Pendant ce temps, le résultat à venir est représenté par un futur. Les futurs sont des entités de première classe qui peuvent être transmises comme paramètres et résultats entre les objets et les activités. La synchronisation est due à un mécanisme naturel dirigé par le flot de donnés : une *attente par nécessité* est effectuée lorsqu'un objet a besoin de la valeur réelle d'un résultat encore sous forme de futur. L'étude formelle de cette attente est un des résultats majeurs de cette thèse.

ASP garantit une propriété de confluence entre des termes compatibles : deux configurations ayant des RSLs (Request Sender List) compatibles sont confluentes. La compatibilité des RSLs est basée sur un ordre préfixe entre les activités envoyant des requêtes à une même activité de destination. Par conséquent, l'exécution est déterminée de façon unique par l'ordre des activités envoyant des requêtes à une même activité de destination. Le fait que l'exécution soit insensible à l'ordre dans lequel ont lieu les retours de résultats (les mises à jour de futurs) rend les propriétés d'ASP réellement intéressantes. Par conséquent, nous avons introduit une relation d'équivalence identifiant les termes avant et après la mise à jour d'un futur. Nous avons défini un sous-calcul d'ASP intitulé *Deterministic Object Networks* (DON) qui se comporte de façon déterministe (Théorème 11.4). Nous avons aussi prouvé que chaque programme communiquant sur un arbre (Théorème 11.3 : Tree Determinacy) se comporte de façon déterministe, même dans le cas du service FIFO qui est pourtant celui ou la concurrence est maximale.

Même si ASP est un calcul nouveau, il a été fortement inspiré par de nombreux langages et calculs concurrents existant. Par exemple les objets sont modélisés comme en ς-calcul, et la propriété DON fortement liée aux Process Networks peut aussi être vue comme une adaptation des canaux linéaires et linéarisés du π-calcul.

Les propriétés d'ASP illustrent la puissance et l'adéquation des futurs aux langages impératifs. Plus précisément, c'est l'équilibre entre les communications asynchrones et la synchronisation par le biais de l'attente par nécessité qui rend ASP confluent. Bien sûr, la topologie des objets assurant que l'on ne peut accéder à une activité que par son objet actif joue aussi un rôle primordial. Le fait que la synchronisation soit liée au flot de données et que les réponses peuvent avoir lieu à tout moment permet de plus de programmer des applications parallèles et concurrentes de façon pratique. En effet, le programmeur est juste tenu d'assurer que la réponse *peut* être calculée[2] au moment où sa valeur est nécessaire pour éviter des deadlocks.

ProActive est une librairie Java qui implémente le modèle de calcul ASP. Un de nos objectifs consiste à pouvoir utiliser en ProActive les propriétés prouvées sur ASP. Par exemple, le fait que le moment où un futur est mis à jour n'est pas observable en ASP peut être utilisé en ProActive pour choisir n'importe quelle stratégie de mise à jour des futurs.

Les preuves présentées dans cette thèse peuvent être adaptées afin de prouver d'autres cas assurant la confluence dans ASP ou dans des calculs similaires. Par exemple, nous expliquons dans cette étude comment adapter les preuves au cas fortement déterministe mais faiblement ouvert où la primitive de service demande d'expliciter l'activité source de la requête à servir.

[2] dans le sens où *il existe* un ordre dans lequel on peut effectuer les calculs pour garantir que ce résultat est déjà calculé lorsqu'on en a besoin.

Part II

Context

Chapter 4

Introduction

4.1 Overview and Orientation

Distributed objects are becoming ubiquitous. *Communicating objects* interact at various levels, and in a wide range of environments. These objects send messages, call methods on each others' interfaces, and receive requests and replies. These objects are stateful and interact with each other. In most of object-oriented frameworks, these objects interact through *method calls*. *Remote method invocation* in industrial platforms, following 15 years of research in academia, has taken off, and appears to be a practical and effective solution.

Large systems, with potentially thousands of interacting entities, cannot accommodate the high coupling induced by synchronous calls. Thus, in a wide range distributed system performing computations, method calls should be *asynchronous*. Of course, this is not fruitful in the case where most of the time is spent in communications. For example, on the web, one always wait synchronously for a page. In the following, those asynchronous method calls are also named *requests*. The term *request* seems rather adequate as asynchronous method calls can represent requests sent to a server, even if, in the most general case of interactions, each object acts both as server and client.

In order to keep a well structured method invocation mechanism, we need to be able to send back results to the sender of a request. This result will be represented by a *future*: the expected result of a given asynchronous method call. Futures turn out to be a very effective abstraction for large distributed systems, preserving both low-coupling and high-structuring.

Unlike most of calculi based on object framework, we do not want to introduce a specific notion of thread or process. All the discussion above considers that computation is performed by objects, thus we want to *unify the notion of threads and objects*. Some objects will have their own process and be called *active objects*. Objects that are not active will all belong to a unique active one. This leads to a *partition of objects* into activities. An *activity* has its own thread and contains a unique active object which is able to handle requests.

All parallel frameworks need a synchronization mechanism. In our case, this synchronization comes in a very natural way both from the model and from the programmer's point of view. Indeed, a natural synchronization *automatically* occurs when one tries to access to a result that has not yet been returned: *a future*. This *data-driven synchronization* only occurs when the real value of the object represented by the future is needed; this mechanism is called *wait-by-necessity* [Car93]. From the model point of view, this principle is natural as it is impossible to evaluate an operation that needs the real value of an object only from the future representing it. From the programmer point of view, no specific synchronization primitive is needed and only necessary synchronizations are transparently performed (just in time). Most importantly, the data-driven aspect implies that non-obvious synchronizations between processes that are not directly connected can be performed automatically.

Even if the notion of *wait-by-necessity* has been introduced in [Car93] it is strongly related to *lazy evaluation* [HM76] (also called *demand-driven* evaluation) and to the notion of futures in functional languages like in *Multilisp* [Hal85]. Indeed futures in Multilisp can be considered as an adaptation of lazy evaluation to parallel computing. Of course futures are less adapted to non-strict functions than lazy evaluation but they provide a parallel evaluation. In other words futures mechanism is eager while lazy evaluation is lazy. The fact that "an operation that needs to know the value of an *undetermined* future is *suspended*" is common to all these mechanisms. In our case, futures provide an elegant and powerful means for performing asynchronous method calls.

Most of theoretical models consider the communications being composed of simple messages traveling over channels. For example, π-calculus and similar calculi are considered as fitting object paradigms but do not include methods and dynamic binding in the model. At the opposite, Abadi and Cardelli proposed a sequential calculus (the ς-calculus [AC96]) more representative of objects structure and typing. Several concurrent languages and calculi have been derived from this one, but none of them fit with our objectives: mainly unification of objects and processes and of method calls and communication.

To sum up, we want to model by a simple and structured calculus the following aspects:

- **Object-orientation**: because of the objects structure and the theory associated to them.
- **Communications through method calls**: because method calls provide structured communications and are representative of object-oriented programming.
- **Asynchronous communications**: for the sake of hiding network latency and decoupling of processes.
- **Futures**: to maintain a structured communications (with returns), between asynchronous processes.

- **Data-driven synchronization**: because it allows a natural and convenient synchronization mechanism well adapted to futures;

- **Any object can become a process**: where active objects uniquely identify processes.

  Actually, the fact that a process is also an object is classical in object-oriented languages. In Java, the *Thread* class inherits from the *Object* class but a thread object is not an applicative object as it has no specific method. But, what makes our calculus innovative is that, moreover, *each* object has the ability to become a process;

- **Service policy**: to allow the specification of the order in which requests should be served.

Starting from widely-adopted object theory, the ς-calculus [AC96], a syntactically light-weight extension is proposed to take into account distribution. Two simple primitives are proposed: *Active* and *Serve*. The former turns an object into an independent and potentially remote activity; the latter allows such an active object to execute (serve) a pending remote call. Upon activation, an object becomes a remotely accessible entity with its own thread of control: *an active object*. In accordance with the above reasoning, we have chosen to make method calls to active objects systematically asynchronous. For synchronization, a *wait-by-necessity* automatically occurs upon a strict operation (like a field access) on a communication result not yet available. A further level of asynchrony and low coupling is reached with the first-class nature of futures; they can be passed between (active) objects as method parameters and returned as results.

The proposed calculus is named *Asynchronous Sequential Processes (ASP)*, reflecting an important property: the sequentiality of active objects. Processes denote the potentially coarse grain nature of active objects. Such processes are usually formed with a set of standard objects under the exclusive control of a root object. The proposed theory allows us to express a fundamental condition for *confluence*, alleviating the programmer of the un-scalable need to consider the interleaving of all instructions and communications. Furthermore, a property ensures *determinism*, stating that whatever the order of communications, whatever the order of future updates, even in the presence of cycles, some systems converge towards a determinate global state. Apart from the close to 40-year-old Process Networks [Kah74, KM77, WWV00], few parallel calculi and languages ensure determinism, and even less in the context of stateful distributed objects interacting with asynchronous method calls.

One objective of the proposed model is to be a *practical* one. An illustration of such practicability is available under an open source Java API and environment, ProActive [Pro, CKV98], which implements the proposed theory using a strategy designed to hide latency in the setting of wide area networks.

The main contributions of this study are:

- The formal definition of an imperative and asynchronous object (ASP) calculus with futures and data-driven synchronization.

- Parallel programming as a smooth extension of sequential objects, mainly due to data-driven synchronizations (wait-by-necessity) and pervasive futures with concurrent out-of-order updates.

- The characterization of sufficient conditions for deterministic behavior in such a highly asynchronous setting.

- A strong link with the practical side both in some practical considerations inside this study and with the fact that the ASP-calculus model is implemented as a Java library.

Shorter presentations of ASP calculus and some of its properties and proofs can be found in [CHS03, CHS04].

4.2 Organization of this Thesis

Part II presents the context of this thesis. In the following, **Chapter 5** presents a brief overview of distribution and parallelism in an object-oriented framework. **Chapter 6** introduces formalisms, calculi, and distributed frameworks related to this thesis. We chose to focus on some of the most popular calculi and languages and to try to give a precise description of them.

Part III describes the ASP calculus. It starts from a sequential calculus, *à la* Abadi-Cardelli, described in **Chapter 7** with a sequential semantics very closed to the one of Gordon and Hankin. Then the syntax and an informal description of the ASP calculus is given in **Chapter 8**. And finally, some classical examples illustrate the parallel calculus in **Chapter 9**. Some of these examples are strongly inspired from process networks and a first link between those two theories can be deduced from those programs.

Part IV presents the ASP semantics and its properties. Firstly, **Chapter 10** describes ASP semantics, this chapter can be seen as a formalization of the ASP principles previously introduced in Chapter 8. From this model, **Chapter 11** gives some properties of the ASP calculus. After some properties about objects topology, a notion of compatibility and an equivalence relation between terms are presented. Informally, compatibility will be a necessary condition for confluence, and equivalence relation identifies terms modulo some futures updates. These two definitions are strongly interconnected and leads to a confluence property. Then a set of confluent term (*DON terms*) is defined. Finally, based on objects topology, a set of programs that behave deterministically is identified.

Part V provides formal details and proofs related to Chapter 11. Firstly, **Chapter 12** formally describes equivalence modulo futures and its properties. Then, **Chapter 13** proves the confluence property of Chapter 11. This part interleaves formal definitions and proofs. Indeed, the part IV can be seen as a summary of this part with more informal details and more intuitive explanations.

Part VI concludes this thesis. **Chapter 14** gives implementations strategies related to ASP or ProActive. **Chapter 15** compares ASP with other concurrent calculi and languages presented in Chapter 6. This chapter both justifies the existence of a new parallel object calculus with asynchronous communications, gives links between existing calculi and ASP, and explains how some notions specific to other calculi could be compared to ASP or adapted in our framework. **Chapter 16** concludes this thesis and **Chapter 17** draws some perspectives to this study. Firstly, ASP could also be used to model components and mobility. Moreover, the confluence property could also be ensured by identifying some objects that do not modify their internal state or by ordering differently the services of requests.

Appendix A gives a proof of determinacy in the case of a tree topology of the object dependence graph.

Chapter 5

Distribution, Parallelism, Concurrence, and Objects

Parallel and distributed languages – including calculi – all fall within the scope of a few important *dimensions*. This chapter identifies those dimensions and attempts an informal analysis of existing frameworks.

We first explore *distribution, parallelism, and concurrency*. In a second phase, we consider the shift of paradigm, if any, brought in by *objects*.

5.1 A Few Definitions

Let us first introduce a few awfully basic definitions.

Parallelism: Execution of several activities or processes at the same time

In the following, an *activity* will denote an unity of parallelism, that is something more general than a *thread* or a *process*; each parallel entity behaving more or less independently (except of communications and synchronization) will be called an activity. Simultaneously, several activities achieve some actions, may they be visible or invisible. Basic examples of parallelism include two multiplications going on at the same time on two different processes, printing simultaneously two different files on two printers, saving the same file redundantly on several discs.

Concurrency: Simultaneous access to a resource, physical or logical [3]

When at least two activities are trying to access a single entity, concurrent actions are being carried out, or at least attempted to. In many cases, concurrency leads to interleaving, as it is rather frequent that concurrent actions actually take place one after the other, or at least appear to be - our world is in appearance at least rather discrete. Two files printing concurrently on the same printer will produce unpredictable output. Observation is a crucial aspect of concurrency as from different

[3] Sometimes, concurrency is also defined as the fact that several different operations can be performed and consequently one have to be chosen.

point of views some aspects may be considered or not as concurrent. To get back to the printer example, further analysis can probably tell apart the interleaving that occurred (pages, words, lines ...). Indexed, when printing starts, nothing can determine (no predefined order) which file will be printed first.

Distribution: Several address spaces

Distributed environments assume a fundamental constraint: no single address space. Examples of distribution include PCs on a Local Area Network (LAN), cluster of machines more and more used for intensive computation, and of course Grid computing over World Area Networks (WAN).

While the definitions above attempt to strongly differentiate three physical realities, it is often the case that they are slightly tangled, intentionally or just by mistake. The strong implications that do exist between them is in all likelihood one reason for the confusion.

Finally, there is an hypothesis, and a reality, that we believe cannot be bypassed: *Asynchronous systems*: No global clock, and unbounded communication time. Each process runs at its own speed, and communications take a non predictable time – not to be confused with synchronous or asynchronous communications. Again, this hypothesis proceeds from the need to scale up to large and global systems. On the contrary, *synchronous systems* take a few strong hypotheses: instantaneous communications (including broadcasting), instantaneous process reaction to inputs. Of course, the synchronous hypothesis does not scale up; no global clock can be assumed all over the world, and communications do take time, especially if you go half-way around the globe. Therefore, there are clearly two correlated challenges still to be tackled for distributed systems:

- asynchronous system that behave deterministically,
- synchronous systems that can scale up.

This study is an attempt to contribute solutions towards tackling the first challenge.

5.2 Parallelism and Concurrence

5.2.1 Parallel Activities

Provided we are beyond the frame of purely sequential execution or fully implicit parallelism, the very first dimension deals with the definition of *activities*. Whether statically or dynamically defined, a language always features concepts and constructs permitting to define parallel executions. Processes (tasks), threads, actors, active objects, or just the parallel evaluation of expressions or functions exemplify the main abstractions.

The nature of activities has a strong impact on the characteristic of activity identities, which in turn influences the language capacity to deal with them in a uniform and flexible manner.

5.2.2 Sharing

A second and strongly differentiating dimension is *sharing* of state between activities. The fact that several activities can or cannot reach the same data, leading to interleaving and potential race conditions, has a dramatic influence on both programming style and intrinsic properties.

5.2.3 Communication

As soon as sharing and interleaving is not the unique option for interactions between activities, *communication* comes into play as a crucial dimension.

Channels are a first option for enabling communications between activities. In that case, communication takes the form of messages being sent and received over channels. Message is a very basic, and somehow primitive, abstraction for interactions. The advent of procedural languages has led to the practical use of a higher level communication pattern: *remote procedure call (RPC)*. The information being communicated is no longer raw data, but a request for a procedure execution, together with its parameters. It is striking to observe that most of the formal work on parallel calculus is still within the paradigm of channel based communications, while programming languages are for a long time now in the era of procedure, functions and method calls. Section 5.3 will exhibit an important object enhancement to RPC-based communications.

Besides the form of communication, the nature of information being exchanged is also influential. The capacity to send and receive either activity identifiers or channel names induces an important characteristic: dynamic topology. A feature both theoretically and practically systematically used nowadays. With respect to exchanging data, the choice between copy or reference semantics arises. Maintaining reference semantics across physically disjoint address spaces leads to generalized references: a kind of Distributed Shared Memory (DSM) leading to complexity and poor scalability. The second solution, (deep) copying to manage distribution, is rather popular and quite effective. From RPC to Java RMI, including Network Objects [BNOW95], the automatic copy of data being referenced within a communication can occurs. While validated by practice, it must be clear that this solution imposes a semantic shift from procedure call to RPC; the user ends up with the task to maintain coherency of copies when needed. An interesting solution has been proposed that diminish this semantic change: copy-restore semantics that put back the parameters value after the call – somehow similar to a call-by-name semantics. Somehow, *copy-restore semantics* bridge some gap between DSM and deep copy. However, if the framework is to be scalable, copies have to be made somewhere. So, the user has to specify where and how the copies will take place, and of course the action needed to reconcile a global semantics. Moreover, copy-restore does not work properly for multiple (asynchronous) calls between two activities.

Not only passive data can be copied along communications: instead of passing references to activities, processes can themselves be deep-copied. A feature we believe captures at its best a practical facet: mobility of activity. Such computational or process mobility is rather present in calculi (e.g. higher-order PI-calculus, Ambient, Obliq), but only in a few languages, probably reflecting the difficulty to implement process mobility in operational frameworks.

With respect to the semantics of communication, there is of course the dichotomy of synchronous versus asynchronous communication. The reality is much more complex with large variations, and to some extent a continuum: purely synchronous, asynchronous with rendezvous, asynchronous FIFO preserving, asynchronous out of order, asynchronous without any guaranty including delivery.

5.2.4 Synchronization

Synchronization always boils down to waiting something, a given *event* or a piece of *data*.

Control based mechanisms provide primitives that allow to explicitly block the course of execution. The programmer makes use of a dedicated instruction to wait for some conditions or occurrence of events. One of the main purpose of such control based synchronization is often to avoid a busy wait. The well known *select with guards* statement is the outmost example, as for instance in CSP or ADA. However, a select is usually a rather complex control structure that also embodies features such as process termination, and non deterministic choice. Moreover, the guard capacity to include rendez-vous based communications leads to a very complex semantics for the programmer, and a rather difficult implementation in distributed environment; cf. for instance the polling nature of the Ada select [GC84]. This study proposes a clear answer to avoid such trouble: systematic asynchronous calls that systematically allows for selection upon a *local* pending queue.

In the quest for powerful synchronization primitive, selection can also be based on the caller identity, breaking the caller anonymity and somehow returning to two-sided communications. Some languages (e.g. Concurrent C [GR89]) even authorized selection, and as such synchronization, based on the value of effective parameters.

Overall, the select statement is an instruction to *filter* upon pending communications, and trigger the appropriate control flow. As there is no reason to wait for a unique communication, filtering can be achieved on several procedure calls or messages. This is the case in frameworks such as the Join calculus [FGL$^+$96], or the polyphonic *C#* [BCF02] language. These synchronizations are still control based, and rather demanding with respect to programmer skill, mainly due to the interleaving to be considered.

Dataflow synchronization is an important step in the direction of more manageable, but also more efficient distributed systems.

The idea is rather simple and powerful: let us trigger operations just based on the data availability. The programmer should say no more than the functional dependencies between computations. Coming from applicative languages, dataflow evaluations naturally allow parallel evaluation. But our focus being on explicit distribution and parallelism, the dataflow idea turns into a language construct: *the future*.

To the best of our knowledge, Multilisp [Hal85] invented the imperative face of dataflow synchronization with a reference to a value yet to be known, a future value. While still in an imperative setting, synchronization become implicit: a strict operation on an unknown future will automatically block until the value is available. While Multilisp is in a functional and shared memory setting, an important contribution of this study will be to apply this idea to a non-shared memory and object settings.

5.3 Objects

In the previous section, we pointed out parallel processes, message or remote procedure communications, and data-flow synchronization. What comes about to those concepts when objects come into play?

5.3.1 Object, Remote Reference and Communications

Let us start with a fairly embraced shift: Remote Procedure Call (RPC) to Remote Method Invocation (RMI, e.g. Java RMI). Method in the RMI acronym denotes the object paradigm. But the major shift is not in the terminology move, from procedure to method. In truth, the move comes from the target: a method call has a peculiar parameter, the target object, hence the dot notation. When turning a call such as `ro.foo (p);` into a communication, `ro` represents, identifies, a remote entity; in contrast RPC requires a specific parameter, a handler, to represent the remote target (e.g. `foo(ro_handler, p);`). As a consequence, references to objects are somehow unified with references to remote entities. So far, the shift is rather syntactical.

5.3.2 Objects vs. Parallel Activities

We said just above that references to objects are unified with references to remote entities. But the important unification is between objects and activities. Is the very notion of object unified with activity, or process?

Let us quickly put aside models such as Java RMI. A Java remote object is not by essence an activity. The fact that several threads can execute method simultaneously within a remote object does illustrate that when writing `ro.foo (p);`, what `ro` identifies is not a remote activity, just a remote object.

On the contrary, the notion of object can be unified with the notion of activity. Then, an activity *is* an object. A remote reference to an object becomes also a remote reference to an activity. This is the case in the functional framework of Actors [Agh86, AMST97]. An activity is identified with an actor, and an *actor address* is just a reference to a remote activity. In the context of imperative object language, such a concept merging an activity and a remote entity is an *active object*.

A final question remains: are all objects activities? The actor language answer to that question is yes, leading to what is call *uniform actor models*. All objects are active. Although theoretically interesting, we believe that solution not to be effective and practical. We prefer the so called *non-uniform* models, where only some objects are turned into active objects. The capacity for the programmer to specifically pinpoint the active objects reflects the role of activities in explicit parallelism, which is most needed for large scale distribution. But, of course, this study could be easily adapted to a uniform model (which would leads to a lot of simplifications).

5.3.3 Objects and Synchronization

At last, let us offer a quick overview of dataflow synchronization in an object world.

When activities are remote objects, asynchronous method call is a first move to decoupled computations and envision scalability. As method calls can return values, a dataflow synchronization can naturally be added to object languages: each result of an asynchronous call can be viewed as a future. It will be updated upon the reception of the result, after the remote method execution. In the meantime, any attempt to use the unknown future will block.

For instance, when `ro` is a remote object, the execution of the asynchronous call `v= ro.foo (p);` will not block the current activity, leading to `v` being a future. Any strict operation on an unknown future, for instance a *v.bar();* method call, will block until the future is filled up.

We will call such an object-oriented dataflow principle *wait-by-necessity* [Car93]. Many issues arise with such a principle. They will be examined at depth along the course of this thesis.

Chapter 6

Formalisms and Distributed Calculi

6.1 Basic Formalisms

This section reviews main calculi and formalism from which most of the (formally described) concurrent languages and calculi have been derived. Note that in [DZ01], Zilio gives a classification of mobile processes different from ours. It classifies mobile processes into mobility of names which are called *"labile processes"* and processes having a notion of movement and explicit locations like Ambient Calculus called *"motile processes"*. The purpose of this study is to examine the impact of different communication and concurrency methodologies, and thus will not focus on the importance of locations. From this point of view, Ambient calculus can be considered as belonging to the same family as π-calculus. Figure 6.1 provides an informal classification of calculi considering the different concurrency principles.

6.1.1 Functional Programming and Parallel Evaluation

λ-Calculus and pure functional languages provide a simple framework for designing parallel languages. In fact, it is well known that the λ-Calculus is confluent [Bar81]. In other words, the absence of side effect allows one to evaluate expressions composing the programs in any order. A parallel evaluation of functional languages is both deterministic and deadlock free.

The parallel functional evaluators have been widely studied, see for example [KPR$^+$92] for a lazy parallel evaluation, or [Ham94] for a survey of parallel functional programming.

6.1.2 Actors

Actors [Agh86, AMST92, AMST97] are rather functional processes relying on asynchronous communications. Actors are interacting by asynchronous *message passing.*

Agha, in [Agh86], presented an actor language. More formal syntax, operational semantics, equivalence, and bisimulation techniques for actors are given in [AMST92]

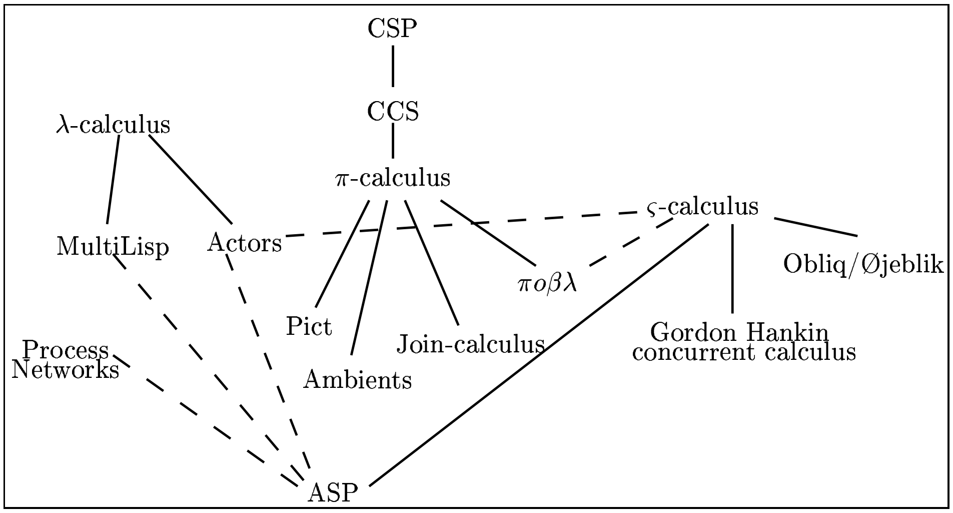


Figure 6.1: Calculi classification (informal)

and [AMST97]. Actors are based on a functional language but are organized in an object-oriented style. An actor is an object for which each method is written in a pure functional language. Communication is ensured by a mailbox mechanism and thus is asynchronous. Fairness is an important requirement of actor specification. According to Agha, fairness is a realistic hypothesis that provides semantics properties. Characteristics specific to actors are:

- An actor may *send* a message to another actor. $send(a, v)$ creates a message with receiver a and contents v.
- An actor may *change* its own behavior. During a message treatment, an actor *must* specify its new behavior (which can be the same as before the message treatment) by the primitive $become(b)$.
- An actor may *create* other actors with the primitives $newaddr()$ for creating an address and $initbeh(a, b)$ for initializing an actor behavior.

The history sensitive aspect of actors is performed by the *become* primitive. In other words, the state of an actor is specified by its behavior. This means that the behavior of an actor at invocation time may differ from its behavior at execution time. In ASP, such behavior modification will be forbidden and replaced by an imperative aspect allowing objects to modify their state (their data). These two approaches are somewhat opposite. Indeed, Agha and al. store the state of the actors inside their functions, while in ASP, the state is only stored in the data part of objects (field).

A notion of futures for actors is also presented in [Agh86].

A λ-calculus based actor calculus is presented in [AMST92, AMST97]. Actors are presented as an open system: they can receive messages from outside actors. These articles specify an operational semantics, an operational equivalence and operational bisimulation techniques for actors.

```
(define (Factorial ())
  (Is-Communication (a eval (with customer ≡ c)
                           (with number ≡ n)) do
      (become Factorial)
      (if (NOT (= n 0))
         (then (send x 1))
         (else (let (x (new FactCust (with customer c)
                                (with number n)))
                 (send Factorial (a eval (with customer x)
                                (with number n-1))))))))
(define (FactCust (with customer ≡ m)
                  (with number ≡ n))
   (Is-Communication (a number k) do
       (send m n*k)))
```

Figure 6.2: A Factorial Actor - [Agh86]

Figure 6.2 defines an actor computing a factorial that distributes the work to customers. More precisely, a chain of `FactCust` objects is created. Each customer performs one multiplication and forwards the result to the following customer, thus each call to factorial first creates a `FactCust` actor and recursively calls the `Factorial` actor with the newly created customer, for $n-1$.

In [KY94], an actor-like concurrent language is presented but is more related to typing theory and can capture "message not understood" errors. It is based on ML-like typing of record calculus. Indeed, this article considers that a concurrent object-oriented programming language is an assemblage where records are added to a concurrent calculus (*à la* π-calculus).

6.1.3 π-calculus

According to Zilio [DZ01], "Milner argues that when one talks about mobility in a system of interacting agents, what really matters is the movement of the access paths to the agents." This study partly follows this idea and will not focus on the notion of location. Even if the notion of location is important, we consider that the interaction between locations and the methodology of communication is out of the scope of this study.

Even if some kind of channels can be explicited in ASP and communicated between processes; this study will be more focused on the concurrency aspects than on mobility of names as presented by Milner. However, a notion of mobility *à la* Milner is intrinsic to ASP through the mobility of global references to some objects: ASP global references can be transmitted between processes.

π-calculus is a concurrent calculus based on communications over channels and introduced by Milner and al. [MPW92, Mil89]. It is a very small calculus where channels are first class entities and communication is due to synchronization between a process performing an output on a channel, and another process performing a blocking input on the same channel. Channel names are first class entities and can be passed over channels.

Several version of π-calculus exist and π-calculus syntax can be presented in different ways. We will use the following syntax and present the main variants of π-calculus below.

$$\begin{array}{rll} P, Q ::= & \mathbf{0} & \text{nil} \\ & \mid P|Q & \text{parallel composition} \\ & \mid (\nu x.P) & \text{restriction of name } x \\ & \mid x(y).P & \text{input} \\ & \mid x\langle y\rangle.Q & \text{output} \\ & \mid [x = y].Q & \text{matching} \\ & \mid P + Q & \text{choice} \\ & \mid !P & \text{replication} \end{array}$$

Informally, $x\langle y\rangle.Q$ sends y along channel x and, after synchronization with a process $x(z).P$ listening on channel x, continues as Q. Similarly and synchronously, $x(z).P$ receives y on channel x and continues as P where z is replaced by y. $\nu x.P$ creates a fresh channel x with lexical scope P. $P|Q$ corresponds to the parallel composition of two processes, $!P$ is an infinite number of P processes running in parallel. $P + Q$ denotes an external choice: as soon as P can be reduced, Q is discarded, or vice versa. Name matching only exists in some variants of π-calculus. In that case $[x = y].Q$ executes Q if x and y are the same channel.

Most of π-calculus derived languages and calculi presented here do not include name matching. Name matching makes equivalent different equivalence relations on π-calculus [FG98], but this is not directly linked to the subject of this study.

Polyadic π-calculus [Mil93] allows to send/receive several channels on a channel (i.e. $x(y).P$ becomes $x(y_1 \ldots y_n).P$).

π-calculus can be either synchronous or asynchronous. Asynchrony is obtained by disallowing choice and output prefix. This last point means that output messages do not have any continuation, that is to say, $x\langle y\rangle.Q$ is replaced by $x\langle y\rangle$ in the syntax above. Asynchronous π-calculus was first proposed by Honda and Tokoro in [HT91] and Boudol in [Bou92]. Synchronous π-calculus can be encoded in asynchronous π-calculus.

Sangiorgi and Merro introduce a “local π-calculus” (Lπ) which is similar to the asynchronous π-calculus in [MS98] (neither output prefix nor choice operator). In [San01], Sangiorgi extends this calculus with the capacity of sending processes through channels (LHOπ: Local Higher Order π-calculus) and shows the compilation from LHOπ to Lπ. A more general compilation of the HOπ into the π-calculus was presented in [San93].

Linear and linearized channels Linear and linearized channels provide confluence for some π-calculus terms. A channel with a linear type [KPT96] can only be used once in input and once in output. Thus communication over a linear channel can not be affected by a third process.

Steffen and Nestmann in [NS97] introduce a generalization of linear channels: “linearized” channels which can be reused after a “unique” usage. This article aims at ensuring, with typing techniques, that at any time only one communication is possible through a given channel.

Communication over linear and linearized channels is deterministic. Thus π-calculus terms only communicating over linear and linearized channels are confluent.

Pict is a language based on π-calculus and is briefly described in Section 6.2.2.

6.1.4 ς-calculus

Abadi and Cardelli [AC96, AC95a, AC95b] present a calculus for modeling object-oriented languages: ς-calculus. They study both functional and imperative behavior, starting from an object-based functional calculus (no classes) without typing, and added imperative aspects and most importantly studied typing. A class-based object calculus can also be translated to ς-calculus. An example of translation is defined in [AC96].

The main contribution of [AC96] deals with typing in object calculi. Even if this aspect is important, we will not focus on it. Indeed in our case, classical typing could be considered as orthogonal with concurrency.

ς-calculus is a base-calculus for several parallel calculi (e.g. Øjeblik [MKN02], concurrent object calculus of [GH98], ...).

ASP calculus is based on an untyped imperative ς-calculus (**imp**ς-calculus of [AC96]). Gordon and al. presented substitution based semantics of **imp**ς-calculus (small step and big step), and proved their equivalence with Abadi and Cardelli closure-based semantics in [GHL97b, GHL97a]. We based our semantics on operational semantics of [GHL97b] because it is more intuitive and concise than the one of Abadi and Cardelli. However, such a semantics is based on a substitution that is more difficult to implement efficiently.

Note that Gordon, Hankin and Lassen [GHL97b, GHL97a] also present compilation and CIU (Closed Instance of Use) equivalence on imperative objects. Moreover Gordon and Rees also present a bisimilarity equivalence of typed object calculi of Abadi and Cardelli in [GR96]. Those aspects deal with equivalence of *static* terms. In the following, we will not use this framework because, to be as general as possible, we were interested in relations that are still defined on (partially) reduced terms.

Also note that in Øjeblik the notion of argument of a method is introduced: a method has two formal parameters, one is the object itself (*self/this/...*), the other is a function parameter. Such an extra argument is only necessary in the context of remote method calls. Indeed, in a local context, a method call can return a function[4] that will be applied to the argument, whereas in concurrent object calculi, it can be useful to protect the state of the object from outside modifications. In that case, returning a function and performing operations “inside” the object[5] are not equivalent because

[4] it is easy to encode λ-calculus in ς-calculus.

[5] or more precisely inside a thread belonging to the object invoked.

only the later solution can modify the state of the invoked object (without loosing coherence of the objects or introducing locks). In ASP, the argument of methods is even more important as it is deep copied in order to preserve a given topology of links between objects : see Section 11.2.

Figure 6.3 presents an example of prime number sieve expressed in ς-calculus.

```
$Sieve \triangleq [m = \varsigma(s)\,\lambda(n)$
                  $let\ sieve' = clone(s)$
                  $in\quad s.prime := n;$
                      $s.next := sieve';$
                      $s.m \Leftarrow \varsigma(s')\,\lambda(n')$
                               $if(n'\ mod\ n) = 0$
                               $then\ []$
                               $else\ sieve'.m(n');$
                      $[],$
        $prime = \varsigma(x)\,x.prime,$
        $next = \varsigma(x)\,x.next];$
```

$\Leftarrow$ denotes the method update: modifies the body of a method.

The sieve can be used in the following way:

$for\ i\ in\ 2..99\ do\ sieve.m(i)$	initializes primes$\leq$ 100
$sieve.next.next.prime$	returns the third prime

Figure 6.3: Sieve of Eratosthenes in ς-calculus - [AC96]

Figure 6.4 presents a binary tree Class example. It has been slightly modified: types were removed.

6.1.5 Process Networks

Process Networks of Kahn [Kah74, KM77, WWV00] are explicitly based on the notion of *channels* between processes, performing *put* and *get* operations on them. Each process is an independent sequential computing station (no shared memory). They are linked with channels (one to one or one to many) behaving like unbounded FIFO queues making the communications asynchronous.

One process network channel can link at most one source process and many destinations. The destinations do not split the channel output, but each one reads every value put in the channel (a kind of broadcast). The reading on a channel is performed by a blocking *get* primitive. The order of reading on channels is fixed by the source program. Process networks provide *deterministic* parallel processes, but require that

$$
\begin{array}{l}
binClass \triangleq [new = \\
\qquad\qquad \varsigma(z)\, [isleaf = \varsigma(s)\, z.isleaf(s), \\
\qquad\qquad\qquad lft = \varsigma(s)\, z.lft(s), \\
\qquad\qquad\qquad rht = \varsigma(s)\, z.rht(s), \\
\qquad\qquad\qquad consLft = \varsigma(s)\, z.consLft(s), \\
\qquad\qquad\qquad consRht = \varsigma(s)\, z.consRht(s)], \\
\qquad isLeaf = \lambda(self)\, true, \\
\qquad lft = \lambda(self)\, self.lft \\
\qquad rht = \lambda(self)\, self.rht \\
\qquad consLft = \lambda(self)\, \lambda(newlft) \\
\qquad\qquad\qquad\qquad ((self.isleaf := false).lft := newlft).rht := self, \\
\qquad consRht = \lambda(self)\, \lambda(newrht) \\
\qquad\qquad\qquad\qquad ((self.isleaf := false).lft := self).rht := newrht]
\end{array}
$$

Figure 6.4: Binary tree in ς-calculus - [AC96]

the order of service is predefined and two processes cannot send data on the same channel.

Figure 6.5 shows the example of a sieve of Eratosthenes written in Process Networks. Note that all communications are performed through explicit and blocking `PUT` and `GET` operations which impose a lot of (not always necessary) synchronizations.

6.2 Concurrent Calculi and Languages

6.2.1 Multilisp

Halstead defined Multilisp [Hal85], a language with *shared memory* and *futures*. The construct $(future\ X)$ immediately returns a future for the value of X and concurrently evaluates X. A future without a value associated to it is said to be *undetermined*, it becomes *determined* when its value has been computed. The combination of shared memory and side effects prevents Multilisp from being determinate.

The main parallelism primitive is a $PCALL$ that performs an implicit *fork* and *join* and evaluates its arguments concurrently: $(PCALL\ F\ A\ B)$ evaluates concurrently F, A, and B to f, a, and b; and applies f to the arguments a and b.

```
Process INTEGERS out QO
    Vars N; 1 → N;
    repeat INCREMENT N; PUT(N,QO) forever
Endprocess;

Process FILTER PRIME in QI out QO
    Vars N;
    repeat GET(QI) → N;
           if (N MOD PRIME)≠0 then PUT(N,QO) close
    forever
Endprocess;

Process SIFT in QI out QO
    Vars PRIME; GET(QI) → PRIME;
    PUT(PRIME,QO);  emit a discovered prime
    doco channels Q;
      FILTER(PRIME,QI,Q); SIFT(Q,QO);
    closeco
Endprocess;

Process OUTPUT in QI;
    repeat PRINT(GET(QI)) forever
Endprocess

Start doco channels Q1 Q2;
    INTEGERS(Q1);SIFT(Q1,Q2); OUTPUT(Q2);
    closeco;
```

Figure 6.5: Sieve of Eratosthenes in Process Networks - [KM77]

In Multilisp, as the addition operator + needs a value of its arguments, the two following expressions yield essentially to the same parallelism:

```
(+ (future A) (future B))


      (pcall + A B)
```

Halstead classifies the programming languages by specifying whether they have explicit parallelism, side effects, and shared memory. For example, CCS is characterized by explicit parallelism, side effects and no shared memory, and Multilisp by explicit parallelism, side effects and shared memory.

