THÈSE

présentée à

L'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

spécialité :

INFORMATIQUE

par

Cédric FOURNET

Sujet de la thèse :

LE JOIN-CALCUL : UN CALCUL POUR LA PROGRAMMATION RÉPARTIE ET MOBILE

The Join-Calculus:
a Calculus for Distributed Mobile Programming

Soutenue le 23 Novembre 1998 devant le jury composé de :

Remerciements

– M. Robin Milner a bien voulu présider le jury ; je l'en remercie chaleureusement.
– MM. Roberto Amadio et Gérard Boudol ont lu cette thèse en détail et ont accepté d'en être les rapporteurs. Qu'ils en soient remerciés, et qu'ils me pardonnent ses longueurs.
– M. Jean-Jacques Lévy a été un directeur de thèse amical, disponible, et de bon conseil. Il m'a persuadé de l'intérêt de la recherche en informatique et m'a suggéré l'étude de la programmation répartie. Au cours de la thèse, il m'a apporté son soutien tout en me laissant une grande liberté. Je lui en suis particulièrement reconnaissant.
– M. Georges Gonthier a également guidé cette thèse. Sa collaboration fut essentielle à la plupart des résultats décrits ici. Je l'en remercie tout particulièrement.
– MM. Luca Cardelli et Gérard Berry ont accepté de participer au jury. Je les remercie de l'attention qu'ils accordent à mon travail.

Je tiens à exprimer ma gratitude envers Martin Abadi, Michele Boreale, Georges Gonthier, Cosimo Laneve, Jean-Jacques Levy, Luc Maranget, et Didier Rémy, avec qui j'ai eu la chance de collaborer dans l'étude du join-calcul. Je remercie encore Luc Maranget, qui a réalisé avec moi une implémentation répartie du join-calcul. Notre prototype n'aurait sans doute pas abouti sans son bon sens ni son expérience de la compilation. Peter Sewell et Carolina Lavatelli ont relu le manuscript. Je les remercie de leur patience.

J'ai bénéficié de vives discussions sur les langages de programmation, les calculs de processus, et leur application à la programmation parallèle ou répartie. Merci à Ilaria Castellani, Damien Doligez, Andrew Gordon, Florent Guillaume, Matthew Hennessy, Kohei Honda, Ole Jensen, Fabrice Le Fessant, Xavier Leroy, Ugo Montanari, Uwe Nestmann, Benjamin Pierce, Jon Riecke, Davide Sangiorgi, Peter Sewell, David Turner, et à ceux que j'oublie ici.

Cette thèse s'est déroulée dans l'excellent environnement scientifique et la bonne ambiance du projet PARA et des projets voisins, à l'Institut National de Recherche en Informatique et en Automatique (INRIA). J'ai été invité par MM. Robin Milner et Benjamin Pierce à l'université de Cambridge pendant l'été 1995. J'ai aussi bénéficié des programmes de recherche européens ESPRIT Basic Research Action 6454 - CONFER, puis ESPRIT Working Group 21836 - CONFER-2.

Ce document, ainsi que l'implémentation répartie du join-calcul, ont été produits à l'aide des logiciels Emacs, TEX , LATEX , et Objective Caml.

Contents

List of Figures

Résumé

Ce résumé en français présente les principales idées de cette thèse, rédigée ensuite en anglais. Les définitions et résultats formels en sont systématiquement omis ; de nombreuses références permettent de se reporter au corps de la thèse.

Notre sujet d'étude est la programmation de systèmes répartis. De tels systèmes comportent de nombreux ordinateurs interconnectés par un réseau ; chacune de ces machines exécute une portion d'un programme de manière concertée, en s'échangeant des valeurs, du code exécutable, voire des processus en cours d'exécution.

Bien que relativement récent ce type de programmation, au sens large, est très répandu : De nombreuses applications utilisent le réseau et font intervenir plusieurs machines, le plus souvent selon des schémas bien établis comme le modèle client-serveur. Dans la plupart des cas ces applications utilisent des langages traditionnels et diverses bibliothèques de protocoles, ce qui obscurcit l'aspect réparti de l'application.

Par ailleurs, une multitude de langages d'avant-garde abordent la programmation répartie de manière plus radicale, en proposant de nouveaux concepts comme les objets mobiles [80], les agents mobiles [149], les applets [64], la portée lexicale globale [44], les ambiances [46]. Malgré leur popularité ces langages expérimentaux sont encore peu utilisés, et leurs avantages respectifs largement controversés.

Quel que soit le choix du langage, la programmation répartie reste notoirement délicate parce qu'elle cumule les problèmes traditionnels du parallélisme et certaines contraintes, comme l'absence d'opérations atomiques globales, ou encore l'hétérogénéité de l'environnement à l'exécution.

Les systèmes répartis sont naturellement parallèles : Chaque machine fonctionne de manière autonome ; de plus chaque programme local a souvent recours au parallélisme pour faire progresser une partie du calcul en attendant certains messages. De manière plus spécifique, ce parallélisme est essentiellement asynchrone : chaque communication prend un temps inconnu, très variable. Ainsi plusieurs ordres de grandeur séparent une communication locale à un processeur d'une communication planétaire. Cette grande variabilité semble inévitable : Les réseaux informatiques ne peuvent s'affranchir de certaines limites physiques comme les temps de propagation. Tandis que le débit d'information échangée augmente rapidement, il ne sera jamais possible d'échanger en une seconde plus d'une vingtaine de messages d'un bout à l'autre de la planète.

Pour des raisons plus structurelles, il n'est pas réaliste (ni sans doute souhaitable) d'imaginer à brève échéance un réseau uniforme et fiable, à l'administration centralisée. Au contraire, la tendance actuelle est plutôt à la coexistence de systèmes différents administrés de manière autonome avec chacun ses particularités. En outre, la structure du réseau est très dynamiques, de nombreux systèmes apparaissent, évoluent, deviennent temporairement inaccessibles. En particulier, la panne d'une machine ou d'une partie du réseau au cours d'un calcul réparti est relativement fréquente, et à défaut d'une programmation particulièrement soignée il suffit en général d'une machine déficiente pour invalider l'ensemble du calcul.

Il existe de nombreux travaux d'algorithmique sur les protocoles répartis fondamentaux, ainsi que de programmation système sur les mécanismes d'implémentation sous-jacents. En revanche, l'étude formelle de la programmation et de ses langages semble dans ce domaine pratiquement inexistante, en tout cas très en retard par rapport à la programmation séquentielle ou parallèle. En l'absence d'un modèle, pourtant, il n'est pas possible de spécifier le comportement d'un programme ou l'impact d'une modification dans un programme, de valider une implémentation répartie, d'en étudier certaines propriétés délicates comme la sécurité du système, ou sa fiabilité en cas de pannes.

Notre objectif est de construire et d'étudier un langage de programmation élémentaire — le join-calcul —, dans lequel chaque étape du calcul correspond naturellement à l'envoi d'au plus un message entre machines. Nous utilisons les termes de langage et de calcul de manière voisine : Un calcul est un langage réduit à sa plus simple expression afin de modéliser précisément certains aspects du langage, en faisant abstraction de tout le reste.

Le join-calcul s'inspire largement de deux modèles bien connus du parallélisme, les calculs de processus comme CCS ou le pi-calcul de Milner Parrow et Walker d'une part [100], et la machine chimique abstraite de Berry et Boudol d'autre part [29]. A première vue ces modèles développés pour formaliser les systèmes parallèles devraient également s'appliquer ici, a fortiori leurs variantes asynchrones.

Ces calculs sont utilisés principalement pour étudier la spécification de systèmes, typiquement pour en énoncer les propriétés de manière formelle pour ensuite les prouver ou les vérifier. Notre objectif ici est plus particulier. Le join-calcul correspond au noyau d'un langage de programmation directement utilisable ; en particulier, il est implémentable de manière répartie. Par construction, notre modèle de parallélisme préserve la plupart des propriétés formelles du pi-calcul, mais il présente de meilleures propriétés de localité au cours du calcul.

Idéalement, notre modèle doit être beaucoup plus simple et général qu'une implémentation particulière, sans pour autant masquer les phénomènes essentiels de la programmation répartie. Son expressivité est encadrée par deux objectifs opposés : l'écriture naturelle de programmes, et l'exécution efficace de tout programme dans un environnement distribué. Par exemple, si chacune des étapes du calcul requiert une synchronisation entre toutes les machines participantes, l'implémentation du langage sera nécessairement inefficace au moins pour certains programmes, ce qui n'apparaît pas dans le modèle. Notre objectif étant de refléter les particularités du calcul distribué, ce type de modèle n'est pas adéquat.

Une fois le join-calcul défini au chapitre 2, nous utilisons ce calcul à la fois comme noyau d'un langage de programmation et comme calcul de processus. Bien que cette thèse présente surtout les aspects théoriques du join-calcul, nous avons développé une implémentation répartie en même temps que le formalisme ; cette correspondance entre la programmation et le calcul est de notre point de vue essentielle. Notre implémentation expérimentale comporte un langage de haut niveau, modulaire et fortement

typé, qui est compilé en code mobile avant d'être exécuté par un ensemble de machines connectées au réseau Internet. L'élaboration du langage ainsi que les propriétés formelles du système de types sont traitées au chapitre 3.

Ensuite, nous nous intéressons aux propriétés formelles du join-calcul, aux équivalences entre processus, aux techniques de preuves, et à leur application. Nous construisons un cadre général pour comparer des processus dans le join-calcul de manière plus ou moins fine (chapitre 4) ; nous élaborons des techniques de preuves adaptées au join-calcul (chapitre 5) ; nous relions également le join-calcul à d'autres calculs de processus, principalement le pi-calcul asynchrone (chapitre 6).

Dans un troisième temps, nous décrivons de manière explicite des primitives supplémentaires pour la programmation mobile distribuée, comme un raffinement du join-calcul. Plus précisément, nous définissons une sémantique opérationnelle qui rend compte des pannes locales au cours du calcul distribué, et permet d'exprimer les propriétés de programmes répartis dans un tel environnement (chapitre 7).

Comment modéliser un calcul réparti ?

De manière générale, un modèle du parallélisme fait intervenir plusieurs agents, ou processus, et décrit leurs interactions ; l'évolution globale d'un système résulte de l'agrégation de ces interactions élémentaires. Le modèle dépend donc essentiellement du mécanisme de communication élémentaire entre agents. Traditionnellement, la communication entre processus est modélisée par la transmission de messages d'un processus émetteur vers un processus récepteur. Plus précisément, un médium intervient également ; il véhicule le message, et le fait parvenir à destination selon diverses modalités. Par exemple, le médium peut préserver l'ordre des messages, garantir une certaine équité, avoir une capacité bornée ... et ces détails varient souvent d'une implémentation à l'autre.

Dans son calcul des systèmes communiquants (CCS), Milner simplifie grandement le modèle en utilisant un médium idéalisé, qu'il nomme l'éther. Les propriétés de l'éther permettent de modéliser la communication d'une manière uniforme, abstraite, qui passe le médium sous silence. Ainsi, deux processus communiquent par *rendez-vous*, cette étape de communication affectant simultanément l'émetteur et le récepteur. Intuitivement, le rendez-vous présuppose que les deux agents sont adjacents, mais quand ce n'est pas réaliste il est toujours possible de réintroduire le médium comme un agent intermédiaire, et de décrire la communication entre l'émetteur et le récepteur comme une succession de rendez-vous intermédiaires.

Par ailleurs, CCS et ses descendants structurent la communication par noms. Ces noms (ou adresses, ou canaux) représentent les capacités de communications élémentaires de chaque processus. Ainsi, l'émission et la réception sur un nom modélisent simplement les opérations de transmission, de routage, et de synchronisation sous-jacentes dans un système parallèle.

Le pi-calcul constitue une amélioration significative en terme d'expressivité, en modélisant l'échange de canaux au cours de la communication. Ainsi, un agent peut créer un canal de communication, puis le communiquer à ses pairs, qui acquièrent alors de nouvelles capacités de communication. Cette *mobilité de nom* permet de modéliser aisément tous les paradigmes de programmation traditionnels.

CCS et le pi-calcul permettent de décrire précisément de nombreux systèmes parallèles répartis ou non, et d'en évaluer les propriétés formelles. Bien sûr, la communication immédiate par l'éther ne correspond pas toujours au système modélisé beaucoup plus limité, mais il est toujours possible de raffiner le modèle pour faire apparaître les problèmes de communication si nécessaire.

De manière à refléter plus fidèlement la communication asynchrone, en particulier présente dans les modèles à objets répartis, Boudol [37] et Honda et Tokoro [70] ont proposé une version simplifiée du pi-calcul qui abandonne la notion de rendez-vous : Seule la réception est décelable, tandis que le processus émetteur d'un message ne peut directement détecter que son message a été reçu. Cette modification rapproche le pi-calcul de la pratique, puisqu'il n'est plus possible d'écrire un protocole qui utilise la communication synchrone [114].

En revanche, il devient possible d'implémenter chacune des étapes de communication de manière raisonnablement efficace, quoique centralisée. Ainsi, le langage de programmation PICT démontre que le pi-calcul peut être effectivement utilisé comme noyau d'un langage de haut niveau pour la programmation parallèle [122, 145]. Comme nous le verrons, pourtant, l'absence de communication synchrone dans le langage n'implique pas nécessairement son absence dans l'implémentation ; par exemple, l'implémentation répartie du π-calcul asynchrone est loin d'être évidente.

Dans la pratique, la programmation répartie est plus compliquée. Pour programmer un ensemble d'ordinateurs connectés en réseau, l'interface des systèmes d'exploitations fournit des primitives de plus bas niveau que la communication par canaux dans l'éther. En général, il s'agit d'envois de messages asynchrones d'un point fixe du réseau à un autre, de manière peu fiable, comme dans le protocole IP ; des primitives de plus haut niveau rendent l'adressage plus commode — par exemple l'appel d'une procédure ou d'une méthode à distance masque le routage et l'encodage des données ; la résolution dynamique de l'adressage permet de rediriger les messages d'un point à un autre — et fournissent davantage de garanties — par exemple, l'absence de duplication des messages, leur intégrité, ou la confirmation ultérieure de leur réception. Par ailleurs, ces mécanismes de communication ne sont pas intégrés aux langages de programmation traditionnels ; ils sont disponibles dans des bibliothèques de protocoles ou dans des extensions, et se juxtaposent aux mécanismes de base du langage.

Il est bien sûr possible de ne pas considérer ces problèmes dans un langage "de haut niveau" fondé sur un calcul de processus, à condition d'implémenter son modèle de communication à partir de primitives de bas niveau. Outre d'évidents problèmes de performance, cette approche n'est pas satisfaisante parce que certains détails de l'implémentation doivent nécessairement être révélés au programmeur, qui ne peut alors plus raisonner uniquement sur son programme indépendamment d'une implémentation particulière. Le modèle abstrait perd alors beaucoup de son intérêt.

L'écart entre l'implémentation et le modèle cache par exemple le nombre de messages et de machines requis pour implémenter une opération "élémentaire". Dès lors, l'efficacité d'un programme mesurée par exemple par le volume de messages émis sur le réseau ne peut pas se déduire facilement du code source.

Dans certaines situations relativement courantes, cet écart devient évident. Lorsque certaines machines sont plus lentes que d'autres, ou sont en panne, l'utilisation de ces machines ralentit ou fait échouer le calcul. Si l'utilisation de ces machines n'apparaît pas dans le langage, ce comportement devient incompréhensible.

Dans le cas particulier de CCS asynchrone, le problème provient de certains aspects dynamiques de la communication : Lorsqu'un message est émis sur un canal par une machine, et qu'il existe un récepteur en attente sur une autre machine, le message devrait en théorie arriver sur cette machine sans autre détour. Cependant, la localisation des récepteurs sur un canal donné varie en fonction des autres étapes de calcul ; elle est une propriété globale dont chaque machine ne peut maintenir qu'une image approximative, par exemple par envoi de messages de mise à jour.

Par exemple, si une machine attend un seul message sur un canal donné, plusieurs autres machines peuvent en même temps produire un message sur ce canal et tenter de le communiquer à cette machine. Dans ce cas, seul le premier message est reçu, tandis que les suivants doivent être réexpédiés à d'autres récepteurs potentiels. Outre la présence de nombreux messages inutiles, cela pose un sérieux problème en cas de panne d'une machine qui a reçu trop de messages : Si un processus reçoit un seul message, en théorie sa disparition peut causer la perte d'au plus un message, tandis que d'autres récepteurs reçoivent les autres messages émis sur ce canal. En pratique, avec l'implémentation esquissée ci-dessus, de nombreux messages peuvent être perdus.

En fait, ces problèmes semblent irréductibles en présence de pannes de machines, quelle que soit la complexité de l'implémentation, du moment que celle-ci n'utilise que des messages asynchrones entre machines : Le choix d'un récepteur parmi plusieurs possibles s'apparente en effet à une élection (le récepteur élu peut ensuite envoyer des messages sur d'autres canaux aux autres machines, pour leur signaler la réception du message), problème pour lequel il n'y a pas de solutions asynchrones sans divergences.

Nous pouvons énoncer un critère simple pour l'implémentation d'un canal, de manière à garantir l'absence de messages inutiles, donc en particulier la possibilité de détecter un écart d'atomicité entre le modèle et l'implémentation : Quand un message est émis sur une machine, cette machine doit pouvoir déterminer une machine "réceptrice" telle que

– ou bien il y aura un processus récepteur pour ce message sur cette machine lorsque le message arrivera, c'est-à-dire indépendamment du reste du calcul ;

– ou bien il n'y aura plus jamais de récepteur sur ce canal.

Dans l'esprit de CCS, il existe une solution formelle à notre problème, qui consiste à faire réapparaître le médium sous forme de processus auxiliaires, de manière à vérifier le critère ci-dessus au prix d'une complexité accrue dans le calcul. En théorie, s'il existe en permanence sur chaque machine un processus-relais par canal qui lit tout message et le ré-émet sur ce même canal, alors l'existence de communications inutiles apparaît dans le modèle, et justifie formellement la vulnérabilité de ces messages à la panne de n'importe quelle machine. L'inefficacité de cette solution est flagrante, puisqu'en particulier le même message peut être réexpédié de relais en relais indéfiniment, et ainsi créer des divergences dans le calcul.

Une autre solution plus réaliste consiste à matérialiser chaque canal de l'éther par un processus auxiliaire chargé de centraliser toutes les requêtes d'émissions et de réception sur ce canal. Ce processus se situe alors sur une machine donnée ; si cette machine tombe en panne, il devient clair dans le modèle que la communication sur ce canal est coupée. (Comme nous le verrons, c'est au fait l'encodage du pi-calcul dans notre modèle.)

Cette analyse reprise plus en détail au chapitre 2 souligne que le pi-calcul asynchrone — et plus généralement les calculs de processus pour le parallélisme — ne nous paraît pas adéquat pour la *programmation* répartie, même s'il permet d'en étudier certains aspects. Nous proposons un calcul similaire qui vérifie notre critère d'implémentation pour tout programme, indépendamment de la localisation des processus.

Pour cela, nous étudions un autre modèle standard du parallélisme, la machine chimique abstraite de Berry et Boudol [28]. Ce formalisme présente la sémantique opérationnelle des calculs de processus sous une forme compacte et intuitive, comme un ensemble de règles de réécritures opérant sur des multi-ensembles de processus élémentaires. Nous montrons que le même problème de routage apparaît pour sélectionner une règle de réécriture lorsque le multi-ensemble est réparti entre plusieurs sites, et suggérons une modification qui nous amène directement au modèle opérationnel du join-calcul, la *machine chimique réflexive.*

Présentation du join-calcul

Le chapitre 2 présente notre modèle pour la programmation répartie. Nous y discutons tout d'abord le cheminement qui nous conduit à proposer ce nouveau modèle plutôt que d'utiliser un modèle existant. Bien que de nombreux calculs de processus existent, nous expliquons pourquoi ces modèles sont inadaptés à la programmation repartie. Nous y présentons ensuite notre modèle en détail.

Un premier exemple

Au chapitre 2, la sémantique opérationnelle est présentée à partir d'un exemple, puis définie de manière générale. Nous reprenons ici l'exemple de manière plus informelle, sans utiliser le formalisme chimique.

Notre exemple est la version simplifiée d'un serveur d'impression sur un réseau local. Chaque station de travail sur le réseau peut envoyer des requêtes d'impression au serveur. D'autre part, chaque imprimante peut signaler sa présence au serveur, qui répartit la charge entre les imprimantes disponibles.

Les seules valeurs dans le join-calcul sont les noms de canaux, ou adresses. De ce point de vue, le join-calcul est une variante du pi-calcul, dont il utilise la mobilité de nom. Plus tard, il sera facile de généraliser le calcul à d'autres valeurs primitives comme les entiers ou les chaînes de caractères.

L'état du calcul est représenté par deux sortes de termes, des *messages*, et des *règles de réaction*. Ces termes peuvent être assemblés par composition parallèle pour former les processus du calcul.

Le message $x\langle y\rangle$ comporte deux noms ; x est l'adresse du message, y est son contenu. Plusieurs messages peuvent être assemblés par composition parallèle.

Par exemple, nous modélisons l'interface du serveur par deux noms, *imprimer* pour les requêtes d'impression, et *accepter* pour les imprimantes disponibles. Ainsi, le processus

$$\mathit{imprimer}\langle 1\rangle \mid \mathit{imprimer}\langle 2\rangle \mid \mathit{accepter}\langle \mathit{laser}\rangle$$

se compose de trois messages, et décrit un état du système où deux impressions des fichiers 1 et 2 sont en attente, tandis qu'une imprimante laser est disponible. L'ordre des messages est sans importance ; formellement la composition parallèle est associative et commutative.

Notée $J \triangleright P$, une règle de réaction consomme un ensemble de messages de la forme décrite dans le filtre J, et déclenche l'exécution d'une copie du processus P dans lequel les paramètres formels de J sont remplacés par les arguments transmis dans ces messages. La même règle peut être utilisée à plusieurs reprises, tant qu'il y a des messages à consommer.

Par exemple, nous définissons le comportement du serveur par une règle de réaction notée D, qui décrit comment les messages envoyés sur les noms *imprimer* et *accepter* sont traités.

$$D \overset{\text{def}}{=} \mathit{accepter}\langle \mathit{imprimante}\rangle \mid \mathit{imprimer}\langle \mathit{fichier}\rangle \triangleright \mathit{imprimante}\langle \mathit{fichier}\rangle$$

Cette règle consomme deux messages, l'un sur *accepter*, l'autre sur *imprimer*, et déclenche l'impression en envoyant le fichier à l'imprimante. Nous pouvons regrouper la définition du serveur d'impression et son état courant dans un seul processus.

$$P \overset{\text{def}}{=} \mathsf{def}\ D\ \mathsf{in}\ \mathit{imprimer}\langle 1\rangle \mid \mathit{imprimer}\langle 2\rangle \mid \mathit{accepter}\langle \mathit{laser}\rangle$$

Suivant notre explication informelle de la règle D, le message $\mathit{accepter}\langle \mathit{laser}\rangle$ et l'un des deux messages $\mathit{imprimer}\langle 1\rangle$ ou $\mathit{imprimer}\langle 2\rangle$ peuvent être consommés par le serveur d'impression. Par exemple, nous avons une étape de réduction

$$P \rightarrow \mathsf{def}\ D\ \mathsf{in}\ \mathit{imprimer}\langle 2\rangle \mid \mathit{laser}\langle 1\rangle$$

Cette réduction a opéré une synchronisation entre les deux messages consommés ; en particulier le choix entre 1 et 2 n'est pas déterminé avant la réduction.

En revanche, le traitement des messages envoyés sur *imprimer* et *accepter* est entièrement défini par D. Ces deux noms sont liés par D, ce qui exclut la réception de tels messages dans toute autre définition, ou la redéfinition du serveur d'impression. Cette propriété, appelée portée lexicale des noms, simplifie beaucoup l'étude formelle du join-calcul et son implémentation. En particulier, toute synchronisation entre messages doit être déclarée au préalable, au moment ou les noms des messages sont définis, et cette information est disponible à la compilation.

Un serveur d'impression plus réaliste pourrait par exemple envoyer le nom *imprimer* aux différentes machines du réseau, en utilisant d'autres noms définis sur ces machines.

De manière plus générale, nous utiliserons la syntaxe suivante pour le join-calcul :

$$\begin{array}{lcll}
P & ::= & & \text{processus} \\
 & & x\langle v_1,\ldots,v_n\rangle & \quad\text{message asynchrone} \\
 & \mid & \mathsf{def}\ D\ \mathsf{in}\ P & \quad\text{définition locale} \\
 & \mid & P \mid P' & \quad\text{exécution parallèle} \\
 & \mid & \mathsf{0} & \quad\text{processus inerte} \\
 & & & \\
D & ::= & & \text{définitions} \\
 & & J \triangleright P & \quad\text{règle de réaction} \\
 & \mid & D \wedge D' & \quad\text{composition de définitions} \\
 & \mid & \mathsf{T} & \quad\text{définition vide} \\
 & & & \\
J & ::= & & \text{filtre} \\
 & & x\langle y_1,\ldots,y_n\rangle & \quad\text{message requis} \\
 & \mid & J \mid J' & \quad\text{synchronisation de messages}
\end{array}$$

Parmi ces termes, nous n'avons pas encore introduit le processus inerte $\mathsf{0}$ utilisé pour représenter l'absence de messages. Nous n'avons pas non plus utilisé la forme générale d'une définition, qui peut comporter un nombre quelconque de règles de réductions. A nouveau, nous utilisons une définition inerte T pour représenter l'absence de règles.

A partir de cette syntaxe, nous décrivons l'état du calcul par une *solution chimique* qui comporte deux multi-ensembles : un multi-ensemble de processus $\mathcal{P}$ en cours d'exécution et un multi-ensemble de règles de réactions actives $\mathcal{D}$.

– Une relation d'équivalence structurelle (notée $\rightleftharpoons$) décrit comment passer d'une solution à une autre en réarrangeant l'ordre des définitions et des processus, et en activant de nouvelles règles de réaction. Toutes ces étapes sont réversibles.

– Une relation de réduction chimique (notée $\rightarrow$) décrit l'utilisation d'une règle de réaction particulière qui consomme un paquet de messages préalablement assemblé par équivalence structurelle pour correspondre au filtre de la règle. Seules ces étapes correspondent intuitivement à une progression du calcul.

Les relations d'équivalence structurelle $\rightleftharpoons$ et de réduction $\rightarrow$ sont définies dans la figure 2.3 page 60. Dans la suite de ce résumé, nous passerons souvent sous silence les étapes intermédiaires d'équivalence structurelle.

Nous terminons cet aperçu en indiquant comment le mécanisme qui permet d'activer et de désactiver des règles de réaction dans une solution permet en particulier de modifier la portée apparente des noms définis, et par conséquent d'implémenter l'*extension de portée lexicale*. Ce mécanisme commun avec le pi-calcul est essentiel pour modéliser la dynamique du calcul.

Pour permettre la réception d'un message en dehors de la portée des définitions de ses arguments, il est possible de déplacer les définitions à condition de ne pas modifier les liaisons : Par exemple,

$$\begin{array}{rl}
 & \mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P\ \mathsf{in}\ (\mathsf{def}\ a\langle\rangle \triangleright Q_a\ \mathsf{in}\ x\langle a\rangle) \mid (\mathsf{def}\ b\langle\rangle \triangleright Q_b\ \mathsf{in}\ y\langle b\rangle) \\
\rightleftharpoons^* & \mathsf{def}\ x\langle u\rangle \triangleright P \wedge a\langle\rangle \triangleright Q_a \wedge b\langle\rangle \triangleright Q_b\ \mathsf{in}\ x\langle a\rangle \mid y\langle b\rangle \\
\rightleftharpoons\rightarrow\rightleftharpoons & \mathsf{def}\ x\langle u\rangle \triangleright P \wedge a\langle\rangle \triangleright Q_a \wedge b\langle\rangle \triangleright Q_b\ \mathsf{in}\ P\{^a/_u, ^b/_v\}
\end{array}$$

(en supposant que a n'apparaît pas dans P et Q_b et que b n'apparaît pas dans P et Q_a, ce dont on peut s'assurer par α-conversion.) Techniquement, toutes les solutions ci-dessus comportent un seul processus et aucune règle active. En revanche, la succession d'étapes d'équivalences structurelles intermédiaires $\rightleftharpoons^*$ active les trois règles $x\langle u\rangle \triangleright P$, $a\langle\rangle \triangleright Q_a$, et $b\langle\rangle \triangleright Q_b$, les réarrange de manière différente, puis les désactive sous cette nouvelle forme, pour permettre l'étape de réduction sans déplacer les lieurs.

Nous donnons quelques exemples typiques de processus dans le join-calcul, avec une description informelle de leurs propriétés.

Relais La primitive de base du join-calcul étant la communication asynchrone, il est facile de définir de simples relais ou multiplexeurs, qui n'effectuent aucun calcul mais se contentent de faire suivre des messages d'un canal à un autre.

Par exemple, dans le processus

$$P \quad = \quad \mathsf{def}\ x\langle u\rangle \triangleright y\langle u\rangle\ \mathsf{in}\ Q$$

le nom y est libre, tandis que le nom x est lié. A chaque fois que Q envoie un message sur x, ce message est consommé par l'unique règle qui définit x, et un autre message adressé à y avec le même contenu est émis. Ainsi, la définition ci-dessus introduit un relais de x vers y. Puisque la définition de x est entièrement déterminée, on peut utiliser x ou y indifféremment dans Q.

Ce type de relais n'affecte pas le calcul ; il peut le ralentir un peu, mais ce ralentissement est imperceptible ; formellement, nous verrons que la présence de relais est indécelable dans le join-calcul, cette propriété sémantique étant essentielle au calcul asynchrone [139].

Une légère modification de l'exemple précédent permet de distribuer le même message à plusieurs destinataires :

$$\mathsf{def}\ x\langle u\rangle \triangleright x_1\langle u\rangle \mid x_2\langle u\rangle \mid \ldots \mid x_n\langle u\rangle\ \mathsf{in}\ Q$$

Chaque message $x\langle v\rangle$ émis par Q déclenche l'émission de n messages, qui peuvent être reçus par n définitions différentes dans un contexte approprié. De manière plus générale, il est possible de modéliser d'autre types de routage, par exemple avec une distribution des messages qui dépend de l'un des champs des messages.

Comment communiquer un processus ? Le join-calcul est un calcul du premier ordre en ce qui concerne les valeurs : Contrairement à d'autres formalismes, seuls des noms peuvent être communiqués, pas des processus. Cependant, il existe un encodage bien connu d'un processus arbitraire P en tant que nom ; il suffit de placer ce processus P sous une garde, puis de communiquer le nom correspondant à la place du processus. Ce nom est une continuation, souvent notée κ dans les exemples suivants. Nous utiliserons donc la règle

$$\kappa\langle\rangle \triangleright P$$

Dans un contexte où κ est ainsi lié, le join-calcul donne des garanties très fortes, puisque l'association entre le nom et le processus ne peut pas être altérée au cours du calcul.

Par exemple, il est facile de définir la réplication d'un processus en définissant une continuation et en la déclenchant à répétition :

$$\begin{array}{rcl} \mathsf{repl}\ Q & \overset{\text{def}}{=} & \mathsf{def}\ \kappa\langle\rangle \triangleright Q \mid \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle \\ & \rightarrow & \mathsf{def}\ \kappa\langle\rangle \triangleright Q \mid \kappa\langle\rangle\ \mathsf{in}\ (Q \mid \kappa\langle\rangle) \\ & \rightleftharpoons^* & Q \mid \mathsf{repl}\ Q \end{array}$$

La réduction consomme le message $\kappa\langle\rangle$, et génère immédiatement un autre message $\kappa\langle\rangle$ avec une copie de Q. En prenant garde de choisir un nom κ qui n'apparaît pas dans Q, nous pouvons utiliser l'équivalence structurelle pour faire sortir Q de la portée de $\kappa\langle\rangle$. Nous obtenons les réductions attendues :

$$\mathsf{repl}\ Q \ \rightarrow\ Q \mid \mathsf{repl}\ Q \ \rightarrow\ Q \mid Q \mid \mathsf{repl}\ Q \ \rightarrow\ \cdots$$

Choix interne Par définition, un processus effectue un choix interne lorsqu'il choisit une alternative parmi d'autres de manière non-déterministe, indépendamment du contexte. Ce type de choix est facilement encodable dans le join-calcul.

Par exemple, le choix entre les processus P et Q peut s'écrire simplement

$$\mathsf{def}\ x\langle\rangle \triangleright P \wedge x\langle\rangle = Q\ \mathsf{in}\ x\langle\rangle$$

(pour un x qui n'apparaît pas dans P ou Q) en utilisant plusieurs règles consommant le même message, ou encore

$$\mathsf{def}\ p\langle\rangle \triangleright P \wedge q\langle\rangle = Q\ \mathsf{in}\ \mathsf{def}\ x\langle\rangle \mid y\langle u\rangle \triangleright u\langle\rangle\ \mathsf{in}\ x\langle\rangle \mid y\langle p\rangle \mid y\langle q\rangle$$

en utilisant deux continuations et plusieurs combinaisons de messages pour la même règle de réaction.

Au passage, nous remarquons que dans le join-calcul une définition répliquée est plus simple qu'une définition à usage unique, pour laquelle un message auxiliaire $x\langle\rangle$ limite le nombre d'utilisations. Cet aspect correspond à ce qui se passe dans l'implémentation, où une définition simple n'a pas d'état.

Programmation impérative Nous terminons notre série d'exemples par un codage un peu plus long mais particulièrement utile. Il s'agit d'une cellule de référence, qui permet de stocker une valeur, d'y accéder, et de la modifier au cours du calcul. La cellule de référence permet de modéliser tout système réparti impératif à base de mémoire partagée. Par ailleurs, cet exemple illustre bien l'usage de la portée lexicale pour contrôler le comportement de chaque cellule.

Nous définissons l'abstraction qui génère une nouvelle cellule par la règle

$$\mathit{cellule}\langle v_0, \kappa_0\rangle \triangleright \left(\mathsf{def} \begin{array}{rl} \mathit{lire}\langle\kappa\rangle \mid s\langle v\rangle \triangleright & \kappa\langle v\rangle \mid s\langle v\rangle \\ \wedge \text{écrire}\langle u, \kappa\rangle \mid s\langle v\rangle \triangleright & \kappa\langle\rangle \quad\mid s\langle u\rangle \end{array} \;\mathsf{in}\; \kappa_0\langle \mathit{lire}, \text{écrire}\rangle \mid s\langle v_0\rangle\right)$$

Chaque message envoyé sur *cellule* déclenche un processus qui contient la définition d'une nouvelle cellule, indépendante des autres. Dans ce processus gardé, trois nouveaux noms *lire*, écrire, s sont définis selon deux règles, puis les deux premiers noms sont envoyés sur un canal de continuation κ_0 fourni par l'appelant. Ces deux noms *lire*, écrire constituent l'interface de la nouvelle cellule. Le troisième nom s reste local (cette propriété est facile à établir, puisque s n'est jamais communiqué à l'intérieur de la portée lexicale de la définition, et n'est pas accessible de l'extérieur) ; de plus, il y a toujours exactement un message sur s, qui contient la valeur courante v de la cellule (cet invariant s'établit aisément ; il est initialement vérifié par le message $s\langle v_0\rangle$, et chacune des deux règles de réduction le préserve), et cette valeur est consultée ou modifiée lors des réceptions de messages sur *lire* ou écrire. Dans le premier cas, le même message $s\langle v\rangle$ est immédiatement ré-émis tandis que la valeur courante est également renvoyée à l'appelant ; dans le deuxième cas la valeur courante est jetée et remplacée par la nouvelle valeur fournie par l'appelant.