According to Halstead [Hal85], the fact that no data is shared between different threads (no shared memory) is one of the failings of CSP. But the interleaving of processes accessing and modifying data can also be considered as disturbing for the programmer as different interleaving between the threads can lead to different results (Multilisp is not deterministic). Another drawback of CSP is that it leads to nonuniform access to data. Indeed local accesses can be performed classically, whereas accesses to data belonging to another process needs a communication through channels. In ASP no memory is shared but the copying of shared data is implicit.

Katz and Weise [KW90] studied the interactions between futures and continuations and problems arising when those two functionalities are mixed.

6.2.2 Pict

Pict [PT00] is a language based on π-calculus. It is based on an asynchronous π-calculus without choice and name matching. Typing (sub-typing and type inference) of Pict and higher level features (than π-calculus) are also presented in [PT00].

A core language of Pict is presented in [PT95] and is used to create more complex objects able to encode classical features. For example, synchronization, locks, choice and, a lot of other primitives can be derived from the core calculus.

From a general point of view, Pict is designed to be used to experiment new designs of concurrent object structures (like for example, the one of [PT95]). Only a few primitives provide the possibility of writing simple objects and their typing. No specific concurrency mechanism for objects has been implemented inside Pict.

6.2.3 Ambient Calculus

Ambient calculus [CG00] is a calculus describing the movement of processes through the explicit notion of *location*. Ambients are convenient for modeling movements through administrative domains (e.g. through firewalls).

An ambient is defined by the following characteristics.

- An ambient is a bounded place.
- Ambients can be nested and can be moved as a whole.
- Computation appears inside ambients, and can control the ambient itself (e.g. make it move).

The actions of ambients are also called *capabilities*. The capability *in* m allows entry into the ambient m, the capability *out* m allows exit out of m, and the capability *open* m allows the opening of m.

Syntax of ambient calculus is the following (n are names, P, Q are processes and M are capacities):

$P, Q ::=$	$(\nu n)P$	restriction
	$\mathbf{0}$	inactivity
	$P \mid Q$	composition
	$!P$	replication
	$n[P]$	ambient
	$M.P$	action

$M ::=$	$\mathit{in}\ n$	can enter n
	$\mathit{out}\ n$	can exit n
	$\mathit{open}\ n$	can open n

Some communication primitives can be added to ambients in a somewhat orthogonal way (in a π-calculus style): $(x).P$ performs an input action that can interact with an asynchronous output action $\langle M \rangle$. Such interactions are only *local* to an ambient. Indeed, long range communications may need to cross firewalls and should not happen automatically. According to Cardelli and Gordon, long range communications should be performed by the movement of a "messenger" ambient. With such communication primitives, ambients can encode the asynchronous π-calculus.

Operational semantics is based on the three following rules:

$$n[in\ m.P|Q]|m[R] \rightarrow m[n[P|Q]|r]$$

$$m[n[out\ m.P|Q]|r] \rightarrow n[P|Q]|m[R]$$

$$open\ n.P|n[Q] \rightarrow P|Q$$

and a local communication rule:

$$(x).P|\langle M \rangle \rightarrow P\{x \leftarrow M\}$$

Note that mobility in π-calculus is a mobility of names: names can be communicated over channels whereas mobility in ambients consists in moving ambients themselves. Thus the notion of mobility in these two calculi are not incompatible and in fact inside each ambient a mobility of names is possible because of the encoding of the asynchronous π-calculus.

The main contribution of [CG00] concerns expressiveness of Ambients. This paper also contains a lot of examples of ambients.

Figure 6.6 shows an example of encoding of locks in ambients [CG00].

$$\begin{array}{l} acquire\ n.P \triangleq open\ n.P \\ release\ n.P \triangleq n[]|P \end{array}$$

Figure 6.6: Locks in ambients - [CG00]

Note that a meaningless term of the form $n.P$ can arise during reduction, and a type system like the one described in [CG99] is necessary to avoid such anomaly.

6.2.4 Obliq and Øjeblik

Obliq [Car95] is a language based on the ς-calculus that expresses both parallelism and mobility. Obliq is an object language based on threads communicating with a shared memory. Øjeblik [NHKM02, BN02, MKN02], is a sufficiently expressive subset of Obliq which has a formal semantics. The main results on Øjeblik concern migration.

```
let sieve =
{ m =>
   meth(s, n)
     print(n);
     let s0 = clone(s);
     s.m := meth(s1,n1)
              if (n1 % n) is 0 then ok
              else s0.m(n1)
              end
            end;
     end};

 print the primes <100
for i=2 to 100 do sieve.m(i) end;
```

Figure 6.7: Prime number sieve in Obliq

Øjeblik and Obliq semantics is based on threads ($fork$ and $join$ operators) and all references are global (when necessary)[6]. As a consequence these languages are based on a shared memory mechanism. Calling a method on a remote object leads to a remote execution of the method but this execution is performed by the original thread (or more precisely the original thread is blocked). Thus the parallelism is only based on threads, and is independent of the location of the objects performing operations.

In Obliq, the interferences between threads can be limited by serialized objects: if an object is *serialized*, then, at any time, only one thread is inside this object. In other words, a second thread entering an object is blocked until the first one has finished. Serialization may be guaranteed with a mutex. An operation is *self-inflicted* if it addresses the current self. Authorizing reentrant mutexes allows self inflicted operations to be performed for serialized objects. This allows recursion but not mutual recursion (no call back).

In Obliq and Øjeblik migration is the composition of cloning and aliasing:

$$surrogate = \varsigma(s)s.alias\langle s.clone\rangle.$$

This composition is deeply studied in Øjeblik, see for example [NHKM99].

An Obliq object can be protected, for example, in [NHKM99] : “based on self infliction, objects are protected against external modification”. That means that for the protected objects, only self-inflicted updates cloning and aliasing are allowed. In Øjeblik every object is *protected* and *serialized* .

In [NHKM02], Nestmann and al. present different semantics for forwarding and updating. The effect of authorizing or not some operations to pass or not through the forwarders is studied.

[6] Indeed, when an object reference is passed through the network, a local reference becomes a global reference.

6.2.5 The $\pi o\beta\lambda$ Language

Inspired from POOL [Ame89, Ame92] Jones designed a concurrent object-oriented calculus named $\pi o\beta\lambda$ [Jon92, JH96].

$\pi o\beta\lambda$ can be considered as a rather synchronous language. Indeed:

- Only one method of a given object can be active at any time. Using the Obliq vocabulary, one could say that every object is serialized.
- Like in Obliq, the calling method is blocked until a result is returned by the called object.

There is no direct notion of thread in $\pi o\beta\lambda$. Instead, parallelism comes from two facts:

- A function can return a value before the end of its execution. In that case, the calling method obtains the result and can continue its execution while the called function terminates its computation (which will have no consequence on the returned value).
- A function can delegate the task of returning a value to another object by using the *yield* primitive. In that case, this object is not blocked any more and the result is directly returned from the last object to the first caller.

These features will have to be compared with automatic and transparent updates of futures in ASP.

Figure 6.8 shows an example of a $\pi o\beta\lambda$ binary tree.

```
class T0
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W);
    return

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then return L!Search(X)
              else return R!Search(X)
```

Figure 6.8: Binary tree in (a language inspired by) $\pi o\beta\lambda$ - [LW95]

A sufficient condition is given for increasing the concurrency of $\pi o\beta\lambda$ programs without losing confluence, it is based on a program transformation. The principle is that an operation can be safely exchanged with a return statement, provided the operation does not *interfere* with the result to be returned. The interference can concern both data flow aspects: the operation should not affect the result; and control flow ones: the operation should terminate and can not invoke methods on public objects (because such calls could interfere with calls performed by the caller object which should occur later).

Under this condition, one can return a result from a method before the end of its execution; then the execution of the method continues in parallel with the caller thread. This sufficient condition is expressed by an equivalence between original and transformed program. $\pi o\beta\lambda$ can be translated to (dialects of) π-calculus (e.g. [Jon93]). From such a translation, Sangiorgi [San99], and Liu and Walker [LW95, LW98] proved the correctness of transformations on $\pi o\beta\lambda$ described in [JH96].

Figure 6.9 shows an example with the result of such a transformation applied to the program of Figure 6.8. Consequently, these two programs behave identically.

```
class TO
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W)

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then commit L!Search(X)
              else commit R!Search(X)
```

Figure 6.9: Binary tree, an equivalent program ($\pi o\beta\lambda$)- [LW95]

There is an equivalent version of the calculus (defining, for example, a Sieve of Eratosthenes), with a very different syntax in [Jon92].

6.2.6 Gordon and Hankin Concurrent Calculus

Gordon and Hankin [GH98] proposed a concurrent object calculus: a parallel composition $\|$ is added to the ς-calculus. Moreover, every object has a name: there is a *denomination* operator. In such a calculus, a method is executed by the thread that has invoked it. Moreover, objects need to be declared as separate processes that do not perform computation. As a consequence, the notion of object and of executing threads are clearly separate (one could define objects and threads in different spaces). Moreover a type system is necessary to distinguish terms from expressions as a denominated object can only be a process, but an expression can either be a top level process (thread) or be included inside another expression.

Note that an additional synchronization mechanism has to be added to the calculus (via mutexes).

Jeffrey [Jef00] introduced a modified version of Gordon and Hankin's concurrent object calculus, and added the notion of *location* in order to make this calculus distributed.

6.2.7 Join-Calculus

The join-calculus [FGL$^+$96, FG96, FBL98] is an asynchronous calculus with mobility and distribution. Synchronization in join-calculus is based on *filtering patterns* over channels. From the communication point of view, join-calculus can be seen as an asynchronous π-calculus with powerful message receivers (called triggers): a process can be triggered by the presence of several messages simultaneously.

The Join-Calculus syntax is composed of processes:

$$\begin{array}{rll} P ::= & | \, x\langle\tilde{y}\rangle & \text{message emission} \\ & | \, \mathbf{def}\ D\ \mathbf{in}\ P & \text{definition of ports} \\ & | \, P|P & \text{parallel composition} \\ & | \, \mathbf{0} & \text{null process,} \end{array}$$

definitions:

$$\begin{array}{rll} D ::= & J \triangleright P & \text{rule matching join pattern } J \text{ (trigger)} \\ & | \, D \wedge D & \text{connection of rules} \\ & | \, \mathbf{T} & \text{empty definition,} \end{array}$$

and join patterns:

$$\begin{array}{rll} J ::= & x\langle\tilde{y}\rangle & \text{message pattern} \\ & | \, J|J & \text{joined patterns} \end{array}$$

The Join calculus semantics is based on a reflexive chemical abstract machine (RCHAM) and can be summarized by the following rules:

$$\begin{array}{rcll} \vdash P|P' & \leftrightarrow & \vdash P, P' & \\ \vdash \mathbf{0} & \leftrightarrow & \vdash & \\ \mathbf{T} \vdash & \leftrightarrow & \vdash & \\ \vdash \mathbf{def}\ D\ \mathbf{in}\ P & \leftrightarrow & D\sigma \vdash P\sigma & \sigma \text{ creates fresh channels} \end{array}$$

$$J \triangleright P \vdash J\sigma \longrightarrow J \triangleright P \vdash P\sigma$$

6.2.8 CML

In [Rep91], Reppy presents an extension of SML (Standard ML) called CML (Concurrent ML) for concurrent programming in SML. CML is a threaded language. The synchronization is performed by a `sync` operator. In the base language, the communications are synchronous but a mailbox (request queue) mechanism can be easily implemented with a buffered channel.

6.2.9 Kell-calculus

Stefani [Ste03] has introduced a calculus that is able to model hierarchical components — especially sub-components control. The *kell-calculus* is based directly on the π-calculus with the possibility to have joins inside triggers (like in *distributed join-calculus* - DJoin [FG96]). Kell-calculus is also intended to overcome the limitations of the **M**-calculus [SS03] which are also presented in [Ste03].

6.3 Other Expressions of Concurrency

Steele [Ste90] expressed a programming model ensuring the confluence of programs by analyzing (mainly dynamically) the 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.

Montanari and al. introduced tile-based semantics [BMM02, FM00] which is based on rewrite rules in side effects. This theoretical framework (based on double categories) has been applied to give a semantics to located CCS in [FM00]. We think such a framework could be used to provide a less heavy (or at least more modular) semantics for ASP calculus but this would require the semantics to be entirely rewritten.

6.4 Short Synthesis

In this chapter, we reviewed some classical concurrent calculi and languages. In the following, a new calculus (ASP) will be presented. This new calculus was necessary for us to have a structured model of an imperative object calculus with asynchronous communications and futures. Indeed none of the calculi presented here has all these characteristics. Furthermore, ASP is a calculus with a more structured syntax than other calculi, and even if ASP programs could be rewritten into another calculus or language, ASP structuring would be lost and the proof of the ASP properties would probably be more difficult or impossible. Chapter 15 will further compare most of the languages and calculi presented here with ASP.

Part III

ASP Calculus

This part presents a calculus named *ASP*: *Asynchronous Sequential Processes*. ASP models an object-oriented language with asynchronous communications and futures, and sequential execution within each parallel process. We start from a purely sequential and classical object calculus (**imp**ς-calculus) [AC96] and extend it with two parallel constructors: *Active* and *Serve*. *Active* turns a standard object into an active one, executing in parallel and serving requests in the order specified by the *Serve* operator. Method calls on active objects are asynchronous: the results of asynchronous calls are represented by *futures* until the corresponding response is returned. Automatic synchronization of processes comes from *wait-by-necessity* [Car93]: a wait automatically occurs upon a strict operation (e.g. a method call) on a future.

The passing of *futures* (results of asynchronous calls) between processes, both as method parameters and as method results is an important feature of ASP.

Chapter 7

An Imperative Sequential Calculus

7.1 Syntax

ASP calculus starts from an imperative sequential object calculus *à la* Abadi-Cardelli. Note that a few characteristics have been changed between **imp**ς-calculus and ASP sequential calculus.

- Because arguments passed to an active objects methods will play a particular role, a parameter is added to every method like in [NHKM02]: in addition to the self argument of the 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 syntax below). In practice, we will use methods with several arguments in our examples, but, for simple, we will not add them in the semantics.

- Method update is not included in ASP calculus because we do not find it necessary and it is possible to express updatable methods in ASP calculus anyway (e.g. updatable fields containing lambda expressions). Moreover, adding updatable methods would not arise any theoretical problem.[7]

- As in [GHL97b], during the reduction, *locations* (reference to objects in a store) can be part of terms in order to simplify the semantics. The locations should not appear in source terms.

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). In the folowing of this study, l_i, $i \in 1..n$[8] range over fields names and m_j, $j \in 1..m$ over method names. In practice, there is one integer n and one integer m for each object, but we will simply denote all these numbers by n and m.

[7] But, in the parallel case, updating methods would unnaturally modify the meaning of requests that have already been sent but are not executed yet.

[8] $i \in 1..n$ classically abbreviates $i \in \mathbb{N} \cap [1, n]$

$a, b \in L ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition,
$\mid a.l_i$	field access,
$\mid a.l_i := b$	field update,
$\mid a.m_j(b)$	method call,
$\mid clone(a)$	superficial copy,
$\mid \iota$	location (not in source terms)

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

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

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

$$let\ x = a\ in\ b \triangleq [m = \varsigma(z, x)b].m(a)$$

$$a; b \triangleq [m = \varsigma(z, z')b].m(a)$$

Lambda expressions, and methods with zero and more than one argument are also easy to encode in the sequential calculus and will also be used in this study.

7.2 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 also called *static terms*. Locations appear when objects are put in the store.

7.2.1 Substitution

The substitution of b by c in a is written: $a\{\!\{b \leftarrow c\}\!\}$. Substitutions are denoted by $\theta ::= \{\!\{b \leftarrow c\}\!\}$. Multiple substitutions are applied from left to right: $a\theta\theta' = (a\theta)\theta'$

In method calls (INVOKE), substitution is applied in a classical way on bounded variables: formal parameter x is replaced by the location of the argument without replacing inside binders $\varsigma(x, z)$ or $\varsigma(z, x)$.

An *injective* substitution of a location by another location will also be called a *renaming*. A renaming is in fact an alpha-conversion of locations.

7.2.2 Store

Reduced objects are objects with all fields reduced (to a location):

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

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 (*store append*). When the domains are not disjoint, $\sigma + \sigma'$ updates the values defined in σ' by those defined in σ(*store update*). It is defined on $dom(\sigma) \cup dom(\sigma')$ by

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

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

7.2.3 Configuration

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

Definition 7.1 (Well formed sequential configuration)

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

Let $\equiv$ be the equality between configurations modulo renaming of locations:

Definition 7.2 (Equivalence on Sequential Configurations)

$$(a, \sigma) \equiv (a', \sigma') \Leftrightarrow \exists \theta,\, (a\theta, \sigma\theta) = (a', \sigma')$$

7.3 Reduction

Table 7.1 defines a small step substitution-based operational semantics for the sequential calculus. 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 [GHL97a]. Table 7.1 applies one rule on the point of reduction represented by the unique occurrence of $\bullet$ in the following *reduction contexts*:

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

$\mathcal{R}[a]$ denotes the substitution inside a reduction context:

$$\mathcal{R}[a] = \mathcal{R}\{\!\{\bullet \leftarrow a\}\!\}$$

.

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

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

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

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

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

Table 7.1: Sequential Reduction

7.4 Properties

Initial Configuration 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.

Well-formedness As a first correctness property, it is easy to show that reduction preserves well-formedness.

Property 7.1 (Well-formed sequential reduction)

$$\vdash (a, \sigma)\ \textsc{ok} \wedge (a, \sigma) \rightarrow_S (b, \sigma) \quad \Longrightarrow \quad \vdash (b, \sigma)\ \textsc{ok}$$

sequential determinism Moreover, a first result towards determinism is to ensure that a sequential reduction is deterministic. Indeed, the reduction contexts specify the order of reduction. Consequently, a sequential reduction is deterministic up to the choice of freshly allocated locations:

Property 7.2 (Determinism)

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

In fact, at most one reduction can be made on each configuration. The only choice is the name of locations created by STOREALLOC and CLONE.

Chapter 8

Asynchronous Sequential Processes

We introduce here a parallel calculus which is based on *activities*. Each ASP object is either *active* or *passive*. There is one active object at the root of each activity. Activities execute instructions concurrently, and interact only through method calls. Method calls toward active objects are always *asynchronous*. Synchronization is due to *wait-by-necessity* on the result of an asynchronous method call (data-driven synchronization).

8.1 Principles

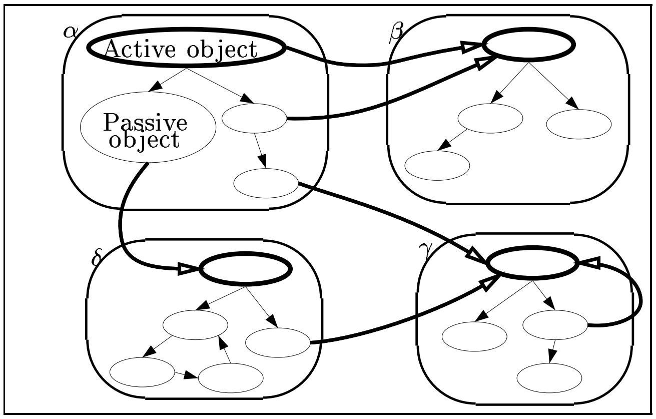


Figure 8.1: Objects and activities topology

An *activity* is a single process (execution thread) associated with 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. ASP activities do not share memory. Figure 8.1 shows an example of objects topology in a configuration containing four activities.

The principles of asynchronous method calls is the following, when an object sends a request to an activity, it is stored in a request queue and a future is associated to this request. A *future* represents the result of a method call to an active object that has not yet been returned. Such a request is called *pending*. Later on this request will be *served* (i.e. taken in the request queue in order to be evaluated), it becomes a *current request*. When the service is finished, a value is associated to the result of this request and the association between the future corresponding to this request and the calculated value is stored in a *future values list*. Such requests are called *served requests*. Afterwards, the distant reference to the future may be *updated* by the calculated value.

The activation of an object ($Active(a, m)$) creates a new activity whose active object is a copy of a. $Serve(M)$ performs a blocking service of requests received by the current active object.

For example, with the point object defined in Section 7.1, $Point.getColor()$ 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 to object p 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 one tries 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*.

Unlike many other concurrent calculus based on ς-calculus, in ASP, the requests are not executed by the process that performs the method call but by the processed associated to the destination of the request.

8.2 New Syntax

We extend the sequential calculus by adding the possibility of creating an active object and of serving a request:

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

$\vert Active(a, m_j)$	activates object: deep copy + activity creation
	m_j is the activity method or $\emptyset$ for FIFO service
$\vert Serve(M)$	Serves a request among a set of method labels,
$\vert a \Uparrow f, b$	a with continuation b (not in source terms)

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

$$M = m_1, \ldots, m_k$$

A parallel configuration is a set of activities, each activity contains several fields that will be introduced informally in the following and formally defined in Chapter 10:

$$P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\ldots] \| \ldots$$

To summarize, the whole syntax of the ASP source terms is the following. m_j, $j \in K$ range over method names:

$$\begin{array}{rll}
a, b \in L ::= x & & \text{variable,} \\
& \mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} & \text{object definition,} \\
& \mid a.l_i & \text{field access,} \\
& \mid a.l_i := b & \text{field update,} \\
& \mid a.m_j(b) & \text{method call,} \\
& \mid clone(a) & \text{superficial copy,} \\
& \mid Active(a, m_j) & \text{activity creation,} \\
& \mid Serve(m_j)^{j \in 1..k} & \text{service primitive.}
\end{array}$$

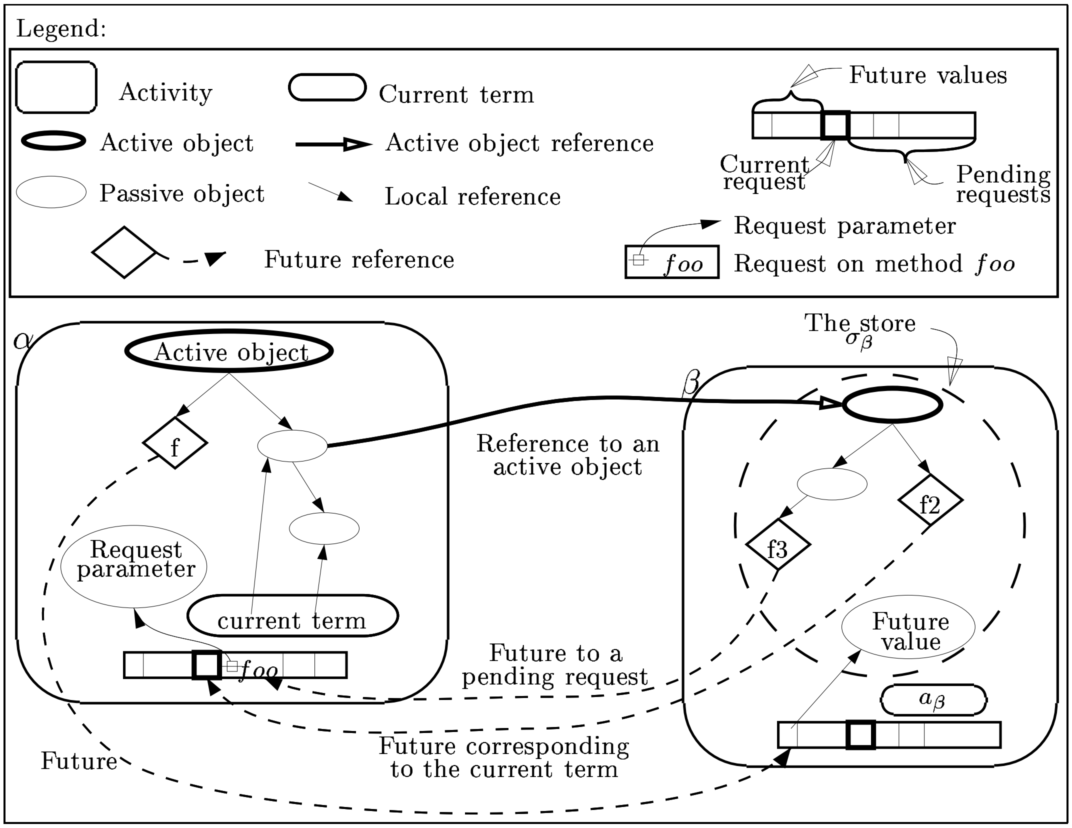


Figure 8.2: Example of a parallel configuration

8.3 Informal Semantics

In every activity α, a *current term* a_α represents the current computation. Every activity has its own *store* σ_α which contains one active and many passive objects. It also contains *pending requests* which store the pending method calls and a *future list* which stores the result of finished requests.

Figure 8.2 gives a representation of a configuration consisting of two activities. It contains three references to futures (one calculated, one current and one pending). The active objects are bold ellipses; the futures references are diamonds; the futures values, the current future and the pending requests 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. The continuation will not appear in the figures.

8.3.1 Activities

The *Active* operator ($Active(a, m_j)$) creates a new activity α with the object a at its root. The object a is copied as well as all its dependencies[9] (deep copy) in a new activity. The second argument to the *Active* operator is the name of a method which will be called as soon as the object is activated. This method 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 8.2, 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 (activity ends). The remote references to the active object of activity α will be denoted by $AO(\alpha)$. $AO(\alpha)$ acts as a proxy for the active object of activity α.

8.3.2 Requests

The communications between activities are due to method calls on active objects and returns of corresponding results. A method call on an active object ($Active(o, \emptyset).foo()$) consists in atomically adding an entry to the *pending requests* 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). The arguments of requests and the values of futures are deeply copied when they are transmitted between activities[9]. Active objects are transmitted with a reference semantics.

8.3.3 Futures

An operation on an object is *strict* if it needs to access to the content of the object: the only strict operations are field and method access, update, clone. For example, transmitting an object as a method parameter is not a strict operation.

Futures are generalized references that can be manipulated classically while no strict operation is performed on the object they represent. In Figure 8.2, the 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 the activity β.

[9] to prevent distant references to passive objects.

A wait-by-necessity occurs when we try to perform a strict operation on a future. This wait-by-necessity can only be released by *updating* the future i.e. replacing the reference to the future by a copy[9] of the future value.

8.3.4 Serving Requests

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 pending requests (a request on one of the method labels 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 stored continuation. The term that had served the finished request continues its execution (it becomes the current term). The *future list* maps futures to their values within the activities that computed them. A future value is called *partial* if its dependencies contain futures references.

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).

Chapter 9

A few Examples

This chapter presents four examples illustrating the ASP calculus. First, Section 9.1 presents a simple binary tree. Then, sections 9.2 and 9.4 presents two examples: a Sieve of Eratosthenes and a Fibonacci numbers computation, inspired from process networks. Thus, Section 9.3 gives a brief idea of the possible translations from process networks to ASP. Finally, Section 9.5 outlines how a more complex program (a bank account server) could be implemented in ASP.

9.1 Binary Tree

Figure 9.1 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 ASP. The definition of classes (new method ...) has already been proposed by Abadi and Cardelli in the **imp**ς-calculus [AC96].

$\underline{add}$ 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.

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

$\underline{new}$ is the method invoked to create a new node.

This example is parameterized by a factory able to create a sequential (sequential binary tree) or an active (parallel binary tree) node.

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: $[a = 6, b = 8]$.

$$\begin{array}{l} let\ tree = (BT.new).add(3,4).add(2,3).add(5,6).add(7,8) in \\ \quad [a = tree.search(5), b = tree.search(3)].b := tree.search(7) \end{array}$$

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

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

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

Figure 9.1: Example: a binary tree

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

Figure 9.2: Example: Sieve of Eratosthenes (pull)

activity, the result of concurrent calls is deterministic. Determinism properties will be detailed in Section 11.

9.2 Distributed Sieve of Eratosthenes

Let us translate the distributed sieve of Eratosthenes described in [KM77] in ASP. In [KM77], 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 9.3). *Repeat* performs an infinite loop and will be defined later on. Figure 9.2 defines a "pull" sieve of Eratosthenes in ASP.

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 (*print*(n) denotes the output of integer n). When a new prime is found, a new *Sieve* in inserted between the *Sift* and the former last *Sieve*.

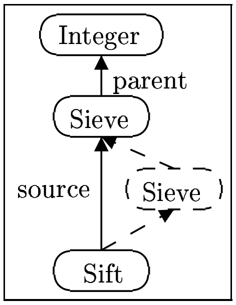


Figure 9.3: Sieve of Eratosthenes (pull)

```
$let\ Sieve = [N = 0, prime = 0; next = []; put = \varsigma(s, n)s.N := n,$
        $act = \varsigma(s, \_)Serve(put); Display.put(s.N);$
                              $s.prime := s.N; s.next := Active(s, act);$
                              $Repeat(Serve(put);$
                              $if\ (s.N \text{ MOD } s.prime \neq 0)\ s.next.put(s.N))]\ in$
$let\ Integer = [n = 1; first = Active(Sieve, act);$
            $act = \varsigma(s, \_)Repeat(s.n := s.n + 1; s.first.put(s.n))]\ in$
  $Active(Integer, act)$
```

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

Figure 9.4: Example: Sieve of Eratosthenes (push)

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 9.5). Figure 9.4 defines a "push" sieve of Eratosthenes in ASP.

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. But the fact that this reference will not be used any more could only be verified by a (complex) control flow analysis.

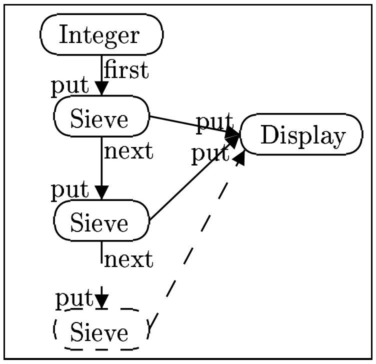


Figure 9.5: Sieve of Eratosthenes (push)

9.3 From Process Network to ASP

These two examples suggest the translation sketched by Figure 9.6.

The translation from a process network to an ASP term is performed either by a *push* strategy allowing pipelining: we call a method *put* on the destination of the channel or by a *pull* strategy: the destination calls a method *get* on the source of the channel. Section 15.6 details the translation and comparisons between ASP and process Networks.

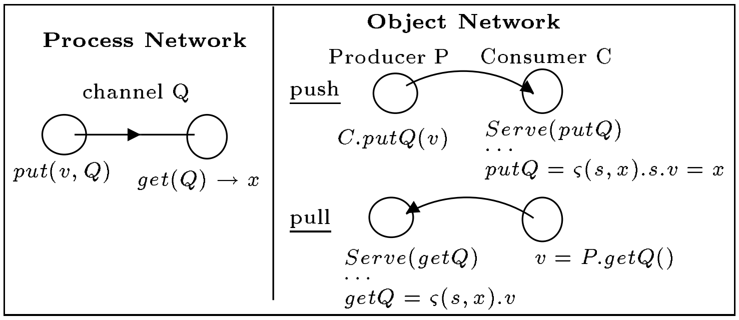


Figure 9.6: Process Network vs. Object Network

9.4 Example: Fibonacci Numbers

Consider the process network that computes the Fibonacci numbers in [PR03]. Let us write an equivalent program in ASP. Figure 9.7 describes the set up of processes that computes the Fibonacci numbers and Figure 9.8 gives the corresponding program.

Display receives the list of Fibonacci numbers. Initialization consists in sending 0 ($fib(0)$) and 1 ($fib(1)$) from $Cons2$ and $Cons1$ respectively. At the opposite from the other examples, this program is rather synchronous because there is at most one pending request in each activity (except the *Display*).

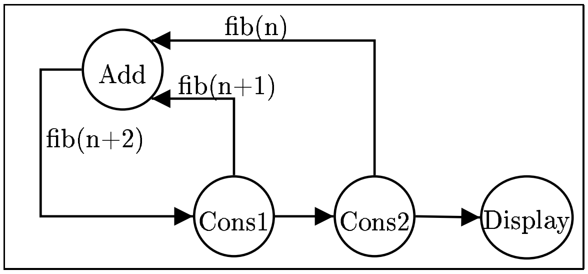


Figure 9.7: Fibonacci Numbers Processes

$$
\begin{array}{ll}
\mathit{let\ Add} = \mathit{Active}([n1 = 0, n2 = 0; & \\
 & \mathit{service} = \varsigma(s, \_) \\
 & \quad \mathit{Repeat}(\mathit{Serve}(\mathit{set}1); \mathit{Serve}(\mathit{set}2); \mathit{Cons}1.\mathit{send}(s.n1 + s.n2)), \\
 & \mathit{set}1 = \varsigma(s, n)s.n1 := n, \mathit{set}2 = \varsigma(s, n)s.n2 := n], \mathit{service}) \\
\mathit{and\ Cons}1 = \mathit{Active}([\emptyset; & \\
 & \mathit{service} = \varsigma(s, \_)\,(\mathit{Add}.\mathit{set}1(1); \mathit{Cons}2.\mathit{send}(1); \mathit{Repeat}(\mathit{Serve}(\mathit{send}))) \\
 & \mathit{send} = \varsigma(s, n)(\mathit{Add}.\mathit{set}1(n); \mathit{Cons}2.\mathit{send}(n))], \mathit{service}) \\
\mathit{and\ Cons}2 = \mathit{Active}([\emptyset; & \\
 & \mathit{service} = \varsigma(s, \_)\,(\mathit{Add}.\mathit{set}2(0); \mathit{Display}.\mathit{send}(0); \mathit{Repeat}(\mathit{Serve}(\mathit{send}))) \\
 & \mathit{send} = \varsigma(s, n)(\mathit{Add}.\mathit{set}2(n); \mathit{Display}.\mathit{send}(n))], \mathit{service})
\end{array}
$$

Figure 9.8: Example: Fibonacci Numbers

9.5 A bank Account Server

Let us imagine a bank application server. Figure 9.9 gives the set up of the different objects, this application should provide 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.

The Figure 9.10 defines the Central Service object. 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.

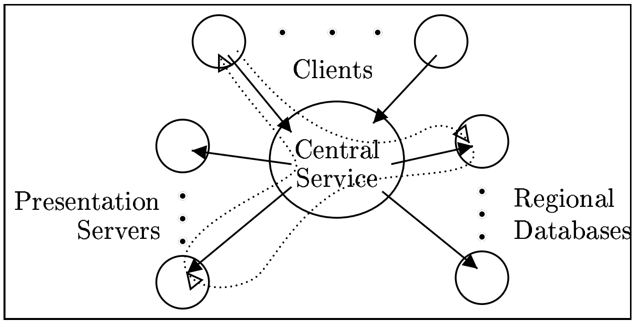


Figure 9.9: Example: a bank application

```
let CentralService = [...;
    regionalDatabase = ς(s, ID)....,
    presentationServer = ς(s, device). ...,
    act = ς(s, _).Repeat(Serve(getStatement)),
    getStatement = ς(self, ID, device).
        let state = (self.regionalDatabase(ID))
                                             .getStatement(ID)
        in   (self.presentationServer(device))
                                             .getPresentation(state)  ]
```

Figure 9.10: Example: Bank account server

The other involved objects are not detailed here. 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 sent to the client might contain several futures coming from Regional Database and Presentation Server. Thus, by writing a classical object oriented program, one obtains a parallel and somewhat lazy execution.

Part IV

Semantics and Properties

This part formally defines the ASP calculus semantics and formalizes determinism properties on this calculus.

As futures can proliferate (they can be passed between activities), 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.

Chapter 10

Parallel Semantics

10.1 Structure of Parallel Activities

Assume now that there are three distinct name spaces: activities ($\alpha, \beta \in Act$), locations (ι) and futures (f_i). Consequently, locations and future identifiers f_i are local to an activity.

A future is characterized by its identifier f_i, the source activity α and the destination activity β of the corresponding request (the activity that receives and handles the request): ($f_i^{\alpha\to\beta}$). The identifier f_i must be chosen in order to ensure that $f_i^{\alpha\to\beta}$ is unique. For example, one can choose to associate the identifier f_i to the i^{th} request received by β, or equivalently to the i^{th} request sent by α.

Each activity $\alpha[a; \sigma; \iota; F; R; f]$ is characterized by:

- a *current term*: $a = b \Uparrow f_i^{\gamma\to\alpha}, b'$ to be reduced. a contains the terms corresponding to the different requests being treated separated by $\Uparrow$. The left part b is the term currently evaluated, the right one $f_i^{\gamma\to\alpha}, b'$ 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). Of course, b' can also contain continuations;

- a *store*: σ contains all the objects of the activity α;

- an *active object location*: ι is the location of the active object of activity α, thus $\sigma(\iota)$ is the active object of α;

- *futures values*: a list associating, for each served request, the corresponding future $f_i^{\gamma\to\alpha}$ and the location ι where the result of the request (also called future value) is stored: $F = \{f_i^{\gamma\to\alpha} \mapsto \iota\}$;

- *pending requests*: $R = \{[m_j; \iota; f_i^{\gamma\to\alpha}]\}$, a list of pending requests. A request can be seen as the "reification" of a method call [Smi84]. Each request $r ::= [m_j; \iota; f_i^{\alpha\to\beta}]$ consists of:

  - the name of the *target method* m_j,

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

- a *current future*: $f = f_i^{\gamma\to\alpha}$, the future associated with the request currently served. To simplify notations, f will denote any future ($f ::= f_i^{\gamma\to\alpha}$).

Empty parts of activities will be denoted by $\emptyset$. $\emptyset$ designates an empty list (futures values or pending requests) or an empty current future (when no request is currently treated).

A parallel configuration is a set of activities

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

Configurations are identified modulo reordering of activities.

Adding a request r at the end of the pending requests (R) will be denoted by $R :: r$ and taking the first request (r) at the beginning of the pending requests by $r :: R$. Similarly, $F :: \{f_i \mapsto \iota\}$ adds a new future association to the future values.

In the store, one has either objects or global references:

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

$fut(f_i^{\alpha\to\beta})$ 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*.

From a more practical point of view, when a reference to a future is encountered 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 $f_i^{\alpha\to\beta}$).

10.2 Parallel Reduction

The terms below 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 $\mathcal{M}$ only needs to contain all the method labels of the concerned (active) object.

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

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

10.2.1 More Operations on Store

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 10.1. The deep copy stops when a generalized reference is encountered. In that case, the new store contains the generalized reference. In Table 10.1, 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.

A more operational definition would consist in marking the location ι at the root of the copy and recursively all the locations that are referenced by marked locations (all locations contained in $\sigma(\iota')$ if ι' is marked). When a fix-point is reached, $copy(\iota, \sigma)$ is the part of store defining marked locations. Note that this part of store is independent: it references only locations defined in $copy(\iota, \sigma)$. In other words

$$\vdash (\iota, \sigma) \text{ OK} \Rightarrow \vdash (\iota, copy(\iota, \sigma)) \text{ OK}.$$

$$\boxed{\begin{array}{c} \iota \in dom(copy(\iota, \sigma)) \\[1ex] \iota' \in dom(copy(\iota, \sigma)) \Rightarrow locs(\sigma(\iota')) \subseteq dom(copy(\iota, \sigma)) \\[1ex] \iota' \in dom(copy(\iota, \sigma)) \Rightarrow copy(\iota, \sigma)(\iota') = \sigma(\iota') \end{array}}$$

Table 10.1: Deep copy

Figure 10.1 shows an activity α and the deep copy of a location ι. The deep copy is inside the dotted circle.

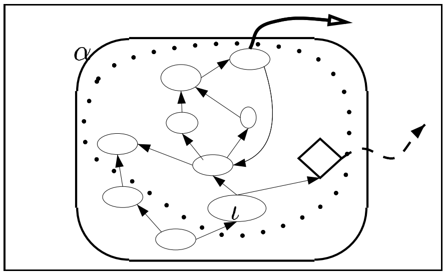


Figure 10.1: Example of a Deep copy: $copy(\iota, \sigma_\alpha)$

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

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

Copy and Merge The following operator adds the part of σ starting at location ι at the location ι' of σ' avoiding collision of locations (only ι' can be updated):

Definition 10.1 (Copy and Merge)

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

The following property is a consequence of the preceding definitions:

Property 10.1 (Copy and Merge)

$$\iota \in \mathit{dom}(\sigma') \land \iota \neq \iota' \Rightarrow \sigma'(\iota) = \mathit{Copy\&Merge}(\sigma, \iota\ ;\ \sigma', \iota')(\iota)$$

10.2.2 Reduction Rules

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

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

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

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

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

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

Table 10.2: Parallel reduction (used or modified values are non-gray)

Table 10.2 describes the reduction rules corresponding to the small step semantics of the parallel calculus. The grayed values are unchanged and unused by reduction rules. A description of these rules is given in the following:

LOCAL Inside each activity, a local reduction can occur following the rules of Table 7.1. Note that sequential rules concerning strict operations: FIELD, INVOKE, UPDATE, CLONE[10] 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 real object (ending a wait-by-necessity)[11].

NEWACT This rule activates an object. A new activity γ is created containing the deep copy of the object $\sigma(\iota)$, an empty pending requests and no futures values. A generalized reference to the created activity $AO(\gamma)$ is stored in the source activity α. The other references to ι in α are unchanged (still pointing to a passive object) because it seems more intuitive to us and it follows the ProActive behavior.

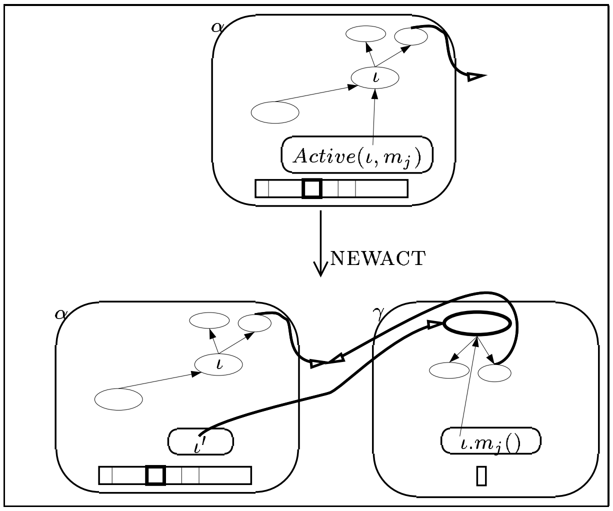


Figure 10.2: NEWACT rule

m_j specifies the service method (the first method executed). The service method has no argument and should perform $Serve$ instructions. If no method m_j is specified, a FIFO service is performed by default. When the execution of the service method is finished, the activity do not execute any more operation. Afterwards, it is only useful for updating the values of futures which have already been calculated. Such an activity only performs REPLY operations.

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

[10] cloning future is considered as a strict operation for determinism reasons (cf page 115 for more details).

[11] To be precise, an update can also transform a future reference into another future reference.

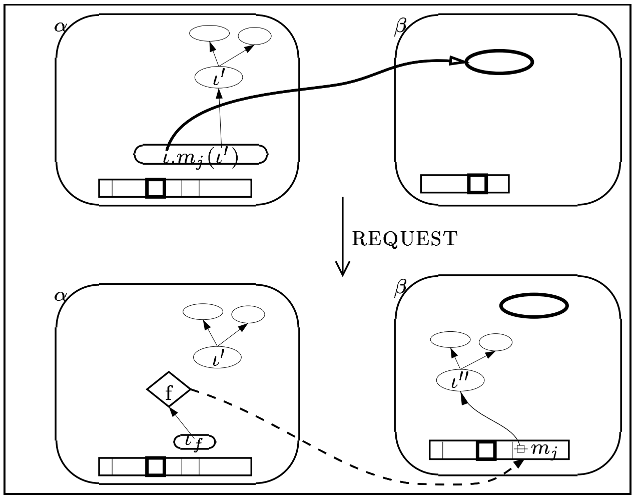


Figure 10.3: REQUEST rule

REQUEST This rule sends a new request from the activity α to the activity β (Figure 10.3). 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 (which is stored in σ_β), and the associated future $[m_j; \iota''; f_i^{\alpha\to\beta}]$ is added to the pending requests R_β.

From a practical point of view, the atomicity of this operation can be ensured by a rendez-vous: the caller process wait for an acknowledgment from the callee activity before continuing its execution. Meanwhile, if the future identifier is created by the callee, it can be returned inside the acknowledgment message.

SERVE When a call to a *Serve* primitive is encountered, SERVE serves a new request (Figure 10.4). 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: current term to be evaluated is a call to the method $(\iota.m_j(\iota_r))$. The activity is stuck until a matching request is found in the pending requests.

ENDSERVICE When the current request is finished (current term is a location), ENDSERVICE associates the location of the result to the current future f. The response is (deep) copied to prevent post-service modification of the value and the next current term and current future are extracted from the continuation (Figure 10.5).

REPLY This rule updates a total or partial future value (Figure 10.6). It replaces a reference to a future by its value. Deliberately, it is not specified when this rule should be applied. It is only required that any activity α contains a reference to a future $f_i^{\gamma\to\beta}$, and another one (β) has calculated the corresponding result. Also, some operations (e.g. INVOKE) need the real object value of some of their operands. Such operations may lead to wait-by-necessity, which can only be resolved by the update of

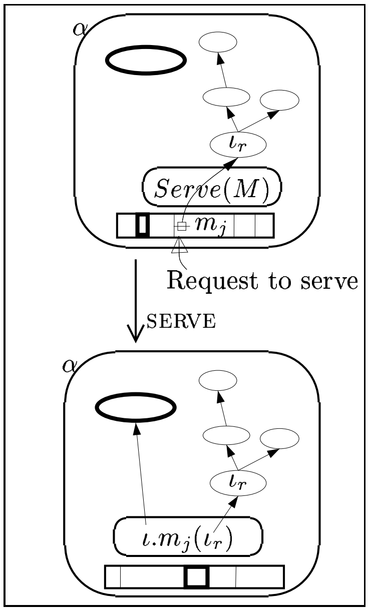


Figure 10.4: SERVE rule

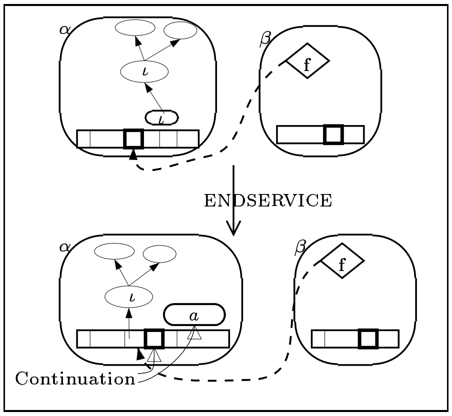


Figure 10.5: ENDSERVICE rule

the future value. Of course, 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 inside the value of a method call parameter and inside returned values (futures values).

After an update, a future cannot be removed from the futures values because the future might have proliferated in other activities; reference counting could be used to perform garbage collection of futures [LQP92, Fes01]. See 14.1 for more details.

Note that both SERVE, LOCAL and ENDSERVICE are local rules (involving a single activity). NEWACT only creates an activity and thus the only communication rules are REQUEST and REPLY.

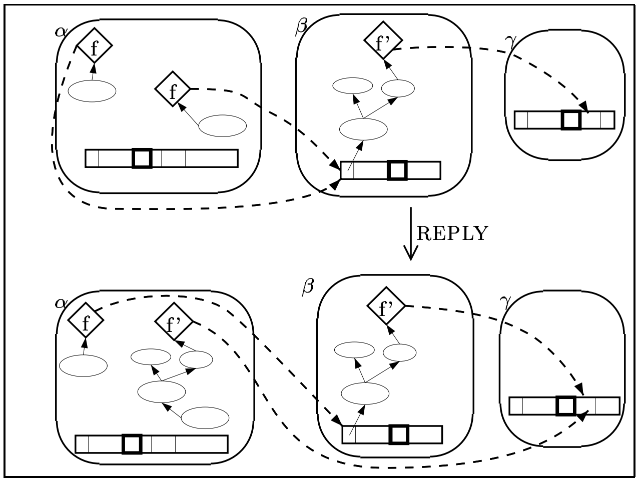


Figure 10.6: REPLY rule

Initial Configuration An *initial configuration* consists of a single activity, called *main activity*, the current term of this configuration is the source term a: $\mu[a;\emptyset;\emptyset;\emptyset;\emptyset;\emptyset]$. This activity will never receive any request. It can only communicate by sending requests or receiving replies.

10.3 Well-formedness

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

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

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

For example, in the Figure 10.7, one has, for the activity α:

$$\mathit{ActiveRefs}(\alpha) = \{\beta, \delta\}$$

$$\mathit{FutureRefs}(\alpha) = \{f^{\alpha\to\beta}, f^{\beta\to\gamma}\}$$

Definition 10.2 (Futures list) *Let $FL(\gamma)$ be the list of futures that have been calculated, the current futures (the one in the activity and all those in the continuation of the current expression) and futures corresponding to pending requests of activity γ. It is depicted by the rectangles of Figure 8.2.*

$$\begin{aligned} FL(\gamma) = \{f_i^{\beta\to\gamma}|\{f_i^{\beta\to\gamma} \mapsto \iota\} \in F_\gamma\} :: \{f_\gamma\} :: \mathcal{F}(a_\gamma) \\ :: \{f_i^{\beta\to\gamma}|[m_j, \iota, f_i^{\beta\to\gamma}] \in R_\gamma\} \end{aligned}$$
$$\mathit{where} \left\{ \begin{array}{ll} \mathcal{F}(a \Uparrow f, b) = f :: \mathcal{F}(b) & \\ \mathcal{F}(a) = \emptyset & \mathit{if}\ a \neq a' \Uparrow f, b \end{array} \right.$$

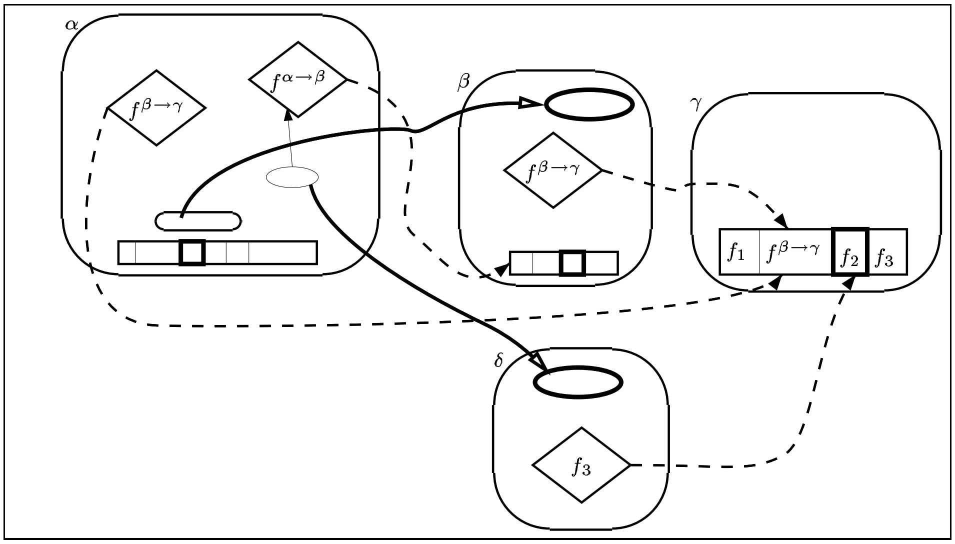


Figure 10.7: Another example of configuration

In the Figure 10.7, the futures list of the activity γ is:

$$FL(\gamma) = \{f_1, f^{\beta\to\gamma}, f_2, f_3\}$$

A parallel configuration is *well formed* if all local configurations are well formed (in the sense of Definition 7.1), 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 10.3 (Well-formedness)

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

Sequential Property 7.1 can be translated to the parallel reduction case. Indeed, it is easy to show that **parallel reduction preserves well-formedness**:

Property 10.2 (Well-formed parallel reduction)

$$\vdash P\,\text{OK} \wedge P \longrightarrow P' \quad\Longrightarrow\quad \vdash P'\,\text{OK}$$

Of course, initial configurations are well-formed and then every term obtained during reduction is also well-formed.

Chapter 11

Properties and Confluence

This chapter starts with properties on topology of objects, first between activities (Section 11.2) and then inside an activity (Section 11.3). Then a notion of compatibility between configurations (Section 11.4) and an equivalence relation on configurations (Section 11.5) are introduced. Different confluence and determinism properties form the main contribution of this chapter.

Confluence properties alleviate the programmer from studying the interleaving of instructions and communications. Very different works have been performed to ensure confluence of calculus, languages, or programs. Linear channels in π-calculus [NS97, KPT96], non interference properties in shared memory systems [Ste90], Process Networks [Kah74] or Jones' technique for creating deterministic concurrency in $\pi o \beta \lambda$ [Jon92] are typical examples. But none of them deals with a concurrent, imperative, object language with asynchronous communications.

The key property of this chapter states that the execution of a set of processes is only determined by the order of arrival of requests (Section 11.7); asynchronous replies can occur in an arbitrary order without observable consequence. This work seems, to some extent, more general and strongly related to both linearized channels in π-calculus [KPT96] and the Process Networks of Kahn [Kah74]. A specification of a set of terms: DON terms, behaving deterministically is given in Section 11.8. It can be seen as a generalization of Process Networks. Then, a set of programs that behave deterministically is identified in Section 11.9. Finally, a discussion on the different strategies for serving requests conclude this chapter.

The proofs, some technical details, and the specification of equivalence can be found in chapters 12 and 13, and also in [CHS03]. Appendix A gives another proof of determinacy in the case of a tree topology which does not necessitate to define an equivalence modulo future updates.

11.1 Notations and Hypothesis

In the following, α_P denotes the activity α of configuration P. We suppose that the *freshly allocated activities are chosen deterministically*: the first activity created by α will have the same identifier for all the possible executions. This condition is necessary

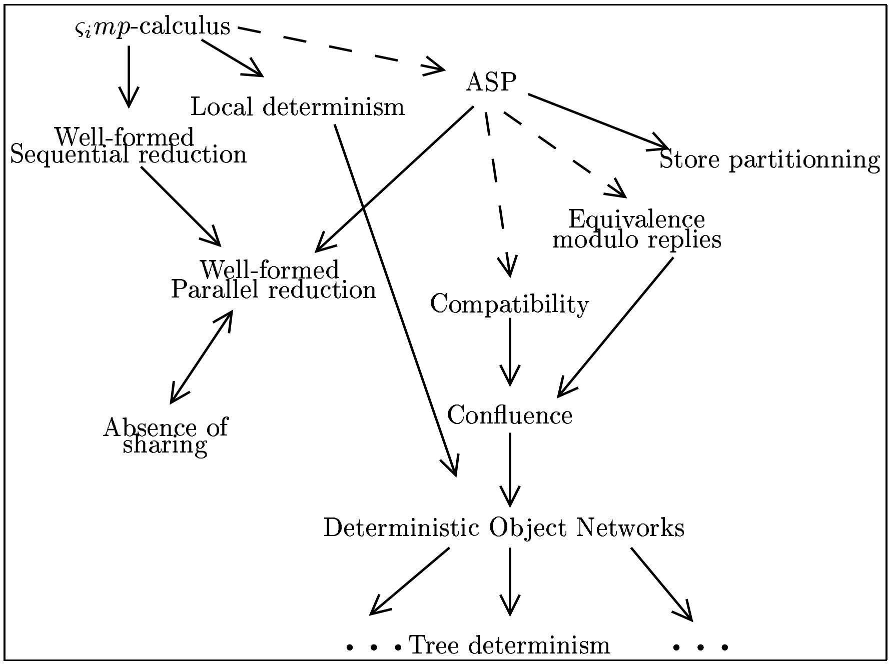

$\varsigma_i mp$-calculus
ASP
Local determinism
Well-formed Sequential reduction
Store partitionning
Equivalence modulo replies
Well-formed Parallel reduction
Compatibility
Absence of sharing
Confluence
Deterministic Object Networks
• • • Tree determinism • • •

Figure 11.1: A simple properties Diagram

to avoid renaming activities. Indeed the renaming of activities leads to complicated considerations to be sure that renamed activities are equivalent. For example, two activities may be exchanged and that may make compatible two configurations that do not converge. To simplify we have chosen to consider that activities are named deterministically. But for example one could safely add a renaming of activities to equivalence modulo futures without technical difficulty. The main interest of renaming activities seems to be the case where, after a set of interleaving requests, a term always reaches the same state and then behaves deterministically. As this case is not studied here, we chose to simplify the following by choosing a deterministic naming of activities.

For example, in order to ensure a deterministic choice of activity, an activity could be characterized by a list of integers. And when an activity [1.2.5] creates its fifth activity, this activity is called [1.2.5.5]. Of course, a more concise (but somewhat equivalent) notation should be used in practice. In the following, to keep notations concise, we still use $\alpha, \beta, \gamma, \delta \ldots$ for activities names.

Moreover, the equivalent activities have the same identifier in equivalent configurations (i.e. if P and Q are equivalent, the activity corresponding to α_P in the configuration Q is α_Q). In practice, this is directly ensured by the facts that configurations to be compared are always derived from the same source terms, and the freshly allocated activities are chosen deterministically.

Let us specify the choice of fresh futures $f_i^{\alpha\to\beta}$. We consider 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 4th request received by β comes from γ and concerns method foo, its future identifier will be $foo_4^{\gamma\to\beta}$. In the following, for notations simplicity, both m_j and f will denote method labels.

$\xrightarrow{*}$ will denote the transitive closure of $\longrightarrow$, and $\xrightarrow{T}$ will denote the application of rule T (e.g. LOCAL, REPLY…). thus, for example, $\xrightarrow{\text{REPLY}^*}$ will denote any number (≥ 0) of applications of the REPLY rule.

Potential Services

Let $\mathcal{M}_{\alpha_P}$ be a *static approximation*[12] of the set of M that can appear in the $Serve(M)$ instructions of α_P. In other words, for a given source program P_0, for each activity α created, consider that there is a set $\mathcal{M}_{\alpha_{P_0}}$ such that if α will be able to perform a $Serve(M)$ then $M \in \mathcal{M}_{\alpha_{P_0}}$. More formally:

Definition 11.1 (Potential services) *Let P_0 be an initial configuration. $\mathcal{M}_{\alpha_{P_0}}$ is any set verifying:*

$$P_0 \xrightarrow{*} P \ \wedge \ a_{\alpha_P} = \mathcal{R}[Serve(M)] \Rightarrow M \in \mathcal{M}_{\alpha_{P_0}}$$

For example, let P_0 be a source program. An activity α created by this program and that may serve either some requests on m_1 and m_2 ($Serve(m_1, m_2)$), or some requests on m_3 ($Serve(m_3)$) will be characterized by $\mathcal{M}_{\alpha_{P_0}} = \{(m_1, m_2), (m_3)\}$.

A static approximation is needed because a $Serve$ primitive can be present in an object received as a request parameter. Thus if one had a dynamic approximation of potential services $\mathcal{M}'_{\alpha_P}$. The service of a new request could modify $\mathcal{M}'_{\alpha_P}$ and, as a consequence the following potential services would be changed and that would make the compatibility relation even more dynamic.

11.2 Object Sharing

In ASP, a shared reference would be an object that could be referenced by objects belonging to different activities. The syntax of intermediate terms guarantees that there are no shared references in ASP except future and active object references.

In other words, the only generalized references are the active objects and the future references. No memory is shared because futures can only be deeply copied when they are updated in another activity (and thus are immutable) and active objects are only accessible through asynchronous method calls. This property on topology is strongly related to the fact that the only communications between activities are the requests sending and the futures updates and implies the using of deep copy in the sending of requests and the receiving of replies.

Note that this property is syntactic: it is directly ensured by ASP syntax. As a consequence, deep copies are necessary to maintain well formedness of ASP configuration without loosing the absence of sharing.

Consequently, it is adequate to choose locations locally to an activity: if some memory could be shared between activities then a location identifier local to an activity would not be sufficient.

[12] For example, it could be approximated and verified by a type system.

11.3 Futures and Parameters Isolation

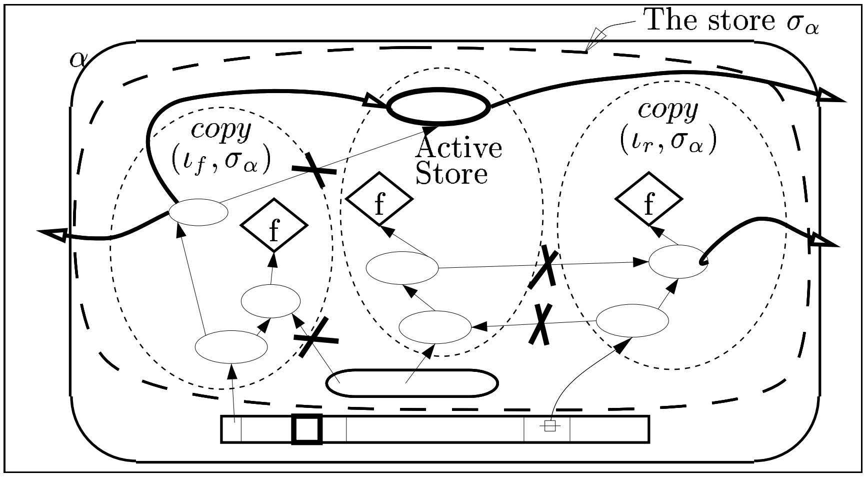

The store σ_α
α
copy
(ι_f, σ_α)
Active
Store
copy
(ι_r, σ_α)
f
f
f

Figure 11.2: Store Partitioning: future value, active store, request parameter

The following property states that the value of each future and each request parameter is situated in an isolated part of a store. Figure 11.2 illustrates the isolation of a future value (on the left) and a request parameter (on the right).

Property 11.1 (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:

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

where $\bigoplus$ is the disjoint union.

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 the parameters partitions is the update of a calculated future value.

Equivalently, one could consider that, instead of a single store by activity, one could have a local store and specialized stores for each future value and each method parameter. In that case, one would need to add two REPLY rules: one that updates futures inside a future value and another one that updates futures inside a method parameter. In fact, this two rules are not necessary to obtain a coherent calculus but without them, we could not express all the future updates strategies.

11.4 Configuration Compatibility

This section introduces notations and concepts that will be useful for establishing confluence of terms in 11.7. Informally, two configurations are compatible if, for all activities present in both, the served, current and pending requests of one is a prefix of the same list in the other. Moreover, if two requests can not interfere, that is to say if no $Serve(M)$ can concern both requests, then these requests can be safely exchanged. The compatibility definition is justified by the fact that the order of evaluation is entirely defined by the order of request sending. More precisely, Theorem 11.1 states that the order of activities sending requests to a given activity determines the behavior of the program. Or in other words that compatible configurations are confluent. This means that *futures updates and imperative aspects of ASP do not act upon the final result of evaluation.* This property can be considered as the main contribution of this study.

Definition 11.2 (Request Sender List) *The request sender list (RSL) is the list of request senders in the* order *the requests have been* received *and indexed by the invoked method.*
The i^{th} element of RSL_α is defined by:

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

The list of futures that have or will be calculated by activity α is $FL(\alpha)$ and has been defined in 10.3. The RSL list is obtained from *futures* associated to *served requests*, *current requests* and *pending requests*. Moreover, if n requests have been received by α, then for each i between 1 and n $(RSL_\alpha)_i$ is well defined. It is important to note that the order of this list is the order of requests *arrivals*; and thus, for example some entries corresponding to served requests can appear after some current or pending requests.

In the examples, for simplicity of notations, in an activity β we will denote by $\delta.foo(b)$ when there is ι such that $\iota.foo(b)$ and $\sigma_\beta(\iota) = AO(\beta)$.

The example of the Figure 11.3 shows four activities. This state (calculated current and pending futures of activity δ) is obtained with the following sequence:

- the activity β invokes a bar request on the activity δ: $\delta.bar()$,
- α performs a $\delta.foo()$,
- γ performs two consecutive $\delta.gee()$,
- α performs a $\delta.foo()$,
- β performs two consecutive $\delta.foo()$,
- γ performs a $\delta.gee()$.

In this example, the RSL of the activity δ is:

$$RSL(\delta) = \beta^{bar} :: \alpha^{foo} :: \gamma^{gee} :: \gamma^{gee} :: \alpha^{foo} :: \beta^{bar} :: \beta^{bar} :: \gamma^{gee}$$

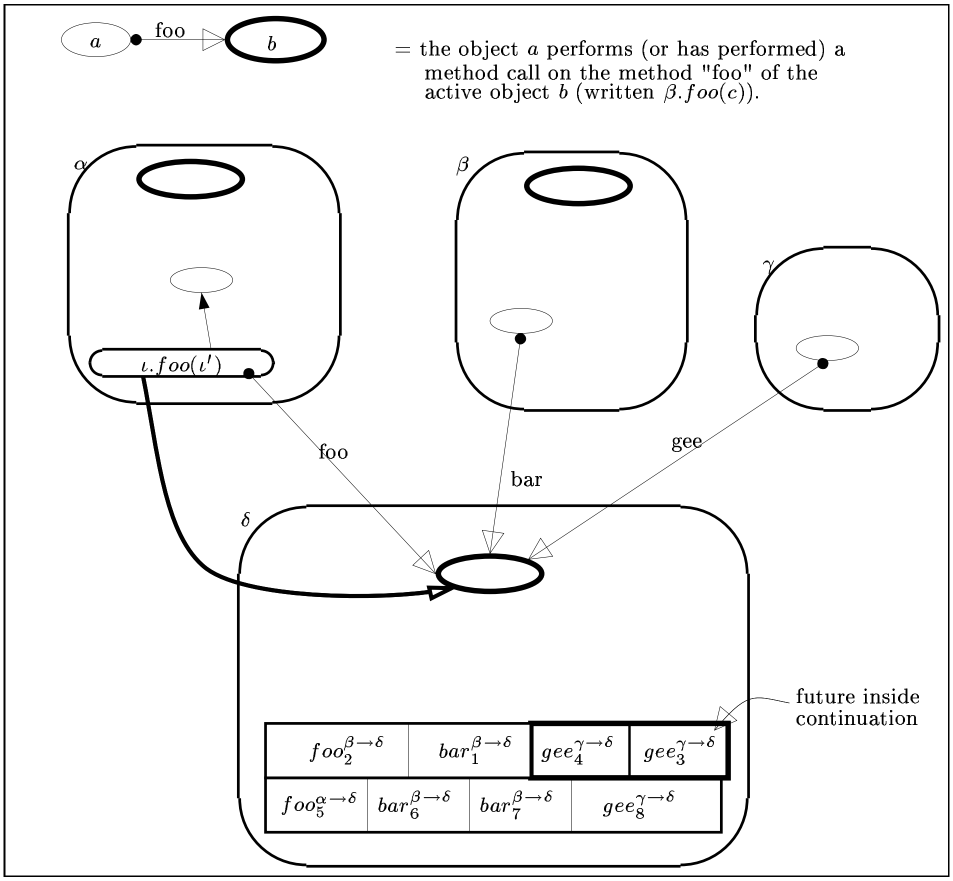

a
foo
b
= the object a performs (or has performed) a method call on the method "foo" of the active object b (written β.foo(c)).
α
β
γ
ι.foo(ι′)
foo
bar
gee
δ
future inside continuation
$foo_2^{\beta\to\delta}$
$bar_1^{\beta\to\delta}$
$gee_4^{\gamma\to\delta}$
$gee_3^{\gamma\to\delta}$
$foo_5^{\alpha\to\delta}$
$bar_6^{\beta\to\delta}$
$bar_7^{\beta\to\delta}$
$gee_8^{\gamma\to\delta}$

Figure 11.3: Example of RSL

Let $RSL_\alpha\big|_M$ represent the *restriction* of the RSL_α list to the set of labels M. For instance $(\alpha^{foo} :: \beta^{bar} :: \gamma^m :: \mu^{foo})\big|_{foo,m} = \alpha^{foo} :: \gamma^m :: \mu^{foo}$.

In the example of the Figure 11.3, one has:

$$RSL(\delta)\big|_{foo,bar} = \beta^{bar} :: \alpha^{foo} :: \alpha^{foo} :: \beta^{bar} :: \beta^{bar}$$

For a FIFO service, and if no service is performed while another request is being served, the order of requests can not be changed when they are served. Thus the RSL is directly obtained from the concatenation of the futures values (in the order they have been calculated), the unique current future, and the pending requests in the order they arrived. Finishing the current service (ENDSERVICE) and serving a new request (SERVE) will put the current future at the end of the futures values and take the first pending request future as current future. Indeed if $f_n^{\beta\to\alpha}$ is the current future then $f_1^{\delta\to\alpha}\ldots f_{n-1}^{\gamma\to\alpha}$ correspond to calculated futures and $f_{n+1}^{\delta\to\alpha}\ldots$ correspond to pending requests.

Definition 11.3 (RSL comparison $\trianglelefteq$) *RSLs are ordered by the prefix order on activities:*

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

Two RSLs are compatible if they are comparable. That means that one of the two RSL is simply the beginning of the other one.

Definition 11.4 (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*

$$\begin{aligned} RSL_\alpha \bowtie RSL_\beta \quad &\Leftrightarrow \quad RSL_\alpha \sqcup RSL_\beta \text{ exists} \\ &\Leftrightarrow \quad (RSL_\alpha \trianglelefteq RSL_\beta) \vee (RSL_\beta \trianglelefteq RSL_\alpha) \end{aligned}$$

Two configurations are said to be compatible if all the restrictions of their RSL that can be served are compatible (have a least upper bound). Suppose that configurations to be compared derive from the same source term. Thus there is P_0 such that $P_0 \xrightarrow{*} P$ and $P_0 \xrightarrow{*} Q$ and then the compatibility of P and Q is defined by:

Definition 11.5 (Configuration compatibility: $P \bowtie Q$)
If P_0 is an initial configuration such that $P_0 \xrightarrow{*} P$ and $P_0 \xrightarrow{*} Q$

$$P \bowtie Q \Leftrightarrow \forall \alpha \in P \cap Q, \forall M \in \mathcal{M}_{\alpha_{P_0}}, RSL_{\alpha_P}\big|_M \bowtie RSL_{\alpha_Q}\big|_M$$

Intuitively, it means that RSLs are in a compatible state if the application of *Serve* operations present in the code will lead to equivalent lists of calculated futures. Then, two configurations are compatible if for every activity α present in both configurations, their RSLs are in a compatible state (served, current and pending requests).

Following the RSL definition (Definition 11.2) the configuration compatibility only relies on the arrival order of requests; the future list (FL) order (Definition 10.2), potentially different on served and current requests, does not matter. Indeed, the behavior of an activity is fully determined by the arrival order of requests.

In the general case, *Serve* operations can be performed while another request is being served; then the relation between RSL order and FL order can only be determined by a precise study. If no *Serve* operation is performed while another request is being served (only the service method performs *Serve* operations), then all the restrictions (to potential services) of the RSL and the FL are in the same order. In the FIFO case, the FL order and the RSL order are the same.

Observe that the restriction of RSLs in Definition 11.4 is only useful for the part of the RSL containing pending requests. If two calculated futures are exchanged (even if they correspond to different labels) they correspond to two requests that have been served in a different order. And thus, they correspond to a non-confluent program. Indeed, these two services could have modified the state of the object concurrently. In fact, such exchanged futures means that there are two RSLs (of another activity) that are not compatible.

For example if the method *foo* sets a field to the value 1 and the method *bar* sets the same field to the value 2, then, if the calculated futures are

$\{foo_1^{\alpha\rightarrow\gamma} :: bar_2^{\alpha\rightarrow\gamma}\}$ then the field has the value 2. Whereas if the calculated futures are $\{bar_2^{\alpha\rightarrow\gamma} :: foo_1^{\alpha\rightarrow\gamma}\}$ then the field has the value 1 even if there is no $Serve(foo, bar)$[13]. Such a configuration can be obtained, if the service method performs a $Serve(foo); Serve(bar)$ in one case and $Serve(bar); Serve(foo)$ in the other case. For example, because the activity that has created this activity has a non-deterministic behavior and, two RSLs (leading to the two service methods) that are not compatible.

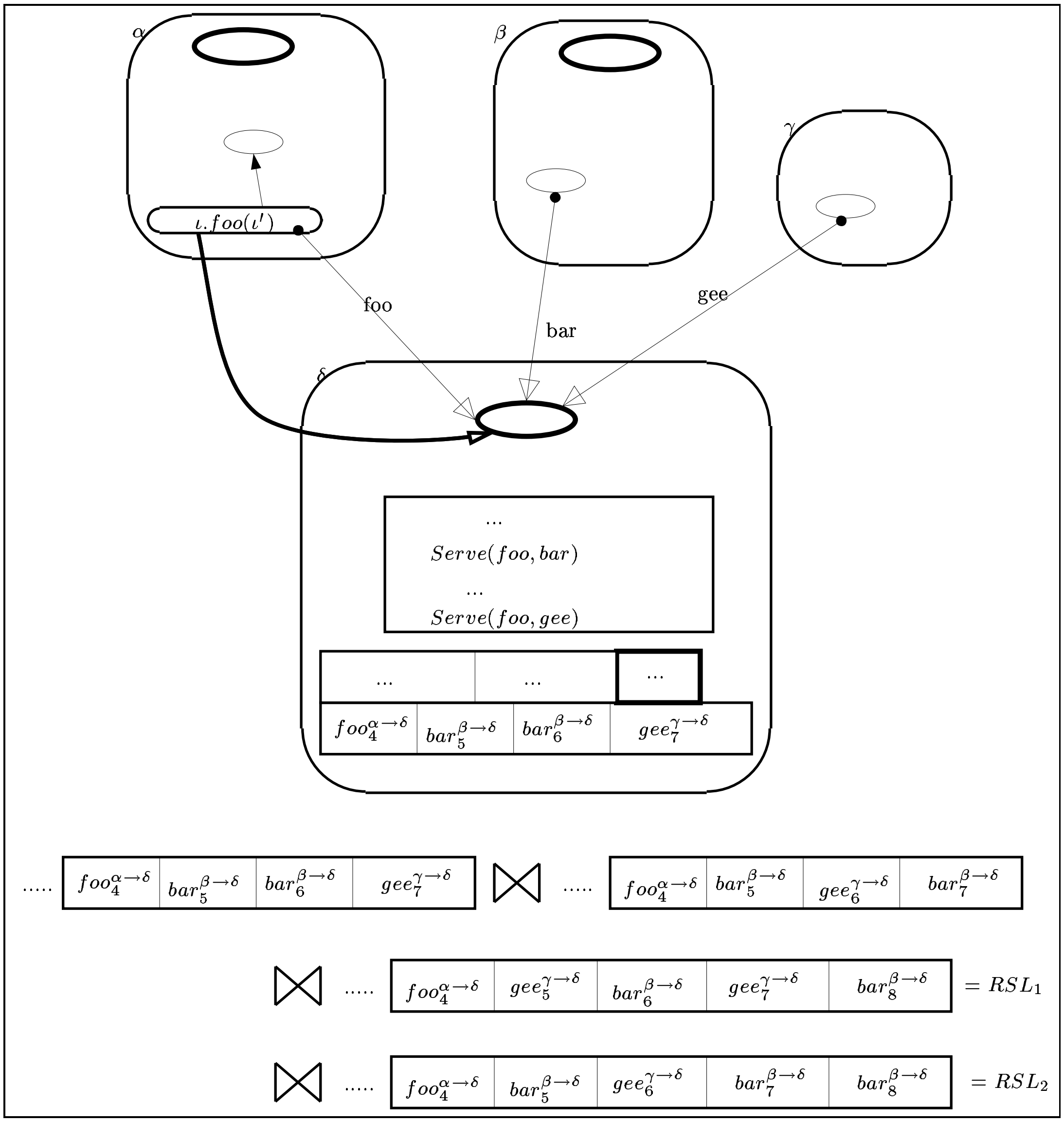

α
β
γ
$\iota.foo(\iota')$
foo
bar
gee
δ
...
$Serve(foo, bar)$
...
$Serve(foo, gee)$
... ...
$foo_4^{\alpha\rightarrow\delta}$ $bar_5^{\beta\rightarrow\delta}$ $bar_6^{\beta\rightarrow\delta}$ $gee_7^{\gamma\rightarrow\delta}$
..... $foo_4^{\alpha\rightarrow\delta}$ $bar_5^{\beta\rightarrow\delta}$ $bar_6^{\beta\rightarrow\delta}$ $gee_7^{\gamma\rightarrow\delta}$ $\bowtie$ ..... $foo_4^{\alpha\rightarrow\delta}$ $bar_5^{\beta\rightarrow\delta}$ $gee_6^{\gamma\rightarrow\delta}$ $bar_7^{\beta\rightarrow\delta}$
$\bowtie$ ..... $foo_4^{\alpha\rightarrow\delta}$ $gee_5^{\gamma\rightarrow\delta}$ $bar_6^{\beta\rightarrow\delta}$ $gee_7^{\gamma\rightarrow\delta}$ $bar_8^{\beta\rightarrow\delta}$ $= RSL_1$
$\bowtie$ ..... $foo_4^{\alpha\rightarrow\delta}$ $bar_5^{\beta\rightarrow\delta}$ $gee_6^{\gamma\rightarrow\delta}$ $bar_7^{\beta\rightarrow\delta}$ $bar_8^{\beta\rightarrow\delta}$ $= RSL_2$

Figure 11.4: Example of RSL compatibility

[13] Moreover, in that case, the two RSLs are the same!

The Figure 11.4 extends the example of Figure 11.3. Suppose, the potential services of the activity δ is:

$$\mathcal{M}_{\delta_{P_0}} = \{\{bar,\ foo\}, \{gee,\ foo\}\}$$

that is to say, the only calls to $Serve$ primitive are $Serve(foo,\ bar)$ and $Serve(foo,\ gee)$. The two last RSLs are:

$$RSL_1(\delta) = \alpha^{foo}.\gamma^{gee}.\beta^{bar}.\gamma^{gee}.\beta^{bar}$$

$$RSL2(\delta) = \alpha^{foo}.\beta^{bar}.\gamma^{gee}.\beta^{bar}.\beta^{bar}$$

Then all the configurations having the RSLs at the bottom of the diagram are compatible. For example, for the two last RSLs:

$$RSL_1(\delta)\big|bar, foo = \ldots\alpha.\beta.\beta \bowtie \ldots\alpha.\beta.\beta.\beta = RSL_2(\delta)\big|bar, foo$$
$$RSL_1(\delta)\big|gee, foo = \ldots\alpha.\gamma.\gamma \bowtie \ldots\alpha.\gamma = RSL_2(\delta)\big|gee, foo$$

11.5 Equivalence Modulo Replies

First, let us generalize the equivalence relation $\equiv$ defined previously (Definition 7.2). Let $\equiv$ denote the equivalence modulo renaming of locations and futures. Furthermore $\equiv$ takes into account the RSL compatibility. Indeed, the equivalence on pending requests allows 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 these allowed permutations, equivalent configurations are composed of equivalent pending requests in the same order.

Let us now introduce an equivalence relation that is insensitive to the update of futures.