Typage des canaux de communication

Le join-calcul résulte de notre insistance à prendre en compte certaines contraintes inhérentes à la programmation répartie. Cependant, ces contraintes s'avèrent également intéressantes pour définir un langage fondé sur le join-calcul.

Ainsi, la communication est largement déterminée statiquement par la définition de nouveaux noms, et cet aspect déclaratif facilite l'analyse des programmes et, de notre point de vue, leur clarté. Comme nous le verrons, il est possible par exemple de ré-interpréter le join-calcul comme une extension naturelle d'un langage fonctionnel de haut niveau à la ML auquel on aurait ajouté le parallélisme. Aussi, l'analyse statique des programmes est sensiblement plus simple que pour les autres calculs de processus avec mobilité de nom, parce que pour un nom donné l'ensemble des récepteurs sur ce nom est connu.

Afin de détecter à la compilation le mauvais usage des noms du join-calcul, nous proposons un système de typage simple et expressif qui bénéficie directement de ces propriétés statiques. En fait, notre système est proche du polymorphisme paramétrique à la Milner, popularisé par le langage ML, et il en conserve tous les avantages.

La communication dans le join-calcul consiste à envoyer et recevoir des messages sur certains noms, ces messages contenant eux-mêmes d'autres noms qui peuvent intervenir dans la suite du calcul. De ce point de vue, le join-calcul est un calcul d'ordre supérieur, où les canaux remplacent les fonctions.

Bien que la sémantique opérationnelle soit très différente, la structure syntaxique des définitions dans le join-calcul généralise celle des définitions mutuellement récursives de ML. La différence la plus notable est la présence de filtres qui reçoivent plusieurs messages de sources différentes au cours de l'exécution, ce qui requiert quelques précautions dans la règle de généralisation.

Un nom x utilisé pour communiquer des messages contenant n valeurs de types respectifs $\tau_1, \ldots, \tau_n$ est naturellement doté du type $\langle \tau_1, \ldots, \tau_n \rangle$. En pratique, il y a d'autre types que les types de canaux ; par exemple, nous ajoutons les booléens, les entiers, les chaînes de caractères, chaque ensemble de valeurs ayant son propre type et un ensemble de fonctions primitives typées.

Dans l'exemple qui suit, nous supposons que *imprimer* est un nom primitif qui affiche sur la console les arguments entiers qu'il reçoit. Si `Entier` est le type des entiers, alors le type de *imprimer* est $\langle \texttt{Entier} \rangle$ (un canal transportant un entier). Nous pouvons définir un nom qui reçoit une paire d'entiers et les imprime, par la règle

$$\mathit{imprimer\_paire}\langle x, y \rangle \triangleright \mathit{imprimer}\langle x \rangle \mid \mathit{imprimer}\langle y \rangle$$

Il alors facile d'inférer que le type de *imprimer_paire* est $\langle \texttt{Entier}, \texttt{Entier} \rangle$ (un canal transportant deux entiers).

Dans certains cas, l'usage d'un nom dans un processus gardé ne détermine pas entièrement son type ; ce nom est alors *polymorphe*. Par conséquent, le type des arguments n'a pas besoin d'être entièrement déterminé pour garantir le bon typage du nom polymorphe ; les parties non-spécifiées sont représentées par des variables de type qui peuvent être remplacées par n'importe quel type. Cette approche bien connue s'appelle le polymorphisme paramétrique. Par exemple, la règle suivante définit un nom polymorphe :

$$\mathit{appliquer}\langle \kappa, x \rangle \triangleright \kappa\langle x \rangle$$

Le nom *appliquer* reçoit deux arguments κ et x, et déclenche le processus $\kappa\langle x \rangle$. Si τ est le type de x, alors κ qui transmet des valeurs x doit être de type $\langle \tau \rangle$. Par conséquent, le nom *appliquer* doit être de type $\langle \langle \tau \rangle, \tau \rangle$, pour tout type τ. Cette généralisation est rendue explicite par un quantificateur, en dotant *appliquer* du schéma de type $\forall \alpha. \langle \langle \alpha \rangle, \alpha \rangle$.

Par la suite, ce schéma de type permet de bien typer des messages qui instancient α de plusieurs manières différentes, par exemple pour typer le processus

$$\mathit{appliquer}\langle \mathit{imprimer}, 4 \rangle \mid \mathit{appliquer}\langle \mathit{imprimer\_chaine}, ''\mathit{bonjour}'' \rangle$$

Contrairement à ML, le join-calcul permet de décrire des calculs parallèles, et en particulier des effets de bord, en utilisant des filtres qui synchronisent plusieurs messages dans une même règle. Par exemple, nous pouvons considérer une variante plus répartie de *appliquer* qui reçoit le canal κ et l'argument x sur deux canaux différents :

$$\mathit{canal}\langle\kappa\rangle \mid \mathit{argument}\langle x\rangle \triangleright \kappa\langle x\rangle$$

Les types des arguments κ et x restent inchangés, et donc les noms *canal* et *argument* ont pour types respectifs $\langle\langle\alpha\rangle\rangle$ et $\langle\alpha\rangle$. Pourtant, ces deux types sont corrélés par l'usage du même α. Cela exclut la généralisation de l'un ou l'autre de ces types, puisqu'autrement deux instances différentes lors de l'envoi indépendant de chacun des messages pourrait conduire à une erreur à l'exécution. Par exemple, le processus

$$\begin{array}{ll} & \mathit{canal}\langle \mathit{imprimer}\rangle \mid \mathit{argument}\langle 4\rangle \\ \mid & \mathit{canal}\langle \mathit{imprimer\_chaine}\rangle \mid \mathit{argument}\langle ''\mathit{bonjour}''\rangle \end{array}$$

n'est pas typable avec la définition ci-dessus, et peut conduire à une erreur manifeste de typage $\mathit{imprimer}\langle ''\mathit{bonjour}''\rangle$ en cas de synchronisation croisée.

Pour tenir compte de la synchronisation par le filtrage, nous définissons notre critère de généralisation pour un ensemble de règles comme suit : Une variable est généralisable tant qu'elle n'apparaît pas dans le type de plusieurs noms co-définis.

Une fois ce critère très simple identifié, le reste du système de typage est standard. L'ensemble des règles de typage sont rassemblées dans la figure 3.2 page 77.

Après quelques lemmes standards, nous établissons que chacune des étapes de réduction de notre sémantique chimique préserve le bon typage d'un programme (théorème 1, page 83) et qu'un programme bien typé ne contient pas d'erreurs d'arité ni d'erreurs de typage lors de l'application d'une primitive (théorème 2, page 85). Ce système de types est intégré à notre implémentation-prototype ; il répond aux besoins d'un langage de programmation moderne ; en particulier les types peuvent être automatiquement inférés à la compilation, et les erreurs de typage sont intelligibles pour le programmeur.

De nombreux systèmes de types plus sophistiqués ont été proposés, en particulier pour le pi-calcul. Ces systèmes n'offrent pas l'inférence de type polymorphe parce que les occurrences contravariantes de chaque nom ne sont pas connues statiquement. En revanche, ces systèmes décrivent plus précisément certains aspects de la communication, comme l'absence de certains blocages à l'exécution, le nombre de messages échangés, où l'usage purement local de certains noms. Ces extensions devraient également s'appliquer au join-calcul considéré comme sous-ensemble contraint du pi-calcul.

Vers un langage de programmation

Le join-calcul comporte un seul mécanisme pour contrôler l'exécution d'un programme, la réception conjointe de messages asynchrones. Les exemples ci-dessus suggèrent que ce mécanisme est suffisamment expressif, mais pas toujours commode pour programmer.

Ainsi, il n'y a aucun opérateur de composition séquentielle, et pour exprimer que deux processus P et Q s'exécutent dans un certain ordre, il faut placer ces processus sous des gardes, les représenter par des continuations, et déclencher celles-ci explicitement dans d'autre parties du programme. Il est bien connu que le passage de continuations permet d'exprimer les diverses stratégies d'évaluation de langages séquentiels comme le λ-calcul ; nous en donnons quelques exemples dans la section 3.4.

Pour le programmeur, la manipulation explicite des continuations est notoirement délicate ; en pratique il est préférable de fixer une stratégie d'évaluation (ici l'appel par valeur) et d'ajouter le contrôle séquentiel sous la forme de sucre syntaxique dans le langage. Cette approche a été par exemple proposée dans le langage PICT [122].

Nous expliquons sa mise en oeuvre dans le join-calcul en reprenant notre exemple *imprimer_paire*. En effet, le message $\mathit{imprimer\_paire}\langle 1,2\rangle$ peut causer l'affichage de "12" ou "21", selon l'ordonnancement des deux appels à *imprimer*, ce qui en général ne conviendra pas au programmeur. Pour contraindre l'ordre d'affichage, il faut modifier la définition de ces noms : Nous utilisons la nouvelle primitive typée

$$\mathit{imprimer} \quad : \quad \langle \mathtt{Entier}, \langle\rangle\rangle$$

où l'argument supplémentaire de type $\langle\rangle$ est la continuation déclenchée après l'affichage ; la nouvelle définition de *imprimer_paire* est :

$$\begin{array}{ll} \mathit{imprimer\_paire}\langle x,y,\kappa\rangle \triangleright & \mathsf{def}\ \kappa_y\langle\rangle \triangleright \kappa\langle\rangle\ \mathsf{in} \\ & \mathsf{def}\ \kappa_x\langle\rangle \triangleright \mathit{imprimer}\langle y,\kappa_y\rangle\ \mathsf{in} \\ & \mathit{imprimer}\langle x,\kappa_x\rangle \end{array}$$

Dans ce cas, la continuation est un message vide, ou signal ; de manière plus générale, la continuation peut aussi retourner à l'appelant un résultat.

L'usage du contrôle séquentiel étant très courant, il est indispensable de masquer cet encodage pénible. Pour cela, nous distinguons deux sortes de canaux : Les canaux asynchrones et les canaux synchrones. Les canaux asynchrones sont des canaux ordinaires du join-calcul ; les canaux synchrones sont des canaux qui transportent implicitement un argument supplémentaire comme continuation, que chaque définition peut utiliser pour retourner un signal ou un résultat. Par convention, nous utilisons une typographie différente pour les noms synchrones. Par exemple, la définition ci-dessus devient simplement

$$\begin{array}{lll} \mathrm{imprimer\_paire}(x,y) & \triangleright & \mathrm{imprimer}(x); \\ & & \mathrm{imprimer}(y); \\ & & \mathsf{reply\ to}\ \mathrm{imprimer\_paire} \end{array}$$

où le point-virgule exprime de manière concise l'attente d'un signal sur la continuation implicite, et où le processus "reply to imprimer_paire" déclenche la continuation implicitement reçue dans le message "imprimer_paire(x,y)".

De manière plus générale, le point-virgule est remplacé par l'expression

$$\mathsf{let}\ v_1,\ldots,v_n = \mathrm{nom\_synchrone}(u_1,\ldots,u_m)\ \mathsf{in}\ P$$

qui crée une continuation pour P, la transmet comme $m+1$-ème argument, et attend sur cette continuation n résultats. En fixant l'ordre d'évaluation des arguments chaque valeur peut être remplacée par une expression à évaluer préalablement à l'envoi du message.

Le langage synchrone obtenu est très proche de ML. En fait, si l'on se limite aux noms synchrones et que l'on exclue la composition parallèle dans les processus et les filtres, on obtient un langage noyau pour ML.

Dès lors, nous pouvons ré-interpréter le langage complet comme une extension de ML pour le parallélisme avec la possibilité d'évaluer plusieurs expressions en parallèle, et d'attendre le résultat de plusieurs expressions avant de poursuivre l'exécution.

Ce style de parallélisme a au fait déjà été proposé dans le cadre plus restreint de la programmation parallèle impérative, sous le nom de multi-fonctions [23]. Par exemple, deux tâches parallèles peuvent s'exécuter indépendamment en échangeant au milieu du calcul leurs résultats partiels ; cette synchronisation intermédiaire s'écrit en join-calcul

$$\mathrm{sync_a}(v_a) \mid \mathrm{sync_b}(v_b) \triangleright \mathsf{reply}\ v_b\ \mathsf{to}\ \mathrm{sync_a} \mid \mathsf{reply}\ v_a\ \mathsf{to}\ \mathrm{sync_a}$$

tandis que chaque tâche s'écrit de manière fonctionnelle (ici pour la tâche a) $\mathsf{let}\ v_a = E_a\ \mathsf{in}\ \mathsf{let}\ v_b = \mathrm{sync_a}(v_a)\ \mathsf{in}\ E'_a$.

L'encodage des noms synchrones s'étend sans problème particulier au système de typage ; nous définissons un nouveau constructeur de types fonctionnels $\rightarrow$ pour les noms synchrones

$$\langle\tau_1,\ldots,\tau_q\rangle \rightarrow \langle\tau'_1,\ldots,\tau'_p\rangle \quad \stackrel{\text{def}}{=} \quad \langle\tau_1,\ldots,\tau_q,\langle\tau'_1,\ldots,\tau'_p\rangle\rangle$$

Par exemple, "imprimer_paire" peut être doté du type $\langle\mathtt{Entier},\mathtt{Entier}\rangle \rightarrow \langle\rangle$.

La comparaison du join-calcul avec noms synchrones et de ML est intéressante pour plusieurs raisons. Ainsi, nous retrouvons les limitations apportées au polymorphisme paramétrique en présence d'effets de bord, traditionnellement représentés par un modèle de la mémoire, simplement en utilisant notre codage de la cellule mémoire. Grâce au sucre syntaxique, cette cellule s'écrit maintenant

$$\mathrm{cellule}(v_0) \triangleright \left(\begin{array}{ll} \mathsf{def} & \begin{array}{rl} \mathrm{lire}() \mid s\langle v\rangle \triangleright & \mathsf{reply}\ v\ \mathsf{to}\ \mathrm{lire} \mid s\langle v\rangle \\ \wedge\ \ \mathrm{écrire}(u) \mid s\langle v\rangle \triangleright & \mathsf{reply}\ \mathsf{to}\ \mathrm{écrire} \mid s\langle u\rangle \end{array} \\ \mathsf{in} & \mathsf{reply}\ \mathrm{lire}, \mathrm{écrire}\ \mathsf{to}\ \mathrm{cellule} \mid s\langle v_0\rangle \end{array} \right)$$

L'allocateur de cellules a un type polymorphe

$$\mathrm{cellule} \quad : \quad \forall\,\alpha.\,\langle\alpha\rangle \rightarrow \langle\langle\rangle \rightarrow \langle\alpha\rangle, \langle\alpha\rangle \rightarrow \langle\rangle\rangle$$

En revanche, chaque cellule allouée est monomorphe, du type de l'argument d'initialisation v_0.

Cette proximité a également de nombreux avantages pratiques, dont bénéficie notre implémentation du join-calcul. Par exemple, le langage utilise un système de modules standard, et le compilateur infère les types des noms du join-calcul. De plus, le langage d'implémentation du join-calcul étant CAML, une variante de ML, des passerelles existent entre les deux langages, et ces passerelles sont bien typées : Ainsi il est relativement facile de générer un code exécutable qui rassemble plusieurs modules écrits en join-calcul et en CAML. De plus, chacun de ces modules peut être compilé indépendamment et interfacé avec le reste du code par une déclaration d'interface. Chaque interface contient les types des valeurs définies dans chaque module, et l'utilisation correcte de ces valeurs peut dont être vérifiée à la compilation de chaque module. A l'exécution, chaque langage accepte des valeurs en provenance de l'autre langage sans tests ou conversions supplémentaires. En particulier, toutes les librairies CAML sont accessibles en join-calcul.

Le chapitre 3 contient enfin une discussion plus prospective sur la programmation orientée-objet dans le join-calcul. Bien que le calcul ne comporte pas de primitives spécifiques à la programmation objet, chaque définition peut être interprétée comme un objet dans un contexte parallèle. Ainsi, la cellule de référence devient un objet dont “cellule” est le créateur, “lire” et “ecrire” les méthodes, et s la représentation de l'état interne ; d'autres exemples plus complexes sont développés dans la section 3.5. La déclaration de filtres combinant méthodes et états est très expressive en ce qui concerne la synchronisation. Pour modéliser directement un calcul à objets, cependant, il est nécessaire d'ajouter des enregistrements extensibles comme valeurs.

Equivalences entre processus

Nous nous intéressons maintenant aux propriétés formelles de processus dans le join-calcul. Cette partie de la thèse est largement inspirée des techniques et des résultats pour CCS et le pi-calcul ; en retour, une bonne partie des résultats présentés ici sont nouveaux et devraient également s'appliquer à la plupart des calculs de processus asynchrones.

Au chapitre 4, nous posons les bases d'une théorie de l'équivalence entre processus dans le join-calcul.

Notre objectif est d'identifier les relations d'équivalences permettant d'énoncer des relations intéressantes et de les prouver. Ainsi une “bonne” équivalence doit être facile à interpréter, ne pas identifier de processus évidemment différents ni séparer de processus intuitivement équivalents dans un environnement réparti, et si possible offrir des techniques de preuves suffisantes pour établir ces équations.

A première vue, il semble que nous n'avons que l'embarras du choix, tant ces aspects théoriques ont été développés. Par exemple, il existe plusieurs centaines de propositions subtilement différentes pour définir l'équivalence entre processus [61].

Cela dit, notre approche intègre certaines spécificités techniques (sémantique à réduction, communication asynchrone, absence de tests de nom), ce qui nous éloigne des résultats bien connus, et nous amène à une présentation originale de l'équivalence entre processus.

Nous posons tout d'abord quelques critères naturels d'observation :

1. Deux processus qui émettent des messages sur des noms libres différents ne sont visiblement pas équivalents.

   Plus généralement, nous supposons données certaines observations élémentaires, les “barbes” dans le jargon des calculs de processus. Dans le join-calcul, un processus a une barbe sur un nom x lorsque ce nom est libre dans le processus et que le processus peut émettre un message sur ce nom, éventuellement après quelques réductions préalables.

2. L'exécution d'étapes de calculs internes n'est pas détectable en elle-même. Intuitivement, une équivalence qui compterait le nombre d'étapes présupposerait une horloge globale. Par ailleurs dans la programmation répartie un message passant par le réseau sera beaucoup plus lent qu'une dizaine de messages locaux. Une

mesure précise du coût du calcul est délicate ; en tout cas le nombre d'étapes de calcul n'est pas significatif dans un calcul asynchrone, parce que le coût de chaque étape n'est pas uniforme.

3. Deux processus équivalents doivent être interchangeables ; en particulier leur équivalence ne doit pas dépendre du contexte dans lequel ils sont placés.

   Cela nous amène à exiger des propriétés de congruence vis-à-vis des opérateurs du join-calcul pour nos équivalences.

Nous retrouvons heureusement au passage de nombreuses notions d'équivalence proposées dans d'autres contextes. Certaines définitions d'apparence fort différentes produisent les même équivalences, ce qui réduit le nombre d'équivalences à considérer. Au terme de notre étude, nous identifions quatre notions d'équivalences de plus en plus précises et nous exposons ce qui les sépare :

La notion la plus grossière d'équivalence est l'équivalence de tests (*may testing*). Cette notion est la plus naturelle ; elle identifie deux processus lorsque dans tout contexte ils peuvent émettre des messages sur les même noms. Elle exprime des propriétés de sûreté, mais est difficile à établir directement et ne dit rien quant à la présence de comportements souhaitables.

Si l'on s'intéresse aux comportements infinis, il est naturel de considérer certaines propriétés d'équité, ou de progression dans le calcul. Par exemple, il est possible d'observer les messages qui peuvent toujours être émis. De telles propriétés s'expriment sous la forme de conditions d'"équité formelle", qui imposent que si un message peut toujours être émis indépendamment du calcul en cours, alors ce message est effectivement émis au cours de n'importe quel calcul équitable. L'équivalence de tests équitable (*fair testing*) s'avère plus précise que le may-testing, mais malheureusement encore plus délicate à manipuler directement. Nous proposons d'autres caractérisations plus efficaces, quoique peut-être moins intuitives, en termes de simulations couplées. Ces caractérisations semblent spécifiques au join-calcul.

Si l'on s'intéresse également à l'évolution interne des processus, telle que reflétée par leur comportement observable, on retrouve la notion de congruence barbue proposée par Milner et Sangiorgi, ainsi que celle proposée par Honda et Tokoro.

Ces deux équivalences sont des bisimulations faibles. Une relation $\mathcal{R}$ est une bisimulation faible lorsque, pour toute paire de processus $P \mathrel{\mathcal{R}} Q$, pour chaque étape de calcul $P \rightarrow P'$ (ou $Q \rightarrow Q'$), il existe des étapes de calcul $Q \rightarrow^* Q'$ permettant à l'autre processus de rejoindre le premier, en restant dans la relation : $P' \mathrel{\mathcal{R}} Q'$. Ainsi, les points de choix internes de P et Q doivent correspondre.

De notre point de vue, l'intérêt majeur de la bisimulation barbue est technique : Ce type d'équivalence peut être établie par co-induction, en considérant une étape de réduction à la fois, et non plus des traces arbitrairement longues.

Bien que le problème soit peu connu, il existe au fait deux manières différentes de définir la congruence barbue. La définition traditionnelle de Milner et Sangiorgi retient la plus grande congruence contenue dans la plus grande bisimulation barbue. La définition de Honda et Tokoro considère la plus grande congruence qui est une bisimulation barbue. Cette seconde définition est plus exigeante, et les deux relations coïncident dans la mesure où la congruence barbue de Milner et Sangiorgi est elle-même une bisimulation. Cependant, cette propriété de congruence est loin d'être évidente ; elle est laissée comme un problème ouvert par Honda. Nous établissons ici la coïncidence de ces deux types de définitions dans le join-calcul (théorème 3, page 114) ; la preuve est complexe, mais les techniques employées s'appliquent également au pi-calcul pour fournir le même résultat [56].

Si l'on distingue les valeurs transmises syntaxiquement, et non plus selon leur comportement, on obtient plusieurs bisimulations étiquetées interchangeables. Ces équivalences abordées au chapitre 5 sont particulièrement faciles à établir parce qu'il n'est plus nécessaire de faire intervenir le contexte comme observateur : Les étiquettes suffisent. En revanche leur interprétation est douteuse, et de nombreuses équations utiles ne sont pas valides pour ces équivalences.

Une fois cette hiérarchie mise en place, la preuve d'une équivalence particulière peut avantageusement s'appuyer sur plusieurs équivalences auxiliaires plus fines et plus faciles à établir.

Toutes les équivalences mentionnées ci-dessous peuvent être définies à partir de diagrammes de simulation faible, de congruence, et de barbes, et la plupart des preuves présentées utilisent des techniques co-inductives. Pour factoriser certains arguments récurrents, nous développons dans ce cadre des techniques de preuve relativement générales, en nous appuyant sur des résultats de confluence.

Un modèle plus explicite du join-calcul

Les équivalences observationnelles définies au chapitre 4 sont relativement faciles à interpréter, mais elles fournissent une sémantique du join-calcul peu explicite. En effet, l'interaction entre un processus et son environnement n'est pas apparente ; elle est révélée par l'application de contextes particuliers, suivie de réductions internes.

Pour obtenir des techniques de preuves plus directes, en particulier pour éviter la quantification sur tous les contextes dans la définition des équivalences, nous raffinons donc notre sémantique en ajoutant des interactions primitives avec l'environnement. Cela nous permet de générer des transitions étiquetées, dont l'enchaînement décrit entièrement le comportement observable des processus.

L'interaction entre un processus et son environnement se décompose en deux actions complémentaires : Un processus peut émettre des messages vers le contexte, ce qui fournit au contexte de nouveaux noms, et réciproquement le contexte peut utiliser les noms qu'il a reçus pour émettre des messages vers le processus. En l'absence du contexte, nous avons besoin d'une syntaxe "ouverte" qui étend la syntaxe du join-calcul pour garder la trace des noms qui ont été communiqués à l'environnement.

Par exemple, le processus $\mathsf{def}\ x\langle u\rangle \triangleright P\ \mathsf{in}\ y\langle x\rangle$ peut émettre le message $y\langle x\rangle$, puisque y est un nom libre, et donc communiquer le nom x à l'environnement. Cependant, x reste exclusivement défini par le processus. Nous notons cette interaction par la transition étiquetée

$$\mathsf{def}\ x\langle u\rangle = P\ \mathsf{in}\ y\langle x\rangle \quad \xrightarrow{\{x\}\overline{y}\langle x\rangle} \quad \mathsf{def}_x\ x\langle u\rangle = P\ \mathsf{in}\ 0$$

L'étiquette indique que la communication a lieu sur le nom libre y, que cette communication exporte pour la première fois x vers l'environnement, et que le message contient simplement ce nom x. Tandis que x était auparavant un nom local, x est devenu visible de l'extérieur, ou "exporté". Il peut maintenant être utilisé pour envoyer un message vers le processus :

$$\mathsf{def}_x\ x\langle u\rangle = P \ \mathsf{in}\ 0 \quad \xrightarrow{x\langle z\rangle} \quad \mathsf{def}_x\ x\langle u\rangle = P \ \mathsf{in}\ x\langle z\rangle$$

L'étiquette indique que la communication a lieu sur le nom exporté x, et que le message contient le nom z, un nom libre ou exporté. Enfin, ce message peut être utilisé comme auparavant par la définition de x pour déclencher une copie du processus P :

$$\mathsf{def}_x\ x\langle u\rangle = P \ \mathsf{in}\ x\langle z\rangle \quad \rightarrow \quad \mathsf{def}_x\ x\langle u\rangle = P \ \mathsf{in}\ P\{^z/_u\}$$

Par la suite, ce nouveau sous-processus peut lui-même extruder certains noms localement définis, et l'environnement peut indépendamment démarrer d'autres copies de P pour d'autres valeurs de u, comme détaillé ci-dessus.

Ainsi, l'interface d'un processus "ouvert" comporte deux ensembles de noms disjoints : Les noms libres et les noms exportés, ces noms servant de supports exclusifs pour l'interaction avec l'environnement. Cette séparation provient de la définition statique des récepteurs. Dans les calculs de processus habituels, au contraire, le même nom peut servir à la fois pour émettre un message vers l'environnement, recevoir un message de l'environnement, ou encore effectuer une réduction interne. L'utilisation ici d'une interface structurée simplifie l'étude des processus, puisqu'il y a moins de transitions parasites à prendre en compte.

A partir de cette sémantique ouverte, nous appliquons la définition habituelle de la bisimulation étiquetée faible : Deux processus sont bisimilaires si lorsque l'un d'eux produit une transition, l'autre peut produire la même transition éventuellement suivie ou précédée de réductions internes, et que les deux processus résultants restent bisimilaires. Cette équivalence s'avère être une congruence (théorème 5, page 158) ; elle est strictement plus fine que la congruence barbue, ce qui la place au sommet de notre hiérarchie d'équivalences.

Par définition, notre sémantique ouverte autorise l'intrusion de tout message sur un nom x dès que ce nom est exporté (pourvu que le message soit bien typé). Cette propriété nous est dictée par la sémantique asynchrone du join-calcul : En effet, l'émetteur d'un message ne peut détecter que ce message est effectivement reçu, donc l'intrusion d'un message ne doit pas dépendre de l'état interne du processus. Malheureusement, cela induit de nombreuses intrusions parasites dans notre modèle ouvert, qui se contentent d'accumuler des messages inutiles dans le processus étudié. En particulier, l'arbre de synchronisation est presque toujours infini, ainsi que la plus petite bisimulation qui contient une paire de processus ouverts.

Pour réduire la taille du modèle, nous modifions notre sémantique en n'autorisant plus l'intrusion de messages que s'ils sont immédiatement consommés par une réduction interne. Puisque la nouvelle sémantique opérationnelle est plus stricte, nous définissons une nouvelle sorte de bisimulation faible plus laxiste, appelée bisimulation asynchrone et initialement proposée pour le pi-calcul : Lorsque deux processus sont bisimilaires et que l'un d'eux effectue une intrusion-réduction, l'autre processus peut soit effectuer une intrusion-réduction, soit stocker le message pour un usage ultérieur, tant que les deux processus restent bisimilaires.

Ce n'est pas encore suffisant, parce que parfois il faut fournir plusieurs messages à la fois pour passer un filtre ; par conséquent, nous considérons des intrusions multiples, qui fournissent plusieurs messages à la fois, à condition que tous ces messages soient immédiatement consommés par une réduction (potentiellement avec d'autres messages locaux requis par le filtre).

Nous établissons que les deux modèles équipés de leurs bisimulations étiquetées respectives sont équivalents (théorème 6, page 161), ce qui suggère l'utilisation généralisée de la bisimulation asynchrone, dont le modèle est nettement plus compact.

Nous étudions enfin en détail le lien entre ces bisimulations étiquetées et les congruences barbues. Puisque ces bisimulations sont des congruences et qu'elles respectent les barbes, elles sont au moins aussi fines que les congruences barbues. De fait, elles séparent davantage de processus, parce que l'égalité des étiquettes permet à l'environnement de comparer les noms de l'interface d'un processus, tandis qu'un contexte du calcul ne peut que les utiliser pour communiquer. Ainsi, la présence d'un relais devient décelable.

La comparaison de noms est une primitive courante dans les calculs de processus, mais elle est indésirable dans le cadre d'un langage de programmation, parce que l'implémentation doit alors respecter l'égalité, et ne peut plus librement introduire des noms intermédiaires—par exemple, notre implémentation introduit dynamiquement des relais lorsqu'un nom acquiert une portée globale, ce qui serait impossible si la comparaison de noms se réduisait à la comparaison de pointeurs.

Quoi qu'il en soit, nous montrons que ce problème de comparaison implicite dans les étiquettes est tout ce qui sépare nos sémantiques extensionnelles et intentionnelles. Dans ce but, nous ajoutons la comparaison de noms dans le calcul, et nous prouvons que la plus grande bisimulation barbue qui est une congruence pour tous les contextes d'évaluation coïncide alors avec la bisimulation étiquetée. La contrepartie de ce résultat dans le pi-calcul asynchrone valide une conjecture de Milner sur la bisimulation barbue [101].

Variantes et Encodages

Le chapitre 6 est presque entièrement consacré à l'exploration des propriétés formelles du join-calcul, à travers l'étude comparative de variantes du join-calcul, puis du pi-calcul. Ces variantes peuvent être reliées par divers encodages ; nous décrivons ces encodages, et nous en étudions les propriétés de correction. Cet ensemble de résultats permet de passer d'une variante à l'autre en préservant certaines équivalences, et fournissent des points de repères techniques utiles. Par ailleurs, ils fournissent de nombreuses illustrations des équivalences et techniques de preuve développées précédemment.

Dans le domaine des calculs de processus, une grande diversité de formalismes et de variantes coexistent ; en effet, chaque problème semble appeler une nouvelle variante, voire un nouveau calcul mieux adapté au problème. Bien que les mêmes idées et les mêmes méthodes s'appliquent, il en résulte une profusion de variantes, dont les liens formels ne sont pas toujours très clairs. Ainsi, en général chaque résultat est exprimé pour une variante particulière, et ne s'applique pas automatiquement aux autres variantes.

En ce qui concerne le join-calcul, cette variété apparaît pour deux raisons :

- En tant que noyau d'un langage de programmation, le join-calcul comporte certaines opérations utiles dont l'implémentation ne pose pas de problèmes particuliers. En théorie, pourtant, il est possible de simplifier davantage ces opérations afin d'obtenir un calcul minimaliste plus facile à étudier.

  Ainsi, la communication polyadique est commode, mais elle nous impose l'usage d'un système de types ; de même, une définition peut lier un grand nombre de noms et comporter de nombreuses clauses, ce qui complique les inductions structurelles.
- En tant que variante du pi-calcul adaptée à la programmation distribuée, le join-calcul semble bénéficier de propriétés formelles similaires. Il est utile de relier précisément ces deux calculs pour faciliter leur comparaison et permettre le transfert de résultats de l'un à l'autre.

Notre objectif est ici surtout théorique. En particulier certains encodages sont précis mais inefficaces ; cela n'est pas grave puisque ces variantes ne sont pas destinées à la programmation.

A propos des encodages

A titre d'exercice, il est souvent intéressant de relier deux formalismes en traduisant les termes d'un langage à l'autre. Cette approche est largement exploitée dans l'étude des calculs de processus.

Evidemment, il est souhaitable que cette traduction reflète et préserve également les propriétés entre les deux calculs. Si ces calculs s'expriment dans le même formalisme, il est possible de relier directement chaque processus P à sa traduction $[\![P]\!]$ par une équivalence. Souvent pourtant, chaque calcul a sa propre notion d'équivalence, ce qui rend impossible une comparaison directe des processus. On combine alors deux types de résultats :

1. La *correspondance opérationnelle* relie directement les réductions du calcul source à celles de l'image de la traduction. Par exemple, chaque réduction source peut donner lieu à une série de réductions dans la traduction initiale qui amène à la traduction du nouveau terme source, peut-être à une équivalence près pour rester dans l'image de $[\![\cdot]\!]$.
2. La complétude (*full abstraction*) relie les équivalences du calcul source à celles de l'image de la traduction : $P \approx Q$ si et seulement si $[\![P]\!] \approx' [\![Q]\!]$. Elle indique ainsi ce que la traduction révèle ou masque par rapport aux termes initiaux. Cette propriété est délicate à obtenir pour des équivalences observationnelles, parce que les contextes sources et objets peuvent être très différents. Souvent par exemple, de nombreux contextes dans le calcul cible ne sont pas des traductions de contextes sources ; ils peuvent en théorie briser des invariants imposés par la sémantique du calcul source, pour détecter des propriétés jusque là invisibles dans l'image de la traduction. Cette situation récurrente complique beaucoup certains codages.

Nos principaux résultats s'expriment en termes de complétude pour les équivalences mises en place au cours des chapitres 4 et 5. Le plus souvent, deux instances de la même équivalence sont utilisées de part et d'autre de la traduction.

Vers un join-calcul minimaliste

En théorie, le join-calcul que nous utilisons comme noyau de notre langage de programmation répartie peut encore être sensiblement simplifié. Outre son intérêt formel, cette entreprise s'avère utile par la suite, puisque les développements ultérieurs peuvent se concentrer sur le calcul réduit.