Equivalence modulo future replies ($P \equiv_F Q$) is an extension of $\equiv$ authorizing the update of some calculated futures. This is equivalent to considering the references to futures already calculated as equivalent to local references to the part of the store which is the (deep copy of the) future value. Or, in other words, a future is equivalent to a part of the store if this part of the store is equivalent to the store which is the (deep copy of the) future value (provided the updated part does not overlap with the remainder of the store).

Figure 11.5 shows two equivalent terms. The second is obtained by updating the future f (applying a REPLY rule).

Chapter 12 and [CHS03] formally defines equivalence $\equiv_F$ and proves its properties. A brief explanation of this formal definition is given below.

First, let Θ be a renaming of some futures identifiers from configuration P to configuration Q.

Let L be a path in the tree formed by the term a that follows references inside the local store. Paths contain field access, dereferencing, ...but are insensitive to the dereferencing of a future reference. Following a path inside α is denoted by $\overset{\alpha}{\mapsto}_L$;

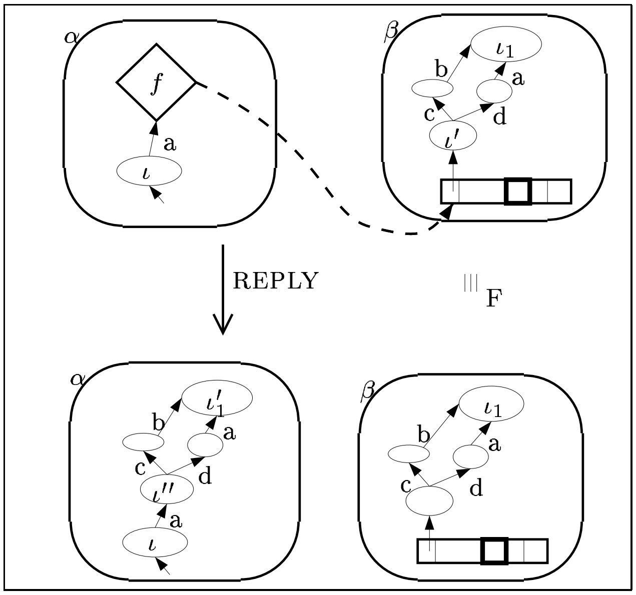


Figure 11.5: Two terms equivalent modulo future update

and following a path containing futures references and thus possibly inside different activities is denoted by $\overset{\alpha*}{\longmapsto}_L$.

In this chapter, we will only informally describe major points that have to be verified by this equivalence.

First let us illustrate what "insensitive to the dereferencing of a future reference" means. In Figure 11.5, suppose in the second configuration $\sigma_\alpha(\iota) = [a = \iota'']$. Then in the second configuration:

$$\alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha}{\longmapsto}_{ref.a} \iota''$$

Which corresponds in the first one to (with $F_\beta(f) = \iota'$):

$$\alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha*}{\longmapsto}_{ref.a} \iota'$$

An important condition is that aliased objects are the same in both configurations, that is if, in the second configuration there are two paths $L_1 = ref.a.ref.c.ref.b$ and $L_2 = ref.a.ref.d.ref.a$ leading to the same location (ι'_1) without following futures references:

$$\exists L_1, L_2 \alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha}{\longmapsto}_{L_1} \iota'_1 \wedge \alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha}{\longmapsto}_{L_2} \iota'_1$$

Then the same two paths lead to the same location in the first configuration with the constraint that the end of these paths does not follow futures references:

$$\exists L, L', L'_1, L'_2, \iota' \begin{cases} L_1 = L.L'_1 \wedge L_2 = L'.L'_2 \wedge L_1 \neq \emptyset \\ \alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha*}{\longmapsto}_{L} \iota' \overset{\beta}{\longmapsto}_{L'_1} \iota_1 \\ \alpha \overset{\alpha}{\longmapsto}_{\ldots} \iota \overset{\alpha*}{\longmapsto}_{L'} \iota' \overset{\beta}{\longmapsto}_{L'_2} \iota_1 \end{cases}$$

That is verified with:$L = L' = ref.a.ref$, $L'_1 = c.ref.b$ and $L'_2 = d.ref.a$.

That is to say those paths are also aliased in the other configuration and the alias occurs (at least partially) in the last activity encountered:

$$\begin{cases} c \overset{\beta}{\longmapsto}_{L'_1} \iota_1 \\ c \overset{\beta}{\longmapsto}_{L'_2} \iota_1 \end{cases}$$

Furthermore, alias condition is necessary to avoid identifying the first and the last configuration of Figure 11.6.

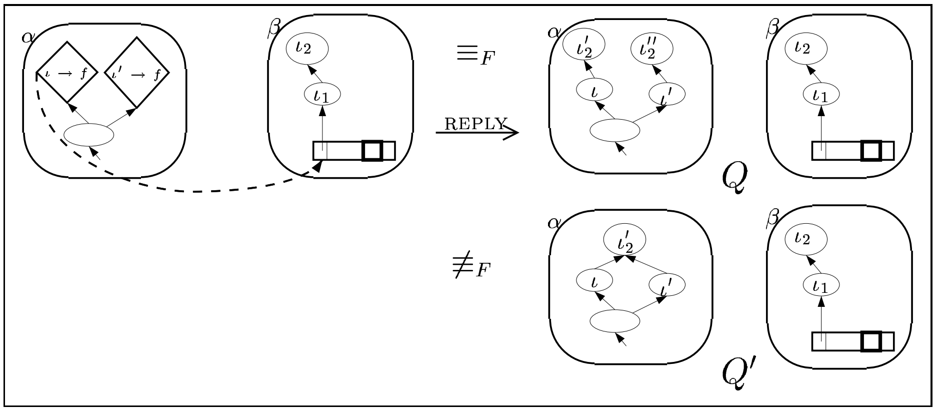


Figure 11.6: Another example

As explained informally in this section, two configurations only differing by some future updates are equivalent:

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

More precisely, we have the following sufficient condition for equivalence modulo future replies:

$$\begin{cases} P_1 \overset{\text{REPLY}}{\longrightarrow} P' \\ P_2 \overset{\text{REPLY}}{\longrightarrow} P' \end{cases} \quad \Rightarrow \quad P_1 \equiv_F P_2$$

But this condition is not necessary as it does not deal with mutual references between futures. On the Figure 11.7 the two bottom configurations are equivalent but there is no configuration P' such that:

$$P_1 \overset{\text{REPLY}}{\longrightarrow} P' \wedge P_2 \overset{\text{REPLY}}{\longrightarrow} P'$$

Indeed all configurations derived from P_1 will only contain future f_1 and similarly, all configurations derived from P_2 will only contain future f_2.

In this example, one can still conclude because of the transitivity of $\equiv_F$, and P_1 and P_2 are derived by REPLY rules from a common configuration, but, in the general case, it is difficult to find this common configuration when one only knows the configurations P_1 and P_2. In the most general case, it seems difficult to infer which futures must be "un-updated" to find a common source configuration.

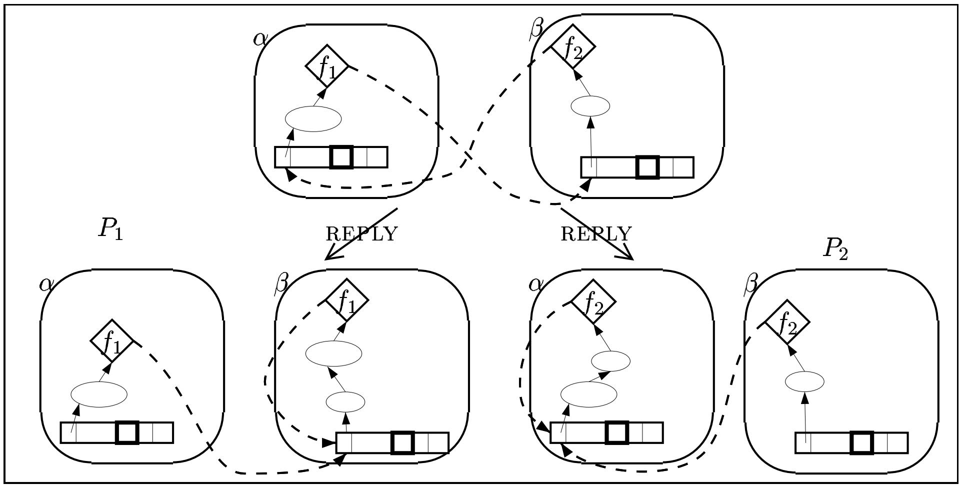


Figure 11.7: Updates in a cycle of futures

11.6 Properties of Equivalence Modulo Replies

In the following, some important properties of $\equiv_F$ are exhibited. Let T be any parallel reduction.

$$T \in \{\textsc{local}, \textsc{newact}, \textsc{request}, \textsc{serve}, \textsc{endservice}, \textsc{reply}\}$$

Then let us denote by $\Longrightarrow$ the reduction $\longrightarrow$ preceded by some applications of the REPLY rule.

Definition 11.6 (Parallel Reduction modulo future updates)

$$\overset{T}{\Longrightarrow} \quad = \quad \xrightarrow{\textsc{reply}*}\xrightarrow{T} \textit{ if } T \neq \textsc{reply} \textit{ and } \xrightarrow{\textsc{reply}*} \textit{ if } T = \textsc{reply}$$

Informally, this reduction allows to achieve any replies necessary to the application of another transition. If the rule is REPLY then $\Longrightarrow$ represents any number of application of the REPLY rule (including zero). The following property is verified:

Property 11.2 (Equivalence modulo futures and reduction)

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

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, after several REPLY rules, the application of a given rule on equivalent terms preserves equivalence (see Section 12.6 for details).

The following corollary states that one can actually apply several REPLY before the reduction $P \xrightarrow{T} Q$ without any consequence on the Property 11.2:

Property 11.3 (Equivalence and generalized parallel reduction)

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

In the following sections, we present sufficient conditions for confluence of ASP configurations and determinism of their behavior.

11.7 Confluence

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

Definition 11.7 (Confluent Configurations: $P_1 \curlyvee P_2$)

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

The next property states that if, from a given term, one obtains two compatible configurations, then these configurations are confluent.

Theorem 11.1 (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 principles of the confluence theorem can be summarized by: non-determinism can only originate from the application of two interfering REQUEST rules on the same destination activity; the order of updates of futures never has any influence on the reduction of a term. The only constraint to the moment when a REPLY must occur is a wait-by-necessity on that future. In fact even if this theorem is natural, it allows a lot of asynchronism and proves that the mechanism of futures is rather powerful, even in an imperative calculus.

Furthermore, the order of requests does not matter if they can not be involved in the same *Serve* primitive, thus some requests on different methods do not interfere; and they can be safely exchanged. This is expressed by the compatibility relation.

On the contrary, 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 modulo future update. Of course, another equivalence relation could be found for which compatibility between terms would not be necessary for confluence. For example, consider a strictly functional server which has no internal state and only serves request. Such a server could be considered as deterministic even if it receives requests in an undetermined order.

The proof of the Theorem 11.1 is presented in Chapter 13. It 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 Chapter 13 and [CHS03] for more details). Thus the configurations can be reduced to a common one (modulo future replies equivalence).

The next sections identify sets of terms that behave deterministically.

11.8 Deterministic Object Networks

The work of Kahn and MacQueen on process networks [KM77] suggested us the following properties ensuring the determinacy of some programs. In process networks, the determinacy is ensured by the facts that channels have only one source, and destinations read the data independently (values are broadcasted to all destination processes). And, most importantly, the order of reading on different channels is fixed for a given program, and the reading of an entry in the buffer is blocking.

In ASP, *Serve* being a blocking primitive, if at some time, two activities can send concurrently to the same activity a request on a given method, then a conflict appears and the reduction is not confluent. A conflict also appears when the two activities send requests on two method labels m_1 and m_2 that appear in the same $Serve(M)$ ($m_1 \in M$ and $m_2 \in M$).

In other words, if at any time two activities can not send concurrently a request to the same third activity; or if such requests can be sent concurrently then they concerns two methods that do not appear in the same $Serve(M)$; then there is no interference and the reduction is confluent.

In order to formalize this principle, Deterministic Object Networks (DON) are defined below.

Definition 11.8 (DON) *A configuration P, derived from an initial configuration P_0, is a Deterministic Object Network -$DON(P)$- if :*

$$DON(P) \Leftrightarrow \left(\begin{array}{r} P \xrightarrow{*} Q \Rightarrow \forall \alpha \in Q,\ \forall M \in \mathcal{M}_{\alpha_{P_0}}, \exists^1 \beta \in Q, \exists m \in M, \\ a_\beta = \mathcal{R}[\iota.m(\ldots)] \wedge \sigma_\beta(\iota) = AO(\alpha) \end{array} \right)$$

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

A program is a deterministic object network if at any time, for each set of labels M on which α can perform a *Serve* primitive, only one activity can send a request on methods of M.

For example the Sieve of Eratosthenes examples verify $DON(P)$. But if the first one is easy to verify statically because in Figure 9.3 the object dependence graph forms a tree.

The second example seems much more difficult to verify because in Figure 9.5 the object dependence graph is no longer a tree. This comes from the fact that all sieve objects keep a reference to the Display object. However dynamically, at each time a single one will be able to send a request on *Display* object.

From the definition of DON one can conclude easily that DON terms always reduce to compatible configurations :

Property 11.4 (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: uniqueness of request sender (β) for every target (α).

This can be proved by contradiction: If two incompatible RSLs could be obtained in γ, one would have two requests R_1 and R_2 at the same place in the two RSLs. Moreover, one has $R_1 = [foo, \iota, f1_i^{\alpha\rightarrow\gamma}]$ and $R_2 = [bar, \iota', f2_{i'}^{\beta\rightarrow\gamma}]$ and $foo \in M$ and $bar \in M$ for a given $Serve(M)$. Then there would exist another term Q' such that $P \xrightarrow{*} Q'$ and in Q' the two concurrent requests R_1 and R_2 can be sent concurrently from β and γ to the activity α:

$$a_\beta = \mathcal{R}[\iota.foo(\ldots)] \wedge \sigma_\beta(\iota) = AO(\alpha)$$

and

$$a_\gamma = \mathcal{R}[\iota'.bar(\ldots)] \wedge \sigma_\gamma(\iota') = AO(\alpha)$$

And P would not be a DON term.

Thus, the reduction of DON terms always leads to the same RSLs, for all orders of request sending: requests are always served in the same order.

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

Theorem 11.2 (DON determinism)

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

Figure 11.8 illustrates the fact that term that does not verify the DON definition can lead to undeterministic behavior. In P two request can be sent concurrently to δ, and we obtain two configuration P_1 and P_2 that are not confluent (and have uncompatible RSLs).

As explained before, the two examples of sieve of Eratosthenes (Figures 9.2 and 9.4) are both DON and thus their execution is deterministic.

The DON definition is dynamic but could be approximated by statically determining the set of active objects that can send a request on method m of activity α. This means that one first has to statically decide whether an object is active or not (by static analysis). Note also that a static approximation of reachable configurations Q is needed. Such static analyses have been (heavily) studied in the literature. However, the dynamic nature of the DON property is rather intrinsic and unavoidable: the dependencies between objects can evolve over time; over different periods, different

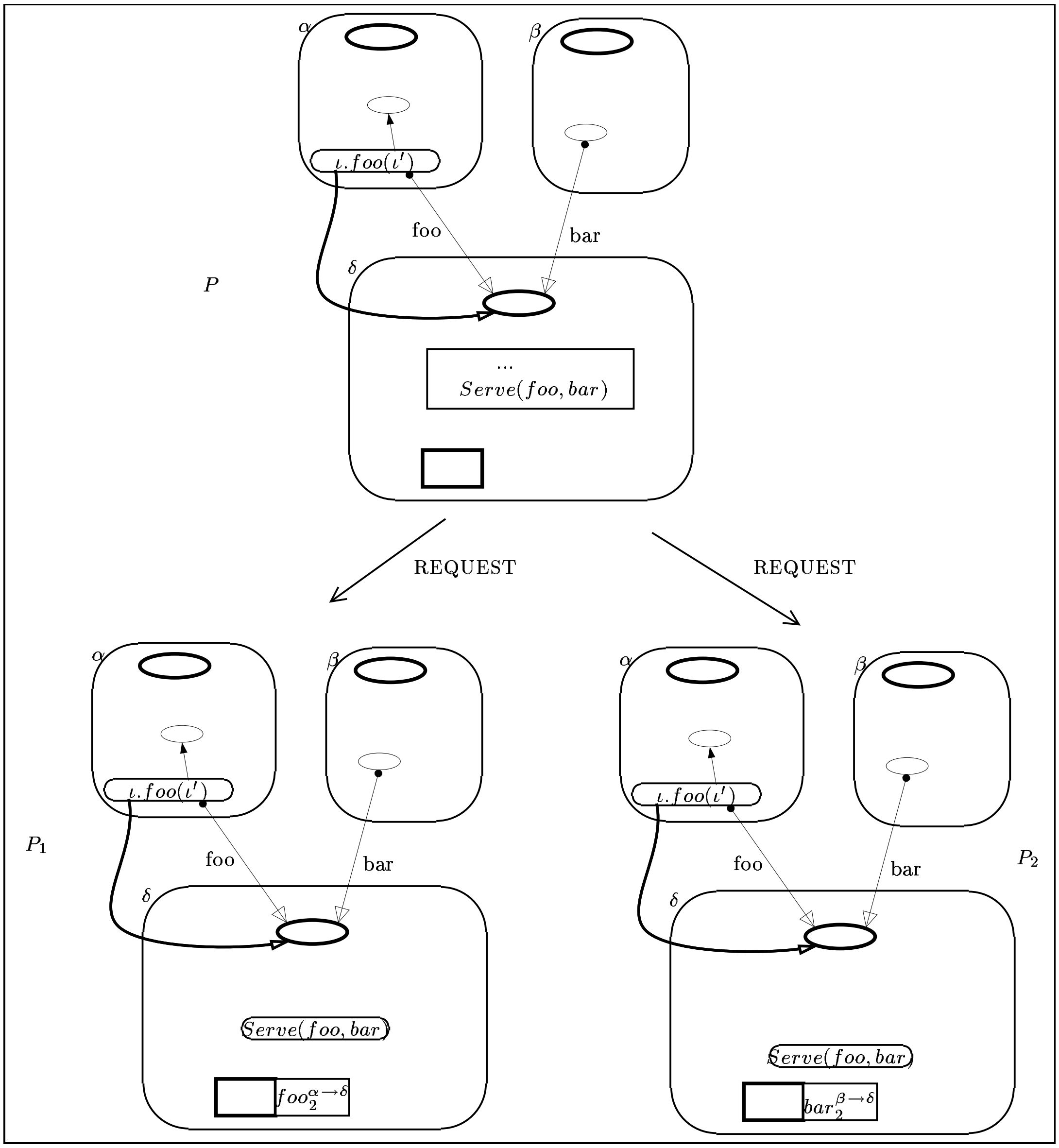


Figure 11.8: a non-DON term

activities can send requests to a given one. These changes in object topology can be related to reconfigurations in Process Networks [KM77].

This section showed that one can identify a sub-calculus (DON terms) of ASP that is deterministic and inspired from process networks. The similarities between DON terms and Process Networks are further studied in Section 15.6. The idea is that, for a DON term, a *channel* is a set $M \in \mathcal{M}_\alpha$ for a given activity α. At any time, only one activity can send requests on this channel (because of the DON definition); remember that the process networks are based on the uniqueness of sender for each channel. Such ASP channels have a behavior similar to the process networks ones and one could simulate such channels by process networks channels. It is important to note that this definition of channels can be considered as a first step towards a static approximation of DON: if one can statically determine such channels and prove that two channels never interfere in the same *Serve* primitive of the same activity then one obtains a confluent sub-calculus that is statically verifiable.

11.9 Tree Topology Determinism

In this section a simple static approximation of DON terms is performed. It has the advantage to be correct even in the highly interleaving case of FIFO services.

The *request flow graph* is the graph where nodes are activities and there is an edge between two activities if one activity has previously sent requests to another one ($\alpha \rightarrow_R \beta$ if α has sent a request to β). Such a graph is only growing with time.

It is easy to prove that if, at every step of the reduction, the request flow graph is a tree then the term verifies the DON definition. Indeed, at any time a unique activity can send a request to a given one. Thus, for each $\alpha \in Q$, RSL_α contains occurrences of at most one activity (the same activities for all possible reductions): $RSL_\alpha = \{\beta.\beta.\beta\ldots\}$. Then for all Q and R such that $P \xrightarrow{*} Q \wedge P \xrightarrow{*} R$, Q and R are compatible ($Q \bowtie R$) as they can only differ by the existing activities and the length of their RSLs (of course, $\beta.\beta \bowtie \beta.\beta.\beta$). As a consequence:

Theorem 11.3 (Tree Determinacy)

If, at every step of the reduction, the request flow graph forms a set of trees then the reduction is deterministic.

What is important here, is to see that the parts of reduction where request flow graph forms a tree are deterministic. Indeed, upon a global synchronization, one can reset the request flow graph to an empty one. Then, in order to prove that a term is confluent, one only has to 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.

11.10 A Deterministic Example: The Binary Tree

The Binary Tree of Figure 9.1 verifies Theorem 11.3 and thus behaves deterministically provided, at each time, at most one client can add new nodes.

Figure 11.9 illustrates the evaluation of the term:

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

This term behaves in a deterministic manner whatever order of replies occurs.

Now consider that the result of a preceding request is used to create a new node (dotted lines in Figure 11.9):

$let\ tree = (BT.new).add(3,4).add(2,3).add(5,6).add(7,8)\ in$
$\quad let\ Client = [a = tree.search(5), b = tree.search(3)]\ in$
$\qquad Client.b := tree.search(7);\ tree.add(1, Client.a)$

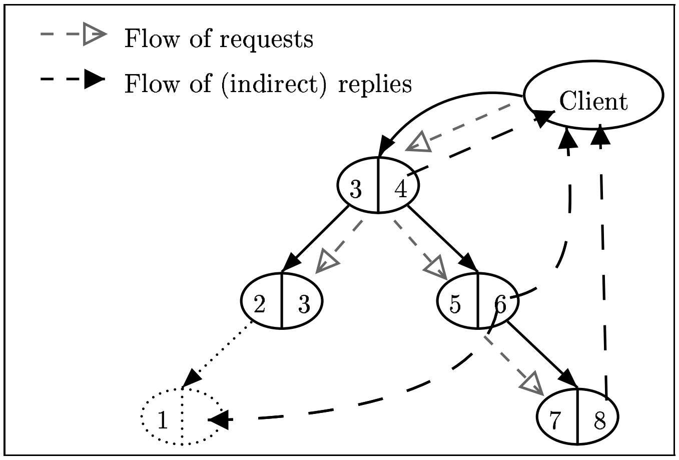


Figure 11.9: Concurrent replies in the binary tree case

Then the future update that fills the node indexed by 1 can occur at any time since the value associated to this node is not needed. Consequently a future update can occur directly from the node number 5 to the node number 1.

11.11 Another deterministic example

To show the interest of DON theorem (Theorem 11.4), let us examine the Fibonacci numbers example of the Section 9.4.

In the following, we will show that this example is fully deterministic but does not verify a tree topology.

The Figure 11.10 shows the Fibonacci numbers example of Figure 9.8 with, the RSLs inside the bottom rectangles of each activity.

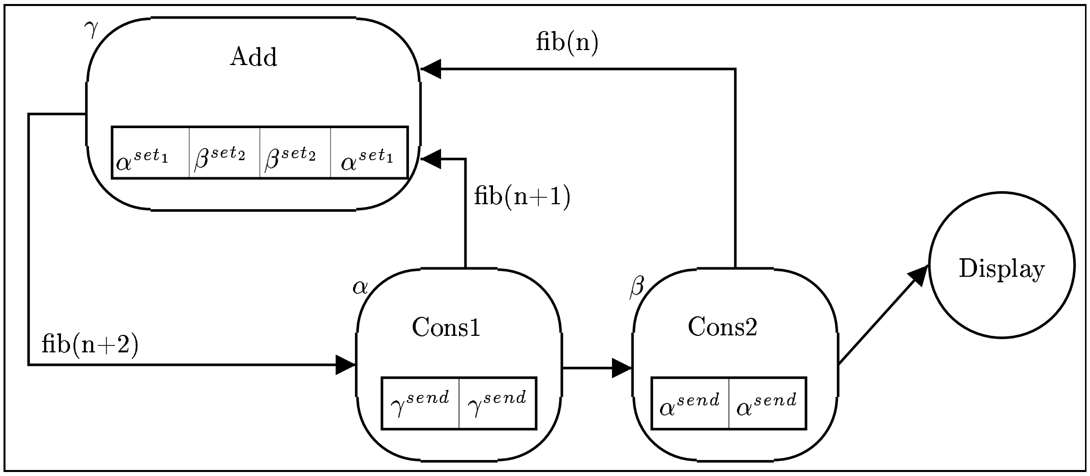


Figure 11.10: Fibonacci Numbers RSLs

The most interesting activity is the one containing the *Add* object, we will focus on this one. In the most general case, its RSL is of the form[14]:

$$RSL(\gamma) = \emptyset \mid \beta^{set_2} \mid \alpha^{set_1} \mid \\ \alpha^{set_1} :: \beta^{set_2} :: RSL_{add} \mid \beta^{set_2} :: \alpha^{set_1} :: RSL_{add}$$

The potential services of activity γ is:

$$\mathcal{M}_{\gamma_{ExempleFibo}} = \{\{set_1\}, \{set_2\}\}$$

Thus, in all possible executions, one has:

$$RSL_{add}\big|set_1 = \alpha^*$$

$$RSL_{add}\big|set_2 = \beta^*$$

But it seems difficult to prove formally and statically that the RSL has the form given above.

By comparison, let us prove that the Fibonacci numbers example verifies the DON definition.

It is easier to verify that only $Cons_1$ invokes the set_1 method on the *Add* object. Similarly, only $Cons_2$ invokes the set_2 method on the *Add* object. Thus, for the activity α:

$$\forall \delta \in Act(Fibo) \neq \alpha,\ a_\delta = \mathcal{R}[\iota.set_1(\ldots)] \wedge \sigma_\delta(\iota) = AO(\gamma) \text{ is impossible}$$

and:

$$\exists^1 \alpha,\ a_\alpha = \mathcal{R}[\iota.set_1(\ldots)] \wedge \sigma_\alpha(\iota) = AO(\gamma)$$

Similarly

$$\exists^1 \beta,\ a_\beta = \mathcal{R}[\iota.set_2(\ldots)] \wedge \sigma_\beta(\iota) = AO(\gamma)$$

Thus Theorem 11.4 states that the Fibonacci numbers example has a deterministic behavior.

This example shows that the DON property is, to our mind, an interesting first step towards the *static detection of confluent programs*. Indeed, from a static approximation of active objects and method calls, it does not seem difficult to verify the DON property (when it is true). At the opposite, some control flow informations are necessary to find the possible RSLs and this is much more difficult to determine statically.

Therefore, we consider that the compatibility between configurations is the most general property ensuring confluence in our context (without information on the behavior of programs). The tree topology determinism is the easiest to verify statically. And the DON property seems the best compromise both ensuring determinism in a lot of cases and adapted to a static approximation.

Details on static analysis of ASP calculus are out of the scope of this study.

[14] One could find a more precise form for the RSL but it is not our purpose here.

11.12 Discussion: Comparing Requests Service Strategies

FIFO service is, to some extent, the worst case with respect to determinism, as any out of order reception of requests will lead to non-determinism.

By contrast, a request service by source activity ($Serve(\alpha)$) is entirely confluent. More precisely, if no FIFO service was allowed and the service determined by method label ($Serve(M)$ service primitive) was replaced by a service determined by the source activity of requests ($Serve(\alpha)$ service primitive) then the resulting calculus would be fully confluent and somewhat closer to process networks..

Indeed, the request service order would be the same whatever the interleaving of request arriving. This aspect is not developed here but could be very interesting if the order of activities sending a request to a given one was known. Such a calculus would be more similar to process networks where *get* operations are performed on a given channel and a channel only has one source process. However, specifying in a language from whom a sever accepts (serve) requests is no longer considered as a general option as clients must remain anonymous for the sake of modularity.

Even if DON definition allows much more flexibility in the general case, it seems difficult to find a more precise property for FIFO services. Determinism Theorem 11.3 is easy to specify but difficult to ensure. For example, a program that selectively serves different methods will still behave deterministically upon out of order receptions between those methods. This is a direct consequence of DON property that is not directly related to object topology and then such a DON program will not verify the Theorem 11.3.

Part V

Proofs

Chapter 12

Equivalence Modulo Futures

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

12.1 Renaming

Remember activities are chosen deterministically. Let us introduce a set Θ of renaming futures from configuration P to configuration Q:

In other words, Θ is an alpha-conversion of futures.

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

where $\alpha \in Act(P)$ and $\alpha \in Act(Q)$.

12.2 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.

Thus, two request queues are equivalent if all their restrictions of requests that can interfere in the same $Serve(M)$ are equivalent. In other words, for every set M of labels belonging to a $Serve(M)$ primitive of α the list of requests that can be captured by $Serve(M)$ is equivalent in both configurations (α_P and α_Q). Moreover, we will only compare terms coming from the same source term P_0 as $\mathcal{M}_{\alpha_P}$ is a static approximation. R_1 is a correct reoreding of the request queue R_2 if and only if $R_1 \equiv_R R_2$ where $\equiv_R$ is defined in Table 12.1.

The first rule expresses the fact that two requests can be exchanged if they do not interfere. Others 2 rules are reflexivity and transitivity rules.

That could also be expressed by finding a renaming φ_α that only permutes requests on methods that cannot be selected concurrently by the same $Serve(M)$.

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

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

Table 12.1: Reordering Requests

12.3 Future Updates

The equivalence modulo replies 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 to a deep copy. Thus, when two futures references concerns the same future, they are not considered as aliases.

The following example (Figure 12.1) illustrates a simple update of future value. Of course the two configurations are equivalent.

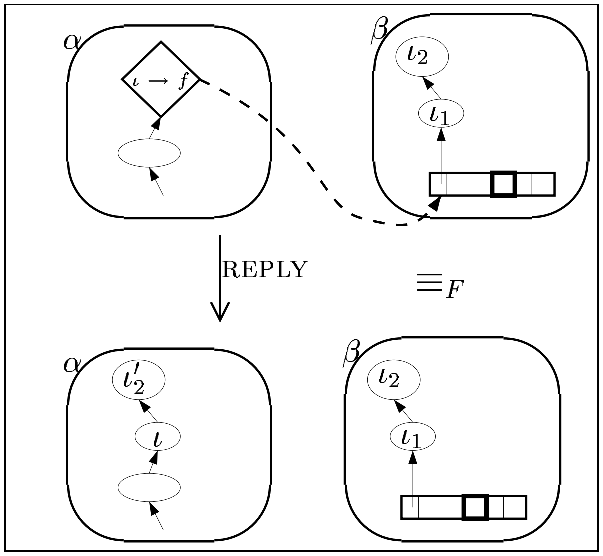


Figure 12.1: Simple example of future Equivalence

12.3.1 Following References and Sub-terms

Let us formalize the idea that "future references can be followed as if they were local references". In the following, the relation $\overset{\alpha}{\mapsto}_L$ is defined.

First, Table 12.2 describes the rules that defines $a \overset{\alpha}{\mapsto}_x b$. Note that there is no b such that $AO(\beta) \overset{\alpha}{\mapsto}_x b$ or $fut(f^{\gamma \to \beta}) \overset{\alpha}{\mapsto}_x b$. The second set of rules define the following of paths starting from an activity and inside lists. The last set of rules defines paths inside lists of requests and futures values.

More precisely, a path starting by an activity begin with an access to the current term a, active object location ι, futures values F, current future f, or pending requests. Ensuring equivalence of pending requests through a reordering equivalence $\equiv_R$ defined in Table 12.1 is easier even if this could be performed directly by a definition with paths. The following of paths inside of the store is not necessary but could also be added. When one does not follow paths inside stores, terms are identified modulo garbage collection inside each activity.

$$\iota \overset{\alpha}{\mapsto}_{ref} \sigma_\alpha(\iota) \qquad [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \overset{\alpha}{\mapsto}_{l_i} b_i$$

$[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \overset{\alpha}{\mapsto}_{m_j(s',x')} a_j\{\!\{x_j \leftarrow s', y_j \leftarrow x'\}\!\}$[15]

$$a.l_i \overset{\alpha}{\mapsto}_{field(l_i)} a \qquad a.l_i := b \overset{\alpha}{\mapsto}_{Update1(l_i)} a \qquad a.l_i := b \overset{\alpha}{\mapsto}_{Update2(l_i)} b$$

$$a.l_i(b) \overset{\alpha}{\mapsto}_{Invoke1(l_i)} a \qquad a.l_i(b) \overset{\alpha}{\mapsto}_{Invoke2(l_i)} b \qquad clone(a) \overset{\alpha}{\mapsto}_{clone} a$$

$$Active(a, m_j) \overset{\alpha}{\mapsto}_{Active(m_j)} a \qquad Serve(M) \overset{\alpha}{\mapsto}_{Serve(M)} \emptyset$$

$$a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow curr} a \qquad a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow f} a \qquad a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow cont} b$$

$$\alpha_p \overset{\alpha}{\mapsto}_{c.t.} a_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{a.o.} \iota_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{c.f.} \iota_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{f.v.} F_\alpha \qquad \frac{R_\alpha \equiv_R R'}{\alpha_p \overset{\alpha}{\mapsto}_{r.q.} R'}$$

$$[m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs\_meth} m \qquad [m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs\_arg} \iota$$

$$[m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs\_fut} f \qquad [m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs\_cdr} R$$

$$\{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs\_id} f_i^{\gamma\to\alpha} \qquad \{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs\_val} \iota$$

$$\{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs\_cdr} F$$

Table 12.2: Paths Definition

[15] Bounded variables are renamed here and thus it avoids to consider alpha-conversion of formal parameters at a higher level. Equivalence on methods ensures both text equality for a non-evaluated method and equivalence of locations inside method body when they appear.

Definition 12.1 *Let L represent the list of references or parts of expressions that must be followed inside activity α (following rules of Table 12.2), one defines inductively:*

$$\begin{array}{l} a \overset{\alpha}{\mapsto}_{\emptyset} a \\ a \overset{\alpha}{\mapsto}_{L.x} b \textit{ if } a \overset{\alpha}{\mapsto}_{L} a' \wedge a' \overset{\alpha}{\mapsto}_{x} b \end{array}$$

More generally, $\overset{\alpha}{\mapsto}_{L_1.L_2}$ denotes the concatenation $\overset{\alpha}{\mapsto}_{L_1}\overset{\alpha}{\mapsto}_{L_2}$.

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

Definition 12.2 ($\boldsymbol{a \overset{\alpha *}{\mapsto}_{L} b}$)

$$a \overset{\alpha *}{\mapsto}_{L_0 \ldots L_n} b \Leftrightarrow \begin{cases} a \overset{\alpha}{\mapsto}_{L_0} b \textit{ for } n = 0 \\ \vee \exists \iota_i,\, f_i,\, \beta_i\, \gamma_i{}^{i \leq n} \begin{cases} a \overset{\alpha}{\mapsto}_{L_0} fut(f_i^{\gamma_i \rightarrow \beta_1}) \wedge F_{\beta_1}(f_i^{\gamma_i \rightarrow \beta_1}) = \iota_1 \\ \sigma_{\beta_1}(\iota_1) \overset{\beta_1}{\mapsto}_{L_1} \iota_2 \wedge \ldots \wedge \\ \sigma_{\beta_n}(\iota_n) \overset{\beta_n}{\mapsto}_{L_n} b \end{cases} \end{cases}$$

This definition consists in, first, following a path inside an activity α, then, following a future reference from α to β_1 and continuing the path in β_1 etc... note that when one follows a future reference, two local (ref) and a future reference are in fact considered as identical to a single local reference. The following of local and future references from ι_0 in α to ι'_1 in β_1 is not taken into account in the path L.

For example, in Figure 12.1 the three arrows of the first configurations that are around the future reference (around the dashed arrow) are considered as equivalent with a single arrow on the second configuration.

Note that trivially:

Lemma 12.1 ($\boldsymbol{\overset{\alpha}{\mapsto}_{L}}$ and $\boldsymbol{\overset{\alpha *}{\mapsto}_{L}}$)

$$a \overset{\alpha}{\mapsto}_{L} b \Rightarrow a \overset{\alpha *}{\mapsto}_{L} b$$

Furthermore, he following of paths is (generally) unique: Following a path from a given term leads to the same expression except if the destination of the path can be a future reference:

Lemma 12.2 (Uniqueness of path destination)

$$a \overset{\alpha *}{\mapsto}_{L} b \wedge a \overset{\alpha *}{\mapsto}_{L} b' \Rightarrow b = b' \vee \exists \iota_i,\, \beta_i,\, \gamma,\, \delta \begin{cases} (\sigma_{\beta_i}(\iota_i) = fut(f_i^{\gamma \rightarrow \delta}) \vee F_{\delta}(f_i^{\gamma \rightarrow \delta}) = \iota_i) \\ b = \iota_i \vee b' = \iota_i \end{cases}$$

Here, the particular case (when $b \neq b'$) is due to the fact that when the destination of the path is a future reference the path does not necessarily follow this reference. In other words, for example, if $b = \iota_i$ in β_i where $\sigma_{\beta_i} = fut(f_i^{\gamma \rightarrow \delta})$ then one can have $b' = \iota_f$ where ι_f is the location of future $fut(f_i^{\gamma \rightarrow \delta})$ in β. A more precise formulation of the preceding lemma would be:

$$a \overset{\alpha *}{\mapsto}_{L} b \wedge a \overset{\alpha *}{\mapsto}_{L} b' \Rightarrow b = b' \vee \exists \iota_i, \iota'_i, \begin{cases} (\sigma_{\beta_i}(\iota_i) = fut(f_i^{\gamma \rightarrow \delta}) \wedge F_{\delta}(f_i^{\gamma \rightarrow \delta}) = \iota'_i) \\ (b = \iota_i \wedge b' = \iota'_i) \vee (b = \iota'_i \wedge b' = \iota_i) \end{cases}$$

12.3.2 Equivalence Definition

Definition 12.3 (Equivalence $P \equiv_F Q$) *Let $R = Q\Theta$. Then*

$$
\begin{array}{lll}
P \equiv_F Q & \Leftrightarrow & \forall \alpha \in Act(P) \cup Act(R),\ \forall L \left(\exists a,\ \alpha_P \overset{\alpha*}{\mapsto}_L a \Leftrightarrow \exists a',\ \alpha_R \overset{\alpha*}{\mapsto}_L a'\right)\\
 & & \wedge\ \forall L,\ L',\ a\Big(\alpha_P \overset{\alpha}{\mapsto}_L a \wedge \alpha_P \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L_0', a', L_1, L_1', \gamma\\
 & & \qquad\qquad \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset\\ \alpha_R \overset{\alpha*}{\mapsto}_{L_0} c \wedge \alpha_R \overset{\alpha*}{\mapsto}_{L_0'} c\\ c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \Big)\\
 & & \wedge\ \forall L,\ L'\ a\Big(\alpha_R \overset{\alpha}{\mapsto}_L a \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L_0', a', L_1, L_1', \gamma\\
 & & \qquad\qquad \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset\\ \alpha_P \overset{\alpha*}{\mapsto}_{L_0} c \wedge \alpha_P \overset{\alpha*}{\mapsto}_{L_0'} c\\ c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L_1'} a' \end{cases} \Big)
\end{array}
$$

The first condition expresses both the equivalence inside an activity and by following futures and the two last conditions express the correctness of aliasing (alias must be the same in both configurations). These two last conditions will be named alias conditions in the following. Note that in the alias conditions the existence of a' and a'' such that $\alpha_P \overset{\alpha*}{\mapsto}_L a'$ and $\alpha_P \overset{\alpha*}{\mapsto}_{L'} a''$ is ensured by the first condition. This rules ensures $a' = a''$ and a' and a'' are “correctly” aliased.