Le join-calcul élémentaire est défini par la grammaire suivante :

$$\begin{array}{lcll} P & ::= & & \text{processus élémentaire} \\ & & x\langle u\rangle & \text{message transmettant un seul nom} \\ & | & P_1 \mid P_2 & \text{composition parallèle} \\ & | & \mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P_1\ \mathsf{in}\ P_2 & \text{définition de deux noms par une règle} \end{array}$$

Nous établissons que ce calcul a exactement la même expressivité que le join-calcul complet en exhibant une série de codages complets (théorème 8, page 181).

Compilation interne des définitions

La première étape de cette réduction concerne les définitions ; en effet, l'équivalence structurelle suffit pour normaliser tout processus du join-calcul en un terme “plat” de la forme def D in M où le processus M est une composition parallèle de messages.

En revanche, la structure des définitions semble complexe, puisqu'une seule définition peut définir un nombre arbitraire de noms et introduire des règles qui joignent un nombre arbitraire de messages en une seule étape de calcul.

Les propriétés asynchrones et statiques du join-calcul permettent pourtant de manipuler cette structure sans modifier son comportement externe. Par exemple, une étape qui joint n messages ne semble pas implémentable dans un calcul où chaque étape joint 2 messages, mais comme tous ces messages sont adressés à la même définition, il suffit de stocker les messages un par un au fur et à mesure de leur arrivée, de représenter ces messages par un état interne, et de vérifier lorsqu'un nouveau message arrive si l'ensemble des messages en attente permet ou non de déclencher la traduction d'une règle n-aire.

Notre codage d'une définition complexe consiste donc à rendre explicite l'automate qui reconnaît les filtres de la définition et à stocker l'état de l'automate dans une cellule mémoire. Diverses précautions permettent de construire un tel codage complet pour la congruence barbue. Le codage est détaillé dans la section 6.3.3 ; il réduit une définition à n noms et m clauses à environ $1 + 2n + 2m$ définitions élémentaires imbriquées. Curieusement, l'usage de cet automate ressemble à la technique de compilation des définitions mise en oeuvre dans notre prototype, où le même automate apparaît, sous une forme plus efficace quoique formellement moins précise. (Une présentation détaillée de ces automates et de leur efficacité est disponible [87].)

Par ailleurs, il est possible de normaliser davantage la structure d'un processus en introduisant systématiquement des relais intermédiaires, ce qui permet par exemple de réduire le nombre de cas à considérer pour établir une bisimulation étiquetée, et de construire des traductions plus résistantes vis-à-vis du contexte (*cf.* section 6.4).

Ainsi, on peut s'assurer par une compilation préalable que tout nom communiqué à l'extérieur est un relais défini par une simple règle du type $x\langle\widetilde{u}\rangle \triangleright y\langle\widetilde{u}\rangle$, ou de manière plus sophistiquée que le même nom n'est jamais communiqué plus d'une fois à l'environnement. Ces compilations successives préservent et reflètent également l'équivalence observationnelle.

Encodage de la communication polyadique

La deuxième étape de notre réduction consiste à restreindre la communication aux messages monadiques : Tandis que chaque message du join-calcul transmet un nombre fixe mais arbitraire de noms, il est possible de remplacer l'émission et la réception de ce message par l'exécution de protocoles complémentaires qui transmettent les mêmes noms un par un.

Ce codage est un classique des calculs de processus, mais sa correction n'a semble-t-il jamais été établie en général pour des équivalences aussi fines que la congruence barbue (souvent, seule une correspondance opérationnelle est mentionnée). De fait, le codage de base ne préserve pas toutes les équations, puisque certaines de ces équations dépendent de l'utilisation exclusive de contextes bien typés, tandis que les traductions de contextes mal typés sont des contextes valides. Par exemple, une définition qui reçoit deux noms de types différents a la certitude que ces noms eux-mêmes sont différents ; en revanche, la traduction de cette définition peut recevoir deux fois le même nom.

En utilisant les encodages préalables décrits ci-dessus, le codage peut être renforcé, ce qui nous permet d'établir finalement sa complétude.

Du pi-calcul au join-calcul, et vice versa

Intuitivement, le join-calcul est une variante réduite du pi-calcul, et malgré une certaine divergence syntaxique les différences essentielles entre les deux calculs sont peu nombreuses :

1. Les trois lieurs du pi-calcul (réception, restriction, réception répliquée) sont regroupés en une seule construction syntaxique dans le join-calcul : La définition.
2. Le join-calcul impose la définition statique de tous les récepteurs, et ne communique donc que la capacité d'envoyer des messages ; au contraire, un nom reçu dans le pi-calcul peut être utilisé comme un nom défini.
3. Le join-calcul ne calcule jamais sur des noms libres.
4. Le pi-calcul n'autorise la réception que d'un message à la fois.

De fait, le join-calcul et le pi-calcul ont la même expressivité en ce qui concerne l'équivalence observationnelle (*cf.* section 6.6, en particulier pour la définition du pi-calcul), mais les codages les plus précis sont compliqués. Nous nous contentons ici de donner une idée de chacun des encodages.

La traduction du join-calcul dans le pi-calcul asynchrone décompose chaque définition en utilisant les trois préfixes du pi-calcul. Par exemple, dans le cas d'une définition binaire, nous avons la traduction

$$[\![\mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P \ \mathsf{in}\ Q]\!] \quad \overset{\text{def}}{=} \quad \nu x.\nu y.(!x\langle u\rangle.y\langle v\rangle.[\![P]\!] \mid [\![Q]\!])$$

(Le reste de la traduction reproduit les constructeurs du join-calcul à l'identique.) Dans cette traduction, le processus $x\langle\rangle.y\langle\rangle.[\![P]\!]$ reçoit un message sur x, puis un message sur y, puis se comporte comme $[\![P]\!]$. L'opérateur de réplication "!" reproduit le même comportement pour toute paire de messages $x\langle\rangle$ et $y\langle\rangle$. La symétrie entre x et y est donc brisée, mais cela n'est pas observable ; en effet la première famille de réductions qui reçoivent les messages sur x peut s'effectuer n'importe quand, tandis que chaque réception d'un message sur y correspond à une réduction source. Cet encodage n'est pas complet, mais il peut être renforcé pour le devenir (théorème 11, page 207).

La traduction du pi-calcul asynchrone dans le join-calcul est nécessairement plus complexe, puisqu'il faut communiquer deux capacités de communication par canal : Un canal du pi-calcul x permet d'effectuer à la fois l'envoi de messages $\overline{x}\langle u\rangle$ et la réception de messages $x(y).P$; son encodage dans le join-calcul comporte donc deux noms x_e et $\mathrm{x_r}$ utilisables pour chacune de ces opérations. Dans le pi-calcul, la communication est possible lorsqu'il y a un message et un récepteur ; cela se traduit naturellement par la règle

$$x_e\langle y_e, \mathrm{y_r}\rangle \mid \mathrm{x_r}() \quad \triangleright \quad \mathsf{reply}\ y_e, \mathrm{y_r}\ \mathsf{to}\ \mathrm{x_r}$$

tandis que chaque traduction d'un récepteur alloue implicitement une continuation pour y placer son processus gardé :

$$[\![x(y).P]\!] \quad \stackrel{\text{def}}{=} \quad \mathsf{let}\ y_e, y_r = \mathrm{x_r}()\ \mathsf{in}\ [\![P]\!]$$

Cette traduction impose la présence d'une définition de x_e et $\mathrm{x_r}$ pour chaque canal x du pi-calcul, ce qui est problématique lorsque x est un nom libre. Néanmoins, il est également possible de renforcer cette ébauche de codage pour le rendre complet vis-à-vis de la congruence barbue (*cf.* théorème 10, page 205).

Localisation et mobilité

Le join-calcul est un modèle élémentaire adapté à la programmation répartie, mais où la localisation des processus et des définitions est implicite. Plus précisément, le modèle est implémentable dans un environnement à plusieurs machines, quelle que soit la répartition des ressources à l'exécution, à condition que toutes les machines participantes puissent s'échanger des messages asynchrones.

Pour étudier cette implémentation répartie, ainsi que pour décrire certains comportements de programmes s'exécutant sur plusieurs machines, il est utile de raffiner le modèle pour faire apparaître la distribution des ressources.

D'autre part, ce raffinement nous permet de présenter un modèle où la localisation des ressources devient dynamique, intégrée au calcul. Dans un programme distribué, il est parfois nécessaire de contrôler la localité, en particulier parce que celle-ci détermine la résistance aux pannes. Ainsi, chaque groupe de processus et de définitions peut migrer d'une machine à l'autre, ou s'arrêter. En termes de langage de programmation, cela correspond à un modèle très expressif d'agents mobiles.

Nous représentons la répartition des ressources en les organisant par *emplacements*. Intuitivement, un emplacement réside sur un site particulier, et peut contenir des processus et des règles de réductions. Un emplacement peut se déplacer d'un site à l'autre ; ainsi un emplacement peut représenter un agent mobile. Un emplacement peut également contenir des sous-emplacements, ce qui donne une structure hiérarchique au calcul et permet de modéliser les sites comme des emplacements particuliers.

Les noms d'emplacements sont des valeurs de première classe, tout comme les noms de canaux. Ces noms peuvent être communiqués à d'autres processus éventuellement dans d'autres places, ce qui permet de programmer la gestion des places tout en contrôlant la migration par la portée lexicale. Chaque emplacement contrôle ses propres mouvements relativement à son sur-emplacement, en désignant son nouvel emplacement. Ce mécanisme permet de simplifier l'analyse des programmes lorsque certains emplacements sont immobiles, et fournit l'ébauche d'un mécanisme plus élaboré de sécurité.

Plusieurs machines chimiques Afin de modéliser la présence de plusieurs sites de calcul, nous raffinons la machine abstraite chimique réflexive en faisant apparaître une machine abstraite par emplacement, et en ajoutant une règle de communication asynchrone entre emplacements : Ainsi, l'état du calcul est maintenant représenté par une famille de paires de multi-ensembles $\{(\mathcal{D}_i, \mathcal{P}_i)\}$ qui contiennent respectivement les règles de réaction et les processus en cours d'exécution dans chaque emplacement.

Conformément à notre intuition de la localité, nous imposons que toutes les règles qui définissent un même nom soient localisées au même emplacement.

Chacune des machines chimiques évolue localement comme précédemment : Certains messages locaux sont consommés par des règles locales et remplacés par de nouveaux processus, tandis que les processus peuvent introduire de nouveaux noms avec leurs règles de réactions. (Dans ce cas, nous imposons, bien sûr, que ces noms n'apparaissent pas dans d'autres emplacements.)

Par ailleurs, une règle supplémentaire décrit la communication globale : Lorsqu'un message est émis dans un emplacement et que ce message est défini dans un autre emplacement, une étape de calcul transporte ce message de l'emplacement émetteur vers l'emplacement récepteur. Cette étape est muette ; elle ne dépend pas du message lui-même ou de l'emplacement émetteur, mais uniquement de l'emplacement récepteur ; elle n'affecte qu'un message à la fois. Par la suite, ce message pourra être consommé localement, peut-être avec d'autres messages.

Informellement, ce nouveau mécanisme de calcul reflète le *routage* des messages d'un point à l'autre du réseau.

Dans le join-calcul et avec la restriction donnée ci-dessus, l'implémentation du routage est élémentaire. En effet, chaque nom du join-calcul est irréversiblement attaché à l'emplacement où il a été introduit, et l'adresse de l'emplacement peut facilement être propagée chaque fois que ce nom est communiqué d'une machine à l'autre. Puisque cet aspect technique n'affecte pas la sémantique du langage, nous en effaçons les détails, et ne conservons que l'étape du calcul, qui est importante pour la synchronisation.

Nous illustrons notre propos en reprenant notre exemple favori, le serveur d'impression. Nous distinguons maintenant trois machines : La machine du serveur s, une imprimante laser p qui contacte le serveur, et la machine d'un utilisateur u sur laquelle une requête d'impression est en attente. Nous conservons la même définition pour le serveur

$$D \stackrel{\text{def}}{=} \mathit{accepter}\langle \mathit{imprimante}\rangle \mid \mathit{imprimer}\langle \mathit{fichier}\rangle \triangleright \mathit{imprimante}\langle \mathit{fichier}\rangle$$

Nous avons la série de réductions suivante :

$$\begin{array}{lllll}
 & D \vdash_s & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{accepter}\langle \mathit{laser}\rangle & \parallel \vdash_u \mathit{imprimer}\langle 1\rangle \\
\overset{\text{Comm}}{\to} & D \vdash_s \mathit{imprimer}\langle 1\rangle & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{accepter}\langle \mathit{laser}\rangle & \parallel \vdash_u \\
\overset{\text{Comm}}{\to} & D \vdash_s \begin{array}{l}\mathit{accepter}\langle \mathit{laser}\rangle,\\ \mathit{imprimer}\langle 1\rangle\end{array} & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p & \parallel \vdash_u \\
\overset{\text{Red}}{\to} & D \vdash_s \mathit{laser}\langle 1\rangle & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p & \parallel \vdash_u \\
\overset{\text{Comm}}{\to} & D \vdash_s & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{laser}\langle 1\rangle & \parallel \vdash_u
\end{array}$$

La première étape transmet le message $\mathit{imprimer}\langle 1\rangle$ de la machine de l'utilisateur à la machine du serveur ; cette réduction a lieu parce que le nom *imprimer* est uniquement défini sur la machine du serveur ; en particulier on peut imaginer que l'information statique "*imprimer* est défini dans s" est propagée en même temps que le nom *imprimer*. De la même manière, la deuxième étape du calcul transmet le message $\mathit{accepter}\langle \mathit{laser}\rangle$ de l'imprimante au serveur. Ensuite, les deux messages maintenant sur le serveur sont conjointement reçus ; cette étape représente une synchronisation locale au serveur, et la seule étape utile si l'on oublie la localisation. Elle déclenche un nouveau message sur le serveur, adressé à l'imprimante. A nouveau, ce message est d'abord transmis à cette machine, puis traité localement.

Cet exemple permet aussi d'illustrer la portée statique globale des noms dans le join-calcul réparti, ainsi que le mécanisme d'extension de portée par la communication. En supposant que le nom *laser* est initialement local à la machine p, une étape de réduction préliminaire sur cette machine peut être

$$\vdash_p \mathsf{def}\ \mathit{laser}\langle f\rangle \triangleright P\ \mathsf{in}\ \mathit{accepter}\langle \mathit{laser}\rangle \quad \overset{\text{Str-def}}{\rightleftharpoons} \quad \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{accepter}\langle \mathit{laser}\rangle$$

Ensuite, la deuxième réduction Comm dans la série ci-dessus étend effectivement la portée de *laser* à la machine du serveur d'impression — il n'est plus possible d'effectuer une étape Str-def pour restreindre sa portée à une seule machine. Ainsi, le serveur d'impression devient capable d'envoyer des messages à l'imprimante qu'il ne connaissait pas auparavant.

Agents mobiles

Les différents emplacements qui apparaissent au cours du calcul sont structurés en arbre ; nous commençons par justifier ce choix. En effet, une alternative serait de considérer un modèle d'emplacements indépendants, dynamiquement liés à la machine sur laquelle ils s'exécutent. Ce modèle pose cependant certains problèmes.

Dans le cas où plusieurs emplacements changent de place au cours du calcul, par exemple, il est souhaitable que la configuration finale de dépende pas de l'ordre des migrations. L'objectif d'une migration est le plus souvent de se retrouver "au même endroit" qu'un autre emplacement. Or, cette liaison entre emplacements n'est pas assurée en cas de migrations multiples : Par exemple, si un programme crée un agent mobile pour se rendre sur un serveur, tandis que le serveur change de machine, il faut programmer explicitement l'agent pour suivre le serveur. De plus, la migration du serveur et celle de l'agent ne sont pas atomiques ; il est ainsi possible que le serveur se déplace avant l'agent, puis que l'ancienne machine du serveur tombe en panne.

En revanche, une structure d'emplacements imbriqués permet de refléter de manière stable la proximité de certaines parties du calcul, ainsi que l'atomicité de leur migration (ou de leur échec). Par ailleurs, cette structure permet de modéliser facilement des systèmes à objets répartis avec des sous-objets. Techniquement enfin, les machines peuvent elles-mêmes être modélisées par des emplacements immobiles, ce qui allège notre formalisme et nous permet de décrire des configurations distribuées complexes par des hiérarchies d'emplacements.

Jusqu'ici, la description de la localité dans le join-calcul reste purement descriptive : La nouvelle sémantique indique précisément où se situe chacun des termes du calcul, mais cette localisation n'affecte pas le calcul lui-même (*cf.* théorème 12, page 234), en particulier pour nos équivalences qui ne mesurent pas l'efficacité du calcul. L'apparition de pannes, en revanche, donne un sens directement observable à la localité.

Pannes

Notre calcul fournit également un modèle élémentaire de pannes. L'arrêt brutal d'une machine peut causer la terminaison des emplacements qui y résident, de manière irréversible. De manière plus générale, chaque emplacement peut s'arrêter, entraînant tous ses sous-emplacements. La terminaison d'un emplacement est détectable à partir d'autres emplacements qui continuent à fonctionner, ce qui permet de programmer explicitement la résistance aux pannes si besoin est.

Le modèle de pannes et le modèle de migration sont loin d'être indépendants ; outre le partage de la notion d'emplacements, la migration peut être utilisée de manière constructive pour se prémunir contre certaines erreurs. Par exemple, un protocole entre deux programmes est considérablement compliqué lorsque chacun des programmes craint une panne de l'autre programme entre chaque message : Il faut alors écrire de nombreux codes de rattrapage de panne. En revanche, si les deux programmes envoient leur agent sur une machine fiable, le problème disparaît presque entièrement (il faut encore vérifier la vivacité des deux programmes à la fin du protocole). Même dans le cas où la machine commune n'est pas fiable, elle donne une garantie très utile : Les deux agents disparaissent ensemble en cas de pannes ; à nouveau, un test unique dans chacun des programmes suffit à se prémunir contre ce type de pannes.

Nous donnons un sens opérationnel précis aux pannes — l'arrêt irréversible d'une branche d'emplacements —, en donnons quelques exemples, et en exhibons quelques propriétés formelles en prouvant certaines équations élémentaires. L'utilisation de telles techniques pour prouver des protocoles distribués plus complexes reste à explorer.

Autres aspects du join-calcul

Cette thèse aborde surtout les aspects formels du join-calcul décrits ci-dessus, et laisse de coté deux développements importants dont nous donnons ici également une idée.

Implémentations Avec Maranget, nous avons prototypé une implémentation du langage réparti décrit aux chapitres 3 et 7 [59].

Cette implémentation comporte un compilateur et un interprète, tous deux écrits dans le langage Objective Caml. Le langage correspond assez fidèlement au join-calcul distribué équipé du système de types polymorphes paramétriques. Il permet également de structurer le code en modules qui peuvent être compilés de manière autonome. Enfin, son interface avec le langage sous-jacent Objective Caml permet d'importer et d'exporter des modules d'un langage à l'autre, en particulier toutes les librairies standards.

Le Fessant et Maranget décrivent de manière détaillée diverses stratégies pour la compilation des filtres du join-calcul, dont celle mise en oeuvre dans notre implémentation.

L'implémentation est répartie, dans la mesure où un nombre arbitraire de machines connectées au réseau Internet peuvent prendre part au calcul, y compris en cours de route, que le code et les ressources peuvent migrer librement d'une machine à l'autre, et que l'arrêt d'une machine n'affecte que les emplacements qui s'y trouvent ou—en l'état actuel—qui s'y sont trouvés.

Bien que le nombre de messages échangés au cours de l'exécution soit en général plus faible que le nombre d'étapes de calcul effectivement globales dans le calcul (règles COMM etGO), l'efficacité à l'exécution reste très perfectible, cet aspect du problème n'étant pas une priorité pour notre prototypage.

Par ailleurs, l'implémentation ne fournit qu'une implémentation incomplète du modèle de pannes pour les pannes "réelles". La détection de pannes est très partielle ; en outre, la migration d'emplacements crée encore certaines dépendances parasites vis-à-vis des machines précédentes, qui peut conduire à la perte de certains messages.

Cette implémentation est disponible gratuitement depuis Juin 1997 [59] ; outre le code source, elle contient un manuel de référence et une présentation informelle de la programmation parallèle et répartie dans le join-calcul, avec de nombreux exemples de programmes.

D'autres implémentations sont disponibles : La première implémentation du join-calcul est sans doute celle de Selinger [138]. Son compilateur accepte en entrée un processus du join-calcul, et génère le code C qui lui-même exécute ce processus localement. En outre, Padovani a écrit une implémentation très complète du join-calcul distribué en C [113]. Il utilise le système PVM pour implémenter la communication globale.

Enfin, Le Fessant implémente actuellement le join-calcul distribué de manière plus complète et plus intégrée au langage Objective Caml [86]. Contrairement à notre prototype qui préservait la distinction entre Objective Caml et le join-calcul distribué tout en proposant des passerelles entre les deux langages, cette nouvelle implémentation intègre les deux langages en un seul qui offre toutes les fonctionnalités de Objective Caml avec les primitives pour le parallélisme et la programmation répartie proposées dans le join-calcul distribué. Le support à l'exécution est nettement plus performant ; il intègre par exemple un glaneur de cellules réparti [88].

La sécurité et la programmation répartie Le problème de l'implémentation répartie devient particulièrement sensible lorsque l'on considère l'ensemble du réseau et des autres machines comme potentiellement hostiles, et non plus seulement faillibles.

De ce point de vue, le join-calcul est adéquat pour étudier certains aspects de ce problème. Plus précisément, son modèle de communication peut être implémenté de manière répartie avec de fortes garanties de sécurité, même en présence d'un attaquant sur le réseau (au prix d'un encodage complexe et coûteux).

En collaboration avec Abadi et Gonthier, nous avons proposé un schéma d'implémentation sécuritaire en compilant formellement tout processus écrit dans le join-calcul dans un langage qui étend le join-calcul avec des primitives de cryptographie à clé publique. L'image de notre traduction utilise uniquement des protocoles cryptographiques qui communiquent sur un réseau public pour faire parvenir les messages du join-calcul de l'émetteur à la définition. Ainsi, la sécurité est garantie quelle que soit la répartition des processus et des définitions à l'exécution.

Techniquement, nous établissons des résultats de complétude entre le join-calcul et sa traduction cryptographique. Pour cela, nous utilisons les équivalences et techniques de preuves développées dans cette thèse.

Conclusion

Nous avons présenté un formalisme adapté à la programmation répartie. Bien que notre motivation initiale ait été d'obtenir un formalisme effectivement implémentable, notre modèle se révèle utile pour la programmation parallèle en général, comme une extension naturelle des langages fonctionnels. Ainsi, la définition locale, statique de tous les récepteurs pour un ensemble de noms définis permet de transférer de nombreuses techniques standards comme le typage polymorphe, ou l'utilisation de fermetures et de piles dans l'implémentation.

L'inconvénient évident d'un nouveau formalisme est qu'il faut re-développer toute une méta-théorie avant de pouvoir pleinement l'utiliser. Dans notre cas, nous pensions pouvoir transférer la plupart des résultats du pi-calcul vers le join-calcul. A posteriori, les équivalences et résultats techniques souhaitables pour le join-calcul n'étaient pas réellement disponibles dans le pi-calcul. Au contraire, certaines techniques développées pour le join-calcul fournissent des résultats nouveaux qui devraient s'appliquer à la plupart des calculs de processus.

Au cours de ce travail, d'autres personnes ont développé diverses variantes asynchrones du pi-calcul dans le même esprit, avec en particulier une attention accrue envers les problèmes de programmation et d'implémentation. Nous percevons une certaine convergence de vue avec notre approche. Bien sûr, il serait préférable de disposer d'outils communs qui puissent s'appliquer à ces diverses variantes, mais la recherche d'un formalisme unifié est particulièrement ardue. Ces approches explorent également d'autres aspects de la programmation des réseaux, comme le problème du routage dynamique.

L'implémentation du join-calcul s'est avéré délicate, et il serait intéressant d'étudier plus en détail certains aspects pratiques, en raffinant notre sémantique opérationnelle, par exemple pour y faire apparaître certaines structures de données locales, tels que les piles, les automates, les fermetures, ou le code lui-même. Des analyses statiques plus sophistiquées devraient permettre une amélioration sensible des performances, en détectant à la compilation certains schémas de communication dans les filtres. Enfin, une meilleure intégration de la programmation orientée-objet permettrait, entre autres, une comparaison avec les systèmes à objets répartis qui constituent de fait la majeure partie des langages répartis.

Le choix des primitives pour la programmation explicitement répartie, en particulier pour la détection des pannes, doit encore être validé par la pratique. Par exemple, la détection des pannes ne va pas de soi, mais l'utilisation de protocoles temporisés semble prometteuse. Formellement, l'analyse de programmes distribués en présence de pannes est délicate ; bien que les équivalences opérationnelles soient en place, des techniques de preuve adaptées au problème font encore défaut. C'est un enjeu important pour aborder ce problème difficile.

De manière plus générale, l'application de techniques sémantiques maintenant bien connues pour la programmation séquentielle est encore largement inexplorée dans le cas de la programmation de systèmes répartis. Notre formalisme fournit un modèle élémentaire qui, nous l'espérons, favorisera cette exploration.

Main Notations

In this dissertation, we use the following standard conventions:

Tuples: $\widetilde{v}$ is the tuple $v_1, v_2, \ldots, v_n$ for some integer $n \geq 0$; when the notation occurs several times in the same formula, we assume a different choice of n for each argument of the notation.

Multisets: We use the same notations for both sets and multisets—the context should prevent any ambiguity. We write $\{\widetilde{x}\}$ for the set or the multiset that contains the $\widetilde{x}$'s, and $\{x \mid p(x)\}$ to denote the set or the multiset that contains all the x's such that the predicate $p(x)$ holds. We use the operators $\cap$, $\cup$ and, $\uplus$ to note intersection, union, and disjoint union of sets and multisets, respectively.

Relations and predicates: A binary relation $\mathcal{R}$ between the sets S_1 and S_2 is a subset of $S_1 \times S_2$. Most of our relations will range over the same sets. We usually adopt an infix notation for all relations. We let the variables $\mathcal{R}, \mathcal{R}', \varphi$ and symbols of equality and equivalences range over relations. We write Id for the identity relation.

Let $\mathcal{R}$ and $\mathcal{R}'$ be two relations. When defined, we write $\mathcal{R}\mathcal{R}'$ for the composition of relations $\{(x, y) \mid \exists z, x \mathrel{\mathcal{R}} z \mathrel{\mathcal{R}'} y\}$, $\mathcal{R}^{-1}$ for the converse relation $\{(y, x) \mid x \mathrel{\mathcal{R}} y\}$, $\mathcal{R}^n$ for the repeated relation inductively defined by $\mathcal{R}^0 = \mathit{Id}$ and $\mathcal{R}^{n+1} = \mathcal{R}\mathcal{R}^n$, $\mathcal{R}^=$ for the reflexive closure $\mathit{Id} \cup \mathcal{R}$, $\mathcal{R}^+$ for the transitive closure $\bigcup_{n \geq 1} \mathcal{R}^n$, and $\mathcal{R}^*$ for the reflexive-transitive closure $\bigcup_{n \geq 0} \mathcal{R}^n$.

A preorder is a transitive reflexive relation; an equivalence is a symmetric transitive reflexive relation. A relation refines another relation when it is included in it.

Every relation $\mathcal{R}$ defines a predicate, also written $\mathcal{R}$, defined as $x \mathrel{\mathcal{R}}$ iff $\exists y \mid x \mathrel{\mathcal{R}} y$. For every predicate T, the predicate $\not{T}$ is its negation.

Strings: A string is a finite sequence of elements from a base set. We let the variables φ, σ range over strings. $\varphi\sigma$ is the concatenation of the strings φ and σ, and ϵ is the empty string.

Variables and Substitutions: We use variables to denote names and provide scoping rules that determine when a variable is bound. Bound variables can be substituted for any variable that does not appear in its scope; we call such internal substitutions α-conversion. A variable is *fresh* with regards to a collection of terms when it does not appear in any of these terms.

We use a postfix notation for substitutions. We write $\{{}^{y_1}\!/\!_{x_1}, \ldots, {}^{y_n}\!/\!_{x_n}\}$ or simply $\{{}^{\widetilde{y}}\!/\!_{\widetilde{x}}\}$ for the substitution that simultaneously replaces every occurence of a variable x_i by the variable y_i, and write σ for an arbitrary substitution. We assume implicit α-conversion on bound variables before substitution to avoid name clashes.

Chapter 1

Introduction

Network-based applications have become pervasive over the last few years, and various avant-garde languages have advocated the use of new constructs for expressing distributed computation. However, little is known about the formal foundations of such computations, at least from a programming language viewpoint. This is in sharp contrast with the situation for sequential programming and concurrent programming, where high-level programming languages can be given formal foundations using small calculi such as the λ-calculus or CCS and the π-calculus, and where these calculi can be used in turn to state and prove correctness properties, both for the programs and for the implementations of these languages.

More precisely, there is a gap between formal models of concurrency and languages for programming distributed and mobile systems. In concurrency theory, process calculi such as CCS or the π-calculus [97, 100] introduce a small number of constructs, and have a thoroughly studied meta-theory. However, they are mostly based on atomic non-local interaction—typically *rendez-vous*—which is difficult to implement fully in a distributed setting, and which makes formal results hard to interpret.

In contrast, programming languages such as Actors [12], Telescript [149], Obliq [44] have separate primitives for transmission and synchronization that directly reflect the implementation mechanisms—such as remote procedure calls and semaphores. However, they also have a much larger set of constructs, usually including imperative primitives, and this hinders their formal investigation. Overall, distributed programming is still more an art than a science; it requires skills in system programming and a working knowledge of the subtleties of each given distributed architecture.

We propose an elementary model of distributed programming. As an attempt to bridge the gap between concurrency theory and distributed programming, we introduce a model of concurrency named the *join-calculus*, use this model as the foundation of a practical programming language, and study this model formally as a process calculus.

The join-calculus is a small calculus in which every computation consists solely of asynchronous message-passing communication. It retains the style of process calculi; in particular, it relies on the elegant technique for managing the scopes of names developed for the π-calculus, and it inherits numerous notions of equivalences for relating the behavior of programs, along with efficient proof techniques. The join-calculus also has some useful properties for the programmer. By construction, the basic communication steps of the calculus are restricted to those that are straightforward to implement in a distributed setting, *independently of the distribution of the computation at run-time.* Technically, this is reflected as a "locality property" that rules out any synchronization between components that may not be located at the same machine.

Independently of distributed implementation issues, locality also induces properties that are appropriate for concurrent programming. It promotes a declarative programming style for synchronization, and makes the static analyses of programs simpler. For example, the join-calculus can be given an implicit type system that combines both polymorphism and type inference. Also, it yields a direct relation with functional programming. For instance, languages like core-ML can be naturally embedded in the join-calculus—which suggests that many techniques and results developed for these languages directly apply to the join-calculus.

In this dissertation we present the join-calculus, give a formal account of its model of concurrency, and explore refined models of distributed programming based on the join-calculus. We give the operational semantics of the join-calculus and of its refinements in terms of *chemical abstract machines*, which yields an effective model of our distributed implementation in a simple setting. We explain how the join-calculus can be used as the basis of a high-level programming language with functional and object-oriented features. The more practical aspects of the join-calculus, as the core of a programming language for distributed systems, are explored in our prototype implementation [59].

As regards semantics, we define a notion of *observation* that naturally captures the semantics of subcomponents in a distributed program, but also raises numerous technical issues. We consider what should be the "right" notion of equivalence based on these observations, and develop the corresponding proof techniques for the join-calculus. Our notions of equivalence are hardly new; they have been proposed and studied in other settings. However, some new results obtained for the join-calculus carry over to more traditional calculi. In particular, some of the results in Chapters 4, 5, and 6 also hold in the π-calculus, where they answer some long-standing questions about equivalence relations.

We complete our study by proposing extensions of the join-calculus that give a more explicit account of distributed programming, with agent-based mobility and a simple model of failure. We refer to the refined calculus as the *distributed join-calculus.* At this point, we make further assumptions about distributed programming in the resulting language.

The extended language provides *network transparency.* In the absence of failure, for instance, the outcome of a program does not depend on the localization of resources. Channel names in the language have a global lexical scope, meaning that a message sent on a given name has the same meaning, independently of its emitter. In some cases the message remains local to the machine of the emitter, in some other cases it triggers some lower-level communication on the network, but anyway the message eventually arrives at the same place. At the same time, the language provides *network awareness.* The localization of any resource is entirely determined by the programmer, who may dynamically move groups of resources from one machine to another. Hence, it is possible when necessary to forecast the effective distribution of resources at runtime, and to ensure that parts of the computation are located at the same machine, which may be useful to achieve better performance, or to control the risks induced by the failure of some machine that takes part to the computation.

1.1 Structure of the dissertation

In Chapter 2 we begin our work by a review of two well-known models of concurrency, namely name-passing process calculi such as the π-calculus of Milner, Parrow, and Walker, and the Chemical Abstract Machine of Berry and Boudol. We recall how concurrent computation can be simply reflected as series of basic steps that represent communication, and we show that this model is not adequate for distributed programming: it turns out that some of these basic steps cannot be effectively implemented on top of an asynchronous network. This makes unclear the connection between the abstract model and its implementation as a programming language. We analyze this mismatch, and introduce a few changes that lead to our solution.

Accordingly, we define the join-calculus, a new process calculus that provides built-in locality, in a setting that retains the style and the formal simplicity of process calculi, but that can be fully implemented in a distributed manner. We give a number of small examples that illustrate the expressiveness of the calculus. We discuss the behavior of our example processes in an informal manner. Many ideas introduced in the examples are developed more formally later in the dissertation.

In Chapter 3, we elaborate on the join-calculus, and show how this calculus can be turned into a high-level programming language. We equip the join-calculus with a practical type system—parametric polymorphism—and establish the standard properties of this type system. Also, we analyze the relation between functions and join-calculus processes, and reinterpret the join-calculus as an extension of functional programming with *fork* and *join* operators for concurrency and synchronization. In particular, the definition of functions can be translated into the definition of communication channels in a type-preserving manner. We conclude the chapter by a discussion of object-oriented features.

The next two chapters lay the technical foundations of the join-calculus by studying a series of equivalences relations on processes, and by placing these relations in a hierarchy of equivalences, according to their discriminative power and their ease of proof. In Chapter 4, we mostly focus on reduction-based equivalences. We generate a family of standard equivalences from a few natural properties (congruence, basic observation of messages, abstraction over internal computations, simulation diagrams), discuss their relative advantages, and establish that many of these definitions yield the same equivalences. Some of these results are rather surprising, and also apply to the π-calculus, where they close conjectures by Milner and Sangiorgi [101] and Honda and Yoshida [73]. The main proof involves the construction of "universal contexts". Besides, we also develop convenient bisimulation proof techniques based on general confluence theorems.