The alias conditions can be expressed in the following way: “if two paths lead to the same term (e.g. the same location) then in the equivalent configuration these paths lead to the same (equivalent) term”. Note that the paths on the left of the implication are local to an activity. Indeed, two references to a future leads to the same future value, and are aliased; while the two updated futures will be two different deep copy of the future value which is both logical and sound. Thus ensuring correctness of aliases is sufficient on local paths.

The paths on the right of the implication can follow the future references but the *last* alias must be local to an activity. This ensures that the last alias will still be aliased when the future value will be updated. For the aliases that can appear before the last one (if $L_1 \neq L_1'$), the correctness of the aliases is guaranteed by the alias conditions with L_1 and L_1': $c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L_1'} a'$

This definitions formalizes the intuition given in Section 11.5. The explanations given with Figure 11.5 can be compared to this definition and the informal points ensuring the equivalence in this example (page 88) can now be considered as hints in order to prove that both terms of Figure 11.5 verify the equivalence definition.

The role of the different paths in the alias conditions are illustrated in the Figure 12.2. One can verify that the alias of paths L and L' in the bottom configuration is simulated by two aliases in the first one. Remark that the last alias is local to an activity.

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 12.3. But this will have no consequence on the subsequent reductions and as explained before one could make this equivalence more precise.

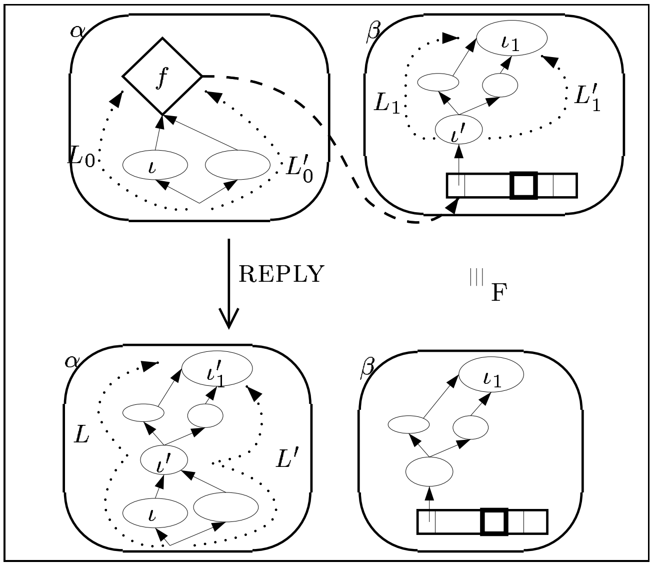


Figure 12.2: The principle of the alias conditions

Property 12.3 (Equivalence relation) $\equiv_F$ *is an equivalence relation.*

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

Definition 12.4 (Equivalence of sub-terms)

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

Thus, the definition of equivalence modulo futures on configurations has the following consequences on the sub-terms:

Lemma 12.4 (sub-term equivalence)

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

In the following proofs, the arguments related to renaming will not be detailed. That is to say, we will always suppose that, when $P \equiv Q$, P and Q use the s futures names (or more precisely, renaming of futures have already been applied). Proofs will focus on parts of proof related to updates of futures. In other words, the alpha conversion part of proofs is considered as straightforward. In the same way, terms are identified modulo renaming of locations. For example, when a fresh location has to be taken, one can choose the location that makes the proof more simple and concise (for example the one that have been chosen in another reduction). This is always possible modulo alpha conversion of the different terms involved.

12.4 Properties of $\equiv_F$

The following property is a direct consequence of Tables 12.1, 12.2 and 12.3:

Property 12.5 (Equivalence and compatibility)

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

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

Lemma 12.6 ($\equiv_F$ and store update)

$$\begin{cases} P \equiv_F Q \quad \wedge \quad a \equiv_F a' \\ \iota \in dom(\sigma_{\alpha_P}) \wedge \iota' \in dom(\sigma_{\alpha_Q}) \quad \wedge \quad \iota \equiv_F \iota' \\ P' = P \textit{ except } \sigma_{\alpha_{P'}} = \{\iota \rightarrow a\} + \sigma_{\alpha_P} \\ Q' = Q \textit{ except } \sigma_{\alpha_{Q'}} = \{\iota' \rightarrow a'\} + \sigma_{\alpha_Q} \end{cases} \Rightarrow \begin{array}{l} P' \equiv_F Q'' \\ \textit{provided } \vdash P' \,\text{OK} \;\wedge\; \vdash Q' \,\text{OK} \end{array}$$

The condition "provided $\vdash P' \,\text{OK} \;\wedge\; \vdash Q' \,\text{OK}$" is useful to ensure that local or generalized references inside a and a' are defined in P and Q. In the following proofs, a and a' will always be sub-terms of P and Q respectively and thus this condition will always be verified.

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

An equivalent version consists in Replacing the condition $\iota \in dom(\sigma_{\alpha_P}) \wedge \iota' \in dom(\sigma_{\alpha_Q}) \wedge \iota \equiv_F \iota'$ by (direct consequence of the definition of sub-term equivalence):

$$\exists L,\, \alpha_P \overset{\alpha}{\mapsto}_L \iota \wedge \alpha_Q \overset{\alpha}{\mapsto}_L \iota$$

Note that the path L leading to ι is not necessarily unique.

Proof :

We will use the existence of the path L and the objective of this proof is to verify:

$$\begin{cases} P \equiv_F Q \\ a \equiv_F a' \quad \wedge \quad \exists L,\, \alpha_P \overset{\alpha}{\mapsto}_L \iota \wedge \alpha_Q \overset{\alpha}{\mapsto}_L \iota \\ P' = P \text{ except } \sigma_{\alpha_{P'}} = \{\iota \rightarrow a\} + \sigma_{\alpha_P} \\ Q' = Q \text{ except } \sigma_{\alpha_{Q'}} = \{\iota' \rightarrow a'\} + \sigma_{\alpha_Q} \end{cases} \Rightarrow \begin{array}{l} P' \equiv_F Q'' \\ \text{provided } \vdash P' \,\text{OK} \;\wedge\; \vdash Q' \,\text{OK} \end{array}$$

Let Θ be the futures renaming that has to be applied to Q to prove $P \equiv_F Q$.

Let $R' = Q'\Theta$. Then if $\beta_{P'} \overset{\beta*}{\longmapsto}_{L'} b$, let us make a recurrence on the number n of times $\overset{\beta*}{\longmapsto}_{L'}$ passes by the location ι of the activity α.

If $n = 0$, the fact that $P \equiv_F Q$ is sufficient to conclude.

If $n > 0$ either $n = 1$ and

$$\alpha = \beta \wedge \alpha_{P'} \overset{\alpha}{\longmapsto}_{L} \iota \overset{\alpha*}{\longmapsto}_{L''} b$$

or

$$\beta_{P'} \overset{\beta*}{\longmapsto}_{L_0} fut(f^{\gamma\to\alpha}) \wedge \sigma_{\alpha}(F_{\alpha}(f^{\gamma\to\alpha})) \overset{\alpha}{\longmapsto}_{L} \iota \overset{\alpha*}{\longmapsto}_{L''} b$$

All these cases are similar, we will focus on the first one (which has a simpler notation).

As $\alpha_{P'} \overset{\alpha}{\longmapsto}_{L} \iota \overset{\alpha*}{\longmapsto}_{L''} b$, one has $\alpha_{R'} \overset{\alpha}{\longmapsto}_{L} \iota'$ by hypothesis and $\iota' \overset{\alpha*}{\longmapsto}_{L''} b'$ by definition of $a \equiv_F a'$ (Lemma 12.4). This proves, in the case where $n = 1$ and $\alpha = \beta \wedge \alpha_{P'} \overset{\alpha}{\longmapsto}_{L} \iota \overset{\alpha*}{\longmapsto}_{L''} b$

$$\exists a,\ \alpha_{P'} \overset{\alpha*}{\longmapsto}_{L} a \Rightarrow \exists a',\ \alpha_{R'} \overset{\alpha*}{\longmapsto}_{L} a'$$

The other cases follows the same reasoning with more complex notations and using a recurrence hypothesis. Finally, one has:

$$\forall \beta \exists a,\ \beta_{P'} \overset{\beta*}{\longmapsto}_{L} a \Rightarrow \exists a',\ \beta_{R'} \overset{\beta*}{\longmapsto}_{L} a'$$

The opposite implication is similar (symmetric).

For alias conditions, suppose

$$\exists L_0, L_0', d\ \alpha_{P'} \overset{\alpha}{\longmapsto}_{L_o} b \wedge \alpha_{P'} \overset{\alpha}{\longmapsto}_{L_o'} b$$

the most interesting case is when there are several L_i such that (of course, L is one of the L_i)

$$\alpha_{P'} \overset{\alpha}{\longmapsto}_{L_i} \iota \wedge \alpha_{P'} \overset{\alpha}{\longmapsto}_{L_{i'}} \iota$$

By hypothesis, as $P \equiv_F Q$, one has:

$$\exists L_1, L_1', L_2, L_2' \begin{cases} L_i = L_1.L_1' \wedge L_{i'} = L_2.L_2' \\ \alpha_{R'} \overset{\alpha*}{\longmapsto}_{L_1} c \overset{\beta}{\longmapsto}_{L_1'} \iota' \\ \alpha_{R'} \overset{\alpha*}{\longmapsto}_{L_2} c \overset{\beta}{\longmapsto}_{L_2'} \iota' \end{cases}$$

Moreover, as L is one of the L_i and $\alpha_{R'} \overset{\alpha}{\longmapsto}_{L} \iota'$ (path local to α) the preceding assertion is simplified to:

$$\alpha_{R'} \overset{\alpha}{\longmapsto}_{L_i} \iota' \wedge \alpha_{R'} \overset{\alpha}{\longmapsto}_{L_{i'}} \iota'$$

That already ensures alias condition for path reaching ι and ι' (when $L_0 = L_i$ and $L_0' = L_{i'}$). Furthermore, for terms inside a and a', the worst case is when, there are L_i, L', L'', b such that ($L_0 = L_i.L'$ and $L_0 = L_{i'}.L''$):

$$\alpha_{P'} \overset{\alpha}{\longmapsto}_{L_i} \iota \overset{\alpha}{\longmapsto}_{L'} b \wedge \alpha_{P'} \overset{\alpha}{\longmapsto}_{L_{i'}} \iota \overset{\alpha}{\longmapsto}_{L''} b$$

the Lemma 12.4 ensures that in R'

$$\iota' \overset{\alpha}{\mapsto}_{L'} b \wedge \iota' \overset{\alpha}{\mapsto}_{L''} b$$

when adding the alias property for $P \equiv_F Q$ one obtains:

$$\alpha_{R'} \overset{\alpha}{\mapsto}_{L_i} \iota' \overset{\alpha}{\mapsto}_{L'} b' \wedge \alpha_{R'} \overset{\alpha}{\mapsto}_{L_{i'}} \iota' \overset{\alpha}{\mapsto}_{L''} b'.$$

Finally, other cases for verification of alias conditions are simpler application of the same hypothesis and lemmas. □

Lemma 12.7 ($\boldsymbol{\equiv_F}$ **and substitution)**

$$\left\{\begin{matrix} P \equiv_F Q \\ \iota \equiv_F \iota' \end{matrix}\right. \Rightarrow a\{\!\{x \leftarrow \iota\}\!\} \equiv_F a\{\!\{x \leftarrow \iota'\}\!\}$$

The proof is straightforward. This lemma will be useful for the cases of the following proofs concerning method invocation. It proves the soundness of $\equiv_F$ with respect to the substitution applied in INVOKE rule.

Recall that (Definition 10.1):

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

The following definition of deep copy is well adapted to the proofs on equivalence relations:

Lemma 12.8 (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:

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

The following lemma is also a consequence of the preceding properties:

Lemma 12.9 (Copy and Merge)
If $P' = P$ *except* $\sigma_{\alpha_{P'}} = \mathit{Copy\&Merge}(\sigma_{\beta_P}, \iota_0 \;;\; \sigma_{\alpha_P}, \iota)$

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

It only means that the part of store that is deeply copied verify the Lemma 12.8 and thus paths starting from the destination of the deep copy in $\alpha_{P'}$ are the same than paths starting from the source location in β_P.

The following property states that adding equivalent deep copies to equivalent configurations produces equivalent configurations.

Lemma 12.10 ($\equiv_F$ and store merge)

$$\left\{\begin{array}{l} P \equiv_F Q \ \wedge \ \iota \in \alpha_P \ \wedge \ \iota' \in \alpha_Q \wedge \iota_0 \in \beta_P \ \wedge \ \iota'_0 \in \beta_Q \\ a \equiv_F a' \quad \wedge \quad \iota \equiv_F \iota' \quad \wedge \quad \iota_0 \equiv_F \iota'_0 \\ P' = P \ \mathit{except} \ \sigma_{\alpha_{P'}} = \mathit{Copy\&Merge}(\sigma_{\beta_P}, \iota_0 \ ; \ \sigma_{\alpha_P}, \iota) \\ Q' = Q \ \mathit{except} \ \sigma_{\alpha_{Q'}} = \mathit{Copy\&Merge}(\sigma_{\beta_Q}, \iota'_0 \ ; \ \sigma_{\alpha_Q}, \iota') \end{array}\right. \Rightarrow P' \equiv_F Q'$$

Proof : There is θ_0 such that:

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

From Lemma 12.8, one has:

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

Thus, using Lemma 12.9, for all $a \in \mathit{copy}(\iota_0, \sigma_{\beta_P})$

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

The same property is also true for configuration Q.

Informally, if a location is in the merged sub-store of $\alpha_{P'}$ then it comes from the sub-store $\mathit{copy}(\iota_0, \sigma_{\beta_P})$ of P and it is equivalent to a location inside $\mathit{copy}(\iota'_0, \sigma_{\beta_Q})$ (because $\iota_0 \equiv_F \iota'_0$) which corresponds to a location in the merged sub-store of $\alpha_{Q'}$. More formally:

$$\begin{array}{lll} \iota \overset{\alpha_{P'}}{\mapsto}_L a\theta_0 & \Leftrightarrow \iota_0 \overset{\beta_P}{\mapsto}_L a & \\ & \Leftrightarrow \iota'_0 \overset{\beta_Q}{\mapsto}_L a' & \text{because } \iota_0 \equiv_F \iota'_0 \\ & \Leftrightarrow \iota' \overset{\alpha_{Q'}}{\mapsto}_L a'\theta'_0 & \text{preceding property for configuration } Q \end{array}$$

Then, let L_0 be a path leading to ι in $\alpha_{P'}$, it also leads to ι' in $\alpha_{Q'}$ because $\iota \equiv_F \iota'$.

$$\left\{\begin{array}{l} \iota \equiv_F \iota' \wedge \\ \left(\iota \overset{\alpha_{P'}}{\mapsto}_L a \Leftrightarrow \iota' \overset{\alpha_{Q'}}{\mapsto}_L a'\right) \end{array}\right. \Rightarrow \left(\alpha_{P'} \overset{\alpha_{P'}}{\mapsto}_{L_0.L} a \Leftrightarrow \alpha_{Q'} \overset{\alpha_{Q'}}{\mapsto}_{L_0.L} a'\right)$$

The general case follows similar reasonings (with a recurrence on the number of time we pass by the location ι like in the proof of Lemma 12.6) and finally we obtain:

$$\forall \beta, \beta_{P'} \overset{\beta_{P'}*}{\mapsto}_L a \Leftrightarrow \beta_{Q'} \overset{\beta_{Q'}*}{\mapsto}_L a$$

The proof of alias conditions follows is similar too and finally P' and Q' are equivalent. □

12.5 Sufficient Conditions for Equivalence

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.

As explained informally in 12.3, two configurations only differing by some future updates are equivalent:

Property 12.11 (REPLY and $\equiv_F$)

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

Proof : One only has to prove that the updated store is equivalent with the old one. The other activities and other parts of the updated activity are unchanged.

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

There are two renamings θ and θ_0 such that:

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

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 12.9).

Let L_0 be such that $\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota$. Thus, when the path passes one time by the location ι, there is L such that:

$$\left(\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \wedge F_\beta(f_i^{\gamma\to\beta}) = \iota_f \wedge \iota_f \overset{\beta}{\mapsto}_L \iota_0\right) \Leftrightarrow \left(\alpha_{P'} \overset{\alpha}{\mapsto}_{L_0.L} \iota'\right)$$

The more general assertion:

$$\forall \gamma_P, L, \quad \gamma_P \overset{\gamma_P *}{\mapsto}_L a \Leftrightarrow \gamma_{P'} \overset{\gamma_{P'} *}{\mapsto}_L a$$

is obtained by a classical recurrence.

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 $L' = L_1.L'' \wedge \alpha_P \overset{\alpha}{\mapsto}_{L_1} \iota$ 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 12.9),

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

□

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

More precisely, we have the following sufficient condition for equivalence modulo future replies:

Property 12.12 (Sufficient condition for equivalence)

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

This is easily proved by transitivity of $\equiv_F$.

Recall this condition is not necessary as it does not deal with mutual references between futures (Figure 11.7).

12.6 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 as it somewhat proves the correctness of $\equiv_F$ with respect to reduction. The proof is decomposed in two parts. First, one 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 by necessity the value of the future. Indeed, some futures may be updated in a configuration and calculated but not updated in an equivalent configuration (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: two equivalent configurations can perform the same reduction and become equivalent configuration. The non-observable transition being REPLY.

In the following, let T be any parallel reduction rule: T range over $\{\text{LOCAL}, \text{NEWACT}, \text{REQUEST}, \text{SERVE}, \text{ENDSERVICE}, \text{REPLY}\}$. $\xrightarrow{T}$ denotes the application of a parallel rule named T (cf Table 10.2).

Property 12.13 ($\equiv_F$ and reduction(1))

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

This property suggests to define the new reduction $\Longrightarrow$:

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

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

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.

Using this new reduction, the Property 12.13 can be rewritten in the following manner:

Property 12.14 ($\equiv_F$ and reduction(2))

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

The properties 12.13 and 12.14 are equivalent. The following proof is valid for both.

Proof : If one cannot apply the same reduction than $P \longrightarrow Q$ (same rule on the same activities ...) on P', $\overset{\text{REPLY}}{\longrightarrow}$ is applied enough times ($P' \overset{\text{REPLY}^*}{\longrightarrow} P''$) to be able to apply this reduction: P' may be performing a wait by necessity on the value of some futures. The principle is that for each awaited future reference, as P can perform the reduction, the future has already been calculated in P, and $P \equiv P'$ implies that the future has also been calculated in P'. Thus P' only needs to update it.

More precisely, it is straightforward to check that if two configurations are equivalent, the same reduction can be applied on the two configurations except if one of them is stuck. 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. This is impossible because P can be reduced.

- In the case of an access to a future (wait-by-necessity): if in an activity of P' one has $a = \ldots \iota' \ldots$ and $\sigma_{\alpha_{P'}}(\iota') = fut(f_i^{\gamma \to \beta})$ and the operation performed on ι' is strict 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 $f_i^{\gamma \to \beta} \in F_{\beta_{P'}}$.

  Then it is possible to update $f_i^{\gamma \to \beta}$ in P': $P' \overset{\text{REPLY}}{\longrightarrow} P'_1$. If in P_1 $\sigma_{\alpha_{P'_1}}(\iota') = fut(f_j^{\gamma \to \beta})$ then, another time, we update the future f_j. After a finite number of updates, we obtain P'' such that $P' \overset{\text{REPLY}^*}{\longrightarrow} P''$ and $\sigma_{\alpha_{P''}}(\iota')$ is not a future reference. Indeed, if the number of updates was infinite, then P could not be reduced, that is contradictory with the hypothesis.

  Then $P' \overset{\text{REPLY}^*}{\longrightarrow} P''$ where $P'' \equiv_F P$ (Property 12.11) and in P'' $\sigma_{\alpha_{P''}}(\iota')$ 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 occurs in P cannot access an object inside a future that has not been updated in P' because

$$P \equiv_F P' \Rightarrow a_{\alpha_P} \equiv_\alpha a_{\alpha'_P} \Rightarrow \forall \iota \in a_{\alpha_P}, \iota\theta_\alpha \in a_{\alpha'_P}.$$

In other words, only the objects accessed directly by the reduction T may have to be updated.

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

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

Where both applications of the rule T are the same (same application points and same activities concerned). In fact, from a given configuration a rule is uniquely specified by the name of the rule and the names of the different activities concerned except in the case of the REPLY rule where the future identifier is also necessary.

This is obtained by a (long) case study. The different cases depend of the reduction applied and the rules applied to prove the equivalence. In the following the proofs will only focus on the cases where one of the locations 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. In the following, no details about renaming of futures, and locations are given: one could easily prove a first step toward the whole proof concerning only renaming:

$$\begin{cases} P \xrightarrow{T} Q \\ P'' \xrightarrow{T} Q' \\ P'' \equiv P \end{cases} \Rightarrow Q' \equiv Q$$

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

- **STOREALLOC** Consequence of Lemma 12.6.

- **FIELD**

  $$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \qquad k \in 1..n}{(\mathcal{R}[\iota.l_i], \sigma) \rightarrow_S (\mathcal{R}[\iota_{i1}], \sigma)} \text{ (FIELD)}$$

  With $a_P = \iota.l_i \equiv_F \iota_2.l_i = a_{P''}$ then $\iota \equiv_F \iota_2$ and $\iota_{i1} \equiv_F \iota_{i2}$ (ι_{i2} is the location of field l_i in P'') because $\iota \overset{\alpha_P}{\mapsto}_{l_i} \iota_{i1}$ and $\iota_2 \overset{\alpha_{P''}}{\mapsto}_{l_i} \iota_{i2}$. Thus $P'' \equiv_F Q$. The following cases will often use reasoning similar to this one, and will not be detailed.

- **INVOKE** Straightforward: Note that the two method bodies must be equivalent and the two arguments too. The final equivalence comes from Lemma 12.7.

UPDATE Direct from Lemma 12.6 and the kind of reasoning of the FIELD case.

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

Indeed, this case justify the fact that cloning a future is considered as a strict operation. The future updates consists in a deep copy of the value whereas the *clone* operator performs a shallow clone. Thus performing a CLONE and then a REPLY reduction creates two deep copies of the future value. At the opposite performing a REPLY before a CLONE reduction creates only one deep copy with two shallow copies of the first object of the future value. Thus, for the coherence of the calculus, in ASP, cloning is considered as a strict operation (cloning a future is blocking: wait-by-necessity). Of course these blocking states can create dead locks.

NEWACT

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

The only interesting case is the presence of futures in the newly created activity. Lemma 12.10 is sufficient to conclude. Indeed in NEWACT, $\sigma_\gamma = copy(\iota, \sigma_\alpha)$ could be written $\sigma_\gamma = Copy\&Merge(\sigma_\alpha, \iota\ ;\ \emptyset, \iota)$

REQUEST

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

Modulo renaming, one can choose the same name for future in P and Q, the same location for the copy of the argument. Lemma 12.10 can be applied to manage with futures that can be present in the deep copy of the requests parameters.

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

SERVE This is the most important case of the proof.

Informally, the equivalence between the two requests lists implies that the served requests are equivalent which is sufficient to conclude.

The fact that the equivalence definition is defined modulo a reordering of requests is essential here. More precisely:

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

Thus the first request of $R_{\alpha_P}\big|_M$ will be equivalent modulo replies in both configurations. Consequently, SERVE will serve equivalent requests.

ENDSERVICE The equivalence between futures lists is straightforward. The proof is based on the application of the Lemma 12.10.

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

Note that without consequence on the proof of the property, one may be unable to apply directly the same rule on the two equivalent terms for three reasons:

- either several future updates may be needed to have the reference on which the concerned future must be updated (several futures updates are needed 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 11.7 one has either only references to future f_1 or to future f_2)

Note that most of this proof is simplified by the important Lemma 12.10. □

The following property is a direct consequence of Property 12.14.

Corollary 12.15 ($\equiv_F$ and reduction)

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

Proof : If T = REPLY then the proof is straightforward. Else $P \overset{T}{\Longrightarrow} Q$ can be decomposed in $P \overset{\text{REPLY}^*}{\longrightarrow} P_1 \xrightarrow{T} Q$. The conclusion comes from the application of the preceding property to P_1 instead of P (because of the transitivity of $\equiv_F$ one has $P_1 \equiv_F P'$). □

12.7 Another Formulation

Let us formalize another definition of equivalence between terms based on renaming and prove its equivalence with the preceding one.

Let us extend the renaming of futures defined in 12.1 with a renaming on locations inside an activity θ_{α_i} and between two activities $\theta_{\alpha_i \rightarrow \alpha'_j, \iota'}$. Recall that θ_{fut} is a bijection from F_{α_P} to F_{α_Q} .

$$\begin{aligned}
\Theta &::= (\theta_{fut}, \theta_{\alpha}^{\alpha \in Act(P)}, \ldots \theta_{\alpha \rightarrow \beta, \iota}^{\alpha \in Act(P),\, \beta \in Q}, \ldots,) \\
\theta_{fut} &::= \{\!\{ f_i^{\beta_P \rightarrow \alpha_P} \leftarrow f_i'^{\beta_Q \rightarrow \alpha_Q}, \ldots \}\!\}; \\
\forall \alpha \in Act(P) \cap Act(Q) \theta_{\alpha} &::= \{\!\{ \iota_1 \leftarrow \iota'_1 \ldots \}\!\};\ \text{where } \iota_1 \in locs(\alpha_P), \iota'_1 \in locs(\alpha_Q)
\end{aligned}$$

$$\begin{aligned}
&\alpha \in Act(P), \beta \in Act(Q), \\
&\qquad \theta_{\alpha_P \rightarrow \beta_Q, \iota'} ::= \{\!\{ \iota_1 \leftarrow \iota'_1, \ldots \}\!\} \text{ where } \iota_1 \in locs(\alpha_P),\ \iota', \iota'_1 \in locs(\beta_Q)
\end{aligned}$$

$$\begin{aligned}
&\alpha \in Act(P), \beta \in Act(Q), \\
&\qquad \theta_{\alpha_P \leftarrow \beta_Q, \iota'} ::= \{\!\{ \iota_1 \leftarrow \iota'_1, \ldots \}\!\} \text{ where } \iota, \iota_1 \in locs(\alpha_P),\ \iota'_1 \in locs(\alpha_Q)
\end{aligned}$$

The last two renaming of locations allow expressing 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, one must provide a renaming $\theta_{\alpha\to\gamma',\iota'}$.

Of course, each renaming must be bijective and

- for each $\beta \in \mathit{Act}(Q)$ the sets $\mathit{codom}(\theta_{\beta i})$, $(\mathit{codom}(\theta_{\alpha\to\beta})_{\alpha\in P}$ are disjuncts;
- for each $\alpha \in \mathit{Act}(P)$ the sets $\mathit{dom}(\theta_{\alpha})$, $(\mathit{dom}(\theta_{\alpha\leftarrow\beta})_{\beta\in Q}$ are disjuncts;

Let us define the following equivalence relation:

Definition 12.5 (Equivalence modulo replies(2))
$\equiv_F$ is the largest equivalence relation closed backward under the rules of Table 12.3 (for any activity α). x is either α or $\alpha \to \beta, \iota'$ or $\alpha \leftarrow \beta, \iota$.

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

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

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

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

$$\frac{\begin{array}{c} a_{\alpha_P} \equiv_\alpha a_{\alpha_Q} \qquad \iota_{\alpha_P} \equiv_\alpha \iota_{\alpha_Q} \qquad F_{\alpha_P} \equiv_\alpha F_{\alpha_Q} \\ \forall M \in \mathcal{M}_{\alpha_P},\, R_{\alpha_P}\big|_M \equiv_F R_{\alpha_Q}\big|_M \qquad f_{\alpha_Q} = f_{\alpha_P}\theta_{fut} \end{array}}{\alpha[a_{\alpha_P}; \sigma_{\alpha_P}; \iota_{\alpha_P}; F_{\alpha_P}; R_{\alpha_P}; f_{\alpha_P}] \equiv_\alpha \alpha[a_{\alpha_Q}; \sigma_{\alpha_Q}; \iota_{\alpha_Q}; F_{\alpha_Q}; R_{\alpha_Q}; f_{\alpha_Q}]}$$

$$\frac{\begin{array}{c} \exists\Theta = (\theta_{fut}, \theta_{\alpha_1}, \ldots) \qquad \mathit{Act}(P) = \mathit{Act}(Q) \\ \forall\alpha \in \mathit{Act}(P), \alpha[a_{\alpha_P}; \sigma_{\alpha_P}; \iota_{\alpha_P}; F_{\alpha_P}; R_{\alpha_P}; f_{\alpha_P}] \equiv_\alpha \alpha[a_{\alpha_Q}; \sigma_{\alpha_Q}; \iota_{\alpha_Q}; F_{\alpha_Q}; R_{\alpha_Q}; f_{\alpha_Q}] \end{array}}{P \equiv_F Q}$$

Table 12.3: Equivalence

All the trivial induction rules corresponding to operators in the syntax are not explicited in this table.

In other words, suppose one want to prove $P \equiv_F Q$, one must provide

$$\Theta = (\theta_{fut}, \theta_{\alpha}^{\alpha \in Act(P)}, \dots \theta_{\alpha \rightarrow \beta, \iota}^{\alpha \in Act(P), \beta \in Q}, \dots,)$$

such that the rules of table 12.3 are verified

$\equiv_\alpha$ denotes the equivalence between (sub-)terms that appears in α in both configurations. $\equiv_{\alpha \rightarrow \beta, \iota'}$ 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 when $x = \alpha$ or between activities when x is of the form $\alpha \leftarrow \beta', \iota$ or $\alpha \rightarrow \beta', \iota'$).

The existence of $\theta_{\alpha \rightarrow \beta, \iota'}$ 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 $\theta_{\alpha \rightarrow \beta, \iota'}$ must be applied to locations of the activity α of the configuration P in order to obtain locations (corresponding to a deep copied future value) in the activity β' of the configuration$Q$.

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

This definition is coinductive because, one may need to use the fact that $\iota \equiv \iota'$ to prove that $\iota \equiv \iota'$. The Figure 12.5 shows the example of a configuration where such kind of definition is necessary

Some Important Remarks:

- $\theta_{\alpha \rightarrow \alpha, \iota}$ 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 one would have to use even more complicated rules and notations.
- The unaccessible parts of the store are not taken into account. These parts represent the locations that can be safely garbage collected.

12.8 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 : Here are some details of the proof. The whole proof is longer but based on the principles given below:

First note that the renaming of future to be applied in both cases is the same.

$\equiv_{F2} \Rightarrow \equiv_F$ If $P \equiv_{F2} Q$, then for all α one has $\alpha_P[\dots] \equiv_x \alpha_R[\dots]$. Renaming of futures is not detailed here. By recurrence and by cases on l, we can 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 $\alpha_P[\dots] \equiv_\alpha \alpha_R[\dots]$ implies that $\alpha_P \overset{\alpha*}{\mapsto}_\emptyset \alpha_P$.

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

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

One can prove $\alpha_P \overset{\alpha*}{\mapsto}_{L.m_j} c \Rightarrow \alpha_Q \overset{\alpha*}{\mapsto}_{L.m_j} c' \wedge c \equiv_x c'$.

Note that the case where a future is only updated in one configuration needs only to be considered in the case $l = ref$.In that case a simple store access is equivalent (according to the definition of $\equiv_F$) to an access to the calculated (but not updated) future.

Let us detail here the case where $l = ref$:

- future reference: Suppose:

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

  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 in Q. Then, because by recurrence hypothesis $\alpha_Q \overset{\alpha*}{\mapsto}_L \iota_0$, finally:

  $$\alpha_Q \overset{\alpha*}{\mapsto}_L \iota_0 \wedge \sigma_\gamma(\iota_0) = fut(f_i^{\delta\to\beta}) \wedge F_{\beta_Q}(f_i^{\delta\to\beta}) = \iota' \wedge \iota' \overset{\gamma}{\mapsto}_{ref} \sigma(\iota').$$

  Thus $\alpha_Q \overset{\alpha*}{\mapsto}_{L.ref} \sigma_{\beta_Q}(\iota')$.

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

- if $x = \alpha \leftarrow \beta, \iota_0$: $b = \iota \equiv_{\alpha\leftarrow\beta,\iota_0} \iota\theta_{\alpha\leftarrow\beta,\iota} = b'$ implies by hypothesis:

  $$\alpha_P \overset{\alpha*}{\mapsto}_L b \wedge \alpha_Q \overset{\alpha*}{\mapsto}_L b'$$

  and by definition of $\equiv_{F2}$:

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

  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 12.3. This can be proved by contradiction by showing that if one condition is not verified then a renaming is not bijective:

First note that one 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:

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

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

The bijectivity of the renaming 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 to follow arrows to determine the renamings. The recurrence cases are trivial. The following proof details 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 the garbage collectable terms, they are ignored by both equivalence relations.

Thus, suppose that P and Q are equivalent according to Definition 12.3.

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 12.3 ensures $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$. One can specify the different renaming from this. Two cases are possible:

- either $\alpha_R \overset{\alpha}{\mapsto}_L \iota'$ then $\{\!\{\iota \leftarrow \iota'\}\!\} \in \theta_\alpha$
- or $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$ and ι' is in the activity γ of R then $\{\!\{\iota \leftarrow \iota'\}\!\} \in \theta_{\alpha \leftarrow \gamma, \iota_0}$ where ι_0 is the location in α_P corresponding to the last reference to future encountered in $\alpha_R \overset{\alpha*}{\mapsto}_L \iota'$. More precisely decompose:

  $$\alpha_R \overset{\alpha*}{\mapsto}_{L_0} \iota'_0 \overset{\gamma}{\mapsto}_{L'} \iota' \text{ where } L = L_0.L',$$

  then one has

  $$\alpha_R \overset{\alpha*}{\mapsto}_{L_0} \iota'_0 \Rightarrow \alpha_P \overset{\alpha*}{\mapsto}_{L_0} \iota_0 \text{ definition of } \equiv_F$$

  and moreover, $\iota_0 \in \alpha_P$ because L_0 is a prefix of L. Thus $\alpha_P \overset{\alpha}{\mapsto}_{L_0} \iota_0$. This defines both ι_0 and γ.

$\{\!\{\iota \leftarrow \iota'\}\!\} \in \theta_{\alpha \rightarrow \beta, \iota_0}$ 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 12.3.

For example, if θ_α is non bijective then one would have $\{\!\{\iota \leftarrow \iota'\}\!\} \in \theta_\alpha$ and $\{\!\{\iota_2 \leftarrow \iota'\}\!\} \in \theta_\alpha$ 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 12.3. In the same way, the above construction never builds renamings such that $\{\!\{\iota \leftarrow \iota'\}\!\} \in \theta_\alpha \wedge \{\!\{\iota \leftarrow \iota_2\}\!\} \in \theta_\alpha$ if $\iota' \neq \iota_2$. □

12.9 Decidability of $\equiv_F$

Property 12.16 (Decidability) $\equiv_F$ *is decidable.*

Proof : In the first case, verifying the equivalence consists in first applying a renaming Θ of futures. The set of such renamings is finite and could be enumerated and applied in a finite time even if it is probably not the best way to proceed.

Then one has to follow arrows starting from each activities of the first configuration and verify that an equivalent arrow can be followed in the second configuration, following future values if necessary (and the same thing starting from the second one). Of course, as soon as a cycle is found, the verification must be stopped. In the worst case one follows, from any activity paths that leads to all locations in this activity and all locations in all the futures values. The set of locations being finite, this location is finite.

The alias conditions can be verified by checking that for all locations such that two sub-terms of this activity points to this location, take the two shorter[16] paths using these two references and verify that they are also aliases in the other configuration.

To prove that the second definition of $\equiv_F$ 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. One concludes easily that the set of possibly valid 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 locations 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, one only has 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

[16] Thus, we avoid having more than one cycle in each path, and we only have a cycle when the aliasing is due to a cycle of references.

by inductively verifying that the rules of Table 12.3 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 one marks locations already visited one has to verify, when arriving at two marked locations that they are equivalent according to Θ). □

12.10 Examples

A list of examples of equivalent terms is given below. The verification of equivalence consist in simply following the arrows on the diagram (trivial). The figures are considered as proofs hints.

In order to illustrate the second definition of equivalence relation, some details on the renaming that have to be used are given in that case.

In the case of Figure 12.3, a simple future value is updated. Thus, one must take:

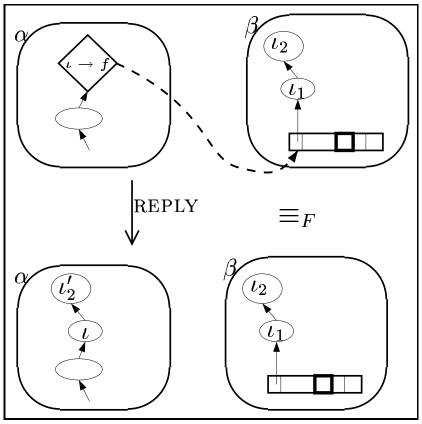


Figure 12.3: Simple example of future Equivalence

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

The Figure 12.4 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:

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

Figure 12.5 illustrates a case of cyclic proof. Note that this may only happen when there is a cycle of local references (locations).

The Figure 12.6 illustrates the importance of the bijectivity properties: in Q' every renaming is bijective but one would need to have $\theta_{\beta\to\alpha,\iota} = \{\!\{\iota_1 \leftarrow \iota, \iota_2 \leftarrow \iota'_2\}\!\}$ and $\theta_{\beta\to\alpha,\iota'} = \{\!\{\iota_1 \leftarrow \iota', \iota_2 \leftarrow \iota'_2\}\!\}$ which would be contradictory with $codom(\theta_{\beta\to\alpha,\iota})$ and $codom(\theta_{\beta\to\alpha,\iota'})$ disjuncts.