In Chapter 5 we give a more extensional account of the join-calculus. We study labeled-based, purely co-inductive proof techniques developed on top of an auxiliary semantics for the join-calculus that makes interaction with the environment explicit, instead of relying on potential reductions in a context. We place the resulting equivalence relations at the top of our hierarchy, assess the formal impact of name-comparison in the calculus, and begin a technical comparison with the π-calculus.

In Chapter 6, we apply the equivalences and the proof techniques developed so far to conduct an analysis of several variants of the join-calculus. We provide a series of fully abstract internal encodings. We also complete our comparison with the π-calculus by exhibiting cross-encodings between the two formalisms and studying their properties.

In Chapter 7 we use the join-calculus as a formal basis to explore several refined models of distributed programming that provide explicit control over the localization of processes and resources. We group these resources in named "locations" that constitute the units of mobility and of failure, and we give their semantics in an extended chemical abstract machine style. We also provide a number of examples of typical distributed programs, and discuss the properties of the distributed join-calculus in the presence of partial failures.

Dependencies The whole thesis uses the definition of the join-calculus given in Chapter 2 (Sections 2.3, 2.4, and partly 2.5.5). Chapters 4 and 5 are initially independent, but really present two sides of the same problem. Chapter 6 strongly relies on these two chapters. Some lemmas rely on the type system developed in Chapter 3. Conversely, most of Chapter 7 can be read independently of Chapters 4–6. The technical comparison with other process calculi assumes a working knowledge of CCS and of the π-calculus.

1.2 Related work on the join-calculus

This dissertation contains revised material from works on the join-calculus in collaboration with other authors. Here is a list of this material in chronological order, with references to the related chapters.

1. *The reflexive chemical abstract machine and the join-calculus*, by Cédric Fournet and Georges Gonthier [55]. The paper contains the original definition of the join-calculus (Chapter 2), a preliminary discussion of functions and objects (Chapter 3), several results on internal encodings, and translations between the join-calculus and the π-calculus (Chapter 6).

2. *A Calculus of Mobile Agents*, by Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget and Didier Rémy [57]. The paper introduces an extended join-calculus with explicit localization of resources and agent-based mobility. It contains numerous examples of mobile agents, and a technical discussion of failure recovery (Chapter 7).

3. *Implicit Typing à la ML for the join-calculus*, by Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy [58]. The paper carries over parametric polymorphism to the join-calculus, and provides a translation of typed functions into typed processes (Chapter 3).

4. *Bisimulations in the Join-Calculus*, by Michele Boreale, Cédric Fournet, and Cosimo Laneve [35]. The paper proposes auxiliary, "open" chemical semantics for the join-calculus, and uses these semantics to define labeled bisimulation equivalences (Chapter 5).

5. *A Hierarchy of Equivalences for Asynchronous Calculi (extended abstract)*, by Cédric Fournet and Georges Gonthier [56]. In a π-calculus setting, the paper

organizes numerous equivalences in a hierarchy according to their discriminative power, and establishes that some definitions yield the same equivalences; in first approximation, the same hierarchy applies to the join-calculus, where it was initially studied (Chapters 4–5).

In addition, two applications of the join-calculus are left outside this dissertation, even as they represent a significant part of the work devoted to the thesis:

Distributed implementations In collaboration with Luc Maranget, we have developed a prototype implementation of a distributed language based on the join-calculus [59].

The system contains a compiler and an interpreter written in Objective-CAML. The language is the distributed join-calculus, with parametric polymorphic types, simple modules, the standard libraries of Objective-CAML, a few specific libraries for distributed programming, and a generic interface to incorporate other Objective-CAML modules. Le Fessant and Maranget give a detailed analysis of the compilation strategies for join-patterns in [87].

The language fully supports our model of distributed and mobile computations. Numerous machines connected through the Internet network may take part to the computation and execute some of its code. The overall performance was not a primary issue, and only partial support for failure-detection is available so far.

The distribution includes the source in Objective-CAML and in join language, a reference manual, tutorials for both concurrent and distributed programming, and a suite of examples.

There are several other implementations of the join-calculus:

- The first implementation of the join-calculus is due to Peter Selinger; the compiler supports the core calculus and generates C code [138].
- Luca Padovani developed a full-fledged implementation of the distributed join-calculus in C; it relies on the PVM system for distributed execution [113].
- Last but not least, Fabrice Le Fessant is conducting a second implementation at INRIA [86]. As opposed to our prototype implementation, the system is based on the standard Objective-CAML distribution supplemented with runtime support for distributed execution—especially a distributed garbage-collector [88]—and all the concurrent and distributed features of the join-calculus. The language is an extension of Objective-CAML with join-calculus constructs (join-patterns, definitions, locations). Programs previously written in ML are supported after recompilation.

Application to security The join-calculus is adequate for modeling some significant security aspects of distributed computation. In particular, its model of communication can be implemented in a distributed manner with strong security guarantees, even if the network is subject to deliberate attacks.

We have conducted a study of the secure implementation of the join-calculus by formally translating any join-calculus program into a language that extends the join-calculus with public-key cryptography, in the spirit of the spi-calculus of Abadi and

Gordon [3, 4, 5]. The image of the translation does not rely on abstract channels. Instead, it involves communication only through a public network. Low-level contexts can express many sophisticated attacks, but several full abstraction results relate such attacks to simpler equivalence properties in the source join-calculus [2].

Chapter 2

The Join-Calculus

This chapter introduces our basic model of concurrency. The model consists of a name-passing process calculus—the join calculus—equipped with an operational semantics—the reflexive chemical abstract machine (RCHAM).

We first discuss the reasons that lead us to this new model, starting from existing formalisms. There already exist numerous models of concurrency, and the join-calculus draws upon some of these models. Nonetheless, these models suffer from serious flaws when applied to distributed programming, which motivates our attempt to design a more adequate formalism.

Process calculi were initially proposed to study the specification of concurrent systems; this underlying idea influenced the design of well-known calculi such as the *Calculus of Communicating Systems* (CCS) of Milner. Most subsequent process calculi have retained this bias toward specification. In contrast, our goal in this dissertation is to study concurrent distributed programming, which raises specific issues. Foremost, we are concerned with the implementation of the model, as the core of a practical programming language. Ideally, the model should be much simpler than the implementation, but should not hide the essential phenomena.

The join-calculus draws most heavily upon two well-known models of concurrency: process calculi—exemplified by CCS and numerous variants of the π-calculus—and chemical abstract machines. These two models are not entirely unrelated, since several recent variants of the π-calculus are equipped with chemical semantics. Our work is more directly related to the recent "asynchronous" trend of the π-calculus [71, 70, 37], and its application to concurrent programming languages [121, 119].

We start our dissertation with a survey of these two models. We recall their main features, then discuss their application to distributed programming. This leads us to an alternate proposal, and to its discussion. Next, we define the join-calculus and its semantics. We give the grammar for processes and the scopes for their names, then we present the basic mechanisms of the RCHAM. Later on, the RCHAM is refined in Chapters 3, 5, and 7 according to our needs. We also illustrate our model by a series of examples to get acquainted with the calculus, and mention a few technical properties of the RCHAM. We finally relate our presentation of the join-calculus to the traditional process calculus framework, and discuss a few other models of concurrency. The comparison with models specifically designed for distributed programming is deferred until Chapter 7.

2.1 The channel abstraction

In a concurrent setting, several entities (usually named *agents*, or *processes*) are placed in parallel and can interact with one another. Conceptually, the global computation consists entirely of these interactions between agents, and thus the definition of a model essentially reduces to the definition of its basic communication mechanism.

The communication of information between agents is naturally modeled as the transmission of messages from one agent—the emitter—to another—the receiver. In practice, there is usually a third entity involved—the medium—where messages reside while in transit. Besides, there are many different ways to specify and to implement the medium: as a shared memory, as buffers, as the interface to some low-level networking library, ... with specificities for each implementation.

In his *Calculus of Communicating Systems* (CCS), Milner proposes that every basic communication occur *in the ether*, an idealized medium that hides the choices and the limitations of the implementation: "an ether is just that which contains an unordered set of messages and enables them to move from source to destination" [97]. As a result, CCS describes communication in a simple, abstract, and uniform manner. Agents are able to synchronize by *handshake communication*, or *rendez-vous*, that atomically affect both the emitter and the receiver. Handshake communication intuitively occurs between entities that are immediate neighbors, hence Milner also suggests that, in the case more complex media are used, these media should be explicitly represented as intermediate agents, and that complex communication should be split into several simpler stages, for instance basic interaction between the emitter and the medium, then between the medium and the receiver.

Starting from CCS and CSP [69], most process calculi also provide some structure to communication by using names that represent communication capabilities. These names are often called *port names*, or *communication channels*. In a sense, they are abstractions of the communication media on which data is exchanged. In CCS, for instance, every process can attempt a rendez-vous at a given channel, and communication may occur whenever two processes attempt to communicate over the same channel. Send and receive operations on channels provide concise abstractions for the transmission, routing, and synchronization that occur in a concurrent system.

In combination, communication channels in the ether yield a simple and expressive model, which is adequate for specification purposes. In practice CCS has been successfully applied for modeling various existing protocols and for reasoning about their formal properties. In such protocols, ethereal communication is still meant to occur locally, which usually happens to be the case because these protocols have already been implemented, or designed with an idea of their implementation. Hence, their abstractions as processes typically make a sensible usage of communication channels. Indeed, every channel might in theory be used by an arbitrary number of processes attempting to communicate at the same time, but in almost all practical examples found in CCS there are just two or three processes involved.

Elaborating on CCS, process calculi can model any kind of computation as basic communications on channels. This is best achieved with two successive improvements:

- Some basic values may be transmitted as part of the communication mechanism. This yields a variant of CCS closer to practice, which is called *value-passing CCS*.

- The π-calculus [100, 98] further improves on CCS by providing an elegant scope management technique for the channels, named *scope extrusion*. It can describe *mobile* systems, in which channels are values that can be exchanged on channels, thus dynamically establishing new communication capabilities between processes.

The additional expressive power of the π-calculus is useful to encode many other paradigms such as functions and objects, which makes the π-calculus a reference calculus for concurrency. Moreover, numerous variants of the π-calculus have been developed to address more specific issues in concurrency, and still retain the formal basis of the π-calculus.

Following our special interest in distributed programming, we focus on the asynchronous π-calculus, a variant introduced both in [37] and in [70] (where it is named the ν-calculus). Taking advantage of name mobility, the asynchronous π-calculus gets rid of several complicated features of the full π-calculus, such as arbitrary summation and recursion. What is more, it simplifies the communication mechanism so that *only the receiver notices that communication occurs.* Message-passing becomes asynchronous, in the sense that it does not require any form of handshake. The symmetry between emitters and receivers, already altered by the communication of values, is definitely broken.

As a first advantage, the resulting calculus is much easier to implement. Indeed, the PICT experiment [119, 122, 121] shows that the asynchronous fragment of the π-calculus can be used as the basis of a practical higher-order programming language, at least in a non-distributed setting.

Further, the asynchronous π-calculus can be used to model asynchronous protocols in a simpler setting. Protocols written in this calculus are closer to their actual implementation because they cannot rely on primitive synchronous communication.

Ethereal communication on asynchronous channels may also seem easy to implement in a distributed asynchronous manner, but this is not the case. Quoting Gérard Boudol in his presentation of the model [37],

> In the asynchronous π-calculus we propose, the communication media are just the (channel) names. However, we shall not give any particular status to these communication media. Instead, we shall regard the messages themselves as elementary agents, freely available for processes waiting for them.

This approach aims at simplicity, but it hides too much in a distributed setting. In a centralized implementation running on a single machine, pending messages may simply be stored in global tables attached to every channel. In a distributed implementation, however, it is hard to make all messages available to all machines, as it involves atomic operations on channels. In short, the asynchronous π-calculus removes the possibility of using synchronous operations in the calculus, but the implementation of an asynchronous channel still relies on synchronous operations.

In the case there are several distant receivers competing for a single message, for instance, these receivers atomically interact as one of them receives the message, thus preventing any other receiver getting this message. Since the successful receiver can communicate its success to all other receivers on some other channel, the initial step of communication yields a direct solution to the well-known problem of reaching a *global consensus* among several machines.

In a practical distributed system, however, the primitives found in system libraries to communicate from one machine to another are much lower-level than ethereal channels or global atomic interactions. Typically, the libraries provide unreliable message-passing from one machine to another. Message-passing is asynchronous, with explicit addressing toward a single receiver. For instance, the model of IP datagrams provides point-to-point communication on wide-area networks, together with some broadcasting on local networks. It uses a machine number plus a port number as static routing information to designate the receiving process.

While it may be tempting to forget about the implementation details of the network, the discrepancy between the asynchronous π-calculus as a programming language and its implementation induces serious problems, because in numerous situations these details must be revealed to the programmer. The distance between the model and its would-be implementation hides important issues, such as the cost of communication primitives, and the machines involved in a communication. Indeed, the cost of sending a high-level message should be related to the cost of a low-level message, rather than the cost of running a sophisticated distributed protocol as powerful as a global consensus in the worst cases. Likewise, basic communication between two machines should involve only the emitter and the receiver, and no other machine.

Unfortunately, the discrepancy becomes apparent in numerous practical situations. Besides overall efficiency, some machines or some parts or the network may be much slower than others, or may even fail. Also, important properties such as security or fault-tolerance crucially depend on the actual low-level messages exchanged by the implementation.

Let us develop our point on an example in a value-passing asynchronous CCS, with four machines labeled $1, 2, 3, 4$ that send and receive on a single channel x. This is modeled by the following processes:

$$\begin{array}{ccccccc} \overline{x}\langle 1\rangle & | & \overline{x}\langle 2\rangle & | & x(u).P & | & x(v).Q \\ \text{on machine 1} & & \text{on machine 2} & & \text{on machine 3} & & \text{on machine 4} \end{array}$$

where the parallel composition operator $|$ is meant to represent the frontiers between different machines connected through an asynchronous network, where the process $\overline{x}\langle 1\rangle$ emits the message 1 on x, and where the process $x(u).P$ receives a message on x then binds u to the contents of the message before running the process P.

According to the semantics of CCS, the specification of a channel guarantees that every message that is emitted is received at most once. Besides, minimal assumptions on fairness guarantee that communication on a channel eventually occur if there is at least an emitter and a receiver on this channel. In our example—and if we further assume that neither P or Q attempt to communicate on x—both machine 3 and machine 4 should get one message. Informally, we would also assume that, in the case a machine goes down, then its host process simply disappears. (Models of failures will be discussed in Chapter 7.)

As we try to sketch an implementation of this system, several questions arise. From the viewpoint of a local runtime in one of these machines, where shall we send messages that are locally emitted on x ? What shall we do when a local process attempts to receive a message on x ? Assuming machine 1 has no information about receivers on other machines, it may send its message to any machine across the network, for instance machine 2, in the hope there might be a receiver there. At this stage, both messages on x depend on machine 2, which may be very slow, or may fail, preventing any message from reaching a receiver. Once machine 1 has sent its message—and even if it detects that machine 2 has failed—it cannot deliver the message to any other machine: there is no simple way to determine whether machine 2 handled the message prior to its failure, hence if the message is re-sent by machine 1 the same message might be received twice on another machine. All of this is hidden in CCS, where either P or Q is triggered even if machine 2 is down—which is represented by deleting its local message $\overline{x}\langle 2\rangle$.

Ideally, each local runtime should forward messages on x only to the machines that have a pending receiver on x. In our virtual implementation, however, the appearance and disappearance of receivers on remote machines may only be signaled asynchronously to other machines. Hence, by the time the message arrives on the machine of its potential receiver, the communication offer may have disappeared, while another communication offer may have popped out on its original machine. Actually, we are still confronted with the issue of implementing synchronous communication in an asynchronous setting, and it seems that any symmetric implementation would introduce diverging computations, as some messages are endlessly forwarded from one machine to another. Note that acknowledgment messages after reception or reception requests from the receivers would not solve the problem, but only postpone it one step further. Multicast or broadcast are not especially useful either, because in the model one receiver can prevent a message being received anywhere else as it consumes the message, thus ensuring mutual exclusion.

In the initial spirit of CCS, we would explicitly model this situation by providing a local *proxy* channel on every machine, and by forwarding local messages from one proxy to another. At least, this approach explains that a proxy on a distant machine may interfere with any communication at a given channel; for instance, if machine 2 hosts a proxy on x, then it can receives the message from machine 1, then fail before forwarding it to another proxy on machine 3 or 4. This solution is more transparent, but still unsatisfactory; it is fairly complex to describe, and immediately introduces divergences in the computation, as messages are communicated from proxy to proxy.

As a last resort, we can use a *centralized implementation*. Intuitively, for every channel x there is somewhere an attached "channel manager" that maintains some state of the computation—for instance pending messages. The practical solution would be to implement every channel on a single machine. This would solve our problems of symmetry, and in particular of diverging computations, but this is entirely centralized, and still does not explain how messages are routed from one emitter to one receiver. Also, communication on x now obviously depends on the machine that hosts the channel manager. This mechanism is hidden in the model, which tells nothing about the creation, the interface, or even the machine where our hypothetical channel manager is located.

In short, asynchronous channels still provide some form of atomic interaction between distant emitters and receivers, hence a faithful implementation in an asynchronous setting becomes problematic. Although the asynchronous π-calculus can be used to write and reason about abstractions of asynchronous protocols, asynchrony à la π-calculus is still too complex for a distributed implementation. This is not the right notion of asynchrony for distributed programming.

In order to capture distributed programming in a simple process calculus, we need a communication mechanism that can be trivially implemented using only a few low-level asynchronous datagrams.

2.2 Chemical machines

Our second reference model is the chemical abstract machine (CHAM), which can be regarded in particular as a computational model of the π-calculus.

Despite their names, abstract machines usually provide a rather detailed model of computation. The machine is abstract because it does not refer to a particular implementation, but its mechanisms and data structures are more explicit than in most operational semantics. Classical examples of such machines are Turing machines in complexity theory, or numerous machines for the λ-calculus and its variants. As regards concurrency, chemical machines have been recently used to model a large variety of situations.

We first present the chemical approach in general, then its formulation as an abstract machine for process calculi—the CHAM. Our new model—the reflexive CHAM—follows the general approach, but significantly departs from the CHAM.

2.2.1 The chemical metaphor

In [24, 25, 26], Banâtre et al. propose a new framework for concurrent distributed programming. Named *Gamma*, this framework is entirely based on *multiset transformation.*

The underlying intuition is to hide the mapping from each individual resource that participates to the computation to a particular machine. All resources are collected in a multiset, then concurrent rewriting is specified as local transformations on small parts of this multiset. With respect to other approaches based on imperative programming languages, the pervasive use of multisets removes artificial sequentiality, hides the physical distribution of resources, and naturally reflects the symmetries of the system, which ultimately makes concurrent programming simpler.

The operational model of [25] is defined by a generic multiset transformer, named the *Gamma function*, and parameterized by a few pairs of functions (R, A), where R is a predicate on multisets—the *reaction condition*—and A is a function on multisets—the *action*. The Gamma function is a partial function from states to sets of states; it can be recursively defined as follows:

$$\begin{array}{rcll} \Gamma(S) & = & \text{if} & \exists (R,A), M \mid M \subseteq S \text{ and } R(M), \\ & & \text{then} & \bigcup_{(R,A),M \mid M \subseteq S \text{ and } R(M)} \Gamma\big((S \backslash M) \cup A(M)\big) \\ & & \text{else} & \{S\} \end{array}$$

That is, the state of the computation is represented as a multiset of resources S, which is repeatedly mutated as long as some of its components match a reaction condition, and finally the resulting stable multisets are returned.

By analogy with elementary chemistry, we can interpret the model as a chemical experiment; resources in multisets become *molecules* floating in a *chemical solution*, or *chemical soup*, while reaction-action pairs become *chemical reactions* that consume specific molecules and produce some other molecules. As in chemistry, we are not directly interested in the individual Brownian motion of molecules in the solution; rather, we describe local reactions that involve a few molecules, and we expect that molecules in the solution will move around until they come into contact, match a reaction rule, and interact.

In this chemical setting, the *locality property* states that every rewriting that operates on a given multiset also operates on any larger multiset. Hence, there is no need to know about other molecules or other reactions when a reaction rule is triggered. Such locality is guaranteed by construction in Gamma, and suggests implementations on massively parallel architectures that can take advantage of this locality by performing numerous rewritings at the same time.

In most examples of Gamma programs, the computation is meant to terminate and to be confluent, but this is not part of the model. Besides, only a few kinds of chemical reactions are used in the programming practice; in [27], these basic patterns of reaction rules are named "tropes"; they perform uniform operations (e.g., filtering, or combination) on singleton or pairs of molecules present in solution.

Restating the metaphor in a distributed setting, each machine in the implementation can be considered as a *reaction site* that handles some or all of the chemical rules, and the implementation must organize the circulation of values from one reaction site to another. Of course, the chemical metaphor does not magically solve the problem of distributed implementation, but at least it clearly identifies the issue. For each reaction (R, A), the delicate part of the implementation searches for elements M in the multiset that match the reaction condition R, and gathers them on a machine that can performs the reaction; conversely, the subsequent computation of $A(M)$ for selected disjoint multisets M can easily be performed in parallel.

In general, the random motion hinted by the chemical metaphor is not very effective. According to Le Métayer, "because of the combinatorial explosion imposed by its semantics, it is difficult to reach a decent level of efficiency in any general implementation of the language." [95]. Indeed, Banâtre and Le Métayer argue that implementation concerns should be tackled only once the chemical specification of the program is written and verified. They provide several examples, and they propose efficient implementations tuned to these particular examples, and also to the particular topology of the interconnection network in their target distributed architecture. Such implementations mostly consist of manually mapping reaction rules to machines, and directing the flow of molecules from one machine to another.

There is another, more effective interpretation of the chemical machine: if *all the chemical rules operate on disjoint kinds of molecules*, then all molecules can travel to a unique reaction site associated with their kind, where they can be locally sorted, matched, and made to react. Figuratively, reactions are "catalyzed" at the reaction sites.

We finally remark that termination detection is not an essential feature of chemical semantics, even if a significant part of the implementation of Gamma actually deals with the distributed termination of chemical rewriting. As we shall see, there are alternative ways to observe the result of the computation, for instance by defining observation itself as chemical interaction. This is one of the enhancements of the chemical abstract machine which is described next.

2.2.2 The chemical abstract machine

In [29, 38], Berry and Boudol revisit the chemical metaphor in a more theoretical manner. They introduce the chemical abstract machine (CHAM) as a convenient device to write the operational semantics of concurrent systems.

As opposed to classical rewriting systems, concurrent systems do not operate on simple redexes; rather, they operate on several sub-terms that would constitute the redex, but that may appear almost anywhere in the term. Hence, the definition of reduction becomes unduly complicated in process calculi. Again, the chemical metaphor is adequate to get rid of the explicit bookkeeping of the associate-commutative structures induced by parallel composition. More generally, *structural rules* can be given that explain how redexes can be assembled from the molecules. Figuratively, these structural rules "overcome the rigidity of the syntax" [38]. As a result, the CHAM provides a clear and concise way to define reduction-based semantics as reductions modulo structural equivalence.

In contrast with the initial, labeled transition semantics of CCS and of the π-calculus, the CHAM focuses on the standard notion of reduction, that is, unlabeled transition steps. The resulting operational semantics is closer to that of sequential languages, and generally simpler. Thus, the CHAM is convenient for presenting and studying variations of a calculus, since most of the chemical mechanisms are common to most calculi. For instance, Milner adopts a minimal chemical semantics for comparing the π-calculus to the λ-calculus [98], and for introducing the polyadic π-calculus [99]. As an abstract machine, the CHAM is also useful to convey some of the implementation issues in concurrent languages, as for instance in FACILE [90]. This is a significant advantage over traditional SOS-style semantics, which lack a direct computational interpretation.

Since we adopt the chemical style of Berry and Boudol in this work, we recall their terminology and set our chemical notations. Chemical semantics use families of chemical rules that operate on multisets of molecules, also called chemical soups, or chemical solutions. The notion of chemical reaction is refined as follows:

- *Reaction rules* are local rewriting rules that describe the particular model. These rules are partitioned into structural rules and reduction rules.

  structural rules $\rightleftharpoons$ are reversible ($\rightharpoonup$ is *heating*, $\rightharpoondown$ is *cooling*). They represent the syntactical rearrangements of molecules in solution. Heating breaks molecules into smaller ones, cooling builds larger molecules from their components. Informally, structural rules do not correspond to an actual computation, but rather to some structural rearrangement.

  reduction rules $\longrightarrow$ consume some specific molecules in the soup, replacing them by some other molecules; they correspond to the basic computation steps.

  In the following, a generic reaction rule is denoted by the symbol $\rightleftharpoons\!\!\!\!\rightarrow$.

- *General laws* explain how local reaction rules apply within larger chemical solutions. They correspond to the context rules in SOS semantics. For instance, the *chemical law* says that every reaction rule applies to any larger multiset of molecules, extraneous molecules being left unchanged by the reaction. The original CHAM also features two other chemical laws, which are discussed later.

We illustrate the chemical approach on a tiny fragment of asynchronous value-passing CCS. We only present two rules:

$$\begin{array}{rrcl} \textsc{Str-join} & P \mid P' & \rightleftharpoons & P, P' \\ \textsc{Red} & \overline{x}\langle\widetilde{v}\rangle \mid x(\widetilde{y}).P & \longrightarrow & P\{\widetilde{{}^{v}\!/_{y}}\} \end{array}$$

The structural rule Str-join accounts for parallel composition; it defines two opposite relations $\rightharpoonup$ and $\rightharpoondown$. Paraphrasing the rule, any molecule of the form $P \mid P'$ can be heated in two smaller molecules P and P', and conversely any pair of floating molecules P and P' can be cooled down into a single compound molecule $P \mid P'$. The reaction rule Red consumes a single molecule that contains two processes glued by parallel composition that can interact at some channel x; the rule produces a single molecule that contains the receiving process after communication. For instance, we have the chemical steps

$$\begin{array}{rcl} \{\ \overline{x}\langle 1\rangle \mid P \mid x(u).Q\ \} & \rightharpoonup\rightharpoonup & \{\ \overline{x}\langle 1\rangle,\ \ P,\ \ x(u).Q\ \} \\ & \rightharpoondown & \{\ P,\ \ \overline{x}\langle 1\rangle | x(u).Q\ \} \\ & \longrightarrow & \{\ P,\ \ Q\{{}^{1}\!/_{u}\}\ \} \\ & \rightharpoondown & \{\ P \mid Q\{{}^{1}\!/_{u}\}\ \} \end{array}$$

where heating is used twice to dissolve the initial process into smaller components, cooling is used to form the redex $\overline{x}\langle 1\rangle | x(u).Q$, communication occurs within this molecule, and eventually cooling yields back a single process.

The cham also emphasizes the management of scopes and scope-extrusion by using additional chemical structure. A *membrane* is the frontier of an active sub-solution, considered as a single molecule in the enclosing soup. Thus, a chemical solution becomes a hierarchy of nested solutions. The membranes materialize the scopes of variables; they are porous, in the sense that controlled interaction can occur through a membrane. To retain locality in reduction rules, molecules exhibit an external communication capability by using an *airlock* mechanism that insulates some communication capabilities and make them available through the membrane in a reversible manner, as a prerequisite to communication in the enclosing solution. The management of membranes and airlocks is a generic attribute of the cham, and is expressed by two additional chemical laws.

While membranes and airlocks render molecules more complex, they offer a tight control on the scopes, and they are useful to define behavioral equivalences on chemical solutions. In this work, we prefer the "flat" structure of a single multiset found in Gamma, hence we drop membranes and airlocks in our presentation of the rcham. These extensions are not required in our setting, and they seem irrelevant from the implementor's point of view. This choice is discussed further in Section 2.5.5.

As regards implementation in a distributed setting, the cham inherits the properties of Gamma. On one hand, the cham conveys some intuition about implementation schemes and implementation costs, in particular for distinguishing between local and global operations. For instance, we can require that every chemical reduction occur on a single machine, and rely on structural rules to convey molecules from one machine to another. Then, we can discuss implementation issues in terms of the shape of the structural rules.

On the other hand, structural rules can specify complex, global rearrangements on the state of the computation, and too much "magical mixing" can make the CHAM very hard to implement, unless we precisely know how to direct chemical rewritings. More precisely, if we reinterpret our attempts to implement asynchronous CCS in a distributed manner using the chemical metaphor, we rediscover the problems of the last section as we try to implement the structural rules that operate on several machines, e.g., that transport a molecule from one machine to another. Unless we globally know the machines where redexes on a given channel can be assembled, we must let molecules float in solution from one machine to another. In particular, if a machine is very slow, or fails, some structural rearrangements become irreversible; they can be observed in the implementation, while in the CHAM such rearrangements should not have any computational content.

2.2.3 Ensuring locality, adding reflexion

Before actually presenting our model of distributed computation, we construct our model by applying some changes on existing models of concurrency, as motivated by our discussion of the implementation issue.

We try to retain the simplicity of name-passing process calculi and of the CHAM, but at the same time we want to ensure *built-in locality.* Pragmatically, *any* program that can be written in our model should be straightforward to implement in a distributed setting, by using only asynchronous communication primitives.

We first remark that the full power of channel-based communication à la CCS is fortunately not required in distributed programming. While channels provide a powerful abstraction, simpler communication schemes usually suffice in the programming practice. In that respect, the PICT experiment is enlightening [122]. While this language is purely founded on channel-based communication, typical channels only have a small number of emitters and receivers, and moreover these agents are often statically known. (For a given channel, agents that may communicate over that channel are named "dynamic" when the channel has been received, and "static" otherwise.) We classify the communication patterns that appear in PICT programs according to the complexity of the receivers:

1. The most common communication pattern features a single static replicated receiver, with a few static and dynamic emitters. This is sufficient to express any higher-order functional computation.

2. Also, numerous channels are used only once, with either a static receiver or a static emitter. This typically represents continuations (*cf.* [82]).

3. A few channels have at most one active static receiver at any time, and some static and dynamic emitters. This is useful to represent mutual exclusion, reference cells, and other forms of imperative control.

4. Some channels invert the communication patterns above, with for instance a single static emitter and numerous receivers. For example, this provides a natural encoding of multicast communication.

5. Finally, there are a few fully-dynamic usages of names. For example, a channel may directly implements unsorted buffers, or similar concurrent data structures.

In the few cases where genuine π-calculus channels are necessary, there are natural encodings that use simpler communication patterns with a few additional messages, so it seems reasonable to make these messages explicit. In all other cases, it is preferable to induce a style of programming that does not use general channels when simpler channels would do as well. This is already the spirit of PICT, where for instance the primitives for input-guarded sums are provided as a library, but not in the core language as usual in the "theoretical" asynchronous π-calculus.

More generally, full-fledged channel-based communication is seldom a primitive in practical programming languages. The trouble is that the underlying protocols are costly, and not much more useful. Other communication patterns, such as some forms of atomic broadcast, may be easier to implement and more useful for the programmer. Anyway, the communication model required in functional languages and in object-oriented programming languages [12, 44] should be much simpler, since remote call or remote method invocation are addressed to a single definition or a single object.

We adopt this restriction in the definition of the join-calculus, and therefore require that each name of the model be attached to a *single receiving agent*; moreover, we require that this agent be statically defined, at the time the name is created. One may wonder what is left of the expressive power of the asynchronous π-calculus. In fact, we do not loose anything as regards concurrency, and thus the join-calculus can be considered as a natural distributed implementation language for the asynchronous π-calculus. This issue is formally studied in Chapter 6.

Our design choice can be given a chemical interpretation, which we use to explain why the resulting model is easier to implement. If we apply the practical implementation strategy described above for a given Gamma program to the CHAM, each rule is allocated on a few machines, and the flow of molecules from one machine to another is directed according to the rules that can affect these molecules. Inasmuch as each rule is centralized on a single machine, and each molecule can be consumed by a single rule, the distributed part of the implementation becomes straightforward: we only have to route each molecule toward its dedicated machine. Under this interpretation, however, the CHAM is not very concurrent: communication is centralized in a fixed set of chemical sites. Besides, the management of each site is still arbitrarily complex and centralized.

To recover fine-grain parallelism, it would be much better to have a larger number of simpler reaction rules. This is exactly what the modifications of the reflexive CHAM bring in, by allowing the dynamic creation of reaction rules, and constraining each rule to express a fixed synchronization pattern.

In summary, the reflexive CHAM model is obtained from the generic CHAM by imposing locality and adding reflexion:

- *locality* means that every message can be consumed by at most one group of statically-defined chemical rules, that supply only minimal synchronization services between pending messages.

- *reflexion* is added by letting reactions extend a machine with new names along with their exclusive reaction rules; this lets our model be computationally complete.

For the time being, distribution and locality are kept mostly implicit, as we use a simple, flat multiset structure to represent the state of the RCHAM. At any point of the computation, however, we can obtain an effective distributed implementation, simply by partitioning processes and reaction rules between the available machines. In Chapters 7 we refine the chemical structure to give a more explicit account of locality.

2.3 The reflexive chemical abstract machine

2.3.1 Overview

We now sketch the basic mechanisms of the reflexive CHAM; the formal definition is exposed in the next section. Our model operates on *higher-order solutions* $\mathcal{D} \vdash \mathcal{P}$ comprising two multisets. The molecules $\mathcal{P}$ represent the processes running in parallel; the reactions $\mathcal{D}$ define the active reaction rules.

Channel names can be used both as addresses to send messages and as message contents. In that sense, the join-calculus belongs to the family of process calculus with name mobility. We write $x\langle y\rangle$ to express that the name y is sent on the name x. Applying the chemical framework, an atom is a pending message $x\langle y\rangle$, and a compound molecule consists of several sub-molecules glued by the parallel operator "$|$". Molecules can be heated into smaller ones, in a reversible way.

As a first example, we consider an idealized print spooler. This spooler handles the print requests issued by workstations and drives the printers on the local network. We assume that the spooler interface consists of two ports *ready* and *job*: available printers like *laser* send their name on the port named *ready*, while users send the filenames *1*, *2* to be printed on the port named *job*. There are three atoms in solution on the first line below, versus one atom and one compound molecule on the second line, where the molecule joins the laser-printer and the file *1*. The structural equivalence $\rightleftharpoons^*$ relates these two solutions, without reactions yet.

$$\begin{array}{rll} & \vdash & \mathit{ready}\langle \mathit{laser}\rangle,\quad \mathit{job}\langle \mathit{1}\rangle,\quad \mathit{job}\langle \mathit{2}\rangle \\ \rightleftharpoons & \vdash & \mathit{ready}\langle \mathit{laser}\rangle|\mathit{job}\langle \mathit{1}\rangle,\qquad \mathit{job}\langle \mathit{2}\rangle \end{array}$$

Written D or $J \triangleright P$, a reaction consumes compound molecules that match a specific join pattern J, and produces new molecules in the solution that are copies of the process P where the formal parameters of J have been instantiated to the transmitted values. Continuing our example, we add a reaction that matches printers and jobs, then sends the filename to the printer.

$$D \quad \stackrel{\text{def}}{=} \quad \mathit{ready}\langle \mathit{printer}\rangle \mid \mathit{job}\langle \mathit{file}\rangle \triangleright \mathit{printer}\langle \mathit{file}\rangle$$

We now add this chemical reaction to the previous solution, so that it can be used to consume the compound molecule and generate a new atom. Notice that non-determinism is induced by $\rightleftharpoons$, which can select either $\mathit{job}\langle 1\rangle$ or $\mathit{job}\langle 2\rangle$—but is actually committed by the reduction.

$$\begin{array}{rrll} & D & \vdash & \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle \mathit{1}\rangle,\quad \mathit{job}\langle \mathit{2}\rangle \\ \longrightarrow & D & \vdash & \mathit{laser}\langle \mathit{1}\rangle,\qquad\qquad\quad \mathit{job}\langle \mathit{2}\rangle \end{array}$$

Our model is reflexive, meaning that reactions can be dynamically created. This is done by our last kind of molecule. The defining molecule $\mathsf{def}\ D\ \mathsf{in}\ P$ can be heated in two parts, a new reaction D and a molecule P. In this case, the newly defined ports can be used in both D and P. The solution we just considered may come from a single molecule, with the structural rules:

$$\begin{array}{rcl} & \vdash & \mathsf{def}\ D\ \mathsf{in}\ \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle \mathit{1}\rangle \mid \mathit{job}\langle \mathit{2}\rangle \\ \rightharpoonup \quad D & \vdash & \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle \mathit{1}\rangle \mid \mathit{job}\langle \mathit{2}\rangle \\ \rightharpoonup\rightharpoonup \quad D & \vdash & \mathit{ready}\langle \mathit{laser}\rangle, \quad \mathit{job}\langle \mathit{1}\rangle, \quad \mathit{job}\langle \mathit{2}\rangle \end{array}$$

A more realistic spooler would send the name *job* to its users, and the name *ready* to its printer drivers. This corresponds to the well-known scope-extrusion of the π-calculus. However, our definitions have a strict lexical discipline: the behavior of *ready* and *job* may not be deterministic, but it is statically defined. Other processes that receive these names may send messages, but they cannot add new reactions for these names. This restriction simplifies the study of join-calculus processes independently of their context, and will be most useful in technical developments.

2.3.2 Syntax and scopes

In this section, we define the syntax for the join-calculus, and we set the scopes for variables. Some of the choices are quite arbitrary; for instance we choose a calculus with polyadic messages, and with rather rich definitions; later in the dissertation we consider other variants and extensions; we discuss the importance of such choices in Chapter 6.

Names Our calculus is a name-passing calculus. We assume given a countable set of port names $\mathcal{N}$. We write name variables in lowercase letters x, y, *foo*, *bar*, ... to denote the elements of $\mathcal{N}$. We also write $\widetilde{x}$ for the tuple of names $x_1, x_2, \ldots, x_n \in \mathcal{N}$.

Names are the only values in the join-calculus; they obey lexical scoping and can be sent in messages. Following the usage for process calculi, we use the words "name", "variable", "port" and "channel" interchangeably. Later in the dissertation, we will introduce other values (location names, integers, booleans) and letters u, v, w will denote values in general.

Names can be used to form messages that carry some other names, such as for instance $x\langle u,v\rangle$. The arity of a message is the number of values being carried, here two. While this is not needed in the join-calculus, we usually assume that for any given name the number of arguments is the same in every message and in every join-pattern. For instance, when we write $x\langle\rangle \mid P$ and let P ranges over "any process of the join-calculus", we implicitly require that all messages on x emitted in P be also $x\langle\rangle$, and exclude processes that may send messages $x\langle u\rangle$ or $x\langle u,v\rangle$. This amounts to attach an arity to every name, and to consider only processes that have messages with consistent arities. We assume that there is an infinite number of names of each arity. We use the notation $\mathcal{N}_0$ for the subset of nullary names, that is, names in $\mathcal{N}$ with null arity.

To ensure this property, we could rely on a recursive sort discipline and consider only well-sorted processes, as for the π-calculus [99, 120]. Also, we can rely instead on the type system described in Chapter 3.

Processes and definitions The grammar for the join-calculus is inductively defined in Figure 2.1. It contains two categories of terms: processes and definitions.

A process P is the asynchronous emission of a polyadic message $x\langle\widetilde{v}\rangle$, the local definition of new names $\mathsf{def}\ D\ \mathsf{in}\ P$, the parallel composition of processes $P \mid Q$, or the null process $\mathsf{0}$. A definition D is a local rule $J \triangleright P$ that matches a given join-pattern J and associates a guarded processes P to it, the composition of definitions connected by the $\wedge$ operator, or the null definition $\top$. A join-pattern J is a parallel composition of formal messages $x\langle y_1, \dots, y_n\rangle$. We say that J is a n-way join pattern when it joins n messages.

We define the precedence of our operators. Processes and definitions of our abstract syntax are denoted as concrete terms, with parentheses when there is an ambiguity. To simplify the notation, we assume that the binary constructors $\mid$ and $\wedge$ are left-associative—anyway they are intended to be associative in the operational semantics—and that parallel composition binds tighter than local definition. For example,

$$\begin{array}{rl} & \mathsf{def}\ x\langle\rangle \triangleright \mathsf{0}\ \mathsf{in}\ y\langle x\rangle \mid z\langle\rangle \mid \mathsf{def}\ z\langle\rangle \triangleright \mathsf{0}\ \mathsf{in}\ x\langle\rangle \mid z\langle\rangle \\ \stackrel{\text{def}}{=} & \mathsf{def}\ x\langle\rangle \triangleright \mathsf{0}\ \mathsf{in}\ \Big(y\langle x\rangle \mid z\langle\rangle \mid \Big(\mathsf{def}\ z\langle\rangle \triangleright \mathsf{0}\ \mathsf{in}\ (x\langle\rangle \mid z\langle\rangle)\Big)\Big) \end{array}$$

We also make use of the following prefix notations, where the index variables may range over any finite set instead of $1 \dots n$.

$$\begin{array}{rcl} \prod_{i=1}^{n} P_i & \stackrel{\text{def}}{=} & P_1 \mid \cdots \mid P_n \\ \bigwedge_{i=1}^{n} J_i \triangleright P_i & \stackrel{\text{def}}{=} & J_1 \triangleright P_1 \wedge \cdots \wedge J_n \triangleright P_n \end{array}$$

Scopes and substitution Intuitively, local rules entirely define how messages are consumed by supplying join-patterns that trigger guarded processes. They can be considered as an extension of named functions with synchronization. For example, the name variables f and x in the process $\mathsf{def}\ f\langle x\rangle \triangleright P_1\ \mathsf{in}\ P_2$ obey similar lexical scoping rules than the variables f and x in the recursive let-binding $\mathsf{let}\ fx = e_1\ \mathsf{in}\ e_2$ found in most programming languages—here in a ML-like syntax. More generally, names that appear in a process P may be captured by an enclosing definition. The only binder is the join pattern $J = x\langle v_1, v_2, \dots, v_n\rangle \mid \dots$, but the scope of its names depends on their position in messages:

- The formal parameters $v_1, v_2, \dots, v_n$ that are received in J are bound in the corresponding guarded process. They must be pairwise distinct.

- The names x that are partially defined in J are recursively bound in the whole defining process $\mathsf{def}\ D\ \mathsf{in}\ P$, that is, in the main process P and recursively in every guarded process inside the definition D. They may appear at most once in every pattern.

Received variables $\mathsf{rv}[J]$, defined variables $\mathsf{dv}[J]$ and $\mathsf{dv}[D]$, and free variables $\mathsf{fv}[D]$ and $\mathsf{fv}[P]$ are specified by structural induction in Figure 2.2. The conditions on the occurrences of names in join-patterns enforces two restrictions: no received variable may appear twice in the same pattern J, which avoid the comparison on names and guarantees that join patterns remain linear; moreover, no pattern may join several

$$
\begin{array}{lllll}
P & ::= & & & \text{processes} \\
 & & x\langle v_1, \ldots, v_n\rangle & & \quad\text{message} \\
 & | & \mathsf{def}\ D\ \mathsf{in}\ P & & \quad\text{local definition} \\
 & | & P \mid P' & & \quad\text{parallel composition} \\
 & | & \mathsf{0} & & \quad\text{null process} \\
 \\
D & ::= & & & \text{definitions} \\
 & & J \triangleright P & & \quad\text{reaction rule} \\
 & | & D \wedge D' & & \quad\text{conjunction of definitions} \\
 & | & \mathsf{T} & & \quad\text{null definition} \\
 \\
J & ::= & & & \text{join-patterns} \\
 & & x\langle y_1, \ldots, y_n\rangle & & \quad\text{message pattern} \\
 & | & J \mid J' & & \quad\text{join of patterns}
\end{array}
$$

Figure 2.1: Syntax for the join-calculus

$$
\begin{array}{rcl}
\mathsf{fv}[x\langle v_1, \ldots, v_n\rangle] & \overset{\text{def}}{=} & \{x, v_1, \ldots, v_n\} \\
\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P] & \overset{\text{def}}{=} & (\mathsf{fv}[P] \cup \mathsf{fv}[D]) \setminus \mathsf{dv}[D] \\
\mathsf{fv}[P \mid P'] & \overset{\text{def}}{=} & \mathsf{fv}[P] \cup \mathsf{fv}[P'] \\
\mathsf{fv}[\mathsf{0}] & \overset{\text{def}}{=} & \emptyset \\
 \\
\mathsf{fv}[J \triangleright P] & \overset{\text{def}}{=} & \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J]) \\
\mathsf{fv}[D \wedge D'] & \overset{\text{def}}{=} & \mathsf{fv}[D] \cup \mathsf{fv}[D'] \\
\mathsf{fv}[\mathsf{T}] & \overset{\text{def}}{=} & \emptyset \\
 \\
\mathsf{dv}[J \triangleright P] & \overset{\text{def}}{=} & \mathsf{dv}[J] \\
\mathsf{dv}[D \wedge D'] & \overset{\text{def}}{=} & \mathsf{dv}[D] \cup \mathsf{dv}[D'] \\
\mathsf{dv}[\mathsf{T}] & \overset{\text{def}}{=} & \emptyset
\end{array}
$$

$$
\begin{array}{rclrcl}
\mathsf{dv}[x\langle y_1, \ldots, y_n\rangle] & \overset{\text{def}}{=} & \{x\} & \mathsf{dv}[J \mid J'] & \overset{\text{def}}{=} & \mathsf{dv}[J] \uplus \mathsf{dv}[J'] \\
\mathsf{rv}[x\langle y_1, \ldots, y_n\rangle] & \overset{\text{def}}{=} & \{y_1, \ldots, y_n\} & \mathsf{rv}[J \mid J'] & \overset{\text{def}}{=} & \mathsf{rv}[J] \uplus \mathsf{rv}[J']
\end{array}
$$

Figure 2.2: Scopes for the join-calculus

$$\begin{array}{llcl}
\textsc{Str-join} & \vdash P_1 \mid P_2 & \rightleftharpoons & \vdash P_1, P_2 \\
\textsc{Str-null} & \vdash \mathsf{0} & \rightleftharpoons & \vdash \\
\textsc{Str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash \\
\textsc{Str-nodef} & \mathsf{T} \vdash & \rightleftharpoons & \vdash \\
\textsc{Str-def} & \vdash \mathsf{def}\ D\ \mathsf{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}} \\
\textsc{Red} & J \triangleright P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\mathrm{rv}}
\end{array}$$

side conditions for the substitutions:

Str-def	σ_{dv} instantiates the port variables $\mathsf{dv}[D]$ to distinct, fresh names: $Dom(\sigma_{\mathrm{dv}}) \cap \mathsf{fv}[\mathcal{S}] = \emptyset$ where $\mathcal{S}$ is the initial solution;
Red	σ_{rv} substitutes the transmitted names for the distinct received variables $\mathsf{rv}[J]$.

Figure 2.3: The reflexive chemical machine (RCHAM)

messages on the same defined name, which is less important but simplifies some discussions; in effect, we can use α-conversion to ensure that no name appear twice in any join-pattern.

2.3.3 Operational semantics

Reflexive solutions The state of the computation is a *chemical soup*; it consists of two separate multisets of molecules $\mathcal{D}$ and $\mathcal{P}$, and is written $\mathcal{D} \vdash \mathcal{P}$. On the right-hand-side, $\mathcal{P}$ is a multiset of running processes; on the left-hand-side, $\mathcal{D}$ is a multiset of active rules. The active definitions in $\mathcal{D}$ are reaction rules that define the possible reductions of processes; the running processes in $\mathcal{P}$ represent the state of the computation. These processes interact only according to the reactions rules, but they can in turn introduce new names and reaction rules, which is emphasized by naming our machine "reflexive".

We use the letters $\mathcal{S}, \mathcal{T}$ to denote chemical solutions. We also extend our definitions of defined variables and free variables to solutions component-wise:

$$\begin{array}{lcl}
\mathsf{dv}[\mathcal{D} \vdash \mathcal{P}] & \overset{\text{def}}{=} & \bigcup_{D \in \mathcal{D}} \mathsf{fv}[D] \\
\mathsf{fv}[\mathcal{D} \vdash \mathcal{P}] & \overset{\text{def}}{=} & \left(\bigcup_{D \in \mathcal{D}} \mathsf{fv}[D] \cup \bigcup_{P \in \mathcal{P}} \mathsf{fv}[P]\right) \setminus \mathsf{dv}[\mathcal{D} \vdash \mathcal{P}]
\end{array}$$

Chemical rules for the RCHAM The semantics is given in Figure 2.3 as a collection of chemical rules that operate on fragments of reflexive solutions.

The first four structural rules state that $\mid$ and $\wedge$ are associative and commutative, with units $\mathsf{0}$ and T. They give processes and definitions their multiset structure. The structural rule Str-def provides *reflexion*, with a static scoping discipline: in the heating direction, a defining process can activate its local reaction rules, provided that fresh names are substituted for its defined variables. In the cooling direction, reaction rules can be folded on a process, as long as their defined names only appear in that process and in these rules.

The single reduction rule Red describes the use of active reaction rules $J \triangleright P$ that appear on the left-hand-side of the chemical solution. For every such reaction rule, the corresponding reduction step consumes a parallel composition of messages that matches J, and releases in the solution a copy of the associated guarded process, where the sent names are substituted for the received parameters. The reaction rule $J \triangleright P$ is left unchanged, and can be used later on to effect further reduction steps.

The context is implicit In our rules and in our examples, we mention only the elements of both multisets that participate in the rule, separated by commas, and we omit the parts of multisets in chemical solutions that are left unchanged by the effect of the presented rule and that do not condition it. That is, *the presentation of every chemical rule assumes an implicit context*, and every rule applies on any matching subpart of the soup. More explicitly, we also have a context rule:

$$\text{(Context)}\quad \frac{\mathcal{D}_1 \vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\rightarrow \mathcal{D}_2 \vdash \mathcal{P}_2 \qquad (\mathsf{fv}[\mathcal{D}] \cup \mathsf{fv}[\mathcal{P}]) \cap (\mathsf{dv}[\mathcal{D}_1]\backslash\mathsf{dv}[\mathcal{D}_2] \cup \mathsf{dv}[\mathcal{D}_2]\backslash\mathsf{dv}[\mathcal{D}_1]) = \emptyset}{\mathcal{D} \cup \mathcal{D}_1 \vdash \mathcal{P}_1 \cup \mathcal{P} \rightleftharpoons\!\!\!\rightarrow \mathcal{D} \cup \mathcal{D}_2 \vdash \mathcal{P}_2 \cup \mathcal{P}}$$

where $\rightleftharpoons\!\!\!\rightarrow$ ranges over all chemical rules. For instance, the verbose Str-def rule would be

$$\mathcal{D} \vdash \mathcal{P} \cup \{\mathsf{def}\ D\ \mathsf{in}\ P\} \ \rightleftharpoons\ \mathcal{D} \cup \{D\sigma_{\mathrm{dv}}\} \vdash \mathcal{P} \cup \{P\sigma_{\mathrm{dv}}\}$$

with the side condition $\sigma_{\mathrm{dv}} : \mathsf{dv}[D] \mapsto (\mathcal{N} \setminus \mathsf{fv}[\mathcal{P}] \setminus \mathsf{fv}[\mathcal{D}] \setminus \mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P])$.

Terms versus chemical soups In the following, we use both process notations and chemical notations interchangeably, according to our needs. As we do so, we rely on the obvious correspondence between a process (P) and the chemical solution that only contains this process ($\emptyset \vdash \{P\}$). Moreover, we define a notion of reduction on processes up to structural rearrangement as follows:

$$P \to Q \quad \stackrel{\mathrm{def}}{=} \quad \emptyset \vdash \{P\} \rightleftharpoons^{*} \longrightarrow \rightleftharpoons^{*} \emptyset \vdash \{Q\}$$

2.4 Examples

In order to get acquainted with the join-calculus, we present a series of simple terms with an intuitive description of their meaning. The formal treatment of observations is deferred until Chapter 4. Along the way, we introduce some processes and definitions and we define some derived constructs used later in the dissertation.

2.4.1 Some wiring

Our first example of definition does not perform any real computation, but merely relays messages from one name to another:

$$P \quad = \quad \mathsf{def}\ x\langle u\rangle \triangleright y\langle u\rangle\ \mathsf{in}\ Q$$

The name y is free in P, while the name x is bound. Operationally, the rule $x\langle u\rangle \triangleright y\langle u\rangle$ receives any message sent on x and sends a message with the same contents on y, and thus the name x behaves as a local relay for y; it can be used within Q instead of y for sending messages or as a value in messages. Independently of the context, the semantics of the join-calculus guarantees that every message emitted on x will be forwarded on y, which makes the names x and y synonyms. Elaborating on our simple relay, we can also multicast a message by forwarding it on several names:

$$\mathsf{def}\ x\langle u\rangle \triangleright x_1\langle u\rangle \mid x_2\langle u\rangle \mid \ldots \mid x_n\langle u\rangle\ \mathsf{in}\ Q$$

Every message $x\langle v\rangle$ emitted in Q yields n messages, meant to be received by n different agents in the enclosing computation. Likewise, we can model the routing of messages through series of relays and filters.

We can also multiplex several messages sent on a few names into larger messages sent on a single name, or split large messages into smaller components:

$$\begin{array}{rcl} P_1 & = & \mathsf{def}\ x_1\langle u\rangle \mid x_2\langle v\rangle \triangleright x\langle u,v\rangle\ \mathsf{in}\ Q_1 \\ P_2 & = & \mathsf{def}\ x\langle u,v\rangle \triangleright x_1\langle u\rangle \mid x_2\langle v\rangle\ \mathsf{in}\ Q_2 \end{array}$$

In P_1, local messages on x_1 and x_2 are jointly received, then their contents u and v are assembled into a single message $x\langle u,v\rangle$; in the case Q_1 emits on x_1 but not on x_2, the message is stuck. Conversely, in P_2 every message emitted on x yields two smaller messages. Note, however, that the two filters above are not exactly inverse. Especially, if we let $Q_1 = P_2$, then the process P_1 that successively applies both filters does not behave like Q_2. For instance if Q_2 emits two messages $x\langle 1,1\rangle$ and $x\langle 2,2\rangle$, then P_1 may either emit the same messages $x\langle 1,1\rangle$ and $x\langle 2,2\rangle$, or the shuffled messages $x\langle 2,1\rangle$ and $x\langle 1,2\rangle$.

2.4.2 Chemical inertness, units, and deadlocks

We now present some inert terms, that is, terms that have no visible effect on the chemical solution they are plunged in. Formally, we can define *chemical inertness* as follows: a process P is inert in the chemical solution $\mathcal{S} = \mathcal{D} \vdash \mathcal{P}$ when, for every series of chemical steps $\mathcal{D} \vdash P, \mathcal{P} \rightleftharpoons\!\!\!\rightarrow^* \mathcal{T}$, there is also a series of chemical steps $\mathcal{D} \vdash \mathcal{P} \rightleftharpoons\!\!\!\rightarrow^* \mathcal{D}' \vdash \mathcal{P}'$ such that $\mathcal{T} \rightleftharpoons^* \mathcal{D}' \vdash P, \mathcal{P}'$.

For instance, a process of the form $P = \mathsf{def}\ D\ \mathsf{in}\ \mathsf{0}$ is clearly inert for any choice of definition D: in solution, the only chemical rule that applies to P is STR-DEF, but the side condition on that rule ensures that all names defined in D are replaced by fresh names. Hence, no message on these names may ever be emitted, and these active rules remain entirely passive, except for reversible syntactic rearrangements using the rules STR-AND and STR-DEF. Informally, the process P represents a deadlock.

In fact, the units $\mathsf{0}$ and T are included in our grammar because they are convenient to reflect the multiset structure of definitions and processes. Nonetheless, they could also be considered as constructs derived from simple deadlocked terms. A good candidate for the null process $\mathsf{0}$ would be the process $\mathsf{def}\ x\langle\rangle \mid y\langle\rangle \triangleright x\langle\rangle\ \mathsf{in}\ x\langle\rangle$, which is inert in any chemical solution, for the same reasons as above. Likewise, a good candidate for the null definition T would be $x\langle\rangle \triangleright x\langle\rangle$ in a solution where the name x does not occur elsewhere.

We present another kind of inert processes; these processes never interact with the enclosing solution, but they can always perform internal reductions. Informally, these processes correspond to *livelocks*. For instance, a simple livelock process is $P \overset{\text{def}}{=} \mathsf{def}\ x\langle\rangle \triangleright x\langle\rangle\ \mathsf{in}\ x\langle\rangle$, which reduces to itself ($P \rightarrow P$), or $Q \overset{\text{def}}{=} \mathsf{def}\ x\langle\rangle \triangleright x\langle\rangle \mid x\langle\rangle\ \mathsf{in}\ x\langle\rangle$, which accumulates local messages on x.

2.4.3 Abstractions of processes

Messages in the join-calculus transmit only names, not processes; in that sense, the join-calculus is a first-order calculus. Nonetheless, there is a simple encoding of processes as names: instead of passing the process P, we pass a name κ defined by the simple reaction rule $\kappa\langle\rangle \triangleright P$. Later on, receivers may issue messages $\kappa\langle\rangle$, and each copy of the message will trigger a copy of the process P.

For instance, we define *process replication* as the derived construct:

$$\mathsf{repl}\ Q \quad \overset{\text{def}}{=} \quad \mathsf{def}\ \kappa\langle\rangle \triangleright Q \mid \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle$$

where κ is a fresh name ($\kappa \notin \mathsf{fv}[Q]$). Each time the message $\kappa\langle\rangle$ is received, the rule forks a new copy of the process Q and regenerates the message $\kappa\langle\rangle$. We thus have the expected behavior:

$$\mathsf{repl}\ Q \ \rightarrow\ Q \,|\, \mathsf{repl}\ Q \ \rightarrow\ Q \mid Q \,|\, \mathsf{repl}\ Q \ \rightarrow\ \cdots$$

Here, the process abstraction plays the role of the process variable X in the process $\mu X.(Q \mid X)$ that we could use instead to define $\mathsf{repl}\ Q$ if we had a fix-point operator μ in the syntax. In our setting, however, the reaction rules naturally account for replication and recursion, since they can be used any number of times.

Similarly, we can define process abstractions that take values as parameters, meant to bind the free variables of P. We consider for example the process

$$Q \quad = \quad \mathsf{def}\ \mathit{plug}\langle x\rangle \triangleright x\langle 1,2\rangle \mid x\langle 3,4\rangle\ \mathsf{in}\ \big(\mathsf{def}\ \kappa\langle u,v\rangle \triangleright P\ \mathsf{in}\ \mathit{plug}\langle\kappa\rangle\big)$$

with the simple behavior

$$\begin{array}{lll} Q & \rightarrow\rightarrow\rightarrow & P\{^1/_u,{}^2/_v\} \ \mid\ P\{^3/_u,{}^4/_v\} \\ & \mid & \big(\mathsf{def}\ \mathit{plug}\langle x\rangle \triangleright x\langle 1,2\rangle \mid x\langle 3,4\rangle\ \mathsf{in}\ 0\big) \,|\, \big(\mathsf{def}\ \kappa\langle u,v\rangle \triangleright P\ \mathsf{in}\ 0\big) \end{array}$$

The inner process $\mathsf{def}\ \kappa\langle u,v\rangle \triangleright P\ \mathsf{in}\ \mathit{plug}\langle\kappa\rangle$ "communicates" P parameterized on the names u and v, while the enclosing definition receives this abstraction on *plug*. The first reduction step consumes the message emitted on *plug*, which releases the two messages $\kappa\langle 1,2\rangle$ and $\kappa\langle 3,4\rangle$. The two steps that follow each consume one of these messages to trigger a copy of P after substituting actual values for the formal parameters u and v. Afterward, we can use structural rearrangement to restrict the scope of *plug* and κ to nothing, which emphasizes that the two definitions are now inert. Formally, these definitions can be discarded up to strong equivalence (*cf.* Chapters 4 and 5), while in practice they would be garbage-collected by our implementation.

The name κ represents the process P, and is called a *continuation* by analogy with functional programming [21]. In the examples above, the continuation could be applied any number of times by the receiver, and thus several instances of the process could be triggered. Every reduction that uses one of these rules is entirely deterministic: since there is no other way to consume messages sent on these names, these messages remain available until they are consumed. In particular, the same name can be sent in several messages to represent the same process, and there will be no interference between invocations. In case we need to ensure that at most one instance of P is triggered, as is usually the case with continuations, we can use the refined abstraction

$$\mathsf{def}\ \kappa\langle\widetilde{v}\rangle \mid \mathit{once}\langle\rangle \triangleright P\ \mathsf{in}\ \mathit{plug}\langle\kappa\rangle \mid \mathit{once}\langle\rangle$$

where *once* is a fresh name. Here, the *state* of the continuation is represented by the message $\mathit{once}\langle\rangle$, which indicates that the continuation has not been triggered yet. As the continuation is used, this message is consumed, and, since there is no occurrence of *once* in any other process, the definition and any further message sent on κ become inert.

We generalize the use of continuations to represent functions in Section 3.4. We also refer to [130] for a more formal encoding of higher-order processes into first order processes, in a π-calculus setting.

2.4.4 Channels of the π-calculus

Continuing our discussion on the need to represent centralized *channel managers* for each channel of the asynchronous π-calculus, we sketch the encoding of a single π-calculus channel z as a join-calculus definition. We use the rule

$$x\langle v\rangle \mid y\langle\kappa\rangle \triangleright \kappa\langle v\rangle$$

which is much like our print spooler presented in the overview. Reinterpreting this example, π-calculus emitters $\overline{z}\langle 1\rangle$ are rendered as messages $x\langle 1\rangle$, while π-calculus receivers $z(v).P$ are rendered as reception requests $y\langle\kappa\rangle$ carrying a continuation that stands for the receiving process P. That is, the two communication capabilities attached to the channel z are implemented as two distinct names x and y together with a channel manager.

This rule is the basis of our encoding of the π-calculus in the join-calculus. Especially, we retain name-mobility, as the encoding of a π-calculus channel z as a value can be transmitted as the pair of join-calculus names (x, y). We detail the encoding and we study its formal properties in Section 6.6.

2.4.5 Representing choice

As illustrated in previous examples, the semantics of the join-calculus is not deterministic. Specifically, non-determinism may stem from the definition of names, whenever this definition features several exclusive manners to consume the messages emitted on its defined names. Then, each reduction step commits a particular synchronization and exclude all synchronizations that were competing for the same messages. We show several forms of non-deterministic choice.

Internal choice is an operator commonly found in process calculi. It expresses that a process may choose between several alternatives independently of its context. It is often written $\oplus$. For instance, $P \oplus Q$ reduces to either P or Q. A simple encoding of this choice is

$$\mathsf{def}\ s\langle\rangle \triangleright P \wedge s\langle\rangle \triangleright Q\ \mathsf{in}\ s\langle\rangle$$

where s is a fresh name. The single message $s\langle\rangle$ may be consumed by either rule of the definition, which starts either P or Q. Later on, the definition of s is inert. Since internal choice often appears in the sequel, we define a derived operator for it. We let

$$\bigoplus_{i \in S} P_i \quad \overset{\text{def}}{=} \quad \mathsf{def} \bigwedge_{i \in S} \mathit{once}\langle\rangle \triangleright P_i \ \ \mathsf{in} \ \ \mathit{once}\langle\rangle$$

where S is a finite set and *once* is a fresh name, and we use the infix notation $P_1 \oplus \cdots \oplus P_n$ instead of $\bigoplus_{i\in[1..n]} P_i$. We also extend the precedence of operators: $|$ binds tighter than $\oplus$, which binds tighter than definitions.

Non-determinism is not only due to definitions with several rules; it can appear in any join-synchronization, because messages may be combined in different manner before being consumed. For instance, we can choose between different values by joining them with a single message in a two-way pattern: the following process behaves as one of the processes $P\{^1/_v\}$, $P\{^2/_v\}$, or $P\{^3/_v\}$.

$$\mathsf{def}\ \mathit{once}\langle\rangle \mid y\langle v\rangle \triangleright P \ \mathsf{in}\ y\langle 1\rangle \mid y\langle 2\rangle \mid y\langle 3\rangle \mid \mathit{once}\langle\rangle$$

More sophisticated variants of choice are available; for instance, we can use continuations to encode *external choice*, where the context—modeled here by the receiver of the message $\mathit{plug}\langle\kappa_1,\kappa_2\rangle$—may choose one of the two branches.

$$\mathsf{def} \begin{array}{rl} & \kappa_1\langle\widetilde{v}\rangle \mid \mathit{once}\langle\rangle \triangleright P \\ \wedge & \kappa_2\langle\widetilde{v}\rangle \mid \mathit{once}\langle\rangle \triangleright Q \end{array} \ \mathsf{in}\ \mathit{plug}\langle\kappa_1,\kappa_2\rangle \mid \mathit{once}\langle\rangle$$

2.4.6 The reference cell

We finish our series of examples with a larger example that illustrates both higher-order and the use of internal messages to store some local state. Our *reference cell abstraction* is defined as

$$\mathit{mkcell}\langle v_0,\kappa_0\rangle \triangleright \left(\mathsf{def} \begin{array}{rrl} & \mathit{get}\langle\kappa\rangle \mid s\langle v\rangle \triangleright & \kappa\langle v\rangle \mid s\langle v\rangle \\ \wedge & \mathit{set}\langle u,\kappa\rangle \mid s\langle v\rangle \triangleright & \kappa\langle\rangle \quad \mid s\langle u\rangle \end{array} \ \mathsf{in}\ \kappa_0\langle \mathit{get},\mathit{set}\rangle \mid s\langle v_0\rangle\right)$$

Each message on *mkcell* triggers the external definition, which allocates a new reference cell. Three fresh names $\mathit{get}, \mathit{set}, s$ are defined, and two rules are activated, then the first two names are passed to the continuation κ_0 for later access or update to the new cell. Thanks to lexical scoping, the last name s remains local, and the initial message $s\langle v_0\rangle$ together with the two internal rules guarantee the invariant of the cell: there is exactly one message on s, which contains the current value.

Using the same style of definition, we can define other abstractions of concurrent objects, such as locks, monitors, counters, concurrent data structures. Further examples are given in Chapter 3, where we also provide a much simpler syntax that hides the details of continuations.

2.5 Basic properties of the reflexive CHAM

In this section, we discuss some chemical properties and we relate our chemical semantics to more traditional operational semantics.

2.5.1 Normal forms

The next remark prepares the identification of processes and single-molecule solutions. From an arbitrary chemical solutions, we can repeatedly apply heating rules until we obtain two flat multisets of single-clause definitions and single messages. Then, we can cool down this solution using rules Str-join and Str-and, and finally Str-def once.

Remark 2.1 *Every chemical solution is structurally equivalent*

- *to a fully-heated solution that contains only simple reaction rules and messages (unique up-to alpha-conversion)*

$$\{\cdots J_j \triangleright P_j, \cdots\} \vdash \{\cdots x_i\langle\widetilde{u_{ik}}\rangle, \cdots\}$$

- *and to the corresponding solution that contains a single, cooled-down process*

$$\emptyset \vdash \left\{\mathsf{def} \bigwedge J_j \triangleright P_j \;\mathsf{in} \prod x_i\langle\widetilde{u_{ik}}\rangle\right\}$$

The analysis of chemical reductions is especially simple on fully-heated solutions, as we only have to assemble a few messages by rule Str-join to match a join-pattern. After the reduction step, we can heat the triggered process to recover a normal form. Up to renaming, these manipulations are independent of other components in solution.

2.5.2 Built-in locality

As a way of bringing the rcham closer to its actual implementation, we informally describe the distributed aspects of the computation as a refinement of our normal forms. In Chapter 7, we elaborate on this distributed implementation scheme by making explicit the boundaries of reactions rules plus their pending messages, which will be called "locations".

We first partition reaction rules so that the sets of names defined in each class of reaction rules are pairwise disjoint, then we attach every message in solution to the unique class of rules that may receive the message.

For every such partition of the rules, two reduction steps that operate on different classes are independent, and thus there is no need for shared information across the different classes of the partition. Moreover, we can use structural rearrangement in a directed manner to obtain a normal form again whenever a new process is fired. In case the new process is a defining process, we can apply rule Str-def then rule Str-and to create a new class of rules. In case the new process is a parallel composition, we apply Str-join and attach every new message to its receiving class of rules.

To sketch an implementation, we still have to describe how "local messages" attached to a class of rules are organized, and how newly-created messages are attached to a class. The first operation is local synchronization, which can be efficiently implemented using automata that operate on queues of messages. The second operation is routing.

2.5.3 Reductions on processes

A chemical semantics naturally induces a structural equivalence $\equiv$ on terms, defined as the smallest structural congruence that contains $\rightleftharpoons$; this leads to a more classical presentation of the semantics as term rewriting modulo equivalence. Using the normal forms of Remark 2.1, structural congruence can be checked efficiently in polynomial time. (The relations $\rightleftharpoons^*$ and $\equiv$ do not entirely coincide because $\rightleftharpoons^*$ is only a congruence for evaluation contexts, and does not allows the rearrangement of terms under join-pattern guards $J \triangleright [\,\cdot\,]$.)

Since we identify processes P and single-molecules chemical solutions $\emptyset \vdash \{P\}$, we can recover notions of structural equivalence and reduction on processes only from our chemical semantics as the traces of chemical relations on solutions of the special form $\emptyset \vdash \{P\}$. This yields the structural equivalence $\rightleftharpoons^*$ and the single-step reduction relation $\rightleftharpoons^* \longrightarrow \rightleftharpoons^*$ on processes. The resulting rewriting system is a more traditional presentation of the join-calculus, which is useful to compare our model to other calculi, and to write terms and reductions in a more compact way.