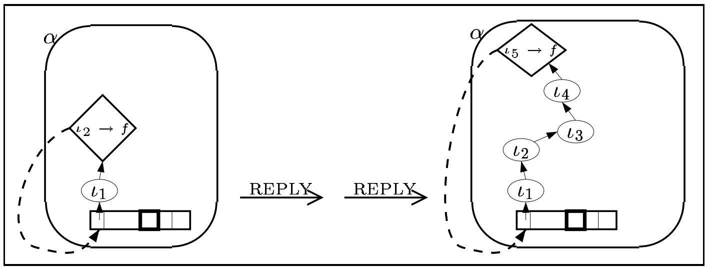


Figure 12.4: Equivalence in case of cycle of futures

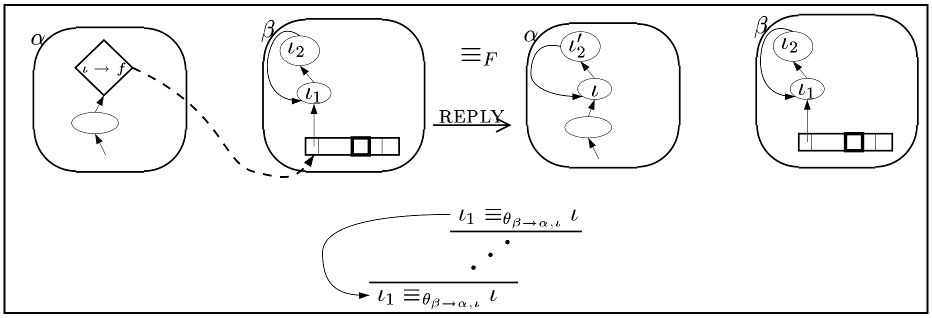

α
$\iota \rightarrow f$
β
ι_2
ι_1
$\equiv_F$
REPLY
α
ι_2'
ι
β
ι_2
ι_1
$\iota_1 \equiv_{\theta_{\beta\rightarrow\alpha,\iota}} \iota$
$\iota_1 \equiv_{\theta_{\beta\rightarrow\alpha,\iota}} \iota$

Figure 12.5: Example of "cyclic" proof

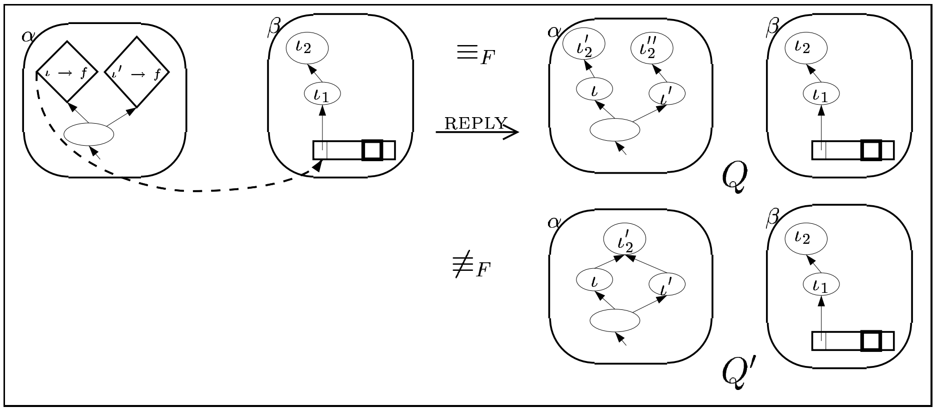


Figure 12.6: Another example

Chapter 13

Confluence Proof

13.1 Context

Let $\xrightarrow{*}$ be the reflexive transitive closure of $\longrightarrow$ and $\xrightarrow{\neg\text{REPLY}}$ any reduction except the REPLY rule.

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

Definition 13.1 (confluent configurations: $P_1 \curlyvee P_2$)

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

Let P_0 be an initial configuration. The goal of this chapter is to prove the following confluence property (which is also Theorem 11.1):

Property 13.1 (Confluence)

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

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.

Let $\mathcal{Q}(Q, Q')$ represent the set of configurations obtained from P_0 and compatible with both Q and Q':

$$\begin{aligned} \mathcal{Q}(Q, Q') &= \{R | P_0 \xrightarrow{*} R \wedge R \trianglerighteq Q \wedge R \trianglerighteq Q'\} \\ &= \{R | P_0 \xrightarrow{*} R \wedge \forall \alpha \in R,\, RSL_{\alpha_R} \trianglerighteq RSL_{\alpha_Q} \wedge RSL_{\alpha_R} RSL_{\alpha_{Q'}}\} \\ &= \{R | P_0 \xrightarrow{*} R \wedge R \bowtie Q \wedge R \bowtie Q'\} \end{aligned}$$

The principle of the proof is that in order to reduce terms Q and Q' to a common one, the terms derived from them will be constrained to stay inside $\mathcal{Q}$. Then completing the missing reductions from Q and Q' will lead to a common term (thus proving confluence).

13.2 Lemmas

The following lemma gives simple consequences of the fact that two stores are disjunct.

Lemma 13.2 (Independent Stores)

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

Lemma 13.3 (Extensibility of Local Reduction)

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

This lemma is trivially proven by checking it on each sequential reduction rule. Note that the using of $\equiv_F$ implies that we are placed in a configuration where all the futures and the active objects of σ, σ_0 and σ' are well defined. That is to say all stores and expressions are parts of a well formed parallel configuration (that is always the case when we use this lemma).

Lemma 13.4 (*copy* and Locations)

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

Proof : Direct consequence of the definition of the deep copy. □

Lemma 13.5 (Multiple Copies)

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

Proof : This proof will only focus on verifying the domain of the deep copy. The content of the store follows the same reasoning but needs more properties on the update (+) of two stores.

First, remark that

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

Thus if $\iota_0 \in copy(\iota', \sigma')$ then $\iota_0 \in copy(\iota', copy(\iota, \sigma) + \sigma')$.

Else if $\iota_0 \in copy(\iota, \sigma)$ then, let us prove $\iota_0 \in copy(\iota', copy(\iota, \sigma) + \sigma')$ by recurrence on the length of the path necessary to reach ι_0, which is the number of times the second rule of Table 10.1 must be applied to prove that $\iota_0 \in copy(\iota, \sigma)$.

- either $\iota_0 = \iota$ then $\iota_0 \in \mathit{copy}(\iota', \sigma')$ and $\iota_0 \in \mathit{copy}(\iota', \mathit{copy}(\iota, \sigma) + \sigma')$
- else $\iota_1 \in \mathit{dom}(\mathit{copy}(\iota, \sigma)) \wedge \iota_0 \in \mathit{locs}(\sigma(\iota_1))$ and
  by recurrence hypothesis $\iota_1 \in \mathit{copy}(\iota', \mathit{copy}(\iota, \sigma) + \sigma')$
  then $\iota_0 \in \mathit{copy}(\iota', \mathit{copy}(\iota, \sigma) + \sigma')$
  because $\iota_0 \in \mathit{copy}(\iota, \sigma) \subseteq \mathit{copy}(\iota, \sigma) + \sigma'$ and
  $\iota_0 \in \mathit{locs}(\sigma(\iota_1)) = \mathit{locs}\left((\mathit{copy}(\iota, \sigma) + \sigma')(\iota_1)\right)$.

We have proved that

$$\mathit{copy}(\iota, \sigma) + \mathit{copy}(\iota', \sigma') \subseteq \mathit{copy}(\iota', \mathit{copy}(\iota, \sigma) + \sigma')$$

The other inclusion (which is more natural) is a little more easy to prove but still uses a recurrence proof. □

Lemma 13.6 (Copy and Store Update)

$$\sigma' + \mathit{Copy\&Merge}(\sigma_1, \iota \ ; \ \sigma_2, \iota') \equiv_F \mathit{Copy\&Merge}(\sigma_1, \iota \ ; \ \sigma' + \sigma_2, \iota')$$
$$\mathit{if} \begin{cases} \mathit{dom}(\sigma') \cap \mathit{dom}(\mathit{Copy\&Merge}(\sigma_1, \iota \ ; \ \sigma_2, \iota')) \subseteq \mathit{dom}(\sigma_2) \\ \iota' \notin \mathit{dom}(\sigma') \end{cases}$$

The preceding lemma is a direct consequence of the definition of $Merge$ operator: In the right side of the equality, the only location that can belong to both the initial and the merged store is ι' (Property 10.1). Thus if $\iota' \notin \mathit{dom}(\sigma')$ the two store define disjuncts locations (but their codomain can be interleaved). For the left side, σ' and the merged store are disjuncts by hypothesis. This implies that the presence of the store σ' has no influence. σ' can be seen as an independent part of σ_2.

In fact we will use the following corollary:

Corollary 13.7 (Copy and Store Update) *If $\iota' \notin \mathit{dom}(\sigma')$ then there is a way of choosing locations allocated by $\mathit{Copy\&Merge}(\sigma_1, \iota \ ; \ \sigma_2, \iota')$ such that:*

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

Note that, in the following we can choose convenient locations and only have to verify $\iota' \notin \mathit{dom}(\sigma')$. That is due to the fact that configurations can be identified modulo renaming of locations.

13.3 Local Confluence

This section presents and proves the diamond property, that is to say the local confluence property necessary to establish confluence properties of Chapter 11.

The following property establishes what is usually called *local confluence*. It is the key property for proving confluence. It is strongly based on the definition of compatibility between configurations ($\bowtie$) because of the using of $\mathcal{Q}$.

Property 13.8 (diamond property) *Let P be a configuration obtained from P_0: $P_0 \xrightarrow{*} P$*

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

Proof : This proof is a (long) case study on the conflict between rules. Cases where one of the applied rules is REPLY will not be detailed. These cases can be verified but are not useful for the proof of the Property 13.11. Indeed, for REPLY rule this proof will only need the Property 12.13. However, conflicts between REPLY and other rules will be detailed (under other hypothesis) in the Appendix A.

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.

In the following, we suppose that one can choose any location or future name when one needs a fresh one. This is justified by the fact that reduction is not sensible (modulo equivalence) to futures and locations names. The fact that activities are chosen deterministically avoids the problem of renaming activities and ensures that the name of an activity will be the same for two application of the same NEWACT rule.

- 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 REQUEST rule: the source activity of the request)

- if the rules are the same, the activities will be indexed by 1 and 2.

The proof can be divided into four parts

13.3.1 Local vs. Parallel Reduction

LOCAL/LOCAL Obvious 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.

LOCAL/REQUEST

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

$\alpha_{\textsc{local}} = \beta_{\textsc{request}}$ let $\alpha = \alpha_{\textsc{request}}$ and $\beta = \alpha_{\textsc{local}} = \beta_{\textsc{request}}$

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

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

One can suppose (up to renaming) that the locations added to σ_β by the two rules are disjuncts. The deep copy of the argument of the request is added in an independent store thus $\sigma_{\beta 2} = \sigma_\beta :: \sigma$. Thus Lemma 13.3 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

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

$\sigma_{\beta 1}$ is obtained by some updates on σ_β:

$$\sigma_{\beta 1} = \sigma_0 + \sigma_\beta.$$

Corollary 13.7 is used for adding the request to the store obtained by local reduction. One can apply the request rule to P_1: let $\sigma'_{\beta 1}$ be the new store :

$$\sigma'_{\beta 1} = Copy\&Merge(\sigma_\alpha, \iota' \; ; \; \sigma_{\beta 1}, \iota'') \equiv_F \sigma_0 + Copy\&Merge(\sigma_\alpha, \iota' \; ; \; \sigma_\beta, \iota'')$$

and obtain a configuration equivalent to P'_2 (Lemma 13.2):

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

LOCAL/ENDSERVICE In ENDSERVICE, $\sigma'_\alpha = \sigma_1 :: \sigma_\alpha$ where $dom(\sigma_1) \cap dom(\sigma_\alpha) = \emptyset$. Thus Lemma 13.3 is sufficient to conclude.

LOCAL/SERVE No conflict.

13.3.2 Creating an Activity

NEWACT/NEWACT No conflict. One may only need to rename activities.

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

NEWACT/ENDSERVICE No conflict.

NEWACT/SERVE No conflict.

13.3.3 Localized Operations (SERVE, ENDSERVICE)

- **SERVE:**

SERVE/SERVE No conflict

REQUEST/SERVE Figure 13.1. If $\alpha_{\text{SERVE}} = \beta_{\text{REQUEST}}$: Informally, if one can perform a $serve(M)$ on P then there is a request matching the labels of M in the request queue so adding a new request to the request queue will not change the served one because SERVE takes the *first* request matching M. A better way of expressing mobility would be to create a new primitive that creates a new activity and replace the service primitive by a forwarder which, instead of treating a request forwards it to the newly created activity. Note that the fact that the first request is taken is essential to ensure confluence. A more complicated service primitive (like serving the last request arrived) would require further studies and would probably not verify confluence properties. Of course a $serve(\alpha)$ serving the *first* request coming from the activity α does not pose any difficulty here.

- **ENDSERVICE:**

ENDSERVICE/ENDSERVICE No conflict.

REQUEST/ENDSERVICE Figure 13.2. There can only be a conflict when $\alpha_{\text{ENDSERVICE}} = \beta_{\text{REQUEST}} = \beta$

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

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

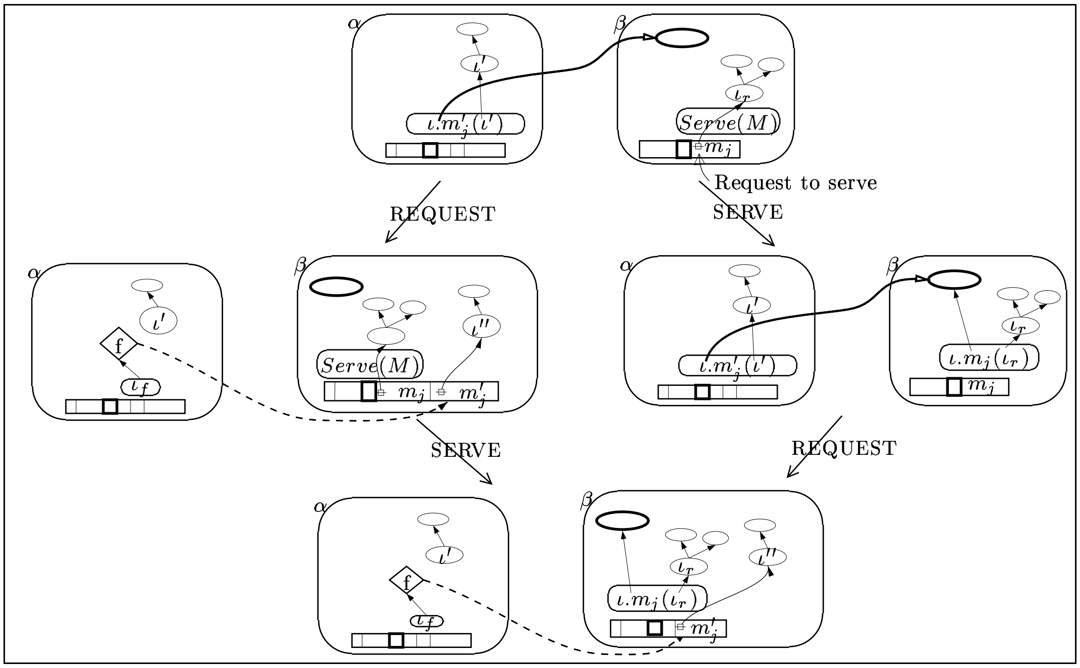


Figure 13.1: SERVE/REQUEST

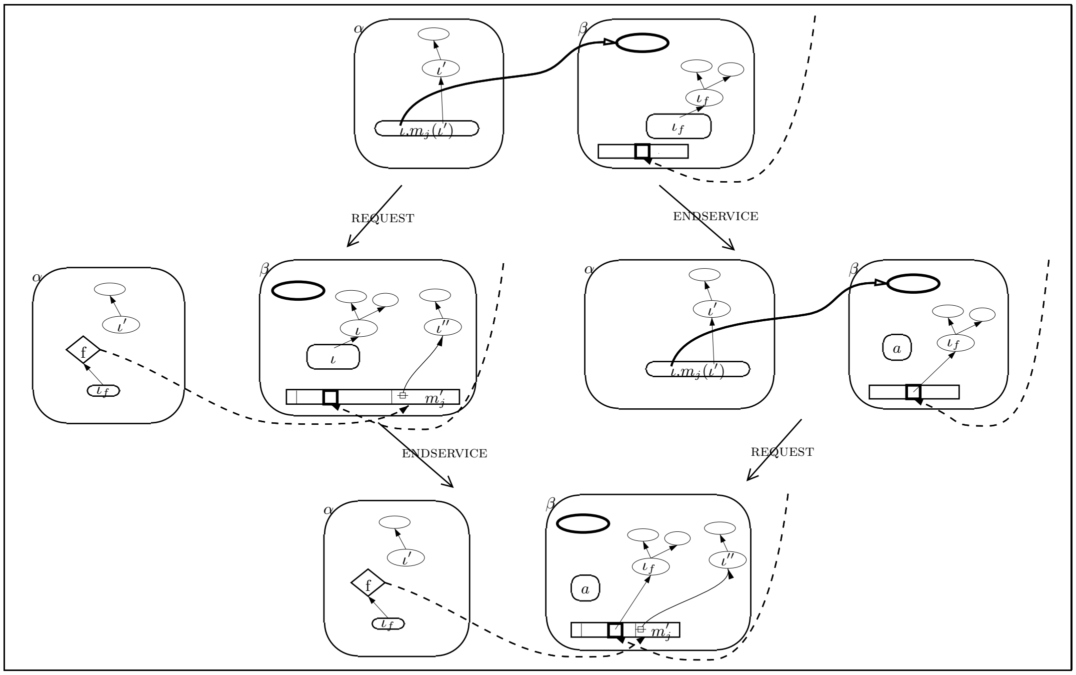


Figure 13.2: ENDSERVICE/REQUEST

The conflict only concerns the store. But the merges that are performed on the store are independent ($\iota'' \notin dom(\sigma_\beta)$). One can suppose that these two operations create disjuncts sets of locations. Then one can perform the missing two rules on the configurations P_1' and P_2'. A configuration with the stores

$$\sigma'_{\beta 2} \equiv_F \sigma + \sigma_{\beta 2} \qquad \text{and} \qquad \sigma' + \sigma_{\beta 1} \equiv_F \sigma'_{\beta 1}$$

is obtained. The crucial point of the proof uses Lemma 13.2 to prove:

$$\begin{aligned}\sigma'_{\beta 2} &\equiv_F \sigma + \sigma_{\beta 2}\\ &= \sigma + \sigma' + \sigma_\beta\\ &= \sigma' + \sigma + \sigma_\beta\\ &= \sigma' + \sigma_{\beta 1}\\ &\equiv_F \sigma'_{\beta 1}\end{aligned}$$

ENDSERVICE/SERVE No conflict.

13.3.4 Concurrent Request Sending: REQUEST/REQUEST

$\alpha_1 = \beta_2$ **or** $\beta_1 = \alpha_2$ Same kind of arguments as in the case LOCAL/REQUEST.

$\alpha_1 = \alpha_2$ No conflict.

$\beta_1 = \beta_2$ 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. □

Note that the preceding proof widely use (indirectly) the fact that the parts of store containing the requests arguments and the futures are isolated.

13.4 Case of the Calculus with $Serve(\alpha)$

In that case, requests can be safely exchanged as soon as they do not come from the same activity. As a consequence, no compatibility relation is necessary. The equivalence modulo futures on requests queues uses the restriction of requests queue to the requests having a given source activity:

$$R_{\alpha_P} \equiv_F R_{\alpha'_Q} \Leftrightarrow \forall \beta \in Act(P),\, R_{\alpha_P}\big|_\beta \equiv_F^l R_{\alpha'_Q}\big|_\beta$$

Note that this new equivalence relation also verifies

$$P \equiv_F P' \wedge P \xrightarrow{\text{SERVE}} Q \Rightarrow \exists Q',\, P' \overset{\text{SERVE}}{\Longrightarrow} Q' \wedge Q' \equiv_F Q$$

because for any $Serve(\beta)$ performed by α, the equivalence modulo futures ensures that the first request from β is the same (modulo future update) in P and P'.

Then, there can be a conflict in the concurrent request sending for $\beta_1 = \beta_2 \wedge \alpha_1 \neq \alpha_2$. But this conflict has no consequence because we still obtain equivalent configurations: the request queues are of the form :

$$R_1' = R :: r1 :: r2 \qquad\qquad R_2' = R :: r2 :: r1$$

with r_1 and r_2 coming from different activities. Thus, the resulting configurations are still equivalent.

To sum up, confluence come from the facts that

- Two requests coming from the same activity can not overtake each other. That is sufficient to ensure equivalence modulo futures.
- Even with this imprecise equivalence modulo futures, the request served by a $Serve(\alpha)$ primitive is the same for two equivalent terms.

13.5 Extension

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

Lemma 13.9 ($\equiv_F$ and $\mathcal{Q}(Q, Q')$)

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

Proof : Direct consequence of the Property 12.5: $P \equiv_F P' \Rightarrow P \bowtie P'$. □

Lemma 13.10 (REPLY vs. other reduction)

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

This lemma could be considered as strongly linked to the Property 12.14 It is more easy to prove (because P' can be reduced directly) but it ensures that P' can be reduced in a unique reduction rule.

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 the proof of the Property 12.14 suffices to conclude. □

Then, global confluence is a consequence of local confluence. The following property is a another formulation of the Theorem 11.1:

Property 13.11 (diamond property with $\equiv_F$)

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

Proof : If one of the $\Longrightarrow$ applies only REPLY rules then one can conclude immediately by corollary 12.14.

Else, both T_1 and T_2 are reduction rules different from REPLY and can be decomposed, there is P_1' such that:

$$P_1 \xrightarrow{\text{REPLY}^*} P_1' \xrightarrow{T_1} Q_1.$$

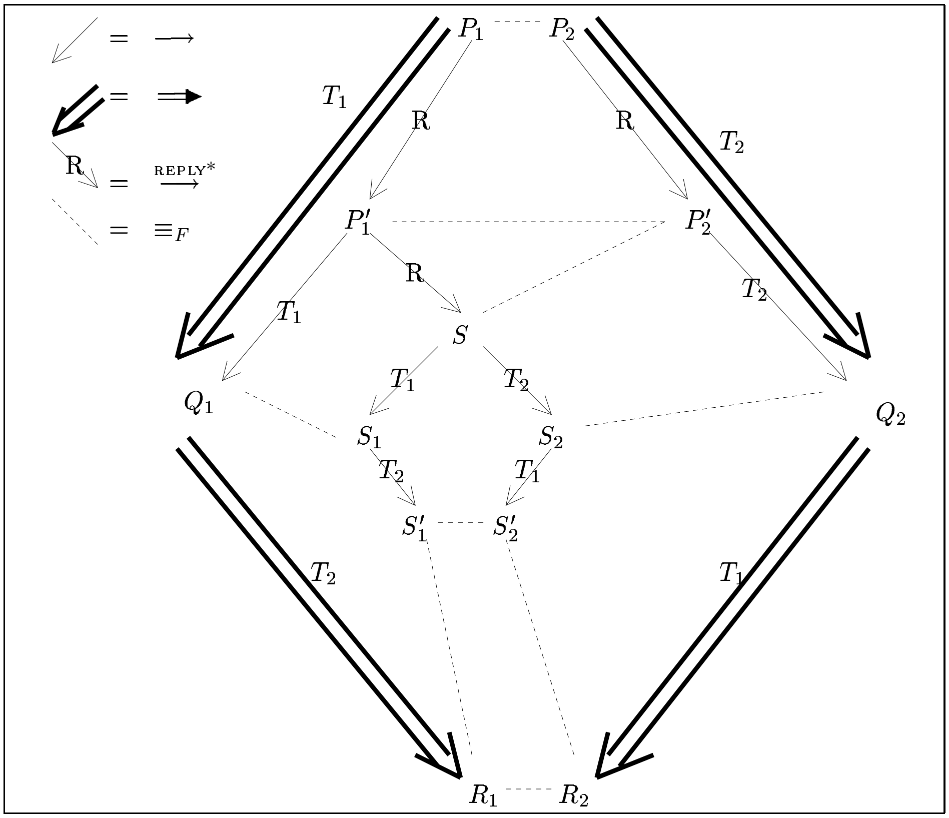


Figure 13.3: The Diamond property proof

Note that one could have $P_1 = P'_1$. In the same way, there is P'_2 such that:

$$P_2 \xrightarrow{\text{REPLY}^*} P'_2 \xrightarrow{T_2} Q_2$$

By Property 12.3, $\equiv_F$ is transitive and then $P'_1 \equiv_F P'_2$. By corollary 12.15:

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

Moreover, by Lemma 13.10:

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

Then, using diamond Property 13.8 (Lemma 13.9 is used to prove $S_1, S_2 \in \mathcal{Q}$):

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

Finally, using Property 12.14:

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

$$S_2 \xrightarrow{T_2} S_2' \wedge S_2 \equiv_F Q_2 \Rightarrow Q_2 \xRightarrow{T_1} R_2 \wedge S_2' \equiv_F R_2$$

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

$$\left\{ \begin{array}{l} Q_1 \Longrightarrow R_1 \\ Q_2 \Longrightarrow R_2 \\ R_1 \equiv_F R_2 \\ R_1, R_2 \in \mathcal{Q} \end{array} \right.$$

□

Part VI

Final Words

Chapter 14

Implementation Strategies

14.1 Garbage Collection

The study of garbage collection mechanisms is out of the scope of this study. In this section, we simply describe the aspects of garbage collection which are linked to ASP. The objective here is just to give informations in order to show how to adapt an existing garbage collector (or at least existing techniques for garbage collection) to the special case of ASP. Because of the objects topology in ASP described in Section 11.2, a garbage collection mechanism on ASP can be separated into three concerns. A local garbage collection phase, and the distributed garbage collection of futures and activities (the only generalized references in ASP). Of course, in practice, the handling of these three concerns should interact.

14.1.1 Local Garbage Collection

First note that parameters of requests and values of futures are situated in *isolated* parts of store and thus could be put in a different store (Property 11.1).

Thus, local garbage collection can be performed easily: useful objects are referenced by the current term (a_α), the active object (ι_α), the futures list (F_α), and the requests lists (R_α). Note that the best moment for a garbage collection step seems to be after the ENDSERVICE. Indeed, every object that has only been allocated for serving the preceding request will not be useful any more.

Note that the global references consists only in futures and active objects. Thus the first step of garbage collection described previously is clearly a purely local one and can be performed by classical and well known techniques.

14.1.2 Futures

First, one must note that futures garbaging depends on the future update strategy.

If the strategy does not update futures as soon as their values are calculated, then, futures have to be kept in the futures values list in order to be potentially used later on for update. From a Garbage collection point of view, immediate future reply is much simpler but it does not allow a future to be updated at any time and we consider this as too restrictive.

Therefore, as the future might have proliferated in a lot of activities, it is difficult to decide when a future can be removed from the list of futures values. Of course, in practice, reference counting or any distributed garbage collector mechanism could be used to perform distributed garbage collection of futures. See, for example, [LQP92] and [Fes01] for the study of distributed garbage collectors. Indeed, future references are particular global references, and all these frameworks dealing with garbage collection of global references can be applied to the particular case of futures.

14.1.3 Active Objects

In fact, garbage collection of active objects could be more generally called garbage collection of activities. Active objects references are also generalized references that can be spread over the different activities. In order to perform garbage collection of active objects one first needs to determine if this active object (or activity) is referenced from "useful" activities (classical garbage collection). Then an activity can be garbage collected if it is no more referenced, the activity does not contain any more pending requests, and does not have any more proper activity (empty current term with no continuation). In that case all the activity *except the futures list* can be garbage collected. Of course references to futures contained within this activity can still exist, their garbage collection is dependent on the constraints described previously.

Chapter 15

ASP Versus Other Concurrent Calculi

This chapters compares ASP and the different calculi and languages introduced in Chapter 6. The objective of this chapter is both to compare ASP communication and parallelism mechanisms with existing ones, and to discuss how the ASP concepts can be adapted to other calculi. We will also present how tools developed on other calculi could be adapted to ASP.

Generally, with respect to shared memory calculi, our objective is to design a calculus with accesses local to a process and between processes as uniform as possible, without using a shared memory mechanism. Note that as soon as no memory is shared, some copying of data is needed, and the semantics of local and remote communications is still different: ASP defines a semantics with implicit data copying upon communication between processes.

15.1 Actors

Relying on the active object concept, the ASP model is rather close to, and was somehow inspired by, the notion of *actors* [AMST92, AMST97]. Both rely on asynchronous communications, but actors are rather functionals, while ASP is in an imperative and object-oriented setting. While actors are interacting by asynchronous *message passing*, ASP is based on asynchronous *method calls*, which remain strongly typed and structured and most importantly more adapted to object-oriented framework. From a more fundamental point of view, Actors and ASP differ by the definition of state. As in ASP the state is encapsulated in objects fields whereas, in Actors, the state is encoded in the actor behavior. Furthermore, in the Actors model every actor acts independently and has its own thread, but ASP adopts a less uniform model where only some objects are active. Generally, the ASP application designer has the possibility, and responsibility to achieve this partition for the sake of distribution and parallelism. Thus, starting from very similar objectives, ASP and Actors are very different calculi where a lot of crucial points differ (activity definition, communications, asynchrony, futures, states, . . .).

ASP future semantics, with the store partitioning property (isolation between future values, the active store, and the pending requests), accounts for the capacity to achieve confluence and determinism in an imperative setting. Finally, the bisimulation techniques used by Agha et al. in [AMST92, AMST97] would have been inadequate to obtain the main result presented in the current thesis: a strong, somehow intrinsic, but dynamic property on processes interacting by asynchronous communications.

To some extent, this study develops the idea introduced in [AMST92, AMST97] that "The behavior of a component is locally determined by its initial state and the history of its interactions". ASP extends this idea on an imperative language. We chose to take into account a more global history in order to be only sensitive on the order of the message senders instead of the complete history of messages. In other words, we do not need to compare all message contents (arguments of method) in the history of an activity but instead we compare the history of just message senders (RSL) of *all* activities. Our properties state that in ASP, the history of an activity interactions is uniquely determined by the RSLs of all activities.

15.2 ς-calculus and related

First note that this study is strongly based on the works of Abadi, Cardelli, Gordon, Hankin and al. about **imp**ς-calculus [AC96, GHL97a, GHL97b].

Proving equivalence between terms can be performed by introducing bisimulation on an object calculus like in [GR96]. This thesis introduces an equivalence relation specific to ASP, and actually some aspects of equivalence modulo replies are close to bisimulation techniques. But CIU (Closed Instance of Use) equivalence introduced in [GR96] deals only with static terms. In order to capture the intrinsic properties of the calculus, we are interested in dynamic properties like confluence, thus CIU equivalence is inadequate to our problem. As a perspective, more static properties could be obtained from confluence property in order to perform static analysis of programs.

From a different point of view, Gordon and Hankin [GH98], and Jeffrey [Jef00] also introduce parallel calculi based on threads and shared memory, it is also inadequate to our case because it would not fit the characteristics of communications that we want to model.

15.3 π-calculus and related

ASP calculus could be rewritten in π-calculus [MPW92, Mil93] but this would not help to prove confluence properties directly. Indeed, in π-calculus, synchronization is based on channels. On the contrary, ASP relies on data-driven synchronization over an imperative object calculus, and thus its semantics is different from π-calculus. Indeed, while the synchronization in ASP is implicit, π-calculus impose to explicitly perform synchronization on channels. Thus, writing ASP programs in π-calculus would require to know the first point where the value of a future is needed, and to write explicit communication for the reception of the replies. Note that, in π-calculus, the information contained in $f^{\alpha\to\beta}$ does not be sufficient to completely build these channels: somehow the receiver of the reply must listen on the channel and is not encoded in $f^{\alpha\to\beta}$. An approximation of the first point in a program using a future could be computed statically, but such a static analysis (strictness analysis) seems to be both complicated and imprecise. In general, finding the exact first point using a future is undecidable.

In the same way, PICT also necessitates channel based synchronization.

For example, the following code would be problematic (b and c are two boolean variables, r.wait access to a field of r performing a wait-by-necessity):

```
foo(bool b, bool c)
    r = oa.m();                      gets a future
    if b then r.wait;                performs a wait by necessity
    if c then r = [a = 1, b = 2];    creates a local object
    oa2.bar(r);                      sends to oa2 a possible future possibly awaited
```

The example above shows that one can not determine whether a future is already used, if a future is still (or may still be) awaited at a point, and even if a variable contains or not a future.

To conclude, we do not think that translating ASP into π-calculus would simplify our specifications or our proofs. As a consequence it was more effective to focus our work directly on ASP rather than obtaining result on translated terms (which will not necessarily give us results on our initial calculus).

Under certain restrictions π-calculus channels are called *linear* and *linearized* [NS97, KPT96]. The terms communicating over linear channels can be statically proved to be confluent and such results could be applicable to some ASP terms. First, the confluence property principle is very closed to linearized channels. But, even if our confluence property is in general not statically verifiable, it is much more powerful and several static approximations of these properties can be performed, some of them would probably be closed to linear(ized) channels.

Channels in ASP

Let us introduce the notion of channels in ASP and precise the relations between π-calculus and ASP channels.

Let a channel be a pair *(activity, method label)*, and suppose every *serve* primitive concerns a single label[17]. If at any time only one activity can send a request on a given channel then the term verifies the *DON* property, and the program behaves deterministically. In π-calculus such programs would be considered as using only linearized channels and would lead to the same conclusion. Note that in ASP, updates of response along non-linearized channels can be performed which makes ASP confluence property more powerful. Moreover, this definition of channels is more flexible because it can contain several method labels and then, one can wait for a request

[17] If several methods can be served by the same primitive, then they must belong to the same channel, and a channel becomes a pair (activity, set of method labels).

on any subset of the labels belonging to a channel, in other words we can perform a *Serve* on a part of a channel without losing determinacy.

A first step towards expression of channels in ASP has been presented on Process Networks in Section 9.3. A more formal translation for ASP channels will be given in the Section 15.6 below. What is surprising here is that the notion of channels in Process Networks and linear channels in π-calculus come from very different frameworks but could be adapted to ASP through similar abstraction of *channels*; to some extent, ASP generalizes both π-calculus and Process Networks channels abstractions

15.4 Ambient Calculus

Ambient calculus is based on locations. The objective of this thesis does not have the same concerns as ambient framework. One of ASP objective is to abstract away locations, and to provide determinate distributed systems insensitive to location, whereas Ambient calculus studies the effect of locations on distributed computations. These two studies could be considered as complementary. For example, ASP communications could replace π-calculus like communications in ambients. Also note that the triggering construct could be simulated provided a non-blocking primitive that inspects the requests queue is added to ASP.

15.5 Join-calculus

Synchronization in the join-calculus is based on filtering patterns over channels. The differences between the channel synchronization and the data-driven synchronization described for the π-calculus also make the join-calculus inadequate for expressing the ASP principles.

15.6 Process Networks

The DON property widely used in this study is somewhat inspired from process networks. Indeed the ASP channel view introduced in Section 15.3 can also be compared to Process Networks channels.

Process Networks of Kahn [Kah74, KM77, WWV00] are explicitly based on the notion of *channels* between processes, performing *put* and *get* operations on them. Process networks provide confluent parallel processes, but require that the order of service is predefined and two processes cannot send data on the same channel, which is more restrictive and leading to less parallelism than ASP.

As shown in 9.3, the process networks channels can be translated in any direction (from process performing a get to process performing a put: *push*, or in the opposite direction: *pull*). But this could not be considered as a systematic translation from ASP to Process Networks, because Process Networks channels can be passed in parameter to processes at creation. In practice, in ProActive, we can use reflection to pass method names as parameters, but the theoretical aspect of this solution has not been studied yet.

The following of this section tries to explicit more formally and more generally the translation from process networks to ASP.

15.6.1 Expressing Process Networks channels

We present here a translation of process networks terms in ASP in the case where each channel has only one destination process. We introduce a channel object:

$$\begin{aligned}Channel \triangleq [values = [];&activity = \varsigma(s,\_)Repeat(Serve(put); Serve(get)),\\ &put = \varsigma(s, val)s.value := val, \quad get = \varsigma(s,\_)s.value]\end{aligned}$$

One just has to create a new channel by *let* $Q = Active(Channel, activity)$ *in* ... and use it by $Q.put(\ldots)$ and $Q.get()$. Such channels are first class entities that can be manipulated with even more expressiveness than process networks ones. Note that the request queue of the channel activity will play the buffering role of process networks channel.

The multiple destination channels declaration does not raise any technical difficulty, but is much longer to describe. As proposed in [KM77] we could have a different reference for each activity that can read on a multiple destination channel (different views of the same channel).

15.6.2 ASP is more expressive

Like in the π-calculus case, ASP channels seem more flexible and our property more general.

First, if we consider a channel as a pair *(activity, method label)*, we can perform a wait ($Serve$) on several channels (several labels) at a given program point. This would be expressed in process networks by the possibility to perform a *get* on several channels and take the first request on one of these channels thus merging these channels.

ASP provides a more structured programming model where the causal flow of data can remain in the program structure, for instance through method parameters and results, while a process networks would require several un-obviously associated channels.

Even more, the fact that future updates can occur at any time accounts very much in ASP expressiveness: return channels do not have to verify any constraint. Thus, futures can be seen as hidden and "automatic" transparent channels. These hidden channels are really difficult to simulate in process networks because their existence is based on a data-driven synchronization. Thus, ASP data-driven synchronization, and more generally the futures mechanism can be considered as one of the most original features of this thesis.

15.7 Obliq and Øjeblik

Let us compare ASP to Obliq shared memory and serialized objects with synchronous method calls. In ASP, the notion of executing thread is linked to the activity, and thus every object is "serialized" but a remote invocation does not stop the current thread. In other words, there is a unique thread by active object and the parallelism in ASP is due to the coexistence of several activities. Finally, in ASP data-driven synchronization is sufficient and no notion of joining threads is necessary. To summarize, the notion of thread is replaced in ASP by the process associated to each activity and a wait-by-necessity for synchronization. Furthermore, data-driven synchronization alleviates the programmer from the explicit insertion of synchronization, thus it seems a very convenient way of programming.

With respect to the confluence, the generalized references for all mutable objects, the presence of threads, and the principle of serialization (with mutexes) make the Obliq and Øjeblik languages very different from ASP. In fact there is no way of ensuring a confluence property similar to ours in Obliq. For example, to avoid concurrent accesses to shared objects we would need to remove mutable objects from the calculus which would be contradictory with one of the main characteristics of ASP: its imperative aspect.

15.8 The $\pi o\beta\lambda$ language

In $\pi o\beta\lambda$, a caller always waits for the method result (synchronous method call) which can be returned before the end of the called method. In ASP, method calls are asynchronous thus more instructions can be executed in parallel: the futures mechanism allows one to continue the execution in the calling activity without having the result of the remote call. A simple extension to ASP could provide a way to assign a value to a future before the end of the execution of a method. Note that in $\pi o\beta\lambda$ this characteristic is the source of parallelism whereas in ASP this would simply allow an earlier future update (and potentially a shorter wait-by-necessity). In ASP the source of parallelism is the object activation and the systematic asynchronous method calls between activities. The condition given in [JH96], stating that the result of a method is not modified after being returned, is balanced in ASP by a deep copy of the result (Property 11.1: Store partitioning). Similarly, the *unique reference* condition from the same work is balanced in ASP with the constraints on objects topology (no remote reference to passive objects).

15.9 Multilisp

With respect to Multilisp, the shared memory mechanism is the main difference between the two languages but this difference has a lot of consequences. Therefore, it seems difficult to compare precisely these two frameworks. The main common point between ASP and Multilisp are the futures, but futures in Multilisp are not global references, and as such much simpler to update than in ASP: they only need to be updated once in the shared memory.

Chapter 16

Conclusion

In this study, we proposed a calculus modeling asynchronous communications in object-oriented systems, and exhibited confluence properties. Such properties simplify programming as they avoid having to study every possible interleaving of instructions and messages to understand the behavior of a given program.

In this thesis, we presented a new calculus named ASP and proved sufficient conditions for confluence. Our objective was to provide a formalized and general framework with general properties suitable for open systems. From those dynamic properties more static and easily verifiable conditions for confluence could be derived. Moreover, the properties proved in this thesis and their formalization already have great practical consequences, at least on ProActive.

The ASP calculus is based on asynchronous *activities* processing *requests* and responding by mean of *futures*. Inside each activity, the execution is sequential and there is a one-to-one corresponding between processes and active objects.

Concerning asynchrony and synchronization, when a process has sent a request, it can perform other operations while the result value is not needed and the result to come is represented by a future. Such futures are first class entities that can be passed as parameter and results. For synchronization, a rather natural *data-driven synchronization* occurs when the real value of the result associated to a future is needed. This mechanism is called *wait-by-necessity* and the formal study of this data-driven synchronization is one of the main contributions of this thesis.

ASP ensures a confluence property of compatible terms: two configurations with *compatible* RSLs (Request Sender List) are confluent. RSL compatibility is based on a prefix order on sender activities. Thus *execution is only determined by the ordered list of activities sending request to a given one.* What makes ASP properties powerful is that the execution is insensitive to the moment when results of requests are obtained. Consequently, an equivalence relation was introduced to consider equivalent a term before and after the REPLY reduction. We defined a sub-calculus of ASP formed of *Deterministic Object Networks* (DON) terms that behave deterministically (Theorem 11.4). We proved that every program communicating over a tree (Theorem 11.3: Tree Determinacy) behaves deterministically, even in the highly concurrent case of a FIFO service.

Even if ASP is a new calculus it has been greatly inspired by most of existing concurrent languages and calculi (e.g. ς-calculus model of objects, DON property strongly related to process networks and can also be linked with linear channels of π-calculus).

The properties of ASP illustrate the fact that futures in an imperative language seem rather powerful and convenient. What actually provides confluence in ASP is the balance between asynchronous communications, and synchronizations due to the wait-by-necessity. Of course the topology of objects stating that we can only access an activity through its active object is also important. Out of order replies and data-driven synchronization also provides a convenient way of programming parallel and distributed applications. Indeed, the programmer only has to ensure that the result *can* be calculated[18] at the moment when the value is needed to avoid deadlocks.

ProActive is a Java API implementing the ASP model. One of our objective is to be able to use in ProActive the properties proved on ASP. For example, the fact that the moment when a future is updated is not observable in ASP, can be used to choose any strategy for updating futures in ProActive.

Note that the proofs detailed in this thesis can also be adapted to prove other cases of confluence in ASP or similar calculus. For example we gave details about the adaptation to the case where the service primitive explicitly specifies the source activity of the request.

[18] In the sense that there is an order for performing computation ensuring that this result is calculated.

Chapter 17

Perspectives

17.1 Static Analysis

This thesis presented dynamic properties ensuring confluence. One important perspective is to deduce static properties from the properties presented here in order to be able to identify statically deterministic programs.

As a first perspective, confluence properties could be ensured statically. A static approximation of DON programs would provide a convenient way of ensuring confluence. A first step would be to specify a precise type system for $\mathcal{M}_{\alpha_P}$, either for static service verification, or for inferring the potential services. Such a type system would be crucial in order to statically identify DON programs.

Indeed, the set of active objects and their topology could be approximated statically. Classical abstract interpretation or static analysis techniques [Deu94, Deu95, SRH96, SP81] can be used here but a more approximate methodology like balloon types [Alm97] could also be useful.

Balloon types [Alm97] express a way of restricting the object topology by typing. The balloon types topology is a sub-case of the topology that is sufficient for confluence of ASP programs and it is simple to verify (typing). Indeed, if we applied a balloon types methodology, it would create an objects topology where references between activities form a tree and these reference only link active objects. In ASP, passive objects can reference active ones. Thus the topology of objects ensured by [Alm97] is too restrictive.

Another static approximations of tree topology properties have been performed in [ACG00], it is less restrictive than balloon types. The topology of object graph have been analyzed by a static analysis [SRH96] in order to parallelize programs deterministically.

17.2 Components

Taking advantage of the ASP properties, this section demonstrates how to build hierarchical, distributed, and deterministic components. This could be considered as a first study in order to provide a calculus modeling asynchronous and distributed components. In order to adapt ASP calculus to component programming, one could first describe a calculus inspired by ASP and the following diagrams. The most interesting perspective consists in studying the problem of reconfigurations of such components which could probably be linked with the reconfiguration of process networks.

As already explained, DON property can be considered as a generalization of process networks in an object-oriented framework. Furthermore, process networks are actually not too far from asynchronous and distributed components. While process networks channels just carry data, component interconnections carry method calls in a much more typed and object-oriented way. Out of order futures updates is also an important feature not present in process networks.

17.2.1 From Objects to Components

We consider a simple component model informally defined as follows. A *primitive component* is defined from an activity α (a root active object and a set of passive objects), a set of *Server Interfaces* (SI), and a set of *Client Interfaces* (CI).

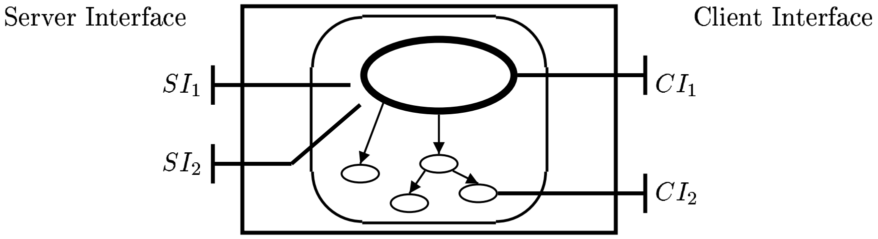


Figure 17.1: A primitive component

Remember that $\mathcal{M}_{\alpha P_0}$ is the set of M (M is itself a set of method labels) that can appear in the $Serve(M)$ instructions of a given activity α. Each Server Interface SI_i is a subset of the served methods ($SI_i \subseteq \bigcup_{M \in \mathcal{M}_{\alpha P_0}} M$)[19]. A client interface (CI_j) is a reference to an(other) activity contained in any (single) attribute (field) of an activity.

From primitive components, *composite components* can be built by interconnecting primitive components and exporting some SIs and CIs.

Figure 17.2 represents a component version of Fibonacci example of 9.4. A primitive component *Add* can be built up from active object *Add*. *Cons1* and *Cons2* have been merged in a composite component (composed of two primitive components). A controller component *Cont* has been added. It exports a server interface (*ComputeFib*(k)) taking an integer k and forwards $k-1$ times its input to *Cons1*. According to the program of 9.4, $Cons2$ sends *send* requests on the client interface. These components are connected in a single composite component FIB exporting a server interface *ComputeFib*(k) and producing *Fib*$(1)\ldots$*Fib*(k) on a client interface *send*(i).

[19] Served methods that do not belong to an interface correspond to asynchronous calls that can only be internal to a component.

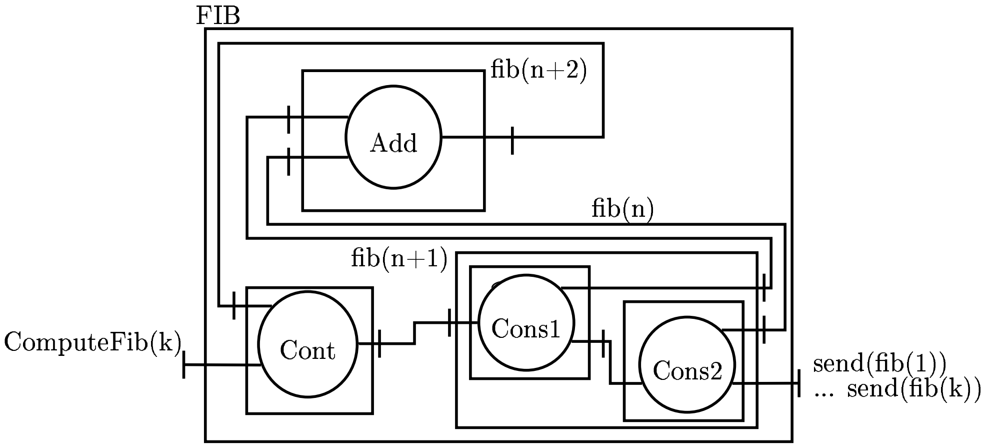


Figure 17.2: A composite component

17.2.2 Deterministic components

Using ASP properties, we introduce deterministic assemblage of asynchronous components.

Definition 17.1 (Deterministic Primitive Component (DPC)) *A DPC is a primitive component defined from an activity α. It verifies that server interfaces SI are disjoint subsets of the served method of the active object of α such that every $M \in \mathcal{M}_{\alpha P_0}$ is included in a single SI_i, let us first note:*

$$\mathcal{M}_{\alpha P_0} = \{M_j | j \in J\}$$

Then, let K_i be disjoint subsets of J, in fact one only needs that:

$$I \subseteq \mathbb{N},\ \forall i \in I,\ K_i \subseteq J$$

Then the SI_i server interfaces should verify:

$$SI_i = \left\{ \bigcup_{j \in K_i, M_j \in \mathcal{M}_{\alpha P_0}} M_j \right\} \text{ such that } \forall k,\ i \neq k \Rightarrow SI_i \cap SI_k = \emptyset$$

This definition implies that the set K_i are disjoint (consequently one only needs $K_i \subseteq J$):

$$i \neq i' \Rightarrow K_i \neq K_{i'} = \emptyset$$

DPC ensure that there is no interference between the services on the different interfaces: request sending on different interfaces will not interfere.

Definition 17.2 (Deterministic Composite Component (DCC))
A DCC is

- *either a DPC,*

- *or a composite components connecting DCC such that at any point in time, the binding between its sub-components* (CI_j, SI_i) *is one to one.*

Of course, the binding between components can be cyclic. The deterministic behavior of a DCC is a direct consequence of properties of Chapter 11. Indeed, a DCC assemblage verifies the DON property: there is no interference between the services on different interfaces inside a primitive component (DPC), and two CI will not be (non-deterministically) merged to a single SI (DCC).

Property 17.1 (DCC determinism) *DCC components behave deterministically.*

Note that the composite components remain deterministic if two CI from the *same primitive component* are merged to a single SI. But this generalization is no more valid in the case of composite components connection.

17.2.3 Components and Futures

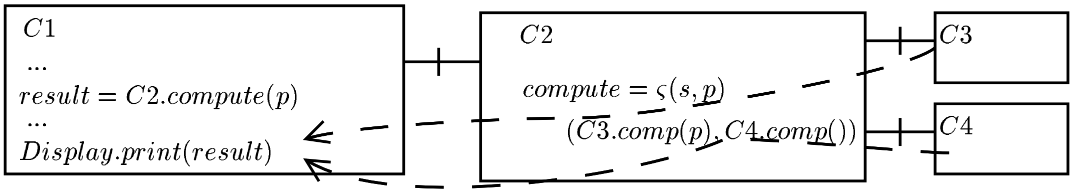


Figure 17.3: Components and Futures

Consider the component assemblage of Figure 17.3; for the sake of simplicity, components and active object names are identified. In this example, $C1$ calls $C2$ on the interface *compute*, the result of this call will be printed. $C2$, within the method *compute*, builds a result that includes parts coming from delegation to $C3$ and $C4$.

First of all, all the calls being asynchronous, the coupling between components remains quite low: $C2$ is not blocked while $C3$ and $C4$ are computing their part of the answer. Furthermore, without any explicit continuation or call back, the part of the result computed by $C3$ and $C4$ will transparently reach $C1$.

This kind of asynchronous assemblage is much more structured and transparent than call back solutions, and it enables unlimited composition. Moreover, whatever the number of delegations, whatever the depth of the composite, if definitions 17.1 and 17.2 are ensured, the composition is deterministic.

17.3 Generalizing Confluence

In the case of a server applications, there may be no way of ensuring the DON property. But this server may behave deterministically because it has a *functional behavior*: its internal state is never modified. In other words, it is a *stateless service*. Formalizing the fact that such an application server has no influence on confluence seems to be an interesting perspective. But a new equivalence relation and a new confluence proof would be needed and a complete study of this case is not obvious. Interesting perspectives consist in studying the interleaving of such functional active objects with imperative ones.

17.4 Temporized Requests

We have made a first study in order to have a general confluence property in ASP without hypothesis on objects topology and behavior. The idea was to have a time expressing the causality between requests. A time is appended to each request and to each activity (*execution time*). The principle is that a request will only be served when its time is greater or equal to the time of its destination activity.

Moreover, we impose that the time appended to a request is greater than the time of the request that creates this request (for *causality* reasons). Furthermore, if a request invokes two requests the second one has a time greater than the first one (*sequential execution* inside a request). That can be performed easily by considering the time as a list of integers ordered lexically.

The problem now is to define the evolution of the time inside each activity without using a centralized server which would be unsatisfactory and unadapted to a wide-range distributed language.

In the case where there is no cycle in the activities dependence graph, a correct algorithm consists in having a time that continuously grows in the root activities. More precisely, the request are treated without condition and the activity time is fixed to the treated request time. For any other activity the time must be either the time of the treated request or (if no request is treated) the greatest lower bound of all activities referencing this activity. With such an algorithm, we ensure that an activity does not receive a request that it should have treated in the past. Equivalently, there is no request sent with a time strictly smaller than the execution time of the destination activity.

We proved this solution allows to order treating of requests for a fixed DAG topology between activities. The main problem is not variable topology, but cycles. In the sub-case where there is no overlapping cycles, another less simple algorithm can be found. In the most general case a much more complex algorithm seems to work but we did not manage to prove it yet and it is too much complex to be used in practice!

17.5 Mobility

ASP is a calculus suited for expressing distributed objects: as references to active objects are global, each activity can be placed on a different machine. We did not explicitly include a primitive for moving objects between different machines. Usually, active objects are identified by their activity name (α, $\beta \ldots$). The *Active* primitive can be used to create a new activity with the same active object. It will create a clone of the initial activity, potentially at a different location, hence simulating mobility. However, it does not move the pending requests which would still be treated in the old activity.

Thus one has to continue the old activity by unceasingly forwarding (in FIFO order) all the requests in the requests queue to the newly created activity. The old activity acts as a forwarder and is also useful for storing and replying previously computed futures values. Using this method, all the request and futures will be automatically forwarded by the old activity.

Note that this methodology first required that any activity can have the location of its active object. This can be encoded in ASP. Then one could migrate an object with:

$$newao = Active(thisActivity); CreateForwarders(newao); Repeat(Serve(M))$$

Where $CreateForwarders(newao)$ replaces the body of each method of the current activity by a forwarded call:

$$m_j := \varsigma(x, y).newao.m_j(y)$$

This necessitate to have updatable methods in ASP. Updatable methods could be added to ASP or be encoded by updatable fields containing lambda-expressions.

Such work is, clearly strongly related to the assertion:

$$surrogate = \varsigma(s)s.alias\langle s.clone).$$

in Øjeblik.

The number of replies could be lowered by authorizing a future to be returned from an activity that was not the original destination of the request. In that case, a new mechanism for finding the activity that has calculated the value of a future would be necessary[20].

[20] In the absence of mobility the activity that calculates the value associated to the future is encoded in the future identifier.

Appendix A

Another Proof of Confluence

A.1 Aims and Interest

This appendix gives another confluence property with its proof. Even if the following confluence property is much less interesting than the general confluence property presented in Chapter 11, It is simpler and more adapted to the tree topology determinism. Indeed, in that case the confluence property is stronger as it is valid modulo a weaker equivalence relation: renaming of futures and locations instead of future updates. Indeed in the case of a tree topology, there is no cycle in the dependence between futures and futures can be exactly updated in all activities. In that case global confluence is a direct corollary of local confluence.

The proofs below will require to study cases that have not been studied in Chapter 13. Indeed, the proof presented in this appendix is more complex but do not require to introduce the equivalence modulo futures which must verify a lot of lemma. More precisely, the cases detailed here are generally balanced with the Property 12.13 ($\equiv_F$ and reduction) which required a long proof.

Finally, we aim at proving the following property

Property A.1 (Tree Dependence Graph Determinacy)
If, at every step of the reduction ($\forall Q$ s.t. $P \xrightarrow{*} Q$), the object dependence graph (of Q) forms a set of trees then the reduction is deterministic:

$$\begin{cases} P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \end{cases} \Longrightarrow \exists R_1, R_2 \begin{cases} Q_1 \xrightarrow{*} R_1 \\ Q_2 \xrightarrow{*} R_2 \\ R_1 \equiv R_2 \end{cases}$$

A.2 Hypothesis

Let us give a formalization of the tree topology previously introduced in this study. We will suppose that the *object dependence graph is a tree.* The idea is that if two objects have a generalized reference on a third one then they will be able to send requests concurrently.

Hypothesis H A.1 *The object dependence graph is a set of trees.*

That is to say, at every time, each active object has a unique father in the dependence graph and there is no cycle of dependencies:

$$\begin{cases}\forall\alpha,\beta,\gamma\in Act, OA(\alpha)\in\beta\wedge OA(\alpha)\in\gamma\Rightarrow\beta=\gamma\\ \nexists n\geq 0,\;\;\nexists\beta_0\ldots\beta_n\in Act,\;\;\forall i<nOA(\beta_i)\in\beta_{i+1},\;\;OA(\beta_n)\in\beta_0\end{cases}$$

We can deduct the following property that ensures that under hypothesis (HA.1) there is no cycle of future references:

Property A.2 ((HA.1) ⇒ no cycle of futures)

$$(HA.1)\Rightarrow\nexists n\geq 0,\;\;\nexists\beta_0\ldots\beta_n\in Act,\begin{cases}\forall i<nfut(f_i^{\gamma_i\rightarrow\beta_i})\in\sigma_{\beta_{i+1}}\\ fut(f_i^{\gamma_i\rightarrow\beta_n})\in\sigma_{\beta_0}\end{cases}$$

A.3 Context

The local confluence property is the following one:

Property A.3 (Local confluence)

$$\begin{cases}P\rightarrow_{\|_2}Q_1\\ P\rightarrow_{\|_2}Q_2\end{cases}\Longrightarrow\exists R_1,R_2\begin{cases}Q_1\rightarrow_{\|_2}R_1\\ Q_2\rightarrow_{\|_2}R_2\\ R_1\equiv R_2\end{cases}$$

Where $\rightarrow_{\|_2}$ is derived from the parallel reduction $\longrightarrow$. $\rightarrow_{\|_2}$ is defined below.

A.3.1 The Special Case of the REPLY Rule

The problem here is the possible propagation of generalized references to each future. To be able to find a common term in one step, one may need to update in parallel several references to a given future ($f^{\gamma\rightarrow\beta}$) both in the same and in different activities. $\rightarrow_{\|_2}$ is equal to $\longrightarrow$ for each rule except REPLY. The new REPLY rule is given below. In the following rule, a set of activities $\{\alpha_j\}$ will have a set of future references $\{\sigma_{\alpha_j}(\iota_{ji}))\}$ updated:

$$\frac{\begin{array}{c}F_\beta(f_i^{\gamma\rightarrow\beta})=\iota_f\\ \forall j\in\{1..m\}\qquad\forall i\in\{1..n_j\}\qquad\sigma_{\alpha_j}(\iota_{ji}))=fut(f_i^{\gamma\rightarrow\beta})\\ \sigma_{\alpha_j i}=Copy\&Merge(\sigma_\beta,\iota_f\;;\;\sigma_{\alpha_j i-1},\iota_{ji})\end{array}}{\begin{array}{c}\alpha_j[a_{\alpha_j};\sigma_{\alpha_j 0};\iota_{\alpha_j};F_{\alpha_j};R_{\alpha_j};f_{\alpha_j}]\parallel\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\parallel P\longrightarrow\\ \alpha_j[a_{\alpha_j};\sigma_{\alpha_j n_j};\iota_{\alpha_j};F_{\alpha_j};R_{\alpha_j};f_{\alpha_j}]\parallel\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\parallel P\end{array}}\;(\text{REPLY}||)$$

This rule is a more general case of the reply rule. ASP calculus still have the same properties; and, to simplify, we will use the new REPLY|| rule only when it is necessary.

Note that one of the most important constraint on the application of the REPLY|| rule is that it must be the same future that is updated several times.Moreover, as it is the same future that is updated and there is no cycle of futures that case, the several replies can be performed in any order without consequence.

For the confluence property, we need:

$$\left\{\begin{array}{l} P \rightarrow_{\|_2} P_1 \\ P \rightarrow_{\|_2} P_2 \end{array}\right. \Longrightarrow \exists P_1', P_2', \left\{\begin{array}{l} P_1 \rightarrow_{\|_2} P_1' \\ P_2 \rightarrow_{\|_2} P_2' \\ P_1' \equiv P_2' \end{array}\right. \tag{A.1}$$

But we will only prove that:

$$\left\{\begin{array}{l} P \longrightarrow P_1 \\ P \longrightarrow P_2 \end{array}\right. \Longrightarrow \exists P_1', P_2', \left\{\begin{array}{l} P_1 \rightarrow_{\|_2} P_1' \\ P_2 \rightarrow_{\|_2} P_2' \\ P_1' \equiv P_2' \end{array}\right. \tag{A.2}$$

Indeed, at first, we can prove that: A.2 ⇒ A.1. Of course the reverse implication is trivial.

Proof : We will only focus on the case where both of the applied rules are REPLY||. Indeed, the case REPLY|| vs. ⟶ is simpler and ⟶ vs. ⟶ is obvious.

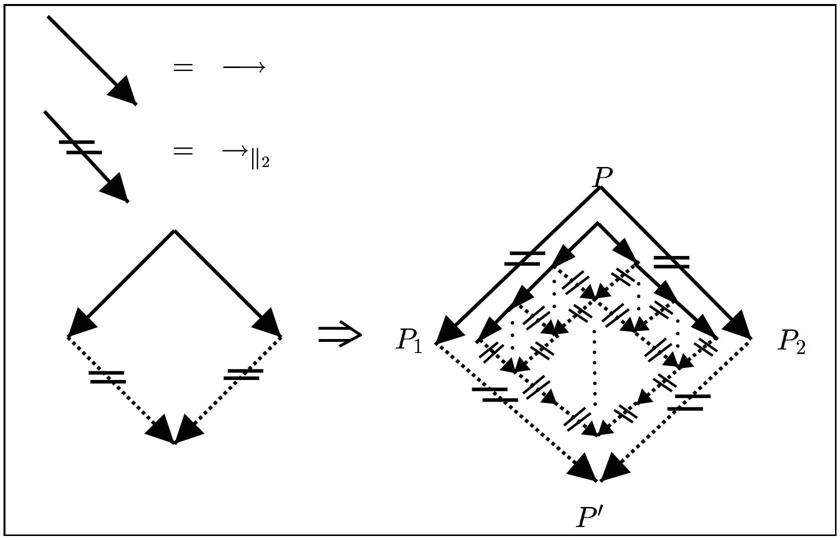


Figure A.1: Diagram of the proof with REPLY||

Let us divide the rule REPLY|| in several REPLY rules.

Then we can apply several times the confluence A.2 to each pair of REPLY rule. In the most general case such a process is not sufficient because it can never terminate[21]. But, here, reduction terminates because one can verify easily that in the case where there is no cycle of future, the number of references to a future $f^{\alpha\rightarrow\beta}$ strictly decreases with every application of the REPLY rule on the future $f^{\alpha\rightarrow\beta}$. Of course, the number of future references is trivially finite.

One would only need to verify that the set of REPLY|| rules applied to P_1 can be merged in a single REPLY|| rule; and the same thing for P_2. This is ensured by:

[21] Recall that in the case where reduction terminates local confluence implies confluence.

- verifying that the future updated is the same in all rules (trivial);
- verifying that all occurrences of this future are present in the intermediate term P_1. Indeed no occurrence of the future f can appear during the update of the future f because there is no cycle of futures.

□

A.3.2 Lemmas

Most of lemmas useful for the proof of this property have been given in 13.2. They generally involve $\equiv_F$, but the lemmas that are used here could be proved with the simple equivalence relation $\equiv$ that is the renaming of futures identifiers and locations.

Recall $\equiv$ is the equality modulo renaming of futures and locations.

In the following we ill not detail technical points corresponding to renamings. Like in the proofs of part V we always suppose that futures and locations involved in reduction can be chosen and renamed in a convenient way. This could be showed to be correct by proving the following lemma.

Lemma A.4 (renaming and parallel reduction)

$$a \equiv a' \wedge a' \longrightarrow b \Rightarrow \exists b', a \longrightarrow b' \wedge b' \equiv b$$

Lemma A.5 (Renaming and *copy*)

$$copy(\iota, \sigma)\theta = copy(\iota\theta, \sigma\theta)$$

The following corollaries are direct consequences of the Property 13.6 and its corollary 13.7 (copy and store update):

Corollary A.6 (Copy and Store Append) *There is a way of choosing locations allocated by $Copy\&Merge(\sigma, \iota\ ;\ \sigma', \iota')$ such that:*

$$\sigma' :: Copy\&Merge(\sigma_1, \iota\ ;\ \sigma_2, \iota') \equiv Copy\&Merge(\sigma_1, \iota\ ;\ \sigma' :: \sigma_2, \iota')$$

,

Corollary A.7 (Independent Copy and Merge) *If $\iota_1, \iota_2 \in dom(\sigma'')$*

$$Copy\&Merge(\sigma, \iota\ ;\ Copy\&Merge(\sigma', \iota'\ ;\ \sigma'', \iota_2), \iota_1) = \\ Copy\&Merge(\sigma', \iota'\ ;\ Copy\&Merge(\sigma, \iota\ ;\ \sigma'', \iota_1), \iota_2)$$

The following lemma can be considered as analogous to Property 13.6 and its corollary 13.7:

Lemma A.8 *If $\iota' \notin dom(\sigma)$ then there is a way of choosing locations allocated by $Copy\&Merge(\sigma_1, \iota\ ;\ \sigma_2, \iota')$ such that:*

$$Copy\&Merge(\sigma_1, \iota\ ;\ \sigma_2 + \sigma, \iota') = Copy\&Merge(\sigma_1, \iota\ ;\ \sigma_2, \iota') + \sigma$$

A.4 Proof of the Local Confluence

The proof of the local confluence (Property A.3) is a case study on the critical pairs between reduction rules. We will label each case in the same way than in the proof of Section 13.3.

Moreover most cases have already been proved in 13.3. Indeed, it is easy to verify that the proofs performed for local confluence did not really necessitate a notion of equivalence modulo futures[22]. More precisely, the cases where no REPLY rule is involved can be proved again with our new hypothesis. These proofs would be the same as the one performed in 13.3.

We will detail below the cases where one of the applied rule is REPLY.

A.4.1 Conflicts Between Localized and REPLY Rules

LOCAL/REPLY

$\alpha_{\textbf{LOCAL}} = \alpha_{\textbf{REPLY}}$

If there is a conflict, then the local rule can only be: INVOKE or UPDATE. Indeed one cannot access to a field or clone a future and STOREALLOC stores a new reduced object that cannot be a future. More precisely, if there is no common location between the store updated by REPLY and the store manipulated by LOCAL then the case is similar to the LOCAL/REQUEST case of page 128. We will focus on the case of the UPDATE rule, the proof in the case of the INVOKE rule is similar.

UPDATE The confluence comes from two facts. First, the updated object cannot be a future (else, a wait-by-necessity would occur). And secondly, as expected, the update operation is not directly influenced by the content of the new field value.

The following rules are applied (the UPDATE rule occur in the activity α):

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

$$\frac{\sigma_\alpha(\iota_2) = fut(f_i^{\gamma\to\beta}) \qquad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_2)}{\begin{array}{c}\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P\end{array}} \quad (\text{REPLY})$$

First, as mentioned earlier, the modified object is not a future, thus $\iota \neq \iota_2$. Then we can apply the corollary 13.7 (Copy and Store Append) to prove that

$$\{\iota \to o'\} + Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_2) = Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \{\iota \to o'\} + \sigma_\alpha, \iota_2)$$

[22] Equivalence modulo futures was more a constraint for these proof, but, of course it was necessary for the remainder of the proofs.

This proves that both applications can be exchanged. Indeed, $\{\iota \rightarrow o'\} + Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_2)$ is the store of α after the sequence $\xrightarrow{\text{REPLY}}\ \xrightarrow{\text{LOCAL}}$ and $Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \{\iota \rightarrow o'\} + \sigma_\alpha, \iota_2)$ after the sequence $\xrightarrow{\text{LOCAL}}\ \xrightarrow{\text{REPLY}}$.

$\alpha_{\textbf{LOCAL}} = \beta_{\textbf{REPLY}}$

In that case, the future isolation property (Property 11.1) is sufficient to conclude.

NEWACT/REPLY

$\alpha_{\textbf{NEWACT}} = \alpha_{\textbf{REPLY}}$

We can prove that a future can be safely activated and the update of the reply can appear after. Conflicts may only appear when the updated future belongs to the deep copy of the activated object. That is to say if $\iota_2 \in dom(copy(\iota'', \sigma))$ in the following rules (else the two reduction would not interfere):

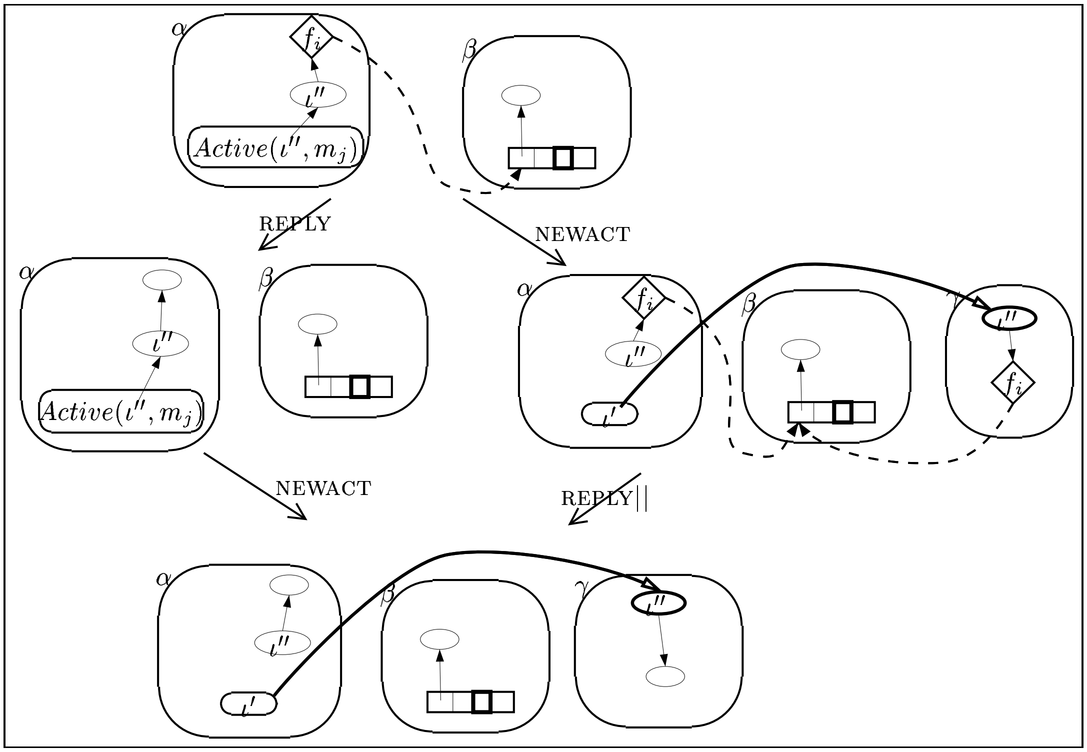


Figure A.2: Activation of an object containing a future

$$\dfrac{\begin{array}{c}\gamma \text{ fresh activity} \qquad \iota' \notin \mathit{dom}(\sigma) \qquad \sigma_1 = \{\iota' \mapsto AO(\gamma)\} :: \sigma \\ \sigma_\gamma = \mathit{copy}(\iota'', \sigma) \qquad \mathit{Service} = (\text{ if } m_j = \emptyset \text{ then } \mathit{FifoService} \text{ else } \iota''.m_j())\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\mathit{Active}(\iota'', m_j)]; \sigma; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \\ \longrightarrow \alpha[\mathcal{R}[\iota']; \sigma_1; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \gamma[\mathit{Service}; \sigma_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel P\end{array}} \text{ (NEWACT)}$$

$$\dfrac{\sigma(\iota_2) = \mathit{fut}(f_i^{\delta\to\beta}) \qquad F_\beta(f_i^{\delta\to\beta}) = \iota_f \qquad \sigma_2 = \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma, \iota_2)}{\begin{array}{c}\alpha[a_\alpha; \sigma; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\mathit{Active}(\iota, m_j); \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel Q \longrightarrow \\ \alpha[a_\alpha; \sigma_2; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel Q\end{array}} \text{ (REPLY)}$$

Where $P = \beta \| Q$.

Confluence is due to the fact that the future update performed by the REPLY rule and the activation of an object (NEWACT) both perform a deep copy.

Moreover, modulo renaming we can suppose that $\iota' \notin \mathit{dom}(\sigma_2)$ thus we can apply the following rule:

$$\dfrac{\begin{array}{c}\gamma \text{ fresh activity} \qquad \iota' \notin \mathit{dom}(\sigma) \qquad \sigma'_2 = \{\iota' \mapsto AO(\gamma)\} :: \sigma \\ \sigma'_\gamma = \mathit{copy}(\iota'', \sigma_2) \qquad \mathit{Service} = (\text{ if } m_j = \emptyset \text{ then } \mathit{FifoService} \text{ else } \iota''.m_j())\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\mathit{Active}(\iota'', m_j)]; \sigma_2; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \\ \longrightarrow \alpha[\mathcal{R}[\iota']; \sigma'_2; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \gamma[\mathit{Service}; \sigma'_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel P\end{array}} \text{ (NEWACT)}$$

Furthermore, as $\iota_2 \in \mathit{dom}(\mathit{copy}(\iota'', \sigma))$ one has $\sigma_\gamma(\iota_2) = \mathit{fut}(f_i^{\delta\to\beta})$ and thus:

$$\dfrac{\begin{array}{cc} & F_\beta(f_i^{\delta\to\beta}) = \iota_f \\ \sigma(\iota_2) = \mathit{fut}(f_i^{\delta\to\beta}) & \sigma'_1 = \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma_1, \iota_2) \\ \sigma_\gamma(\iota_2) = \mathit{fut}(f_i^{\delta\to\beta}) & \sigma''_\gamma = \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma_\gamma, \iota_2)\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota']; \sigma_1; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\mathit{Active}(\iota, m_j); \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \\ \gamma[\mathit{Service}; \sigma_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel Q' \longrightarrow \\ \alpha[\mathcal{R}[\iota']; \sigma'_1; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \\ \gamma[\mathit{Service}; \sigma''_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel Q'\end{array}} \text{ (REPLY||)}$$

Firstly $\sigma'_2 \equiv \sigma'_1$ is trivial because the two rules modify independent parts of the store and applying corollary A.6 is sufficient to equate the two stores:

$$\begin{array}{rll}\sigma'_1 & = \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma_1, \iota_2) & \\ & = \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \{\iota' \mapsto AO(\gamma)\} :: \sigma, \iota_2) & \\ & \equiv \{\iota' \mapsto AO(\gamma)\} :: \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma, \iota_2) & \text{corollary A.6} \\ & = \sigma'_2 & \end{array}$$

Now, one only has to prove that: $\sigma''_\gamma \equiv \sigma'_\gamma$. Remember that .

$$\begin{aligned}
\sigma'_\gamma &= copy(\iota'', Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \sigma, \iota_2)) \\
&= copy(\iota'', Merge(\iota_2, \sigma, \ copy(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_2\}\!\})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{Lemma A.5} \\
&= copy(\iota'', copy(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_2\}\!\}\theta + \sigma) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \theta \text{ ensures that only } \iota_2 \text{ is updated} \\
&\equiv copy(\iota'', copy(\iota_2, \sigma_\beta\{\!\{\iota_f \leftarrow \iota_2\}\!\}\theta) + \sigma) \\
&\equiv copy(\iota_2, \sigma_\beta\{\!\{\iota_f \leftarrow \iota_2\}\!\}\theta) + copy(\iota'', \sigma) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{Lemma 13.5 with } \iota_2 \in dom(copy(\iota'', \sigma)) \\
&\equiv copy(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_2\}\!\}\theta + copy(\iota'', \sigma) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{Lemma A.5} \\
&\equiv Copy\&Merge(\sigma_\beta, \iota_f \ ; \ copy(\iota'', \sigma), \iota_2) \\
&= \sigma''_\gamma
\end{aligned}$$

$\alpha_{\text{NEWACT}} = \beta_{\text{REPLY}}$
The Property 11.1 (futures and parameters isolation) ensures that $\iota_{\text{NEWACT}} \notin copy(\iota_f, \sigma)$ and the independence between the activated object and the future value.

REPLY/NEWSERVICE

No conflict.

ENDSERVICE/REPLY

There could be a conflict because the two rules access to the futures values list but, as each future value is independent, the storage of a future and the reading of another one are, in fact, independents.

A.4.2 Concurrent replies: REPLY/REPLY

$\alpha_1 = \alpha_2$
In that case both future updates are necessarily independent. Thus there is no conflict. Note that a possible future reference inside the updated future value cannot cause an interference here (consequence of corollary A.7).

$\beta_1 = \beta_2$
No conflict because the sending of a reply does not modify the sender state.

$\alpha_1 = \beta_2$ **and** $\alpha_2 \neq \beta_1$
Let us denote $\beta = \alpha_1 = \beta_2$, $\alpha = \alpha_2$ and $\gamma = \beta_1$.

The future updates interfere if the update of a future performed in a REPLYmodify the future value updated by the other REPLYrule

In other words, if $\iota_2 \in copy(\iota_f, \sigma_\beta)$ in the following rules:

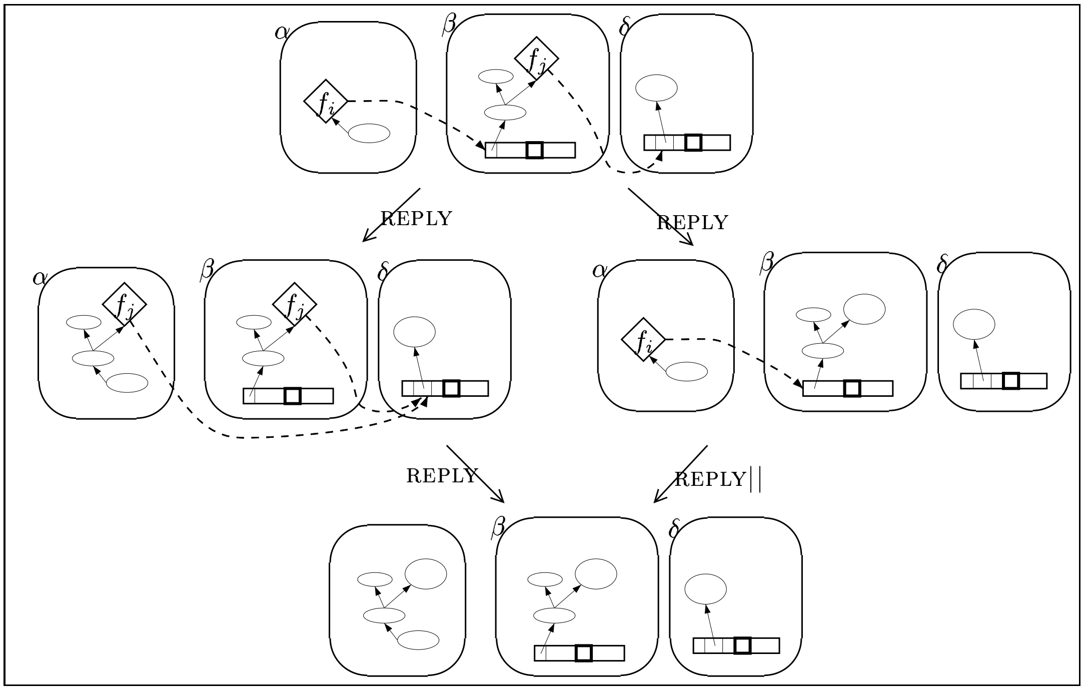


Figure A.3: Concurrent Replies

$$\frac{\sigma_\alpha(\iota_1) = fut(f_i^{\gamma\to\beta}) \qquad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_1)}{\begin{array}{c}\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow\\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P\end{array}} \quad (\text{REPLY})$$

$$\frac{\sigma_\beta(\iota_2) = fut(f_j^{\gamma'\to\delta}) \qquad F_\delta(f_i^{\gamma'\to\delta}) = \iota'_f \qquad \sigma'_\beta = Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ \sigma_\beta, \iota_2)}{\begin{array}{c}\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel \delta[a_\delta;\sigma_\delta;\iota_\delta;F_\delta;R_\delta;f_\delta] \parallel Q \longrightarrow\\ \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel \delta[a_\delta;\sigma_\delta;\iota_\delta;F_\delta;R_\delta;f_\delta] \parallel Q\end{array}} \quad (\text{REPLY})$$

Where, P and Q are such that (let us introduce a new sub-configuration R):

$$\delta\|P = \alpha\|Q = \alpha\|\delta\|R.$$

In the following we suppose that $\alpha \neq \delta$ thus the first reply does not modify the store of δ. If it was not the case, because Property A.2 ensures that there is *no cycle of future*, the update of the future f_i in the first rule (in α) cannot modify the value of the future updated by the second rule (f_j in δ) too[23], even if $\alpha = \delta$ (else we would have the case of Figure 11.7). Then even if the two activities were the same there

[23] Remember that we already consider that the update of the future f_j modify the value of the future f_i.

could not be any cyclic dependencies in the future updates and the fact that $\alpha = \delta$ would not show technical difficulties. Finally, for simplicity we consider that $\alpha \neq \delta$, the case $\alpha = \delta$ can be easily deduced from the following proof.