In the following, we usually consider processes modulo structural rearrangement $\rightleftharpoons^*$, and we use the shorter notation $\rightarrow$ instead of $\rightleftharpoons^* \longrightarrow \rightleftharpoons^*$ to denote a reduction step on processes modulo structural rearrangement. More generally, any relation that is closed by $\rightleftharpoons^*$ can be seen either as a relation on chemical solutions or as a relation on processes.

2.5.4 Case analyses on reduction steps

Whenever we perform a case analysis on reductions, we implicitly use a bijection between reaction rules in solutions and reaction rules in cooled-down processes. Formally,

1. we use Remark 2.1 and we keep track of the series of structural rearrangements from the original process to the normal form;

2. we perform the reduction or the structural rearrangement in question;

3. we check, by induction on the length of the series of rearrangements, that the converse steps can be performed in the other direction and lead back to a derivative that has the same shape as the original process.

   All structural steps are left unchanged, except for α-conversion, for the use of rule Str-join which are used to assemble a join-pattern, and for the rule Str-def in the case scope-extrusion occurred.

We omit this common argument in the following, and directly partition reductions according to their syntactic join-patterns.

2.5.5 SOS-style semantics

We now give a alternative, direct characterization of $\equiv$ and $\rightarrow$, which we prove equivalent to our chemical characterization.

The structural congruence relations on processes and definitions are syntactically re-defined in a mutual recursive way, as the smallest equivalences that satisfy all the rules listed in Figure 2.4; they are both denoted $\equiv$. The axioms P1–P3 and R1–R3 equip processes and definitions with a multiset structure. The axioms D1–D3 deal with the scopes of definitions. The axioms A1–A2 describe α-conversion on received names and defined names, respectively. The rules C1–C3 are context rules.

The syntactic transitions on processes $\xrightarrow{\delta}$ is the smallest transition system that satisfies all the rules listed in Figure 2.5, where the "wide labels" δ range over reaction rules plus τ. The syntactic reduction step $\rightarrow$ is the relation $\xrightarrow{\tau}$. The axiom RED describes join-reduction, the rules E1–E4 are context rules.

As previously mentioned, structural rearrangements cannot occur under a join-pattern guard, but if we remove the context rule for guards, we obtain the expected coincidence of chemical and syntactic relations

Proposition 2.2 *In the absence of rule C2, we have the correspondences*

1. $P \equiv Q$ *iff* $\emptyset \vdash \{P\} \rightleftharpoons^* \emptyset \vdash \{Q\}$.
2. $P \rightarrow Q$ *iff* $\emptyset \vdash \{P\} \rightleftharpoons^* \rightarrow \rightleftharpoons^* \emptyset \vdash \{Q\}$.

Proof: The proof is simple but tedious because the structures of the derivations are not the same in both cases.

1. We check that $\rightleftharpoons^*$ satisfies all the axioms and rules of Figure 2.4, and thus obtain that $\equiv \subseteq \rightleftharpoons^*$. Each of the syntactic rules P1–P3, R1–R3, D1–D3 can be derived from the chemical rules STR-* by composing a few chemical heating and cooling steps. For the context rules C1 and C3 we use the same series of heating and cooling in-between an initial heating step and a final cooling one. This series of heating and cooling is enabled in the presence of additional processes and definitions inasmuch as we perform α-conversion to prevent clashes in the application of STR-DEF in the heating direction.

   Conversely, we use Remark 2.1 for some canonical ordering of all processes and definitions to associate to every chemical solution its unique cooled-down single-molecule solution. For every heating and cooling step, we check that we can perform matching structural rearrangement on the cooled-down processes.

   - STR-NULL and STR-JOIN correspond to P1–P3
   - STR-NODEF and STR-AND correspond to R1–R3
   - STR-DEF corresponds to D1–D3

   Finally, we check that every process is structurally equivalent to its unique flat cooled-down process.

2. We first restrict the proof requirement to fully-diluted normal forms by using the same argument, then we have a direct correspondence between chemical reduction steps and syntactic reduction steps RED–E1–E2. □

The side conditions in the chemical rule STR-DEF may seem to compromise the locality property by requiring that names be fresh in the entire chemical solution. From the implementor's point of view, this is not a problem, as it is easy to generate new identifiers distinct from any others. More formally, the inductive definition of $\rightarrow$

$$
\begin{array}{llcll}
\text{P1} & P \mid \mathsf{0} & \equiv & P & \\
\text{P2} & P \mid Q & \equiv & Q \mid P & \\
\text{P3} & (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
\\
\text{R1} & D \wedge \mathsf{T} & \equiv & D & \\
\text{R2} & D_1 \wedge D_2 & \equiv & D_2 \wedge D_1 & \\
\text{R3} & (D_1 \wedge D_2) \wedge D_3 & \equiv & D_1 \wedge (D_2 \wedge D_3) & \\
\\
\text{D1} & \mathsf{def}\ \mathsf{T}\ \mathsf{in}\ P & \equiv & P & \\
\text{D2} & P \mid \mathsf{def}\ D\ \mathsf{in}\ Q & \equiv & \mathsf{def}\ D\ \mathsf{in}\ P \mid Q & (\mathsf{fv}[P] \cap \mathsf{dv}[D] = \emptyset) \\
\text{D3} & \mathsf{def}\ D_1\ \mathsf{in}\ \mathsf{def}\ D_2\ \mathsf{in}\ P & \equiv & \mathsf{def}\ D_1 \wedge D_2\ \mathsf{in}\ P & (\mathsf{fv}[D_1] \cap \mathsf{dv}[D_2] = \emptyset) \\
\\
\text{A1} & J \triangleright P & \equiv & J\sigma_{\mathrm{rv}} \triangleright P\sigma_{\mathrm{rv}} & (\sigma_{\mathrm{rv}} \text{ injective, on } \mathsf{rv}[J]) \\
\text{A2} & \mathsf{def}\ D\ \mathsf{in}\ P & \equiv & \mathsf{def}\ D\sigma_{\mathrm{dv}}\ \mathsf{in}\ P\sigma_{\mathrm{dv}} & (\sigma_{\mathrm{dv}} \text{ injective, on } \mathsf{dv}[D]) \\
\\
\text{C1} & P \equiv Q & \Longrightarrow & P \mid R \equiv Q \mid R & \\
\text{C2} & P \equiv Q & \Longrightarrow & J \triangleright P \equiv J \triangleright Q & \\
\text{C3} & P \equiv Q, D_1 \equiv D_2 & \Longrightarrow & \mathsf{def}\ D_1\ \mathsf{in}\ P \equiv \mathsf{def}\ D_2\ \mathsf{in}\ Q &
\end{array}
$$

Figure 2.4: Structural congruence on processes and definitions

$$
\begin{array}{llcll}
\textsc{Red} & x_1\langle\widetilde{v_1}\rangle \mid \cdots \mid x_n\langle\widetilde{v_n}\rangle & \xrightarrow{x_1\langle\widetilde{y_1}\rangle \mid \cdots \mid x_n\langle\widetilde{y_n}\rangle \triangleright P} & P\left\{\widetilde{{}^{v_1}/_{y_1}}, \cdots, \widetilde{{}^{v_n}/_{y_n}}\right\} & \\
\\
\text{E1} & P \xrightarrow{d} P' & \Longrightarrow & \mathsf{def}\ d \wedge D\ \mathsf{in}\ P \xrightarrow{\tau} \mathsf{def}\ d \wedge D\ \mathsf{in}\ P' & \\
\text{E2} & P \xrightarrow{\delta} P' & \Longrightarrow & P \mid Q \xrightarrow{\delta} P' \mid Q & \\
\text{E3} & P \xrightarrow{\delta} P' & \Longrightarrow & \mathsf{def}\ D\ \mathsf{in}\ P \xrightarrow{\delta} \mathsf{def}\ D\ \mathsf{in}\ P' & (\mathsf{fv}[\delta] \cap \mathsf{dv}[D] = \emptyset) \\
\text{E4} & Q \equiv P \xrightarrow{\delta} P' \equiv Q' & \Longrightarrow & Q \xrightarrow{\delta} Q' &
\end{array}
$$

d ranges over definition clauses $J \triangleright P$; δ ranges over d and τ.

Figure 2.5: Syntactic transitions with wide labels

clarifies the issue by decomposing the global freshness requirements into several local steps (α-conversion may be required to apply rule D3).

We can tighten the structure of processes during reduction, preventing definitions being merged or exchanged. This is achieved by removing the rule D3 from the definition of structural equivalence. Also, we can further limit the use of structural congruence to those that contribute to a reduction step. More precisely, the reduction step obtained by retaining from the two above tables only the rules

- P2, P3, Red, E2, E3
- a merge of A2 and P2 that moves a process within a defining process after α-conversion, only when the first process contains a message consumed in the reduction step.
- an extended E1 that embeds R2,R3 on $d \wedge D$.

still expresses all derivations, up to structural rearrangement on the resulting process only. This suggests another definition of normal forms, where the historical nesting of all definitions is preserved through reductions.

2.5.6 Refined chemical machines

We close the technical discussion of chemical semantics versus structured operational semantics by references to other parts of this work where chemical mechanisms are refined.

- In Chapter 3, we slightly restrict the chemical machine to preserve the grouping of reaction rules in definitions, in order to establish the subject-reduction property for our typing system. The resulting machine corresponds to a weaker rule D3 that allows definition to be swapped but not merged.
- In Chapter 5, we supplement the chemical machine with rules that directly model interaction with the environment, and obtain a more intensional model.
- In Chapter 7, we use refined chemical machines that keep track of some explicit locality information, and we model both process migration and failures in that setting. The chemical notation is especially convenient there, since structural rearrangements become rather complex.

2.6 Other models of concurrency

We sketch a comparison with similar models of concurrency; models that specifically address distributed computation are discussed in Section 7.4.

Our calculus focuses on mobility in a minimal setting. This contrast with languages for concurrent programming and distributed programming that extend a functional kernel [60, 124, 81, 12] or an object-oriented kernel [44, 63].

Other choices of communication primitives are possible. Instead of directed communication with a functional flavor, some calculi rely for instance on unification and broadcast. This is the case for Oz [143], and for linear objects [20, 18, 19].

2.6.1 Higher-order Gamma

Gamma and the chemical metaphor have been used in numerous contexts. We refer to [27] for a recent survey of its applications. More specifically, in [95] Le Métayer extends the Gamma model to higher-order, which generalizes the CHAM of Berry and Boudol, and actually also the RCHAM.

Higher-order Gamma is a very expressive calculus; its *named multisets* are roughly equivalent to asynchronous channels, but they provide internal multiset rewritings in conjunction with primitives (access, union, difference) that are triggered by the termination of internal rewritings.

The additional structure suggests efficient implementations in special cases, because nested solutions may provide a better locality in the computation. However, it seems hard to give a general implementation of the model, even for a centralized system. We believe that lexical scoping on names is formally much simpler, and suffices for general-purpose programming.

2.6.2 Data flow languages

The data flow programming model also provides a fine-grained parallelism, in a more implicit manner than in the join-calculus: each argument of a function call is received in a low-level message, and each call to a strict n-ary primitive performs a n-way synchronization on its arguments before evaluation. However, the argument messages are not first-class values, hence it is not possible to manipulate synchronization in the language. In data flow languages such as Id [54], the basic semantics is the one of a parallel but deterministic functional language. Additional synchronization mechanisms are introduced through specific data structures, such as mutable data structures, or synchronization barriers. These specific mechanisms are easily implemented in the join-calculus.

Despite this difference of expressiveness, the implementation mechanisms for data flow languages are surprisingly similar to those for the join-calculus. For instance, the techniques developed for controlling the number of threads running in parallel also applies to our implementation.

2.6.3 Multi-functions

To our knowledge, Banâtre [23] was the first to suggest "multi-functions" as primitives for synchronization. To benefit from a multiprocessor architecture, *execution blocks* may fork into a number of threads running in parallel, and symmetrically any thread may attempt a synchronization with all its siblings. The threads are blocked until all other threads join the synchronization, then a few values are exchanged and all threads resume their independent executions. Overall, multi-functions extend the standard procedural model, with more flexibility than in the data-parallel model. These concepts have been integrated in the GOTHIC programming language.

The multi functions correspond to a first-order version of our join definitions (after applying the call-by-value CPS of Section 3.4). As an interesting programming exercise, it is possible to encode the block-building and the synchronization primitives of GOTHIC into the join-calculus language developed in the next chapter.

2.6.4 Petri nets

The connection between petri nets and chemical machines is already stressed by Berry and Boudol in [29]. Informally, the RCHAM corresponds to a dynamic variant of colored Petri nets. Places are given names; markings of different colors are represented as messages on those names; reaction rules consume and generate markings. Actually a very small part of the chemical machine is needed, as it coincides with the RCHAM without definition of local rules (that is, no defining processes under guards).

In addition, the reflexive aspect of the RCHAM can be represented by the unfolding of a new subnet whenever a guarded process defines new names. The formal connection between the join-calculus and higher-order Petri nets is studied in [22]. Interestingly, the locality property of the join-calculus is not enforced in higher-order Petri nets, which gives the latter more expressiveness, but renders its implementation problematic.

2.6.5 Other variants of the π-calculus

Despite their different syntaxes, the join-calculus is essentially a variant of the asynchronous π-calculus, with two main differences: (1) the locality property is built-in, and (2) join-patterns can define richer, n-way synchronization. (The technical comparison with the π-calculus is continued in Sections 5.6 and 6.6.)

The extended, n-way synchronization of the join-calculus seems more general than the input-output synchronization of the π-calculus, and is reminiscent of the *polynomial π-calculus*, a variant of the π-calculus introduced by Milner with more general communication prefixes that may involve several channels in one communication. In fact, n-way joins are not more expressive than plain π-calculus communication, because the locality property and the asynchronous semantics render this synchronization invisible (*cf.* Section 6.6). Nestmann gives a more detailed account on the expressiveness of join inputs without locality in [108].

Since we proposed the join-calculus to address distributed programming, several other variants of the π-calculus have been proposed to a similar effect, in a more conservative syntax [135, 142, 94, 13, 14]. The join-calculus syntactically separates receiving definitions and processes, in (multi) functional programming style. Conversely, these variants retain the input guards of the π-calculus, but they restrict their use.

In [33], a fragment of the π-calculus is considered where received names cannot be used for input (i.e., cannot be redefined). Communication patterns are further restricted in [135] by also demanding that names be available in input-replicated form as soon as created. The resulting notion of *uniform receptiveness* for the π-calculus almost coincides with our locality property, but it is enforced by means of a type system, while locality is syntactic in the join-calculus.

In the A-π-calculus of Merro and Sangiorgi [94] only output capabilities can be communicated, hence all receiving prefixes are statically known. In the π_1-calculus of Amadio [13], there is a unique receiving agent for every channel. In both cases, these properties are enforced by a refined type discipline. As observed in [94], such restrictions make the resulting calculus very similar to the join-calculus. If we measure the distance between two calculi in terms of the complexity of fully abstract encodings, then the A-π-calculus is much closer to the join-calculus than it is to the asynchronous π-calculus.

Chapter 3

Adding Types and Functions

As discussed in the previous chapter, our main motivation in the design of the join-calculus is to guarantee a transparent distributed implementation of channel-based communication. To this end, we introduced an asynchronous process calculus that enforces a more static control over communication than traditional process calculi, as a way of reflecting the need for locality present in realistic distributed systems.

In this chapter, we show that locality is also a sound design choice toward a practical programming language, not just a technical device to support the constraints of distributed programming. We expand our model of concurrency into a simple programming language and we illustrate some of its features. We propose a type system for the join-calculus whose simplicity owes much to locality. We also relate our language to typed functional programming, and to object-oriented programming. From the programmer's point of view, the join-calculus becomes the core of a high-level concurrent language with lexical scope and asynchronous messages. Most of these ideas have been integrated to our prototype implementation of the join-calculus [59]. Our programming environment provides several useful static semantics that analyze programs written in the join-calculus and determine their properties at compile-time instead of checking them at run-time.

As can be expected, our static semantics relies on a type system. The types we need should be expressive enough for most useful programs and easy to understand for programmers. This goal is achieved by adapting the Damas-Milner typing discipline developed for ML [50] to the join calculus. Also known as *implicit parametric polymorphism*, this type discipline is widely used in functional programming languages, as it strikes a good balance between simplicity and expressivity.

From the typing point of view, ML and the join-calculus have a similar structure: definitions in the join-calculus are a generalized form of let expressions in ML and polymorphism can be introduced right after type checking the clauses of a join-definition. However, their semantics are different: synchronization on channels is more demanding than plain function calls, as it interacts with polymorphism. The main technical contribution here is a generalization criterion for the join-calculus that addresses this issue. We prove the correctness of the resulting typing rules with regard to our concurrent semantics by adapting standard techniques to the chemical framework.

While the join-calculus can express various synchronization schemes in a declarative manner, it lacks some syntactic support to express simple sequential control. We identify function calls as a special case of message passing in continuation-passing style

(CPS) and we analyze two reduction strategies for their evaluation. It turns out that the sequential deterministic subset of the join-calculus is basically the continuation-passing style λ-calculus; hence we can embed the λ-calculus using any CPS transform.

Next, we choose a call-by-value CPS and we define a convenient syntactic sugar for sequential control, then for arbitrary functional expressions. The syntactic extension carries over typing, and allows a direct comparison between our type system and the type systems of functional languages with imperative constructs. Overall, the resulting join-calculus language extends a higher-order sequential language with parallelism in expressions (with fork calls) and in function patterns (with join patterns). Join patterns are consistent with lexical scope: they statically bind (joint) function calls to a body of code, whereas the binding of messages to receptors is dynamic.

The language also has object-oriented features. Simple concurrent objects correspond to the definition of names in the join-calculus that represent both the state of the object and its methods. The behavior of the object is declared in the rules of the definition; in particular, concurrent objects can express elaborate synchronization schemes as join patterns on their methods. Our firm commitment to lexical scoping makes our objects very static, but hopefully more primitive object features can be merged to the model with a richer type system.

Contents of the chapter

In section 3.1 we give an overview of our type system. In section 3.2, we define the typed join-calculus. The RCHAM that is presented in Figure 2.3 has a defect as regards typing; we introduce a variant and we relate it to the original. We present the typing rules, and briefly discuss type inference and recursive types. In section 3.3, we establish our main results on types: we prove subject reduction in a chemical setting and we show that well-typed programs cannot go wrong at run-time. In section 3.4, we extend the join-calculus with support for functions and expressions, and we generalize the type system accordingly. In section 3.5 we discuss the representation of objects in the join-calculus. In section 3.6, we compare our work to other type systems that have been proposed for concurrent calculi.

3.1 Polymorphism in the join-calculus

The join-calculus is essentially a name-passing calculus: port names are defined, then used as addresses in messages that convey other names. In that sense the join-calculus is a higher-order calculus (in concurrency theory, however, it would be labeled first-order because processes are not first-class values).

Our polyadic messages are of the form $x\langle x_1, \ldots, x_n\rangle$; the type of a name x carrying n objects of types $\tau_1, \ldots, \tau_n$ is written $\langle \tau_1, \ldots, \tau_n\rangle$. Traditional languages come with system-supplied primitives, which can be used in the programming practice. Similarly, we could assume system-supplied primitive names for a language based on the join-calculus, such as *print_int* that outputs its integer argument on the console. The following rule defines a new name *print_two_ints* that prints two integers:

$$\mathit{print\_two\_ints}\langle x, y\rangle \triangleright \mathit{print\_int}\langle x\rangle \mid \mathit{print\_int}\langle y\rangle$$

When the name *print_two_ints* receives a couple of arguments x and y, it activates two processes $\mathit{print\_int}\langle x\rangle$ and $\mathit{print\_int}\langle y\rangle$ running concurrently. The type of the primitive *print_int* is $\langle\texttt{Int}\rangle$ (i.e. a name that carries one integer) hence the type of the new name *print_two_ints* is $\langle\texttt{Int},\texttt{Int}\rangle$ (i.e. a name that carries two integers).

In this context, a name with a polymorphic type in the join-calculus is reminiscent of a polymorphic function in ML: both do not necessarily perform fully type-specific operations on their arguments. Thus, the types of the arguments are not completely specified and unspecified parts are represented by type variables that stand for just any type. This framework is known as *parametric polymorphism.* Let us consider the following rule:

$$\mathit{apply}\langle\kappa, x\rangle \triangleright \kappa\langle x\rangle$$

The name *apply* takes two arguments κ and x and activates the process $\kappa\langle x\rangle$. Thus, if x is of type τ, then κ must carry names of type τ, i.e., be of type $\langle\tau\rangle$. The name *apply* can be given the type $\langle\langle\tau\rangle,\tau\rangle$ for any type τ. As in ML, this is emphasized by giving *apply* the type scheme $\forall\alpha.\,\langle\langle\alpha\rangle,\alpha\rangle$. Quantification of the type variable α is called *generalization.* Therefore *apply* can take *print_int* and 4 as arguments by the call $\mathit{apply}\langle\mathit{print\_int},4\rangle$, thereby instantiating α with the type `Int`. Given another primitive *print_string*, another legitimate invocation $\mathit{apply}\langle\mathit{print\_string},\text{“foo”}\rangle$ would instantiate α with the type `String`.

While ML is a purely sequential language, the join-calculus describes parallel computations, and features synchronization between messages in parallel through join-patterns. Consider for instance a variant of *apply* that receives κ and x from different sources.

$$\mathit{port}\langle\kappa\rangle \mid \mathit{arg}\langle x\rangle \triangleright \kappa\langle x\rangle$$

The concurrent activation of the co-defined names *port* and *arg* fires $\kappa\langle x\rangle$. The names *port* and *arg* can be given the types $\langle\langle\alpha\rangle\rangle$ and $\langle\alpha\rangle$, respectively. Still, the names *port* and *arg* are correlated by the use of the same type variable α in their types. This forbids to give *port* and *arg* the type schemes $\forall\alpha.\,\langle\alpha\rangle$ and $\forall\alpha.\,\langle\langle\alpha\rangle\rangle$. Otherwise, their types schemes could be instantiated independently, loosing their correlation. Clearly, sending the primitive *print_string* on *port* and an integer on *arg* may cause a run-time type error: attempting to print an integer as a string.

As a consequence, our generalization rule copes with synchronization in an abstract way: a type variable is generalized as long as it does not occur free in the type of *several* co-defined names.

3.2 The typed join-calculus

3.2.1 Syntax

We supplement our grammar for the untyped join-calculus with a definition of types, typing environment, and typing judgments.

The grammar for types and typing environments is defined in Figure 3.1. We assume given a finite set of primitive types $b \in \mathcal{T}$ such as `Int` or `String` and a countable set of type variables α. A type τ is either a primitive type, a type variable, or a message type conveying a fixed number of types; a type scheme σ may quantify over type variables. We use the notations $\mathsf{fv}[\tau]$ and $\mathsf{fv}[\sigma]$ for the free type variables that occur in τ and σ.

We collect bindings between names and types into typing environments. A typing environment A associates type schemes to names, while a simple environment B associates types to names. Given an environment A that already associates some type scheme to a name u, the new environment $A + (u : \sigma)$ is well formed and associates σ to u. More generally, we use the notation $A + A'$ for the new environment obtained by extending A with every binding in A' in turn. We also write $\mathsf{fv}[A]$ for the union of the free variables in any binding of A.

We use three kinds of typing judgments for the terms of the join-calculus: names, processes, and definitions:

- $A \Vdash u : \tau$ states that the name u has type τ in A;
- $A \Vdash P$ states that the process P is well-typed in A;
- $A \Vdash D :: B$ states that the definition D is well-typed in $A + B$, where B is a simple typing environment for the names defined in D ($\mathsf{dv}[D] \subseteq \mathsf{dom}\,(B)$).

3.2.2 Typing rules

The rules of Figure 3.2 describe valid proofs for our judgments. They are much inspired by the typing rules for the (polyadic) λ-calculus plus `let rec`, the real innovation being the generalization in rule (DEF). Our rules use the following definitions:

- in rule (AND), $B_1 \oplus B_2$ is $B_1 + B_2$, and requires that the restrictions of the environments B_1 and B_2 on the names $\mathsf{dom}\,(B_1) \cap \mathsf{dom}\,(B_2)$ be identical.
- in rule (DEF), $\mathrm{Gen}(B, A)$ is the generalization of the simple environment B with respect to A: let $(x_i : \tau_i)\ ^{i \in 1..n}$ enumerate the bindings in B, and let $B - x$ denote the environment B after removing the binding for x. Then $\mathrm{Gen}(B, A)$ is $(x_i : \forall\, (\mathsf{fv}[\tau_i] \setminus \mathsf{fv}[A + B - x_i])\,.\tau_i)\ ^{i \in 1..n}$.

Intuitively, $\mathrm{Gen}(B, A)$ binds every name in $\mathsf{dv}[D]$ to the simple type collected in B generalized on every type variable that occurs only in this simple type.

3.2.3 External primitives

While names already provide enough expressiveness to encode basic values such as integers or strings, it is far more convenient to supplement names with constants that represent basic values 1, 2, ... ,"foo", ... along with their basic types such as `Int`, `String` and their primitive operations *add*, *string_of_int*, *print*. In our implementation, these primitive are mapped to the programming environment and its libraries.

In the whole chapter, values are either port names $x \in \mathcal{N}$ or constants $k \in \mathcal{K}$. We use $u, v \in \mathcal{N} \cup \mathcal{K}$ to denote a value in general.

Constants in $\mathcal{K}$ are parameterized by a family of relations $(\delta_k)_{k \in \mathcal{K}}$ that map values $u_i\ ^{i \in 1..p}$ to processes P. We add a reduction rule that specifies the behavior of constants to the RCHAM defined in Figure 2.3:

$$\text{RED-}\delta \quad \vdash k\langle \widetilde{u} \rangle \quad \longrightarrow \quad \vdash P \quad \text{when } (u_i\ ^{i \in 1..p}, P) \in \delta_k$$

τ	$::=$		type
		b	primitive type
	$\mid$	α	type variable
	$\mid$	$\langle \tau_1, \ldots, \tau_p \rangle$	channel type
σ	$::=$		type scheme
		τ	basic type
	$\mid$	$\forall \alpha . \sigma$	generalized type
A	$::=$		typing environment
		$\emptyset$	empty environment
	$\mid$	$A + (u : \sigma)$	overwrite/extension
B	$::=$		simple typing environment
		$\emptyset$	empty environment
	$\mid$	$B + (u : \tau)$	overwrite/extension

Figure 3.1: Syntax for the types

(Inst)
$$\frac{(u : \forall \alpha_i^{\ i \in 1..n} . \tau) \in A}{A \Vdash u : \tau\{{}^{\tau_i}/_{\alpha_i}{}^{\ i \in 1..n}\}}$$

(Message)
$$\frac{A \Vdash u : \langle \tau_i^{\ i \in 1..n} \rangle \qquad (A \Vdash u_i : \tau_i)^{\ i \in 1..n}}{A \Vdash u\langle u_i^{\ i \in 1..n} \rangle}$$

(Par)
$$\frac{A \Vdash P \qquad A \Vdash Q}{A \Vdash P \mid Q}$$

(Def)
$$\frac{A + B \Vdash D :: B \qquad A + \mathrm{Gen}(B, A) \Vdash P}{A \Vdash \mathsf{def}\ D\ \mathsf{in}\ P}$$

(Null)
$$\frac{}{A \Vdash \mathsf{0}}$$

(Rule)
$$\frac{A + u_{ij} : \tau_{ij}^{\ i \in 1..n, j \in 1..m_i} \Vdash P}{A \Vdash x_1\langle u_{1j}^{\ j \in 1..m_1} \rangle \mid \ldots \mid x_n\langle u_{nj}^{\ j \in 1..m_n} \rangle \triangleright P :: (x_i : \langle \tau_{ij}^{\ j \in 1..m_i} \rangle)^{\ i \in 1..n}}$$

(And)
$$\frac{A \Vdash D_1 :: B_1 \qquad A \Vdash D_2 :: B_2}{A \Vdash D_1 \wedge D_2 :: B_1 \oplus B_2}$$

(Nodef)
$$\frac{}{A \Vdash \mathsf{T} :: B}$$

Figure 3.2: Typing rules for the join-calculus

In the functional tradition, we interpret the presence of a message $k\langle\widetilde{u}\rangle$ when no δ-rule applies as a run-time error. We can still maintain an artificial distinction between erroneous "stuck" messages such as $3\langle \mathit{print\_int}\rangle$ and correct idle messages by providing a δ-rule that leaves idle messages unchanged. Another, more explicit approach would be to parameterize constants in $\mathcal{K}$ by a family of valid arguments $(\epsilon_k)_{k\in\mathcal{K}}$ for each primitive name, independently of primitive reductions.

It is useful to account for the constants in a type-preserving manner. To this end, constants in $\mathcal{K}$ are given with a primitive typing environment $A_\mathcal{K}$ of domain $\mathcal{K}$. The next definition gathers our requirement on the typed primitive environment; it requires that every typable primitive message be reducible—which rules out immediate errors on primitives—and that every primitive reduction be sound.

Definition 3.1 *The typed primitive environment $(\mathcal{K}, (\delta_k)_{k\in\mathcal{K}}, A_\mathcal{K})$ is correct when, for every typing environment A, if we can derive $A + A_\mathcal{K} \Vdash k\langle u_i\ {}^{i\in 1..p}\rangle$, then*

1. *for some process P, we have $(u_i\ {}^{i\in 1..p}, P) \in \delta_k$*
2. *for all processes P with $(u_i\ {}^{i\in 1..p}, P) \in \delta_k$, we have $A + A_\mathcal{K} \Vdash P$.*

In particular, the second requirement ensures that the free names of P are either constants or among the u_i.

3.2.4 Types and chemistry

For our type system to be of some use, we must show its consistency with respect to the semantics of the join-calculus. Before that, we modify our chemical semantics by slightly restraining structural rearrangements, and we extend typing judgments to chemical solutions.

Typing mutually-recursive definitions We would expect every typing judgment to be preserved by structural equivalence, but this is not the case with the structural rules of the RCHAM. The trouble lies in the grouping of definitions that changes outer bound occurrences into recursive ones. Given two definitions D_1 and D_2 defining disjoint sets of names and such that some names defined by D_1 occur free in D_2, but not the converse, we have the structural equivalence

$$\mathsf{def}\ D_1\ \mathsf{in}\ \mathsf{def}\ D_2\ \mathsf{in}\ P \quad\equiv\quad \mathsf{def}\ D_1\ \wedge\ D_2\ \mathsf{in}\ P$$

Unfortunately, the valid typing judgments for the names defined in D_1 and used in D_2 are not the same on each side of the equivalence. In the process $\mathsf{def}\ D_1\ \mathsf{in}\ \mathsf{def}\ D_2\ \mathsf{in}\ P$, names that are defined in D_1 can be generalized by rule DEF, then several instantiations may be used for typing the guarded sub-processes in D_2. On the contrary, in the process $\mathsf{def}\ D_1 \wedge D_2\ \mathsf{in}\ P$ only simple types for names defined in D_1 can be used in D_2 because generalization is disabled in rule AND. For instance, the problem arises for the rather useless process $\mathsf{def}\ x\langle u\rangle \triangleright \mathsf{0}\ \mathsf{in}\ \mathsf{def}\ y\langle\rangle \triangleright x\langle 3\rangle \mid x\langle\text{"foo"}\rangle\ \mathsf{in}\ \mathsf{0}$. In fact, we run across the classical limitation of typing for mutually-recursive functions.

A restricted chemical machine To solve this problem, we introduce a variant of the RCHAM that is better suited to our typing purposes. In the new machine, definitions with several clauses are not heated; more specifically, the structural rule STR-AND disappears and the reduction rule RED is generalized to access clauses within non-diluted definitions. We replace the two chemical rules

$$\begin{array}{lrcl} \textsc{Str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash \\ \textsc{Red} & J \triangleright P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\mathrm{rv}} \end{array}$$

by the single, generalized reduction rule

$$\textsc{Red-And} \quad \ldots \wedge J \triangleright P \wedge \ldots \vdash J\sigma_{\mathrm{rv}} \quad \longrightarrow \quad \ldots \wedge J \triangleright P \wedge \ldots \vdash P\sigma_{\mathrm{rv}}$$

(with the same side conditions as in rule RED) where the notation $\ldots \wedge J \triangleright P \wedge \ldots$ stands for a definition D that contains the clause $J \triangleright P$. This notation now expresses the commutativity and the associativity of $\wedge$, which were previously conveyed separately by the structural rule STR-AND.

In addition, and for every chemical soup $\mathcal{D} \vdash \mathcal{P}$, we require that every name be defined in at most one definition ($\forall D, D' \in \mathcal{D}, \mathsf{dv}[D] \cap \mathsf{dv}[D'] = \emptyset$). This requirement holds for any process before dilution, and is preserved by all chemical reactions in the new machine.

We now relate this restricted machine to the original one. Let us first consider machines that operate on completely diluted solutions (i.e., heating rules cannot apply anymore). There is a straightforward correspondence between chemical solutions of the two formalisms: the processes are the same messages; the non-diluted definitions in the new machine are a partition of the diluted clauses in the original machine. Given this correspondence between solutions in the two chemical frameworks, both reduction rules RED and RED-AND followed by heating yield equivalent fully-diluted solutions. In the general case, cooling in the original machine may lead to more processes. However, we still have:

$$\overset{\textsc{Red}}{\rightarrow} \;\subseteq\; \overset{\textsc{Red-and}}{\rightarrow} \;\subseteq\; \left(\overset{\textsc{Str-and}}{\rightharpoondown}\right)^* \overset{\textsc{Red}}{\rightarrow} \left(\overset{\textsc{Str-and}}{\rightharpoondown}\right)^*$$

In the remaining parts of this chapter, we use the restricted chemical machine without further discussion. We omit the explicit labels on top of the chemical relations. For instance, $\rightleftharpoons^*$ stands for the transitive-reflexive closure of all the structural rules except rule STR-AND.

As an alternate approach, we could rearrange compound definitions using STR-AND to see a program as globally mutually recursive rules plus messages (*cf.* remark 2.1), and independently type them in the "best" structured way after rearranging the rules into several nested definitions. Pragmatically, this would correspond to a dependency analysis before typing, as for instance in the language Miranda. We reject this approach because we believe globally recursive definitions are a source of errors and we do not want to encourage them, as would be the case if definitions were silently flattened by our compiler.

Type-checking solutions The typing of programs easily extends to chemical solutions. Typing chemical solutions simplifies our proofs by avoiding some of the technicalities induced by the more common formalism of term-rewriting modulo structural equivalence. In particular, the chemistry allows us to treat structural rearrangements and proper reductions in the same manner.