First, one can apply the first REPLY rule on the configuration obtained by the second one:

$$\frac{\begin{array}{c}\sigma'_\alpha(\iota_1) = fut(f_i^{\gamma\to\beta}) \qquad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \\ \sigma_{\alpha 1} = \mathit{Copy\&Merge}(\sigma'_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_1)\end{array}}{\begin{array}{c}\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \delta \parallel R \longrightarrow \\ \alpha[a_\alpha; \sigma_{\alpha 1}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \delta \parallel R\end{array}} \ (\text{REPLY})$$

where $\sigma'_\beta \neq \sigma_\beta$ and

$$\begin{aligned}\sigma_{\alpha 1} &= \mathit{Copy\&Merge}(\sigma'_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_1) \\ &= \mathit{Copy\&Merge}(\mathit{Copy\&Merge}(\sigma_\delta, \iota'_f \ ; \ \sigma_\beta, \iota_2), \iota_f \ ; \ \sigma_\alpha, \iota_1)\end{aligned}$$

Regarding the other rule, the future f_j appears both in σ'_α and in σ_β because $\iota_2 \in \mathit{copy}(\iota_f, \sigma_\beta)$ implies:

$$\begin{aligned}\sigma'_\alpha(\iota) &= \mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_1)(\iota) \\ &= \mathit{Merge}(\iota_1, \sigma_\alpha, \mathit{copy}(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_1\}\!\}))(\iota) \\ &= (\mathit{copy}(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_1\}\!\}\theta + \sigma_\alpha)\,(\iota) \\ &= (\mathit{copy}(\iota_f, \sigma_\beta)\{\!\{\iota_f \leftarrow \iota_1\}\!\}\theta)\,(\iota_2\theta) & \text{if } \iota = \iota_2\theta \\ &= \sigma_\beta(\iota_2) = fut(f_j^{\gamma'\to\delta})\end{aligned}$$

In the following, $\iota = \iota_2\theta$ will denote the location of the future reference $fut(f_j^{\gamma'\to\delta})$ in activity α. Of course, we can suppose that ι is a fresh location and $\iota \notin \mathit{dom}(\theta)$.

Then we update the future in both activities with the rule:

$$\frac{\begin{array}{c}F_\delta(f_i^{\gamma'\to\delta}) = \iota'_f \\ \sigma_\beta(\iota_2) = fut(f_j^{\gamma'\to\delta}) \qquad \sigma'_\beta = \mathit{Copy\&Merge}(\sigma_\delta, \iota'_f \ ; \ \sigma_\beta, \iota_2) \\ \sigma'_\alpha(\iota) = fut(f_j^{\gamma'\to\delta}) \qquad \sigma_{\alpha 2} = \mathit{Copy\&Merge}(\sigma_\delta, \iota'_f \ ; \ \sigma'_\alpha, \iota)\end{array}}{\begin{array}{c}\alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \\ \delta[a_\delta; \sigma_\delta; \iota_\delta; F_\delta; R_\delta; f_\delta] \parallel R \longrightarrow \\ \alpha[a_\alpha; \sigma_{\alpha 2}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel \\ \delta[a_\delta; \sigma_\delta; \iota_\delta; F_\delta; R_\delta; f_\delta] \parallel R\end{array}} \ (\text{REPLY}||)$$

Let us denote by θ' the renaming applied in the update of the future f_i (in $\mathit{Copy\&Merge}(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota_1)$): $\theta' = \{\!\{\iota_f \leftarrow \iota_1\}\!\}\theta$ and ψ the symmetrical renaming applied in the update of the future f_j (in $\mathit{Copy\&Merge}(\sigma_\delta, \iota'_f \ ; \ \sigma'_\alpha, \iota)$) thus $\theta'(\iota_f) = \iota_1$, $\theta'(\iota_2) = \iota$ and $\psi(\iota'_f) = \iota$. Modulo renaming, we can consider that $\mathit{codom}(\psi) \cap \mathit{dom}(\theta') = \emptyset$.

We can also suppose that ψ can also be applied in $Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ \sigma_\beta, \iota_2)$ that is to say, first $\iota \notin \sigma_\beta$ and more generally, $codom(\psi) \cap \sigma_\beta = \emptyset$. More precisely, we have

$$Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ \sigma_\beta, \iota_2) = copy(\iota'_f, \sigma_\delta)\psi' + \sigma_\beta$$

where $\psi' = \psi\{\!\{\iota \rightarrow \iota_2\}\!\}$ in order to ensure $\psi'(\iota'_f) = \iota_2$.

It is important to note that:

$$((\iota'_f)\psi')\theta' = \iota_2\theta' = \iota = \iota\theta' = ((\iota'_f)\psi)\theta' \quad \text{because } \iota \notin dom(\theta)$$

And thus:

$$\psi'\theta' = \psi\theta'.$$

In the same way, we extend θ' in order to be applied in the update future f_i in $Copy\&Merge(\sigma'_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_1)$ that is to say θ' must rename more locations. More precisely but less intuitively, we could have taken θ' valid for $Copy\&Merge(\sigma'_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_1)$ and show that it could be applied in $Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_1)$.

$$\begin{aligned}
\sigma_{\alpha 2} &= Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ \sigma'_\alpha, \iota) \\
&= Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota_1), \iota) \\
&\equiv Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ (copy(\iota_f, \sigma_\beta)\theta' + \sigma_\alpha), \iota) \\
&\equiv Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ (copy(\iota_f, \sigma_\beta)\theta'), \iota) + \sigma_\alpha \\
&\qquad\qquad\qquad\qquad\qquad\qquad \iota \notin dom(\sigma_\alpha) \text{ and Lemma A.8} \\
&\equiv copy(\iota'_f, \sigma_\delta)\psi + (copy(\iota_f, \sigma_\beta)\theta') + \sigma_\alpha \\
&\equiv (copy(\iota, \sigma_\delta\psi) + copy(\iota_1, \sigma_\beta\theta')) + \sigma_\alpha \qquad\qquad \text{Lemma A.5} \\
&\equiv (copy(\iota_1, copy(\iota, \sigma_\delta\psi) + \sigma_\beta\theta')) + \sigma_\alpha \\
&\qquad\qquad\qquad \iota \in dom(copy(\iota_1, \sigma_\beta\theta') \text{ and Lemma 13.5 (multiple copies)} \\
&\equiv \Big(copy(\iota_f, copy(\iota'_f, \sigma_\delta)\psi + \sigma_\beta)\theta'\Big) + \sigma_\alpha \\
&\qquad\qquad\qquad\qquad\qquad codom(\psi) \cap dom(\theta') = \emptyset \text{ and Lemma A.5} \\
&\equiv \Big(copy(\iota_f, copy(\iota'_f, \sigma_\delta)\psi' + \sigma_\beta)\theta'\Big) + \sigma_\alpha \qquad\qquad \text{because } \psi'\theta' = \psi\theta' \\
&= Copy\&Merge(Copy\&Merge(\sigma_\delta, \iota'_f\ ;\ \sigma_\beta, \iota_2), \iota_f\ ;\ \sigma_\alpha, \iota_1) = \sigma_{\alpha 1}
\end{aligned}$$

And finally the configurations obtained by the two sequences of applications of the REPLY rules are the same. Thus the critical pair REPLY/REPLY is confluent.

A.4.3 Interfering requests and replies: REQUEST/REPLY

$\beta_{\textbf{REPLY}} = \alpha_{\textbf{REQUEST}}$ **or** $\beta_{\textbf{REQUEST}} = \beta_{\textbf{REPLY}}$

Corollary of the Property 11.1 about futures and parameters isolations. Any adding of request parameter to the store is then independent from other operations on this store.

$\beta_{\textbf{REQUEST}} = \alpha_{\textbf{REPLY}}$

No interference because the two modifications of the store are completely independent.

$\alpha_{\textbf{REQUEST}} = \alpha_{\textbf{REPLY}}$

Let us denote $\alpha = \alpha_{\text{REQUEST}} = \alpha_{\text{REPLY}}$ $\beta = \beta_{\text{REQUEST}}$ and $\gamma = \beta_{\text{REPLY}}$.

This case is very similar to the concurrent replies. Thus we will briefly describe the proof because technical details are strongly similar to the case REPLY/REPLYwith $\alpha_1 = \beta_2$.

The two store modifications are not independent when $\iota_2 \in dom(copy(\iota', \sigma_\alpha))$. Indeed, $\iota_2 = \iota$ is impossible, and else, if $\iota_2 \notin dom(copy(\iota', \sigma_\alpha))$ then the future update and the request send are independent.

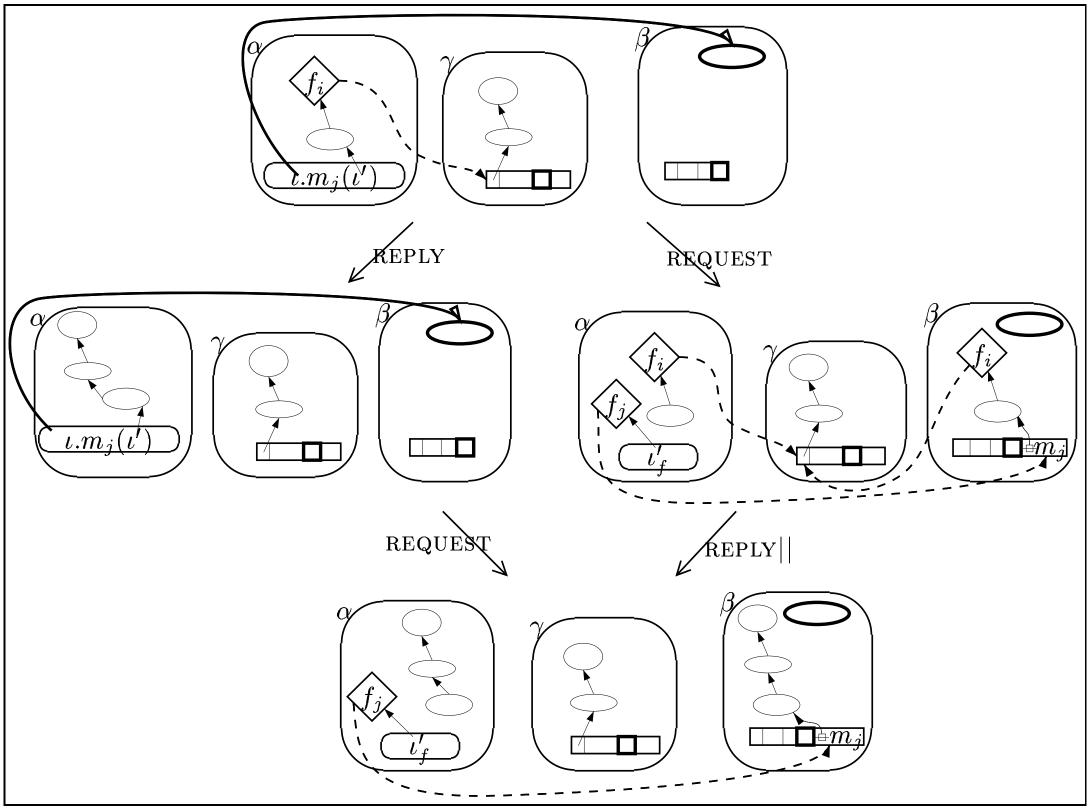


Figure A.4: Interfering requests and replies: $\alpha_{\text{REQUEST}} = \alpha_{\text{REPLY}} = \alpha$

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

$$\frac{\sigma_\alpha(\iota_2)=fut(f_i^{\delta\to\gamma}) \qquad F_\gamma(f_i^{\delta\to\gamma})=\iota_f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\gamma,\iota_f\ ;\ \sigma_\alpha,\iota_2)}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \gamma[a_\gamma;\sigma_\gamma;\iota_\gamma;F_\gamma;R_\gamma;f_\gamma] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \gamma[a_\gamma;\sigma_\gamma;\iota_\gamma;F_\gamma;R_\gamma;f_\gamma] \parallel P\end{array}} \quad (\textsc{reply})$$

Where $\gamma\|P=\beta\|Q=\gamma\|\beta\|R$.

First, one can apply the REQUEST rule on the configuration obtained by the application of the REPLY rule:

$$\frac{\begin{array}{c}\sigma_\alpha(\iota)=AO(\beta) \qquad \iota''\notin dom(\sigma_\beta) \qquad f_j^{\alpha\to\beta}\text{ new future} \qquad \iota'_f\notin dom(\sigma_\alpha) \\ \sigma_{\beta 1}=\mathit{Copy\&Merge}(\sigma'_\alpha,\iota'\ ;\ \sigma_\beta,\iota'') \qquad \sigma_{\alpha 1}=\{\iota'_f\mapsto fut(f_j^{\alpha\to\beta})\}::\sigma'_\alpha\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel Q \longrightarrow \\ \alpha[\mathcal{R}[\iota'_f];\sigma_{\alpha 1};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_{\beta 1};\iota_\beta;F_\beta;R_\beta::[m_j;\iota'';f_j^{\alpha\to\beta}];f_\beta] \parallel Q\end{array}} \quad (\textsc{request})$$

Then, one has:

$$\begin{aligned}\sigma_{\alpha 1} &= \{\iota'_f\mapsto fut(f_j^{\alpha\to\beta})\}::\sigma'_\alpha \\ &= \{\iota'_f\mapsto fut(f_j^{\alpha\to\beta})\}::\mathit{Copy\&Merge}(\sigma_\gamma,\iota_f\ ;\ \sigma_\alpha,\iota_2)\end{aligned}$$

and:

$$\begin{aligned}\sigma_{\beta 1} &= \mathit{Copy\&Merge}(\sigma'_\alpha,\iota'\ ;\ \sigma_\beta,\iota'') \\ &= \mathit{Copy\&Merge}(\mathit{Copy\&Merge}(\sigma_\gamma,\iota_f\ ;\ \sigma_\alpha,\iota_2),\iota'\ ;\ \sigma_\beta,\iota'')\end{aligned}$$

Concerning the configuration obtained by the REQUEST rule, f_i appears both in σ'_α and in σ_β because $\iota_2\in dom(copy(\iota',\sigma_\alpha))$ implies, there is ι such that (same reasoning than in page 164):

$$\begin{aligned}\sigma'_\beta(\iota) &= \mathit{Copy\&Merge}(\sigma_\alpha,\iota'\ ;\ \sigma_\beta,\iota'')(\iota) \\ &= \ldots = \sigma_\alpha(\iota_2) = fut(f_i^{\delta\to\gamma})\end{aligned}$$

where $\iota=\iota_2\theta$ and θ is the renaming used in $\mathit{Copy\&Merge}(\sigma_\alpha,\iota'\ ;\ \sigma_\beta,\iota'')$.

Then, the reply rule have to be applied to update the two references to future f_i:

$$\frac{\begin{array}{cc} & F_\gamma(f_i^{\delta\to\gamma})=\iota_f \\ \sigma_\alpha(\iota_2)=fut(f_i^{\delta\to\gamma}) & \sigma_{\alpha 2}=\mathit{Copy\&Merge}(\sigma_\gamma,\iota_f\ ;\ \sigma''_\alpha,\iota_2) \\ \sigma'_\beta(\iota)=fut(f_i^{\delta\to\gamma}) & \sigma_{\beta 2}=\mathit{Copy\&Merge}(\sigma_\gamma,\iota_f\ ;\ \sigma'_\beta,\iota)\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota'_f];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \\ \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta::[m_j;\iota'';f_j^{\alpha\to\beta}];f_\beta] \parallel \gamma[a_\gamma;\sigma_\gamma;\iota_\gamma;F_\gamma;R_\gamma;f_\gamma] \parallel R \longrightarrow \\ \alpha[\mathcal{R}[\iota'_f];\sigma_{\alpha 2};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \\ \beta[a_\beta;\sigma_{\beta 2};\iota_\beta;F_\beta;R_\beta::[m_j;\iota'';f_j^{\alpha\to\beta}];f_\beta] \parallel \gamma[a_\gamma;\sigma_\gamma;\iota_\gamma;F_\gamma;R_\gamma;f_\gamma] \parallel R\end{array}} \quad (\textsc{reply}||)$$

Then, we must prove that $\sigma_{\alpha 1} = \sigma_{\alpha 2}$ and $\sigma_{\beta 1} = \sigma_{\beta 2}$.

$$\begin{aligned}
\sigma_{\alpha 2} &= \mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \sigma''_\alpha, \iota_2)\\
&= \mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \{\iota'_f \mapsto fut(f_j^{\alpha\rightarrow\beta})\} :: \sigma_\alpha, \iota_2)\\
&\equiv \{\iota'_f \mapsto fut(f_j^{\alpha\rightarrow\beta})\} :: \mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \sigma_\alpha, \iota_2) && \text{corollary A.6}\\
&= \sigma_{\alpha 1}
\end{aligned}$$

The proof of $\sigma_{\beta 1} = \sigma_{\beta 2}$ is more complicated but strongly similar to the case REPLY/REPLY of page 165:

$$\begin{aligned}
\sigma_{\beta 2} &= \mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \sigma'_\beta, \iota_2)\\
&= \mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \mathit{Copy\&Merge}(\sigma_\alpha, \iota' \;;\; \sigma_\beta, \iota''), \iota)\\
&\equiv \ldots && \text{cf. page 165}\\
&\equiv \mathit{Copy\&Merge}(\mathit{Copy\&Merge}(\sigma_\gamma, \iota_f \;;\; \sigma_\alpha, \iota_2), \iota' \;;\; \sigma_\beta, \iota'')\\
&\qquad \text{because } \iota = \iota_2\theta \text{ and } \theta \text{ appears in } \mathit{Copy\&Merge}(\sigma_\alpha, \iota' \;;\; \sigma_\beta, \iota'').\\
&= \sigma_{\beta 1}
\end{aligned}$$

A.4.4 Concurrent Requests Sending: REQUEST/REQUEST

The only case different from the confluence proof in 13.3 is the case where $\beta_1 = \beta_2$. That is to say, two requests come from different activities and have the same destination. The hypothesis (HA.1) disallows such interferences.

Index of Notations

Concepts

Syntax: ASP source terms

M	list of method labels	56
$let\ x = a\ in\ b$	$[m = \varsigma(z,x)b].m(a)$	52
$a; b$	$[m = \varsigma(z,z')b].m(a)$	52
$Repeat(a)$	$[repeat = \varsigma(x).a; x.repeat()].repeat()$	70
$FifoService$	$Repeat(Serve(\mathcal{M}))$	70

ASP intermediate terms and semantics structures

(a, σ)	Sequential configuration	53
α, β	Activities: $\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha]$ [current term, store, active object, future values, pending requests, current future]	55
ι	Locations	52
$locs(a)$	Set of locations occurring in a	52
P, Q	Configuration	70
$a \Uparrow f, b$	a with continuation b f is the future associated with the configuration	56
$AO(\alpha)$	Generalized reference to the activity α	58
$f_i^{\alpha\rightarrow\beta}$	Future identifier	69
$fut(f_i^{\alpha\rightarrow\beta})$	Future reference	70
$r = [m_j; \iota; f_i^{\alpha\rightarrow\beta}]$	Request: asynchronous remote method call	55
$R_\alpha = \{[m_j; \iota; f_i^{\alpha\rightarrow\beta}]\}$	Pending requests: a queue of requests	69

General Notations

$\{a \mapsto b\}$	Association/finite mapping	53
$\theta ::= \{\!\{b \leftarrow c\}\!\}$	Substitution	52
$\xrightarrow{*}$	Transitive closure of any reduction $\rightarrow$	81
$\bigoplus$	Disjoint union	82
$L\big\|_M$	Restriction of (RSL) list L to labels belonging to M	84
L_n	n^{th} element of the list L	83
$\sqcup$	Least upper bound	85
$\exists^1$	There is at most one	92

Stores

σ	Store: finite map from locations to objects (reduced or generalized reference) $\sigma ::= \{\iota_i \mapsto o_i\}$	53
$dom(\sigma)$	set of locations defined by σ	53
$\sigma :: \sigma'$	Append of disjoint stores	53
$\sigma + \sigma'$	Updates the values defined in σ' by those defined in σ) $(\sigma+\sigma')(\iota) = \begin{array}{ll} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{array}$	53

$Merge(\iota, \sigma, \sigma')$	Store Merge: merges independently σ and σ' except for ι which is taken from σ' $Merge(\iota, \sigma, \sigma') = \sigma'\theta + \sigma$ where $\theta = \{\!\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \backslash \{\iota\}, \iota''$ fresh$\}\!\}$	71
$copy(\iota, \sigma)$	Deep copy of $\sigma(\iota)$	71
$Copy\&Merge(\sigma, \iota \; ; \; \sigma', \iota')$	Appends in $\sigma'(\iota')$ a deep copy of $\sigma(\iota)$ $= Merge(\iota', \sigma', copy(\iota, \sigma)\{\!\{\iota \leftarrow \iota'\}\!\})$	72

Semantics

$\mathcal{R}$	Reduction context	53,70
$\mathcal{R}[a]$	Substitution inside a reduction context	53
$\rightarrow_S$	Sequential reduction	54
$\longrightarrow$	Parallel reduction	72
$\xrightarrow{T}$	Parallel reduction where rule T is applied	90
$\Longrightarrow$	Parallel Reduction with future updates: Parallel reduction preceded by some reply rules	90
$\overset{T}{\Longrightarrow}$	Parallel Reduction with future updates where rule T is applied: $\xrightarrow{\text{REPLY}*}\xrightarrow{T}$ if $T \neq$ REPLY and $\xrightarrow{\text{REPLY}*}$ if $T =$ REPLY	90
$FL(\alpha)$	Futures list of α	76
RSL_α	Request Sender List of α: $(RSL_\alpha)_n = \beta^f$ if $f_n^{\beta \rightarrow \alpha} \in FL(\alpha)$	83
$\trianglelefteq$	RSL comparison: prefix order on sender activities	85
$\mathcal{M}_{\alpha_P}$	Potential services. Static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P: $P \xrightarrow{*} Q \wedge Q = \alpha[\mathcal{R}[Serve(M)], \ldots] \parallel \ldots$ $\Rightarrow M \in \mathcal{M}_{\alpha_P}$	81
$ActiveRefs(\alpha)$	Set of active objects referenced by α: $\{\beta \mid \exists \iota \in dom(\sigma_\alpha), \sigma_\alpha(\iota) = AO(\beta)\}$	76
$FutureRefs(\alpha)$	Set of futures referenced by α: $\{f_i^{\beta \rightarrow \gamma} \mid \exists \iota \in dom(\sigma_\alpha), \sigma_\alpha(\iota) = fut(f_i^{\beta \rightarrow \gamma})\}$	76

Equivalences

$\equiv$	equality modulo renaming (alpha-conversion) of locations, futures, activitities and reordering of pending requests	87
$\equiv_F$	Equivalence modulo future replies/updates	87

Properties

$\vdash P\ \textsc{ok}$	Well formed configuration	77
$RSL_\alpha \bowtie RSL_\beta$	RSL compatibility	85
$P \bowtie Q$	Configuration compatibility	85
$P_1 \curlyvee\!\!\!\!\downarrow P_2$	Configuration confluence: $\exists R_1, R_2, P_1 \xrightarrow{*} R_1 \wedge P_2 \xrightarrow{*} R_2 \wedge R_1 \equiv_F R_2$	91
$DON(P)$	Deterministic Object Network	92

Syntax of ASP

Source terms

$$\begin{array}{rll}
a, b \in L ::= & x & \text{variable,}\\
& \mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} & \text{object definition,}\\
& \mid a.l_i & \text{field access,}\\
& \mid a.l_i := b & \text{field update,}\\
& \mid a.m_j(b) & \text{method call,}\\
& \mid clone(a) & \text{superficial copy,}\\
& \mid Active(a, m_j) & \text{activates object:}\\
& & \text{deep copy + activity creation}\\
& & m_j \text{ is the activity method}\\
& & \text{or } \emptyset \text{ for FIFO service}\\
& \mid Serve(M) & \text{Serves a request among}\\
& & \text{a set of method labels,}
\end{array}$$

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

$$M = m_1, \ldots, m_k$$

Intermediate terms

Terms

$$\begin{array}{rll}
a, b \in L' ::= & x & \text{variable,}\\
& \mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} & \text{object definition,}\\
& \mid a.l_i & \text{field access,}\\
& \mid a.l_i := b & \text{field update,}\\
& \mid a.m_j(b) & \text{method call,}\\
& \mid clone(a) & \text{superficial copy,}\\
& \mid Active(a, m_j) & \text{object activation,}\\
& \mid Serve(M) & \text{service primitive,}\\
& \mid \iota & \text{location}\\
& \mid a \Uparrow f, b & a \text{ with continuation } b
\end{array}$$

Activities

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

Requests

$$R ::= \{[m_j; \iota; f_i^{\alpha \rightarrow \beta}]\}$$

Future values

$$F ::= \{f_i^{\gamma \rightarrow \alpha} \mapsto \iota\}$$

store

$$\sigma ::= \{\iota_i \mapsto o_i\}$$

$$\begin{array}{lll} o ::= & [l_i = \iota_i^{i<n}; m_j = \varsigma(x_j, y_j).a_j^{j<m}] & \text{reduced object} \\ & |AO(\alpha) & \text{active object reference} \\ & |fut(f_i^{\alpha \rightarrow \beta}) & \text{future reference} \end{array}$$

Operational Semantics

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

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

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

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

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

Table 9: Sequential Reduction

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

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

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

Table 10: Deep copy

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

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

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

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

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

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

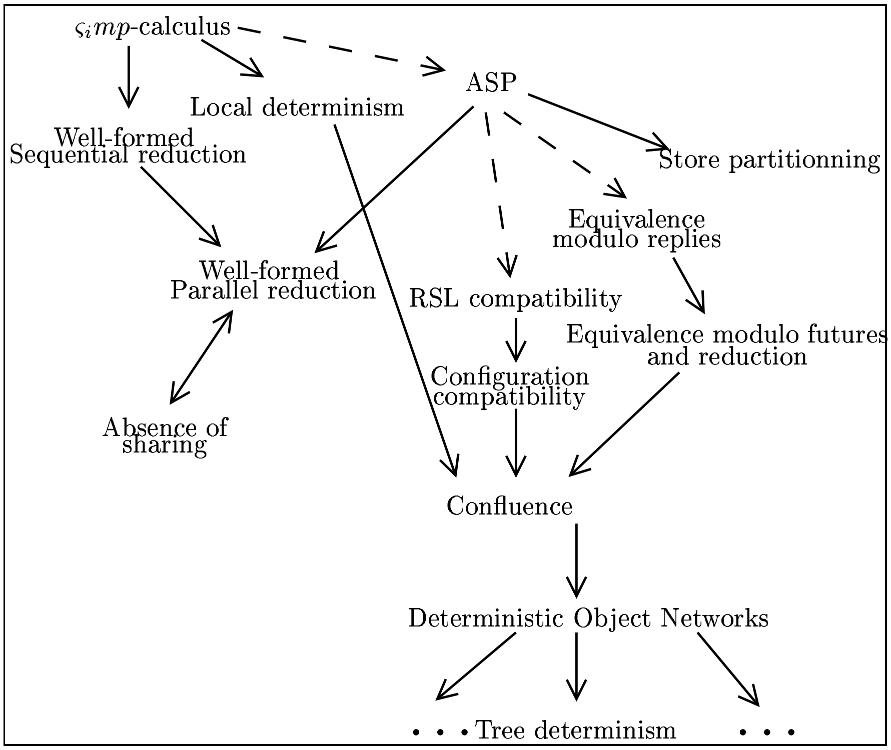
Table 11: Parallel reduction (used or modified values are non-gray)

$$\frac{\begin{array}{c}F_\beta(f_i^{\gamma\rightarrow\beta}) = \iota_f \\ \forall j \in \{1..m\} \qquad \forall i \in \{1..n_j\} \qquad \sigma_{\alpha_j}(\iota_{ji})) = fut(f_i^{\gamma\rightarrow\beta}) \\ \sigma_{\alpha_j i} = Copy\&Merge(\sigma_\beta,\iota_f \;;\; \sigma_{\alpha_j i-1},\iota_{ji})\end{array}}{\begin{array}{c}\alpha_j[a_{\alpha_j};\sigma_{\alpha_j 0};\iota_{\alpha_j};F_{\alpha_j};R_{\alpha_j};f_{\alpha_j}] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha_j[a_{\alpha_j};\sigma_{\alpha_j n_j};\iota_{\alpha_j};F_{\alpha_j};R_{\alpha_j};f_{\alpha_j}] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P\end{array}} \quad (\textsc{reply}\|)$$

Table 12: Parallel replies

Overview of Properties

The idea of the following diagram is to give the dependences between properties and definitions.



$\varsigma_i mp$-calculus
ASP
Local determinism
Well-formed Sequential reduction
Store partitionning
Equivalence modulo replies
Well-formed Parallel reduction
RSL compatibility
Equivalence modulo futures and reduction
Configuration compatibility
Absence of sharing
Confluence
Deterministic Object Networks
• • • Tree determinism • • •

Figure 5: properties Diagram (*very informal*)

Bibliography

[AC95a] Martín Abadi and Luca Cardelli. An imperative object calculus. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in LNCS, pages 471–485. Springer-Verlag, 1995.

[AC95b] Martín Abadi and Luca Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 - Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[ACG00] 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.

[Agh86] Gul Agha. An overview of actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, 1986.

[Alm97] 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.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[Ame92] Pierre America. Formal techniques for parallel object-oriented languages. *Lecture Notes in Computer Science*, 612:119–??, 1992.

[AMST92] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation (extended abstract). In W. R. Cleaveland, editor, *CONCUR'92: Proc. of the Third International Conference on Concurrency Theory*, pages 565–579. Springer, Berlin, Heidelberg, 1992.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[Bar81] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathematics*. North Holland, Amsterdam, The Netherlands, 1981.

[BCF02] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440. Springer-Verlag, 2002.

[BMM02] R. Bruni, J. Meseguer, and U. Montanari. Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. *Mathematical Structures in Computer Science*, 12(1):53–90, 2002.

[BN02] Sébastien Briais and Uwe Nestmann. Mobile objects "must" move safely. In *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2002, University of Twente, the Netherlands.* Kluwer Academic Publishers, 2002.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, 1995.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[Car95] Luca Cardelli. A language with distributed scope. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 286–297, San Francisco, January 22–25, 1995. ACM Press.

[CG99] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92. ACM, 1999.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*: 140–155.

[CHS03] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. Technical report, INRIA Sophia Antipolis, 2003. RR-4753.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2004. To appear.

[CKV98] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998. Proactive available at http://www.inria.fr/oasis/proactive.

[Deu94] Alain 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).

[Deu95] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 226–229, La Jolla, California, June 21–23, 1995.

[DZ01] S. Dal Zilio. Mobile processes: a commented bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*. Springer, 2001.

[FBL98] Cédric Fournet, Michele Boreale, and Cosimo Laneve. Bisimulations in the join calculus. In *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1998.

[Fes01] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC)*, Rhodes Island, August 2001.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* [POP96], pages 372–385.

[FG98] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In Larsen et al. [LSW98], pages 844–855.

[FGL$^+$96] 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.

[FM00] GianLuigi Ferrari and Ugo Montanari. Tiles for concurrent and located calculi. In C. Palamidessi and J. Parrow, editors, *Electronic Notes in Theoretical Computer Science*, volume 7. Elsevier, 2000.

[GC84] N. H. Gehani and T. A. Cargill. Concurrent programming in the ada language: The polling bias. *Software – Practice and Experience*, 14(5):413–427, May 1984.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*. Elsevier ENTCS, 1998.

[GHL97a] Gordon, Hankin, and Lassen. Compilation and equivalence of imperative objects. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 17, 1997.

[GHL97b] 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.

[GR89] Narain Gehani and William D. Roome. *The concurrent C programming language*. Silicon Press, 1989.

[GR96] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* [POP96], pages 386–395.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction (invited paper). In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation (PASCO'94), Linz, Austria*, pages 181–193. World Scientific Publishing, 1994.

[HM76] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103. ACM Press, 1976.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–??, 1991.

[Jef00] Alan Jeffrey. A distributed object calculus. In *ACM SIGPLAN Workshop Foundations of Object Oriented Languages*, 2000.

[JH96] 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.

[Jon92] Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, 1992. UMCS-92-12-1.

[Jon93] Cliff B. Jones. Process-algebraic foundations for an object-based design notation. Technical report, University of Manchester, 1993. UMCS-93-10-1.

[Kah74] 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.

[KM77] 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.

[KPR$^+$92] Owen Kaser, Shaunak Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Fast parallel implementation of lazy languages - the EQUALS experience. In *LISP and Functional Programming*, pages 335–344, 1992.

[KPT96] 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.

[KW90] Morry Katz and Daniel Weise. Continuing into the future: on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 176–184. ACM Press, 1990.

[KY94] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programing. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 31–45. ACM Press, 1994.

[LQP92] 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.

[LSW98] Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors. *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *LNCS*. Springer, July 1998.

[LW95] 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.

[LW98] Xinxin Liu and David Walker. Partial confluence of processes and systems of objects. *Theoretical Computer Science*, 1998.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[Mil93] 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.

[MKN02] Massimo Merro, Josva Kleist, and Uwe Nestmann. Mobile objects as mobile processes. *Information and Computation*, 177(2):195–241, 2002.

[MPW92] 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.

[MS98] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Larsen et al. [LSW98], pages 856–867.

[NHKM99] Uwe Nestmann, Hans Huttel, Josva Kleist, and Massimo Merro. Aliasing models for object migration. In *European Conference on Parallel Processing*, pages 1353–1368, 1999.

[NHKM02] Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. Aliasing models for mobile objects. *Information and Computation*, 175(1):3–33, 2002.

[NS97] Uwe Nestmann and Martin Steffen. Typing confluence. In Stefania Gnesi and Diego Latella, editors, *Proceedings of FMICS '97*, pages 77–101. Consiglio Nazionale Ricerche di Pisa, 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.

[POP96] *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[PR03] Thomas Parks and David Roberts. Distributed Process Networks in Java. In *"Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS2003)"*, Nice, France, April 2003.

[Pro] Proactive API. Available at http://www.inria.fr/oasis/proactive (under LGPL).

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings Theory and Practice of Parallel Programming (TPPP 94)*, pages 187–215, Sendai, Japan, 1995. Springer LNCS 907.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000.

[Rep91] John H. Reppy. Cml: A higher concurrent language. In *Proceedings of the conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).

[San99] Davide Sangiorgi. The typed π-calculus at work: A proof of Jones's parallelisation theorem on concurrent objects. *Theory and Practice of Object-Oriented Systems*, 5(1), 1999. An early version was included in the *Informal proceedings of FOOL 4*, January 1997.

[San01] Davide Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253(2):311–350, February 2001.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 15–18, 1984. ACM SIGACT-SIGPLAN, ACM Press.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[SS03] Alan Schmitt and Jean-Bernard Stefani. The m-calculus: a higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 50–61. ACM Press, 2003.

[Ste90] 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.

[Ste03] Jean Bernard Stefani. A calculus of higher-order distributed components. Technical report, INRIA Rhones Alpes, 2003. RR-4692.

[WWV00] D.L. Webb, A.L. Wendelborn, and J. Vayssiere. A Study of Computational Reconfiguration in a Process Network. In *Proceedings of the 7th Workshop on Integrated Data Environments Australia (IDEA'7)*, February 2000.

Index

Résumé

L'objectif de cette thèse est de concevoir un calcul d'objets permettant d'écrire des applications parallèles et distribuées, en particulier dans un cadre à grande échelle, tout en assurant de bonnes propriétés. Le calcul proposé s'intitule ASP : Asynchronous Sequential Processes.

Les principales caractéristiques de ce calcul sont : des communications asynchrones, la présence de futurs et une exécution séquentielle dans chacun des processus. Ce calcul exhibe de fortes propriétés de confluence et de déterminisme. Cette thèse a donc aussi pour objectif de prouver de telles propriétés dans un cadre aussi général que possible.

ASP est basé sur une répartition des objets en différentes activités disjointes. Une activité est un ensemble d'objets gérés par un unique processus. Les objets actifs sont des objets accessibles par des références globales/distantes. Ils communiquent à travers des appels de méthodes asynchrones avec un mécanisme de futurs. Un futur est une référence globale désignant un résultat qui n'est pas encore calculé. Cette thèse modélise ces différents aspects, leurs principales propriétés et les conséquences de ces mécanismes sur la notion de comportement déterministe des programmes. Le résultat principal consiste en une propriété de confluence et son application à l'identification d'un ensemble de programmes se comportant de façon déterministe.

Du point de vue pratique, ASP peut aussi être considéré comme une modélisation de la librairie ProActive. Cette librairie fournit des outils pour développer des applications parallèles et distribuées en Java.

Mots-clés: Parallélisme, Concurrence, distribution, asynchronisme, langages et calculs à objets, confluence, déterminisme, futurs.

Abstract

The objective of this thesis is to design an object calculus that allows one to write parallel and distributed applications, particularly on wide range networks, while ensuring good properties. This calculus is named ASP: Asynchronous Sequential Processes.

The main characteristics of ASP are: asynchronous communications, futures, and a sequential execution within each process. ASP presents strong confluence and determinism properties, proved in a context as general as possible within this thesis.

A first design decision is the absence of sharing: objects live in disjoint activities. An activity is a set of objects managed by a unique process and a unique active object. Active objects are accessible through global/distant references. They communicate through asynchronous method calls with futures. A future is a global reference representing a result not yet computed. This thesis models those aspects, their main properties, and the consequences of these mechanisms on the deterministic behavior of programs. The main result consists in a confluence property and its application to the identification of a set of programs behaving deterministically.

From a practical point of view, ASP can also be considered as a model of the ProActive library. This library provides tools for developing parallel and distributed applications in Java.

Keywords: parallelism, concurrency, distributed, calculus, asynchrony, object languages, confluence, determinacy, futures.