First, we introduce a judgment $A \Vdash D$ to state that the assumptions made in A on the names defined in D are the same as if those names had been added to A after typing the definition D. We add the typing rule

$$\frac{\begin{array}{l}(\textsc{Multi})\\ A + B \Vdash D :: B \qquad \mathrm{Gen}(B, A) \subseteq A\end{array}}{A \Vdash D}$$

That is, we type D in the environment A extended with new assumptions B that must be exactly the typing environment produced by D as in rule Def; then, we check that the generalization of B in A is equal to A restricted to $\mathsf{dom}\,(B)$, i.e., $\mathrm{Gen}(B, A)$ is a subset of A. Observe that the "extended" environment $A+B$ is also $(A-\mathsf{dom}\,(B))+B$, and overrides generalized types with simple ones.

We also introduce a typing judgment $A \Vdash \mathcal{D} \vdash \mathcal{P}$ to state that the chemical solution $\mathcal{D} \vdash \mathcal{P}$ is well-typed in environment A. This happens when all the terms in the multisets $\mathcal{D}$ and $\mathcal{P}$ are independently well-typed in the environment A:

$$\frac{\begin{array}{l}(\textsc{Soup})\\ \forall P \in \mathcal{P}, A \Vdash P \qquad \forall D \in \mathcal{D}, A \Vdash D\end{array}}{A \Vdash \mathcal{D} \vdash \mathcal{P}}$$

3.2.5 Types at work

We briefly discuss typed extensions toward a more practical programming language. Most of these features can be trivially transposed from ML to the join-calculus and do not present much theoretical interest (although they are very useful, if not essential, in practice). Specifically, recursive types and data-types are part of our prototype implementation.

Others features of ML, such as region and effect type inference, might be more interesting, since their adaptation to the join-calculus could yield useful techniques for static analysis of concurrent programs.

Recursive types Recursive types naturally arise in many programs written in pure message-passing process calculi. In particular, they are needed in most encodings of this dissertation. Besides, the subset of processes that can be given monomorphic recursive types appears in the statement of several full abstraction results (*cf.* [2], and also Chapter 6).

In core ML, cyclic structures are allowed only in data-type definitions such as lists or trees. This approach carries over the join-calculus extended with data-types, and would suffice to our needs. In this work, however, we choose a more liberal approach that do not rely on extensions of the untyped calculus. We add *implicit recursive types* to our type system.

Informally, recursive types can be thought of as infinite regular trees. Technically, recursive types are represented finitely with a μ binder; they are equivalence classes of terms generated by the extended grammar

$$\begin{array}{rcll} \tau & ::= & & \text{recursive type} \\ & & b & \quad\text{primitive type} \\ & | & \alpha & \quad\text{type variable} \\ & | & \langle \tau_1, \ldots, \tau_p \rangle & \quad\text{channel type} \\ & | & \mu\alpha.\langle \tau_1, \ldots, \tau_p \rangle & \quad\text{recursive channel type} \end{array}$$

and quotiented by the equivalence on terms defined as follows

1. the *unfolding relation* is the congruence for all channel constructors generated by the axiom
$$\mu\alpha.\langle \tau_1, \ldots, \tau_p \rangle \quad = \quad \langle \tau_1, \ldots, \tau_p \rangle \left\{ {}^{\mu\alpha.\langle \tau_1, \ldots, \tau_p \rangle}/_{\alpha} \right\}$$

2. two types are equivalent when for an arbitrary depth n they have unfoldings that coincide on the first n levels of channel constructors.

(This notion of equivalence identifies the two recursive types $\mu\alpha.\langle \mathtt{Int}, \langle \mathtt{Int}, \alpha \rangle \rangle$ and $\langle \mathtt{Int}, \mu\alpha.\langle \mathtt{Int}, \langle \mathtt{Int}, \alpha \rangle \rangle \rangle$ while these types are not related by folding or unfolding relations.) For example, the message $x\langle x \rangle$ cannot be typed without recursive types, but becomes typable in the environment $A = (x : \mu\alpha.\langle \alpha \rangle)$ by using the derivation:

$$\frac{A \Vdash x : \mu\alpha.\langle \alpha \rangle \qquad A \Vdash x : \langle \mu\alpha.\langle \alpha \rangle \rangle}{A \Vdash x\langle x \rangle} \ (\textsc{Message})$$

As an advantage, implicit recursive types induce minimal changes in our setting. Except for the underlying notion of equality among types, the typing rules are unchanged, the typing derivations only deal with sufficiently unfolded types, and most properties of the type system extend smoothly. For instance, the proof of subject-reduction in the next section carries over recursive types without any change, because it relies only on syntactic matching on terms appearing in typing derivations.

Several more general techniques have been proposed to deal with recursive types, usually in a more explicit manner. In [15] for instance, recursive types interfere with subtyping properties. In a polyadic π-calculus setting, Turner gives in his dissertation a detailed bisimulation-based account of implicit recursive types [145]. Also, it is possible to enrich the syntax with explicit constructs to fold and unfold the recursive types, which makes type inference much easier, in particular in the presence of subtyping. This is the case in the PICT language [122].

Type inference Since our types and typing rules have the same structure as those of ML, our type system also allows for type inference.

Precisely, there exists an algorithm that, given a soup $\mathcal{D} \vdash \mathcal{P}$ and a typing environment A_0 that binds the free names of $\mathcal{D}$ and $\mathcal{P}$ with the exception of the defined names $\mathsf{dv}[\mathcal{D}]$, either returns a typing environment A of domain $\mathsf{dv}[\mathcal{D}]$ such that $A_0 \oplus A \Vdash \mathcal{D} \vdash \mathcal{P}$, or fails if no such typing environment exists. Moreover, if the algorithm succeeds, then A is principal, that is, for any other typing environment A' of domain $\mathsf{dv}[\mathcal{D}]$ such that $A_0 \oplus A' \Vdash \mathcal{D} \vdash \mathcal{P}$, then A is more general than A'.

Our implementation proceeds by first order-unification plus generalization on selected free variables. The type inference algorithm is part of our compiler; it supports recursive types by using circular unification. The complete formalization is a straightforward adaptation of the one for ML [50].

3.3 Correctness of the evaluation

From a quite abstract point of view, let us assume that some evaluation steps of a program P yield a new program P'. Typing and evaluation agree when two facts hold: first, a typing derivation of P' can be constructed from a typing derivation of P. Second, messages present in P' cannot cause "run-time type errors" such as adding a string to an integer—no type mismatch for primitives—or sending one argument only on a binary name—no arity mismatch on defined names.

3.3.1 Basic properties for the typing

We will need the following standard lemmas.

Lemma 3.2 (Useless variable) *Let u be a name that is fresh in D and P ($u \notin \mathsf{fv}[D] \cup \mathsf{fv}[P]$). We have:*

$$\begin{array}{rcl} A \Vdash P & \textit{iff} & A + (u : \sigma) \Vdash P \\ A \Vdash D :: B & \textit{iff} & A + (u : \sigma) \Vdash D :: B \end{array}$$

Lemma 3.3 (Substitution) *Let φ be a substitution on free type variables. We have:*

$$\begin{array}{rcl} A \Vdash P & \textit{implies} & \varphi(A) \Vdash P \\ A \Vdash D :: B & \textit{implies} & \varphi(A) \Vdash D :: \varphi(B) \end{array}$$

We say that a type $\forall \widetilde{\alpha}.\tau$ is *more general* than $\forall \widetilde{\alpha}'.\tau'$ when τ' is of the form $\tau\{\widetilde{\tau''}/\widetilde{\alpha}\}$. We lift this notion to typing environments as follows: A' is more general than A when A and A' have the same domain and for each u in their domain, $A'(u)$ is more general than $A(u)$.

Lemma 3.4 (Generalization) *If $A \Vdash P$ and A' is more general than A, then we have $A' \Vdash P$.*

Lemma 3.5 (Substitution of a name in a process) *If $A + (u : \tau) \Vdash P$ and $A \Vdash v : \tau$, then we have $A \Vdash P\{^v/_u\}$.*

These lemmas are easily established by examining the use of each typing rule in type derivations on both sides of the statements that appear in the lemmas.

3.3.2 Subject reduction

We say that two typing environments A and A' *agree* when their restrictions on primitive names are equal. We define the relation $\sqsubseteq$ between RCHAMs as the preservation of typing, that is, $\mathcal{S} \sqsubseteq \mathcal{S}'$ if for any typing environment A such that $A \Vdash \mathcal{S}$, there exists a typing environment A' such that $A' \Vdash \mathcal{S}'$ and A and A' agree.

Theorem 1 (Subject reduction) *Chemical reductions preserve typing for any correct primitive environment.*

Proof: We must prove that $\rightleftharpoons\!\!\!\!\!\Rightarrow$ is a sub-relation of $\sqsubseteq$. In fact, we prove the stronger property that typing environments also agree on variable names that appear in both solutions. That is, for every chemical rule $\rightleftharpoons\!\!\!\!\!\Rightarrow$, for every single step $\mathcal{S} \rightleftharpoons\!\!\!\!\!\Rightarrow \mathcal{S}'$ and $A \Vdash \mathcal{S}$, we show that $\mathcal{S}'$ is well-typed in an environment A' that possibly differ from A only on $(\mathsf{dv}[\mathcal{S}] \setminus \mathsf{dv}[\mathcal{S}']) \cup (\mathsf{dv}[\mathcal{S}'] \setminus \mathsf{dv}[\mathcal{S}])$. In contrast with classical proofs, there is no induction on the derivation of the reduction step. Instead, we check the property for every chemical step, then uniformly for the chemical law Context.

Basic case: We first consider the basic case for every chemical rule.

Str-join The step is $\vdash P_1 \,|\, P_2 \rightleftharpoons \;\vdash P_1, P_2$; by using the rules Par and Soup, we clearly have $A \Vdash P_1 \,|\, P_2$ iff $A \Vdash P_1, P_2$.

Str-null, Str-nodef are as easy, using the rule Soup and the corresponding typing axioms Null and Nodef.

Str-def Possibly after α-conversion, the step is $\vdash \mathsf{def}\ D\ \mathsf{in}\ P \rightleftharpoons D \vdash P$.

Heating: Assuming that $A \Vdash \mathsf{def}\ D\ \mathsf{in}\ P$, there is a derivation ending with:

$$\frac{A + B \Vdash D :: B \qquad A + \mathrm{Gen}(B, A) \Vdash P}{A \Vdash \mathsf{def}\ D\ \mathsf{in}\ P}\ (\textsc{Def})$$

Clearly, Def and Soup give $A + \mathrm{Gen}(B, A) \Vdash D \vdash P$.

Cooling: Let $A \Vdash D \vdash P$. Then A is of the form $A' + \mathrm{Gen}(B, A')$ and we have both $A' + B \Vdash D :: B$ and $A' + \mathrm{Gen}(B, A') \Vdash P$. Thus, by Def, $A' \Vdash \mathsf{def}\ D\ \mathsf{in}\ P$.

In both cases, the two typing environments agree on primitive names and on names defined both in the solution to the left and to the right of the structural rule.

Red-And The reduction is $D \vdash \varphi(J) \longrightarrow D \vdash \varphi(Q)$. We first assume that D is simply $J \triangleright Q$ where $J = x_1\langle \widetilde{u}_1\rangle \,|\, \ldots \,|\, x_n\langle \widetilde{u}_n\rangle$. Therefore let $A \Vdash J \triangleright Q \vdash \varphi(J)$. By the rule Soup, we have

$$A \Vdash \varphi(J) \tag{3.1}$$

$$A \Vdash J \triangleright Q \tag{3.2}$$

We decompose the environment A into $A' + \mathrm{Gen}(B, A')$ where B is $(x_i : \langle\widetilde{\tau}_i\rangle)^{\ i \in 1..n}$, $\mathrm{Gen}(B, A')$ is $(x_i : \forall \widetilde{\alpha}_i.\, \widetilde{\tau}_i)^{\ i \in 1..n}$, and $\widetilde{\alpha}_i$ is $\mathsf{fv}[\widetilde{\tau}_i] \setminus (\mathsf{fv}[A'] \cup \bigcup_{j \neq i} \mathsf{fv}[\widetilde{\tau}_j])$. By the rules Multi and Rule, judgment (3.2) yields a judgment for the guarded process

$$A' + B + (\widetilde{u}_i : \widetilde{\tau}_i)^{\ i \in 1..n} \Vdash Q \tag{3.3}$$

Besides, the derivation of judgment (3.1) must have the shape:

$$\frac{\dfrac{A \Vdash x_i : \langle\widetilde{\tau}'_i\rangle \qquad A \Vdash \varphi(\widetilde{u}_i) : \widetilde{\tau}'_i}{A \Vdash x_i\langle\varphi(\widetilde{u}_i)\rangle}\ (\textsc{Message}) \qquad i \in 1..n}{A \Vdash \varphi(J)}\ (\textsc{Par}) \tag{3.4}$$

where each type $\widetilde{\tau}'_i$ is an instance $\theta_i(\widetilde{\tau}_i)$ of the type $\forall\,\widetilde{\alpha}_i.\tau_i$, for some substitution θ_i on $\widetilde{\alpha}_i$.

Since generalizable variables never occur in two different bindings, the domains of θ_i's are disjoint. We define the substitution θ as the composition of the θ_i's for $i = 1, \ldots, n$.

Applying Lemma 3.3 to the judgment (3.3) for the substitution θ—which leaves A' unchanged— we get the judgment

$$A' + (x_i : \langle\theta(\widetilde{\tau}_i)\rangle)^{\;i\in 1..n} + (\widetilde{u}_i : \theta(\widetilde{\tau}_i))^{\;i\in 1..n} \Vdash Q$$

By lemma 3.4, we generalize the assumptions in the above judgment, and obtain

$$A + (\widetilde{u}_i : \theta(\widetilde{\tau}_i))^{\;i\in 1..n} \Vdash Q$$

Finally, this judgment and the hypothesis $A \Vdash \varphi(\widetilde{u}_i) : \theta(\widetilde{\tau}_i)$ of (3.4) allow us to derive $A \Vdash \varphi(Q)$ by iterating Lemma 3.5 for every received name u_i.

We now consider the general case for a definition $D = J \triangleright Q \wedge D'$. By the rules Soup and Multi, the hypothesis $A \Vdash J \triangleright Q \wedge D' \vdash \varphi(J)$ yields

$$A' + B \;\Vdash\; J \triangleright Q \wedge D' :: B \tag{3.5}$$
$$A' + \mathrm{Gen}(B, A') \;\Vdash\; \varphi(J) \tag{3.6}$$

where $A = A' + \mathrm{Gen}(B, A')$. By judgment (3.5) and rule And, and for some partition $B = B' \oplus B''$ we have $A' + B'' \Vdash J \triangleright Q :: B''$ and $A' + B' \Vdash D :: B'$. By lemma 3.2 applied to $A' + B'' \Vdash J \triangleright Q :: B''$ we get $A' + B \Vdash J \triangleright Q :: B''$. We reduce to the special case above by instantiating Rule with this last judgment.

Red-δ holds by hypothesis on primitives.

Context rule: The chemical law Context introduce unrelated terms in chemical solutions. We prove the corresponding inductive step. We assume $A \Vdash \mathcal{D} \cup \mathcal{D}_1 \vdash \mathcal{P} \cup \mathcal{P}_1$. By rule Def and Soup, we know that $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{P}$, and $\mathcal{P}_1$ are all well-typed in A. In particular, $A \Vdash \mathcal{D}_1 \vdash \mathcal{P}_1$.

In the case $\mathcal{D}_1 \vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\Rightarrow \mathcal{D}_2 \vdash \mathcal{P}_2$, by inductive hypothesis there exists A' such that $A' \Vdash \mathcal{D}_2 \vdash \mathcal{P}_2$, and moreover A and A' coincide on all names except names in $X = (\mathsf{dv}[\mathcal{D}_1] \setminus \mathsf{dv}[\mathcal{D}_2]) \cup (\mathsf{dv}[\mathcal{D}_2] \setminus \mathsf{dv}[\mathcal{D}_1])$.

- By rule Soup, $A' \Vdash \mathcal{D}_2 \vdash \mathcal{P}_2$ implies $A' \Vdash \mathcal{D}_2$ and $A' \Vdash \mathcal{P}_2$.
- Since the solution $\mathcal{D} \cup \mathcal{D}_1 \vdash \mathcal{P} \cup \mathcal{P}_1$ is well-formed, we have $\mathsf{dv}[\mathcal{D}] \cap (\mathsf{dv}[\mathcal{D}_1] \cup \mathsf{dv}[\mathcal{D}_2]) = \emptyset$, therefore by Lemma 3.2 applied to $A \Vdash \mathcal{D}$ we have $A' \Vdash \mathcal{D}$.
- Furthermore, the side condition of the rule S-Def yields $\mathsf{fv}[\mathcal{P}] \cap X = \emptyset$. Thus, by lemma 3.2 applied to $A \Vdash \mathcal{P}$. we also have $A' \Vdash \mathcal{P}$.

We conclude $A' \Vdash \mathcal{D} \cup \mathcal{D}_2 \vdash \mathcal{P} \cup \mathcal{P}_2$ by rule Soup. $\square$

3.3.3 No run-time errors

We state the correctness of a computation from what can be observed on running chemical machines, independently of typing properties. We assume that all the names that appear in solution are either defined names or primitive names—intuitively the running program has been entirely linked to its runtime environment. Especially, free variables are turned into primitive names, which ensures that the environment specifies their type.

When ill-formed messages are released in a solution, there is no reduction that would consume them, so they remain visible, exactly as barbs on free names in an untyped setting. In this case the computation has failed.

Definition 3.6 *A chemical solution $\mathcal{D} \vdash \mathcal{P}$ has failed when $\mathcal{P}$ contains either:*

- *A message $k\langle u_i \ ^{i \in 1..n}\rangle$ when no δ-rule applies;*
- *A message $x\langle u_i \ ^{i \in 1..n}\rangle$ when x is defined in $\mathcal{D}$ with arity $m \neq n$.*

In a sequential setting, errors are stuck configuration of programs. They occur only in the presence of primitive values, namely when a primitive is applied to a value outside its domain. This is mimicked in the join-calculus by our definition of correct environments, where stuck messages on primitive names are interpreted as run-time errors.

Stuck messages may also appear in the absence of primitive names, since the calculus we consider is polyadic. That is, sending messages with the wrong arity is an error. An examples of misuse of names is $\mathit{apply}\langle f\rangle$ in the case *apply* is defined as in Section 3.1. Of course, well-formed messages on defined names can still be deadlocked; they are not considered as errors.

Theorem 2 (Correct computation) *A well-typed chemical machine cannot fail as the result of chemical rewriting in a correct primitive environment. In particular, a well-typed program cannot fail.*

Proof: The messages excluded in Definition 3.6 cannot occur in a well-typed chemical soup. Immediate errors on primitive names are excluded in Definition 3.1. Arity mismatch on defined names rules out any typing derivation for the chemical solution because the use of rules Message and Def are in mutual exclusion.

By subject-reduction, the absence of failed messages in a solution is preserved by chemical rewriting. $\square$

3.4 Functional constructs

As emphasized by our treatment of polymorphism, join-calculus processes of the particular form $\mathsf{def}\ f\langle x\rangle \triangleright P\ \mathsf{in}\ Q \,|\, R$ are rather similar to the general expressions in functional languages $\mathsf{let}\ f(x) = E\ \mathsf{in}\ E'; E''$. In particular, these terms share the same static scoping discipline, even as their semantics are quite different: join-calculus processes are built using parallel composition, while functional expressions are built using sequential composition, with a tighter control on the order of evaluation.

In our basic model, synchronization happens only as molecules are consumed, and this suffices to express control flow. To encode functions as processes, we explicitly create and send continuations that enforce a fixed evaluation strategy. This style of programming is known as continuation-passing-style (CPS), and has been advocated as a simple unifying framework to compile functional programs [21]. For any given CPS, we obtain a simple embedding of higher-order functional programming. We present two reduction strategies for the λ-calculus, and their encoding in clear-cut subsets of the join-calculus.

In practice, however, the direct manipulation of continuations is verbose, error-prone, and generally too low-level for practical programming. Instead, we make the sequential control apparent: we fix a call-by-value CPS, and we provide it as syntactic sugar in the language. We supplement the join-calculus with functions and expressions, and we refine the type system accordingly. This language design style has been experimented first in the PICT language [122]. We describe a core programming language based on these ideas, and we give some programming examples. Such extensions turn the join-calculus into a convenient language that can be seen as a concurrent extension of a higher-order typed functional language à la ML, supplemented with concurrent evaluation of expression (forks) and synchronization in patterns (joins).

Alternatively, it is possible to extend the process-calculus to higher-order; for instance in [39] functions can be directly expressed, without the need for continuation channels. This approach would be interesting, in particular, to model the direct implementation of functions in our implementation.

3.4.1 Sequential control in the join-calculus

Resuming our informal discussion of Section 3.1, programmers may feel uncomfortable with the non-deterministic behavior of the *print_two_ints* example; they would prefer their program to print the arguments x and y in a fixed order. Indeed, our implementation provides a synchronous *print_int* primitive that takes two arguments: an integer to be output, and a continuation to be triggered thereafter. This continuation is used for synchronization only; it carries no argument, and has type $\langle\rangle$. Thus, the type of the synchronous *print_int* is $\langle \mathtt{Int}, \langle\rangle\rangle$. The synchronous version of *print_two_ints* also takes an extra continuation argument κ meant to represent the termination of the call, and has type $\langle \mathtt{Int}, \mathtt{Int}, \langle\rangle\rangle$:

$$\begin{array}{rl} \mathit{print\_two\_ints}\langle x, y, \kappa\rangle \triangleright & \mathsf{def}\ \kappa_y\langle\rangle \triangleright \kappa\langle\rangle\ \mathsf{in} \\ & \mathsf{def}\ \kappa_x\langle\rangle \triangleright \mathit{print\_int}\langle y, \kappa_y\rangle\ \mathsf{in} \\ & \mathit{print\_int}\langle x, \kappa_x\rangle \end{array}$$

Continuations can also convey arguments back. For instance, assuming a synchronous *plus* primitive of type $\langle \mathtt{Int}, \mathtt{Int}, \langle \mathtt{Int}\rangle\rangle$ the synchronous process successor of type $\langle \mathtt{Int}, \langle \mathtt{Int}\rangle\rangle$ could be defined by the rule

$$\mathit{succ}\langle x, \kappa\rangle \triangleright \mathit{plus}\langle x, 1, \kappa\rangle$$

The continuation passing style idiom is so common in process calculi that it deserves a convenient syntax that avoids writing explicit continuations. In our language, continuation arguments become implicit in both primitive names and user-defined names. The synchronous version of *print_two_ints* becomes

$$\mathrm{print\_two\_ints}(x,y) \triangleright \mathrm{print\_int}(x); \mathrm{print\_int}(y); \mathsf{reply\ to}\ \mathrm{print\_two\_ints}$$

The sequencing operator ";" avoids the definition of explicit continuations inside the body of print_two_ints. The final call to continuation is left explicit.

More generally, several synchronous names may be defined in the same join-pattern, and thus several named continuations may appear in the same guarded process. We write $\mathsf{reply}\ u_1, \ldots, u_p\ \mathsf{to}\ x$ for each synchronous name x, by analogy to the C statement "`return` expression;". In the extended syntax, for instance, we can set up a *rendez-vous* between two computations by the defining the rule:

$$\mathrm{a}(u) \mid \mathrm{b}(v) \triangleright \mathsf{reply}\ v\ \mathsf{to}\ \mathrm{a} \mid \mathsf{reply}\ u\ \mathsf{to}\ \mathrm{b}$$

The two synchronous names a and b can be passed to two threads of computation running in parallel, so that at some point of their computations they perform the calls $\mathrm{a}(x)$ or $\mathrm{b}(y)$, respectively, to get synchronized and share their partial results with the other thread, in the spirit of multi-functions [23].

We also provide a sequencing binding $\mathsf{let}\ x_1, \ldots, x_p = E\ \mathsf{in}\ P$, where E is an expression, i.e., a process meant to return some value on a given continuation. For instance, the successor process is written

$$\mathrm{succ}(x) \triangleright \mathsf{let}\ y = \mathrm{plus}(x, 1)\ \mathsf{in}\ \mathsf{reply}\ y\ \mathsf{to}\ \mathrm{succ}$$

or, simply, $\mathrm{succ}(x) \triangleright \mathsf{reply}\ x + 1\ \mathsf{to}\ \mathrm{succ}$. As regards the types of synchronous names, we do not refrain from the temptation of defining

$$\langle \tau_1, \ldots, \tau_q \rangle \rightarrow \langle \tau'_1, \ldots, \tau'_p \rangle \quad \stackrel{\text{def}}{=} \quad \langle \tau_1, \ldots, \tau_q, \langle \tau'_1, \ldots, \tau'_p \rangle \rangle$$

Hence, the type of succ can eventually be written $\langle \mathtt{Int} \rangle \rightarrow \langle \mathtt{Int} \rangle$.

3.4.2 Two evaluation strategies of the λ-calculus

Before hiding continuations under syntactic sugar, we present two encodings of the λ-calculus. For a given CPS, we encode λ-terms as named guarded processes and we relate their respective behavior. Our purpose here is to illustrate the tight connection between functions and join-definitions. The grammar for the λ-calculus is as usual:

$$T \quad \stackrel{\text{def}}{=} \quad x \mid \lambda x.T \mid TT$$

Encodings of the λ-calculus have been previously studied in detail in similar π-calculus settings [98, 37, 132]. These encodings use only a small fragment of the π-calculus, where every communication complies with the restrictions that are syntactically enforced in the join-calculus. Thus, we believe that their adequacy results carry over the join-calculus, where they are stated in a simpler setting. For instance, we easily check that the terms and their translations converge or diverge accordingly, as is the case in [98]. We postpone any formal development to future work.

Call-by-name In this reduction strategy, λ-terms are reduced in leftmost-order and no reduction may occur under a λ. Our encoding is:

$$\begin{aligned}
[\![x]\!]_v &\overset{\text{def}}{=} x\langle v\rangle \\
[\![\lambda x.T]\!]_v &\overset{\text{def}}{=} \mathsf{def}\ \kappa\langle x,w\rangle \triangleright [\![T]\!]_w\ \mathsf{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v &\overset{\text{def}}{=} \mathsf{def}\ x\langle u\rangle \triangleright [\![U]\!]_u\ \mathsf{in}\ \mathsf{def}\ w\langle\kappa\rangle \triangleright \kappa\langle x,v\rangle\ \mathsf{in}\ [\![T]\!]_w
\end{aligned}$$

Intuitively, the process $[\![T]\!]_v$ sends its value on v, a value is represented as a process abstraction that serves evaluation requests sent on κ; and evaluation requests supply two names: x to send requests for the value of the argument, and w to eventually return a value when evaluation converges.

The image of the translation is exactly the *deterministic subset* of the join-calculus, defined as the set of processes that contain no parallel composition, and neither join-pattern nor "$\wedge$" in definitions. As expected, reductions for processes in this subset are entirely sequential.

To check the operational correspondence, we detail the mechanism of a β-reduction. The same reduction would occur in any translated evaluation context of the λ-calculus.

$$\begin{aligned}
[\![(\lambda x.T)U]\!]_v &= \mathsf{def}\ x\langle u\rangle \triangleright [\![U]\!]_u\ \mathsf{in} \\
&\quad\ \mathsf{def}\ w\langle\kappa\rangle \triangleright \kappa\langle x,v\rangle\ \mathsf{in} \\
&\quad\ \mathsf{def}\ \kappa\langle x,w\rangle \triangleright [\![T]\!]_w\ \mathsf{in} \\
&\quad\ w\langle\kappa\rangle \\
&\to\to \mathsf{def}\ x\langle u\rangle \triangleright [\![U]\!]_u\ \mathsf{in}\ [\![T]\!]_v \mid P
\end{aligned}$$

The process P contains the definitions of w and κ; it is inert, and would be discarded by any equivalence on processes ($P \sim 0$). The behavior of the resulting process is similar to the behavior of the translation of the resulting λ-term $[\![T\{^U/_x\}]\!]_v$: each occurrence of $[\![x]\!]_z$ reduces in a single step to an instance of $[\![U]\!]_z$. Anticipating on the equivalences defined in the next chapters, we easily establish that weak bisimulation is preserved through β-reduction.

Parallel call-by-value In this reduction strategy, the λ-term (TU) can be reduced as soon as both T and U have been reduced to values, but the function and the argument can be evaluated in parallel. Again, no reduction may occur under a λ. Using a larger subset of the join-calculus, we encode this confluent but non-deterministic reduction strategy as follows:

$$\begin{aligned}
[\![x]\!]_v &\overset{\text{def}}{=} v\langle x\rangle \\
[\![\lambda x.T]\!]_v &\overset{\text{def}}{=} \mathsf{def}\ \kappa\langle x,w\rangle \triangleright [\![T]\!]_w\ \mathsf{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v &\overset{\text{def}}{=} \mathsf{def}\ t\langle\kappa\rangle \mid u\langle w\rangle \triangleright \kappa\langle w,v\rangle\ \mathsf{in}\ [\![T]\!]_t \mid [\![U]\!]_u
\end{aligned}$$

Again, the encoding $[\![T]\!]_v$ sends its value on v and a value is a process that serves evaluation requests sent on κ, but evaluation requests now supply the value of the parameter along with a name for the value of the term.

The image of the translation uses parallel composition to capture the non determinism of the strategy. The symmetry between the evaluation of the function and of the argument is made apparent, backed by the two symmetries, on the fork of evaluation requests and on the join of their results.

P	$::=$		processes
		$v\langle v_1, \ldots, v_n\rangle$	message
	$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	local definition
	$\mid$	$P \mid P$	parallel composition
	$\mid$	$\mathsf{0}$	null process
	$\mid$	$\mathsf{let}\ x_1, \ldots, x_m = v\langle v_1, \ldots v_n\rangle\ \mathsf{in}\ P$	synchronous call
	$\mid$	$\mathsf{reply}\ v_1, \ldots, v_n\ \mathsf{to}\ x$	synchronous reply
τ	$::=$		type
		b	primitive type
	$\mid$	α	type variable
	$\mid$	$\langle \tau_1, \ldots, \tau_n\rangle$	asynchronous channel type
	$\mid$	$\langle \tau_1, \ldots, \tau_n\rangle \rightarrow \langle \tau'_1, \ldots, \tau'_m\rangle$	synchronous channel type

Figure 3.3: Extended syntax with synchronous names

3.4.3 Synchronous names

We now supplement the join-calculus with primitive sequencing constructs. To this end, port names are partitioned in two families according to their calling conventions: *synchronous names* and *asynchronous names*. We emphasize that a name is synchronous by using a different font: we write x instead of x, and foo instead of *foo* for synchronous names.

Asynchronous names are defined and used for asynchronous messages as before; in addition, synchronous names transmit an implicit continuation within every message and every join-pattern. Whenever a message is sent to a synchronous name, a continuation channel is defined as the remaining part of the current instruction sequence, and the continuation is added to the message. The synchronous invocation of a name f is performed by the new process

$$\mathsf{let}\ x_i\ ^{i\in 1..p} = \mathrm{f}(u_j\ ^{j\in 1..q})\ \mathsf{in}\ P$$

Conversely, whenever such a message is received as part of a join pattern, the continuation is bound in the corresponding guarded process, and may be used once to send back results. The asynchronous invocation of a continuation attached to f is performed by the process

$$\mathsf{reply}\ u_i\ ^{i\in 1..p}\ \mathsf{to}\ \mathrm{f}$$

We give the grammar for processes and types in a calculus extended with synchronous calls in Figure 3.3. Patterns, clauses and typing environments are defined as before. We let the sequencing operator ".; ." abbreviate the let-binding $\mathsf{let}{=}\ .\ \mathsf{in}\ .$ obtained for $m = 0$. We extend the precedence rules for let-binding and sequencing in a different manner, though: ";" binds tighter than "|", which binds tighter than the two binding constructs $\mathsf{let} \ldots = \cdot\ \mathsf{in}\ \cdot$ and $\mathsf{def} \ldots \mathsf{in}\ \cdot$.

We provide a new functional type constructor $\langle \tau_1, \ldots, \tau_n\rangle \rightarrow \langle \tau'_1, \ldots, \tau'_m\rangle$ for synchronous names, and we supplement the typing rules of Figure 3.2 with specific rules for the new constructs.

$$\textsc{(Let-Val)}\quad \frac{A \Vdash u : \langle \tau_j{}^{j\in 1..q}\rangle \rightarrow \langle \tau'_i{}^{i \in 1..p}\rangle \qquad (A \Vdash u_j : \tau_j)^{j \in 1..q} \qquad A + (x_i : \tau'_i)^{i\in 1..p} \Vdash P}{A \Vdash \mathsf{let}\ x_i{}^{i\in 1..p} = u\langle u_j{}^{j \in 1..q}\rangle\ \mathsf{in}\ P}$$

$$\textsc{(Reply)}\quad \frac{A \Vdash u : \langle \tau_j{}^{j\in 1..q}\rangle \rightarrow \langle \tau'_i{}^{i \in 1..p}\rangle \qquad (A \Vdash u_i : \tau'_i)^{i \in 1..p}}{A \Vdash \mathsf{reply}\ u_i{}^{i\in 1..p}\ \mathsf{to}\ u}$$

The resulting type system guarantees that synchronous and asynchronous invocations on the same name do not mix. Moreover, a user-defined name x must be invoked synchronously when its definition includes type-consistent occurrences of the $\mathsf{reply}\ u_i{}^{i\in 1..p}\ \mathsf{to}\ x$ construct.

While in theory we distinguish two classes of names, the asynchronous or synchronous status for all defined names may also be determined by typing, with the following convention: every name whose synchronous usage is not detected by the type inference system is considered asynchronous. Similarly, our type system does not prevent multiple invocations of the same continuation, which is easily ruled out by a compile-time verification.

3.4.4 A typed CPS encoding

As usual for process calculi, we can translate functional names back into the plain join-calculus. Our translation applies on well-typed programs once synchronous names have been identified.

We assume that the names introduced by the translation do not mix with the names in the source programs: we use the reserved name κ for intermediate continuations generated while translating let constructs; we assume that there is an injective function between synchronous names f and asynchronous continuation names κ_f that do not appear in the source program.

The call-by-value translation $[\![\cdot]\!]$ is defined by structural induction on the syntax; we mention only the clauses that modify the terms.

- in join-patterns, we add a continuation to any synchronous message:

$$[\![\mathrm{f}(u_i{}^{i\in 1..p})]\!] \quad \stackrel{\text{def}}{=} \quad f\langle u_i{}^{i\in 1..p}, \kappa_f\rangle$$

- in processes guarded by a join-pattern that defines f, we translate:

$$[\![\mathsf{reply}\ u_i{}^{i\in 1..p}\ \mathsf{to}\ \mathrm{f}]\!] \quad \stackrel{\text{def}}{=} \quad \kappa_f\langle u_i{}^{i\in 1..p}\rangle$$

- in let-binding processes, we define and send a local continuation

$$\begin{aligned} & [\![\mathsf{let}\ x_i{}^{i\in 1..p} = \mathrm{f}(u_j{}^{j\in 1..q})\ \mathsf{in}\ P]\!] \\ \stackrel{\text{def}}{=} \quad & \mathsf{def}\ \kappa\langle x_i{}^{i\in 1..p}\rangle \triangleright [\![P]\!]\ \mathsf{in}\ f\langle u_j{}^{j\in 1..q}, \kappa\rangle \end{aligned}$$

- in types, we remove functional type constructors:

$$\langle \tau_j{}^{j\in 1..q}\rangle \rightarrow \langle \tau'_i{}^{i\in 1..p}\rangle \quad \stackrel{\text{def}}{=} \quad \langle \tau_j{}^{j\in 1..q}, \langle \tau'_i{}^{i\in 1..p}\rangle\rangle$$

Note that the encoding for both traditional functions—identified to synchronous names defined in a single rule with a single-message pattern—and for continuations provides guarantees on the usage of these names. For instance, a process that receives a name that encodes a function can use it only as a function; it cannot re-define the function, or even detect other calls to the function. Said otherwise, the substitution lemma holds for every functional definition. This would not be the case in the π-calculus, where more detailed types are required after translation to enforce the same guarantees [120, 82].

We now relate the typing properties through the translation: we translate the two additional typing rules component-wise and obtain

(TRANSLATED LET-VAL)

$$\frac{A \Vdash u : \langle \tau_j\ ^{j \in 1..q}, \langle \tau'_i\ ^{i \in 1..p} \rangle \rangle \qquad (A \Vdash u_j : \tau_j)\ ^{j \in 1..q} \qquad A + (x_i : \tau'_i)\ ^{i \in 1..p} \Vdash P}{A \Vdash \mathsf{def}\ \kappa\langle x_i\ ^{i \in 1..p}\rangle \triangleright P\ \mathsf{in}\ u\langle u_j\ ^{j \in 1..q}, \kappa\rangle}$$

(TRANSLATED REPLY)

$$\frac{A \Vdash u : \langle \tau_j\ ^{j \in 1..q}, \langle \tau'_i\ ^{i \in 1..p} \rangle \rangle \qquad (A \Vdash u_i : \tau'_i)\ ^{i \in 1..p}}{A \Vdash \kappa_u\langle u_i\ ^{i \in 1..p}\rangle}$$

These two rules are easily derived from the basic type system of Figure 3.2: the translation of rule LET-VAL is obtained from DEF, RULE, and MESSAGE; the translation of rule REPLY is obtained from MESSAGE. Hence, the translation preserves valid typing derivations. (However, the translation may enable valid type derivations on ill-typed programs that confound explicit and implicit CPS.)

3.4.5 Toward a concurrent functional language

Now that we have added synchronous names and their primitives, further extensions of the syntax toward a convenient high-level language are mostly a delicate matter of taste. For instance, we can proceed as in our implementation, where we basically adopt (a subset of) the syntax of ML supplemented with join-patterns in functional definitions, and with forks in expressions. This smoothly integrates some concurrent programming with a functional framework, in declarative style. As observed in the design of PICT [122], this integration is pragmatically important, because even distributed programs tend to be mostly functional. In the same manner, external libraries can be embedded provided that they have a functional API, which can be turned into a typed interface.

A complete grammar for processes and expressions is proposed in Figure 3.4, in the spirit of our prototype language [59]. Clauses and definitions are unchanged; they are omitted from the figure. Again, expressions are only a convenient syntactic sugar, which can be removed. This new translation amounts to introducing explicit bindings of the kind of the previous section for all subexpressions, nested calls being translated top-down, left-to-right.

$$\begin{array}{rcl}
\mathsf{reply}\ E_i\ ^{i \in 1..p}\ \mathsf{to}\ \mathsf{f} & \stackrel{\text{def}}{=} & \kappa_f\langle E_i\ ^{i \in 1..p}\rangle \\
u\langle E_i\ ^{i \in 1..p}\rangle & \stackrel{\text{def}}{=} & (\mathsf{let}\ x_i = E_i\ \mathsf{in})\ ^{i \in 1..p}\ u\langle x_i\ ^{i \in 1..p}\rangle \\
\mathsf{let}\ x = u\ \mathsf{in}\ P & \stackrel{\text{def}}{=} & P\{^{u}/_{x}\} \\
\mathsf{let}\ x_i\ ^{i \in 1..p} = \mathsf{f}(E_j\ ^{j \in 1..q})\ \mathsf{in}\ P & \stackrel{\text{def}}{=} & \mathsf{def}\ \kappa\langle x_i\ ^{i \in 1..p}\rangle \triangleright P\ \mathsf{in}\ f\langle E_j\ ^{j \in 1..q}, \kappa\rangle
\end{array}$$

$$\begin{array}{lcll}
P & ::= & & \text{processes} \\
 & & v\langle E_1, \ldots, E_n\rangle & \quad\text{asynchronous message} \\
 & | & \mathsf{def}\ D\ \mathsf{in}\ P & \quad\text{local definition} \\
 & | & P \,|\, P & \quad\text{parallel composition} \\
 & | & \mathsf{0} & \quad\text{null process} \\
 & | & E; P & \quad\text{sequence} \\
 & | & \mathsf{let}\ x_1, \ldots, x_m = E\ \mathsf{in}\ P & \quad\text{synchronous call} \\
 & | & \mathsf{reply}\ E_1, \ldots, E_n\ \mathsf{to}\ x & \quad\text{synchronous reply} \\
\\
E & ::= & & \text{expressions} \\
 & & v\langle E_1, \ldots, E_n\rangle & \quad\text{synchronous call} \\
 & | & \mathsf{def}\ D\ \mathsf{in}\ P & \quad\text{local definition} \\
 & | & E; E & \quad\text{sequence} \\
 & | & \mathsf{let}\ x_1, \ldots, x_m = E\ \mathsf{in}\ E & \quad\text{synchronous call}
\end{array}$$

Figure 3.4: Syntax for a language with processes and expressions

In practice, we would introduce new typing judgments for expressions ($A \Vdash E : \tau_i\ ^{i\in 1..p}$), along with new typing rules. The typing rules for expressions are derived from the previous ones and are omitted.

3.4.6 Types and side effects

If we remove join-composition in patterns and parallel-composition in processes from our extended language, we get a polyadic functional kernel similar to core-ML: both the reductions and the typing rules do correspond. Let us consider in detail how we would translate the `let` binder of ML. According to the `let`-bound expression, there are two cases with distinct typing properties. When the syntax suffices to identify functions either directly or as aliases, we use a generalizing definition:

$$\begin{array}{rcl}
[\![\mathsf{let}\ \mathrm{f}(x) = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{def}\ \mathrm{f}(x) \triangleright \mathsf{reply}\ e_1\ \mathsf{to}\ \mathrm{f}\ \mathsf{in}\ e_2 \\
[\![\mathsf{let}\ \mathrm{g} = \mathsf{let}\ \mathrm{f}(x) = e_1\ \mathsf{in}\ \mathrm{f}\ \mathsf{in}\ e_2]\!] & = & \mathsf{def}\ \mathrm{f}(x) \triangleright \mathsf{reply}\ e_1\ \mathsf{to}\ \mathrm{f}\ \mathsf{in}\ e_2\{^{\mathrm{f}}/_{\mathrm{g}}\}
\end{array}$$

For other values such as the results of function calls, we use a continuation message to convey the result, which forces this result to be monomorphic. Hence, polymorphism is made available only on syntactic values, which is equivalent to Wright's restriction for ML [151].

The language as a whole is more expressive than ML; it provides support for general, concurrent programming, including imperative constructs and side effects as messages. For instance, reference cells need not be taken as primitives; they are programmable in the join-calculus using the following rule

$$\mathrm{mkcell}(v_0) \triangleright \left(\begin{array}{lrl}
\mathsf{def} & \mathrm{get}() \,|\, s\langle v\rangle \triangleright & \mathsf{reply}\ v\ \mathsf{to}\ \mathrm{get} \,|\, s\langle v\rangle \\
 & \wedge\ \ \mathrm{set}(u) \,|\, s\langle v\rangle \triangleright & \mathsf{reply}\ \mathsf{to}\ \mathrm{set} \,|\, s\langle u\rangle \\
\mathsf{in} & \multicolumn{2}{l}{\mathsf{reply}\ \mathrm{get}, \mathrm{set}\ \mathsf{to}\ \mathrm{mkcell} \,|\, s\langle v_0\rangle}
\end{array}\right)$$

The mutable content of the cell is represented as the contents of a message on name s, which is consumed and updated each time the cell is accessed or changed through the functional names get or set. (The translation of this rule in the pure join-calculus has been presented and explained in Section 2.4.6.)

Using our type system, each instance of the reference cell is monomorphic, and the three inner defined names get : $\langle\rangle \to \langle\alpha\rangle$, set : $\langle\alpha\rangle \to \langle\rangle$, and $s : \langle\alpha\rangle$ are typable in an environment with a shared type variable α representing the contents of the cell. Any generalization here would break the assumption that they have been jointly defined and thus jointly typed. For instance the process

$$\mathsf{let}\ \mathrm{get}, \mathrm{set} = \mathrm{mkcell}(1)\ \mathsf{in}\ \mathrm{set}(\text{“world”}) \mid \mathrm{print\_int}(\mathrm{get}())$$

cannot be typed in our system, and may lead to a run-time error. Fortunately, some polymorphism can be recovered by the time the reference cell is allocated. Since a single name mkcell is being defined, its type $\langle\alpha\rangle \to \langle\langle\rangle \to \langle\alpha\rangle, \langle\alpha\rangle \to \langle\rangle\rangle$ can obviously be generalized on α. And thus mkcell can be used polymorphically in the main body of its definition, where several cells containing values of incompatible types can safely be accessed by using different functional names, as is the case in the process

$$\begin{array}{ll} & \mathsf{let}\ \mathrm{get}, \mathrm{set} = \mathrm{mkcell}(1)\ \mathsf{in}\ \mathrm{print\_int}(\mathrm{get}()) \\ \mid & \mathsf{let}\ \mathrm{get}, \mathrm{set} = \mathrm{mkcell}(\text{“hello”})\ \mathsf{in}\ \mathrm{print\_string}(\mathrm{get}()); \mathrm{set}(\text{“world”}) \end{array}$$

More generally, join-calculus definitions may describe protocols that involve sophisticated synchronization of numerous methods and/or partial states, but this is largely independent of the typing, as long as side effects are tracked using the sharing of type variables.

This is in contrast with the classical approach in ML, where references are introduced in a “pure” language as dangerous black boxes that cannot be given polymorphic types, and that communicate with a global store by magic.

In some recent expositions of imperative constructs [152], references are introduced as local stores that can be extruded, which is closer to the join-calculus, but again references are a new special construct. On the contrary, the store can be identified as some part of the chemical machine that consists of the instances of cell definitions on the left-hand-side, and of their state messages on the right-hand-side, but there is no reason to do so for typing. Our approach is uniform and allows us to type at least as much as ML with references allocated by a primitive `ref` constructor.

3.5 Concurrent objects as join-definitions

Informally, name-passing calculi correspond to the message-passing interpretation of objects in a concurrent setting, which is widely used to model distributed object-oriented programming [32, 11]. Actually, some asynchronous variants of the π-calculus have been introduced as abstract models for concurrent objects rather than as process calculi [71, 70], and this analogy has been formalized for the π-calculus and its variants [148].

As is the case for functions, objects are usually not primitive in process calculi, but they can be either added to the calculus, or internally encoded. Accordingly, several object-oriented idioms have been recently expressed in a process calculus setting [74, 136, 121, 76, 147] either as pure encodings or as hybrid calculi, typically with primitive extensible records [49] or with an object-calculus kernel [63].

As a process calculus, the join-calculus provides the essential features of objects. We first describe the kind of objects that can already been expressed in the join-calculus. Using message-passing and pattern-matching in our definitions, we encode objects as servers that receive requests to execute their methods. Nonetheless, the design of a full-fledged object-oriented language would require some extensions. For instance, inheritance (or cloning) is not primitive, and there is no convenient data-type to represent objects as extensible records of methods. We sketch some features to support more general objects with dynamic definitions and inheritance. These extensions can of course be encoded on top of the join-calculus, but it may be interesting to check how they interact with types. In this section, we mostly present direction for future work; there is no formal treatment of the proposed extensions of the calculus.

3.5.1 Primitive objects

We interpret the join-calculus with functions in an object-oriented manner. We identify synchronous names and methods, definitions and concurrent objects.

Objects are created in definitions, whose port names may be either returned and made public, or kept private in the body of their definition. In that sense, our cell example is a simple imperative object with two methods get and set, and a current state—the contents of the cell—held on a message on the internal name s. More generally, the current state of an object can be split into several components held on internal messages, according to the critical sections. Besides, the interface may feature several states with different synchronization capabilities. The resulting objects supports rich synchronization capabilities, which may for instance involve several states of their interface. As regards control, the declarative pattern-matching on join messages is richer than a fixed control strategy, such as the serialization of method calls.

We illustrate the combination of concurrency and synchronization with a larger example of concurrent object: the *priority queue*. The definition of a constructor for a queue that is initially empty may be:

```
mk_priority_queue() ▷
 def  empty() | none⟨⟩ ▷
           {none⟨⟩ | reply true to empty}
  ∧   empty() | some⟨x, e, a, r⟩ ▷
           {some⟨x, e, a, r⟩ | reply false to empty}
  ∧   add(x) | none⟨⟩ ▷
           {reply to add | let e, a, r = mk_priority_queue() in some⟨x, e, a, r⟩}
  ∧   add(x) | some⟨y, e, a, r⟩ ▷
           {reply to add | a(max(x, y)); some⟨min(x, y), e, a, r⟩}
  ∧   remove() | some⟨x, e, a, r⟩ ▷
           {reply x to remove | if e() then none⟨⟩ else some⟨r(), e, a, r⟩}
 in
      none⟨⟩ | reply empty, add, remove to mk_priority_queue
```

The queue as a value is represented by a tuple of three synchronous methods, with the following types and intuitive meanings:

empty : $\langle\rangle \rightarrow \langle$`Bool`$\rangle$ checks whether the queue is empty;

add : $\langle \alpha \rangle \rightarrow \langle\rangle$ inserts a new value in the queue and (always) returns; and

remove : $\langle\rangle \rightarrow \langle \alpha \rangle$ retrieves the smallest value present in the queue, if any, or blocks until a value is inserted otherwise.

There are two internal states, $\mathit{none}\langle\rangle$ when empty, and $\mathit{some}\langle x, \mathrm{e}, \mathrm{a}, \mathrm{r}\rangle$ when containing the smallest value x in its head and another priority queue with methods $\mathrm{e}, \mathrm{a}, \mathrm{r}$ in its tail. Statically, we can check that there is always exactly one state message available for each definition. Values can be concurrently tested, added, and removed; in particular, a new message $\mathit{some}\langle\cdots\rangle$ is released after at most one comparison when a new value is added, while the update propagates toward the tail in parallel. When the tail is eventually reached, a new, empty priority queue is created using the recursive definition mk_priority_queue, which returns three fresh methods $\mathrm{e}, \mathrm{a}, \mathrm{r}$ on an empty priority queue to be stored in the message $\mathit{some}\langle\cdots\rangle$ of the last-but-one component of the concurrent queue. It is easy to check that the priority queue enforces the expected properties, e.g., that series of values that have been added are removed in increasing order.

This example of concurrent data structure is simple, but not very efficient, since for instance insertion is linear on the size of the queue. Possible enhancements would be an indirection in every cell—so that we can add a small value without traversing the whole queue, and the use of a tree structure instead of a linear one—so that the number of nodes being traversed is kept small for long queues. This would use more complicated definitions, with the same join-synchronizations. More examples of data structures are described in the documentation of our implementation, in particular in the first part of the tutorial [59].

More generally, several standard strategies for synchronization within an object can be encoded as multi-clause definitions. We briefly give the generic shape of such definitions.

A partially-serialized object is defined as

$$\bigwedge_{\mathrm{m}} \left(\mathrm{m}(\widetilde{v}) \mid \prod_{s \in \mathrm{locks}(\mathrm{m})} s\langle \widetilde{u}_s \rangle \triangleright \begin{array}{l} \mathsf{let}\ \widetilde{w}, \widetilde{u}'_s = E\ \mathsf{in} \\ \prod_{s \in \mathrm{locks}(\mathrm{m})} s\langle \widetilde{u}'_s \rangle \mid \mathsf{reply}\ \widetilde{w}\ \mathsf{to}\ \mathrm{m} \end{array} \right)$$

where m ranges over the methods, the $\widetilde{u}_s$ partition the state, and each method grabs locks on all the required parts of the state, executes its body, then returns and releases all the locks, possibly mutating parts of the state.

More concurrency can be achieved by releasing parts of the locks before the end of the computation. For instance, the method m may be defined by the rule

$$\mathrm{m}(x) \mid s_0\langle y \rangle \mid s_1\langle z \rangle \triangleright s_0\langle y \rangle \mid s_1\langle \mathrm{f}(x, y) \rangle \mid \mathsf{reply}\ z\ \mathsf{to}\ \mathrm{m}$$

where the first part of the state is unchanged, and can be immediately released, while the second part is released only as the computation of $\mathrm{f}(x, y, z)$ returns. Even as each method call may return at once, concurrent calls on m will be deferred until the second part of the state is made available again. Conversely, concurrent calls to other methods that do not require access to y may still occur in parallel.

An object with several states and transitions between states at method invocation is defined as

$$\bigwedge_{\mathrm{m}} \left(\mathrm{m}(\widetilde{v}) \mid \prod_{s \in \mathrm{pre}(\mathrm{m})} s\langle \widetilde{u}_s \rangle \triangleright \begin{array}{l} \mathsf{let}\ \widetilde{w}, \widetilde{z}_{s'} = E\ \mathsf{in} \\ \prod_{s' \in \mathrm{post}(\mathrm{m})} s'\langle \widetilde{z}_{s'} \rangle \mid \mathsf{reply}\ \widetilde{w}\ \mathsf{to}\ \mathrm{m} \end{array} \right)$$

where the transition (meta) functions pre and post specify the state transition in terms of resources being consumed and released.

Some objects may also provide a finer control on invocation of several methods. Extending the multi-function paradigm of [23], such objects would feature multi-methods that have to be jointly triggered, and thus guarantee atomicity in simultaneous method invocations, or even multiple copies of the state messages, accounting for a finite number of resources being concurrently available.

Our approach is more declarative than the traditional object-as-server encoding, where there is a single port name where all method calls are sent, and where the decoding of methods is entirely dynamic and imperative, as is the case with Actors for instance.

3.5.2 Values, classes and inheritance

So far, our calculus lacks convenient values that can encapsulate the tuple of methods representing our objects. In order to integrate objects in the join-calculus, this strongly suggests that we supplement values with extensible records. Accordingly, we would obtain a pure object-oriented calculus by changing our naming scheme from port names x defined in $\mathsf{def}\ D\ \mathsf{in}\ \cdot$ into labels x applied to named definitions $\mathsf{def}\ o = D\ \mathsf{in}\ \cdot$. The synchronization mechanism and the scoping properties remain the same, yet we need to implement anonymous method invocation, and provide subtyping that enforce restricted access to the internal "methods" hosting the state.

Besides, our primitive objects are very static, because the lexical scope of their definitions forbids any extension of the synchronization patterns on an existing object, and because definitions as a whole are not first-class values that can be overloaded or cloned.

It is well-known that inheritance and synchronization for concurrent objects do not merge gracefully [92]. In our case, we can easily express synchronization, but there is no support for inheritance.

We can recover some dynamicity using indirections; pre-methods are stored in an extended state, and method update are just state updates. Likewise, one can substitute state overwriting for method overriding in many cases, and mix freely static and dynamic components within the same objects.

We can also adapt the more sophisticated approach to typed objects that has been proposed for ML in [126, 127]. Again, we believe that the locality property would enable a smooth integration of concurrency. Should we adopt a class-based inheritance mechanism, for instance, we obtain a static mechanism to assemble fragments of definitions, which seems useful from a programming language viewpoint.

3.6 Related type systems for concurrent languages

While parametric polymorphism suffices to our present needs, more sophisticated types systems have been developed as a way to capture additional static properties of processes. We briefly survey some of these type systems and their additional benefits.

3.6.1 Typing communication patterns

In the area of name-passing process calculi, the first step was taken by Milner in [99]. Milner introduces an improvement of the π-calculus—the *polyadic π-calculus*—where channels are allowed to carry tuples of messages. Polyadicity naturally supports a concept of *recursive sorting*. Maintaining the sort discipline enforces channels to always carry tuples of the same length and nature. Recursive sorts correspond to the recursive-monomorphic variant of our system.

The first extension of Milner's system has been undertaken by Pierce and Sangiorgi, who distinguish between input-only, output-only, and input-output channels. This extension naturally leads to recursive types with subtyping [120]. Since then, more and more elaborate extensions have been proposed and experimented, mostly around the PICT language [122, 145]. For instance, further extensions capture linearity information in channel types [82]. This provides a finer account on communication patterns, and static type inference leads to a more efficient compilation.

The type systems of all these authors are usually more sophisticated than ours. Some of this sophistication is due to the higher dynamicity of the π-calculus semantics and is thus irrelevant in our case. Nevertheless, it would be interesting to refine our basic type system to include some of these elaborate static analysis such as deadlock or linearity analysis, and to make use of this information in an optimizing join-calculus compiler.

The basic theory of polymorphic extensions of Milner's sort discipline for π-calculus has been developed by Turner in his dissertation [145]. We recall that Turner's polymorphism is *explicit*: inputs and outputs are always annotated with sorts. For example, the π-calculus process $\overline{x}[y,z] \mid x[u,w].\,\overline{u}[w]$ is tagged as follows:

$$\overline{x}[\mathtt{Int}; y, z] \mid x[\alpha; u :\uparrow \alpha, w : \alpha].\,\overline{u}[\alpha; w]\ .$$

Consequently, explicit abstraction and application of types are interleaved with communication: in the above example the sort α in the output $\overline{u}[\alpha; w]$ depends on the message received on the channel x. This commitment to explicit polymorphism in π-calculus follows from the absence of a place where sort generalization may occur.

3.6.2 Implicit polymorphism and control

Typing à la ML for concurrent languages is not new; proposals have been defined for languages that combine functional and concurrent primitives in a symmetric manner, such as for instance Concurrent ML [128] and FACILE [144]. In these languages channels are always monomorphic, and polymorphism is only allowed under functional abstractions. An analogous approach has been taken by Vasconcelos for an extension of π-calculus with agent names [146]. In his setting, channels can be generalized when they precisely encode functions (a single, replicated receiver). Processes can thus be parameterized by arguments of different types. However, two processes can never communicate values of different types over the same channel, which restricts the expressiveness of the language. In particular, it is impossible to implement polymorphic services, such as for instance a remote authentication server that would take any piece of data, sign it, and return its signed certificate, and that would also check such certificates upon request from a third party.

As a simplified example, we consider a computing server

$$\mathsf{def}\ run\langle f,x,r\rangle \triangleright f\langle x,r\rangle\ \mathsf{in}\ \ldots$$

where r*un* is a channel—not a function—meant to accept remote evaluation requests, run them locally, then return the answer on a continuation. In our type system, the name run can be given the generalized type $\forall\alpha,\beta.\langle\langle\alpha,\langle\beta\rangle\rangle,\alpha,\langle\beta\rangle\rangle$.

Of course, this channel is much like a function, but the programmer may hesitate between functions or channels for expressing sequential control; he may also have to choose channels because the control structure gets more involved, or because functions and channels do not have the same properties in a distributed setting.

For instance, we can alter our pure rendez-vous multi-function to allow for some more functional style; we would write

```
def sync_1(a) | sync_2(b) ▷
   reply a to sync_1 |reply b to sync_2 in
... f(sync_1(...)) ... | ... g(sync_2(...)) ...
```

where the two "identities" sync_1 and sync_2 can both be given the generalized type $\forall\alpha.\langle\alpha\rangle\rightarrow\langle\alpha\rangle$, and still provide additional control as a side effect: the results of the inner computations are communicated to f and g only when both computations have completed.

The following program models a scheduler, and illustrates a case where polymorphism and control get intertwined:

```
def job(f, x) | token(n, z) ▷
   let result = f⟨x, n, z⟩ in
   token⟨n + 1, z⟩ |reply result to job in
token⟨1, "red"⟩ | token⟨1, "blue"⟩ | job⟨f_1, 12⟩ | job⟨f_2, "join.inria.fr"⟩
```

The scheduler provides some control over the degree of parallelism of a computation; there are at most two jobs running concurrently; moreover, these jobs are labeled with a color "red" or "blue"—standing for some more useful resource—and with a serial number. The name *token* that represents the state of our controller is kept local, while the access method job is made available to the context. Running our type-inference algorithm, we obtain the types

$$\begin{aligned}\mathrm{job} &: \forall\alpha\beta.\langle\langle\alpha,\mathtt{Int},\mathtt{String}\rangle\rightarrow\langle\beta\rangle,\alpha\rangle\rightarrow\langle\beta\rangle\\ token &: \langle\mathtt{Int},\mathtt{String}\rangle\end{aligned}$$

We might still implement job as a function (but not as a channel) by representing the state in an imperative manner, or by playing a continuation game to separate values from control. Yet, there is no simple way to define such a service in CML, Facile, or the language proposed in [146]. In fact, this limitation has been known in CML. A natural solution would be to use first-order, explicit *existential types* such as in [85]. Then after CPS, the channel run could be given the monomorphic type $\exists\alpha,\beta.\langle\langle\alpha,\langle\beta\rangle\rangle,\alpha,\langle\beta\rangle\rangle$. The translation of the example in PICT would give run a similar type.

Chapter 4

Equivalences and Proof Techniques

This chapter and the next one are devoted to the study of equivalences for the join-calculus. In this chapter we focus on reduction-based equivalences; in the next chapter we depart from reductions to consider labeled transitions and develop purely bisimulation-based proof techniques.

Now that we have settled the operational semantics of the join-calculus and explored its expressiveness, we need tools to state and prove the properties of distributed programs. To this end, we must equip our calculus with equivalence relations that have both a sensible discriminating power and some convenient proof techniques. As an extreme example, syntactic equality is immediate to check but it separates numerous processes that have the same behavior; structural equivalence ($\equiv$) is also easily checked—for instance by computing unique normal forms—and equates terms that are intuitively equivalent, but still discriminates too much.

Our main goal is to use these equivalences to relate distributed programs written in the join-calculus or in its extensions. A more specific issue is to discuss the properties of the implementation, and various program transformations performed by the compiler or the run-time system. From a more theoretical point of view, these equivalences provide a deeper understanding of the calculus, and they yield a formal basis to compare variants of the join-calculus to other formalisms, in particular to the π-calculus.

The notions of equivalence that we use are hardly new; they draw upon a large body of theoretical work on process algebra and process calculi; they also have been successfully applied to a broad range of practical issues, such as automated or computer-aided verification of concurrent systems and distributed protocols. Actually, there are numerous proposals for the "right" equivalence for concurrent processes—see for instance [61] for an impressive overview. Choosing the proper equivalence with which to state a correctness argument often means striking a delicate balance between a simple, intuitively compelling statement, and a manageable proof.

For instance, there are many effective, sometimes automated techniques for proving bisimulation-based equivalences. Nonetheless, it can be quite hard to prove that two processes are *not* bisimilar—and to interpret this situation—because bisimulation may not directly correspond to the operational model, and may fail to capture some subtle identities that are key in a protocol. On the contrary, it can be quite hard to prove a coarser, testing equivalence, but the proof that two processes are not testing equivalent is simply a failure scenario. In practice, the proper choice is likely to vary according to the problem being studied. Besides, several related equivalences may be useful for tackling the same problem, as indirect proof techniques or intermediate lemmas rely on finer equivalences than those of the main results.

In this chapter, we pick a few significant equivalences that are used in the dissertation, we describe their main features, and we establish their connections. We are mostly interested in reduction-based, weak equivalences, which is not the traditional approach in process calculi; besides, specific problems arise as we deal with asynchronous, mobile calculi. To our knowledge, there is no guideline available so far in this setting.

Pragmatically, we arrive at a simplified hierarchy of equivalences, where each tier introduces a useful tradeoff between expressiveness and ease of proof, and each tier can have several, sometimes very different formal characterizations. The four main tiers are, with increasing discriminative power,

- may testing
- fair testing (and coupled simulations congruences)
- barbed bisimulation congruences
- labeled bisimulations

In this framework, one can start a proof effort at the upper tier with a simple labeled bisimulation proof; if this fails, one can switch to a coarser equivalence by augmenting the partial proof—typically by considering more processes and reductions; if the proof still fails for the testing equivalences in the last tiers, then at least meaningful counter-examples can be found.

Our main technical results relate definitions stated in different styles: trace-based equivalences versus bisimulation-based equivalences, labeled semantics versus reduction semantics, fairness conditions versus coupled simulations. Several results—or, more to the point of the comparison, the corresponding results for the π-calculus—are new. Specifically, some of the standard definitions of equivalences are related by unexpected identities, some of which close conjectures of Milner and Sangiorgi [101], and Honda and Yoshida [73]. We mention similar results for the π-calculus at the end of the chapter, but we refer to [56] for a detailed analysis of reduction-based equivalences centered around the asynchronous π-calculus and for the corresponding proofs. The direct comparison of the join-calculus and the π-calculus is deferred till Chapter 6.

Contents of the chapter

Before discussing technical subtleties, we spend some time to sketch a general picture and to motivate our choices. In Section 4.1 we describe the general setting of reduction-based systems and we discuss observation predicates and congruence properties. The three next sections detail the first three tiers of our hierarchy of equivalences. In Section 4.2 we begin our study of testing semantics. In Section 4.3 we focus on *fair-testing*. In Section 4.4 we incorporate bisimulation requirements and we explain the issue of *barbed bisimulation congruence*. In Section 4.5 we propose *coupled-barbed simulations* as a coarser alternative to barbed bisimulation, and we relate it to fair-testing. In Section 4.6 we complete our overview of reduction-based equivalences by a general picture of our hierarchy that summarizes our results (page 123), and we also discuss the main differences with the π-calculus, for which the same programme leads to a roughly similar hierarchy of equivalences.

The two final sections contain more specific technical developments. In Section 4.7 we develop useful diagram-chasing techniques for bisimulations. In Section 4.8 we prove the coincidence of the two barbed bisimulation congruences.

4.1 Reduction-based semantics

In this section, we recall standard notions in reduction-based semantics. The general approach in concurrency is to isolate the process from its environment and to focus on their visible interaction. Ideally, observing a process should be the same as communicating with it. This is achieved through the definition of two notions that are common to most process calculi: a *reduction relation* $\rightarrow$ that represents internal evolution and an *observation predicate* $\downarrow_x$ that detects the ability of interacting at a given channel. Based solely on these two notions, numerous observational semantics can then be defined.

4.1.1 Abstract reduction systems

We begin with a very general definition of reduction-based systems, which we use in this chapter to minimize our dependence on specific syntaxes or semantics as we define our hierarchy of equivalences.

Definition 4.1 *An* abstract reduction system *(ARS) is a triple $(\mathcal{P}, \rightarrow, \downarrow_x)$, where $\mathcal{P}$ is a set of terms, $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ is a relation on terms, and $\downarrow_x$ is a family of predicates on terms.*

The reduction relation induces a few standard properties used in further discussions: a process P is *stable* when it has no reduction ($P \not\rightarrow$); a process has a divergent computation when it has unbounded sequences of reductions ($\forall n. P \rightarrow^n$); a process *diverges* when it has no finite sequences of reductions (not $P \rightarrow^* \not\rightarrow$); for example the join-calculus process $\mathsf{repl}\ 0 = \mathsf{def}\ \kappa\langle\rangle \triangleright \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle$ reduces to itself, and thus diverges.

The observation predicates $\downarrow_x$ are usually simple syntactic properties meant to detect the outcome of the computation, e.g., "success", convergence, deadlock. As usual, we adopt a suffix notation for predicates ($P\downarrow_x$). These predicates induce an equivalence and a preorder on terms: let $P, Q \in \mathcal{P}$ be two processes; P is equivalent to Q when they pass the same tests ($\forall x. P\downarrow_x$ iff $Q\downarrow_x$); P is smaller than Q when P passes less tests than Q ($\forall x. P\downarrow_x$ implies $Q\downarrow_x$). We refer to these relations when we say that an equivalence or a preorder is a refinement of the observation predicates.

The most studied reduction system is probably the λ-calculus, with $\beta\eta$-reduction as reduction, and syntactic properties on terms such as "is in normal form", or "is in head-normal form" as predicates [91]. As regards concurrency, process calculi such as CCS or the π-calculus can also be considered as reduction systems with processes as terms and silent (τ) transitions as reductions, even if their operational semantics is traditionally obtained from richer, labeled transitions systems; in this case, the observation predicates are immediate communication capabilities—also known as the *barbs* since Milner and Sangiorgi's paper on barbed bisimilarity [101]. Finally, every formalism with a chemical semantics induces a reduction system with chemical solutions as terms, and reduction up to chemical rearrangement ($\rightleftharpoons^* \longrightarrow \rightleftharpoons^*$) as reduction.

While ARS have little structure, the operational semantics usually provides a good idea of what immediate communication capabilities are. Besides, minor variations in the definition of observation predicates do not usually affect the resulting equivalences, and thus this style of definition is relatively independent of the syntactic details.

Our hierarchy of equivalences is defined for every ARS. More particularly, all definitions and most results carry over the numerous variants and refinements of the join-calculus that we consider in this work. In Chapter 6 for instance, we use this common framework to relate the expressive power of several calculi; we exhibit a translation from one language to the other, and we prove that this translation is fully abstract with regards to each instance of the same reduction-based equivalence. In the following, we try to maintain the distinction between properties that are generic, properties that are particular to asynchronous process calculi, and properties that are specific to the join-calculus.

With respect to transition systems (LTS), which are traditionally preferred for CCS and the π-calculus, purely reduction-based semantics lack extensional proof techniques and models. On the other hand, ARS are simpler and more intuitive, in particular in an asynchronous setting, or in the absence of name-testing; ARS also carry over higher-order settings such as the λ-calculus or higher-order variants of the π-calculus, which are often proposed to model mobile agents.

In this chapter, we argue in favor of reduction-based semantics for process calculi. Nonetheless, both approaches are complementary and it is worthwhile to obtain descriptions of the same calculus both in terms of reductions in context and of labeled transitions. When this is the case, more intuitive results can be stated, and easier proofs can be conducted with the support of labels. In general, little is known about how to infer good LTS from a given ARS; for instance it is hard to derive labels such that labeled bisimulation is a congruence (see [141] for a general approach to the problem). Even in specific settings, a precise correspondence may be difficult to achieve, since one wants to capture properties initially stated in a pure reduction-based setting. Two properties are especially interesting for the join-calculus; they are discussed further in the next chapter: how to accommodate asynchronous semantics? [73, 16, 35] (*cf.* Section 5.3); how to deal with the absence of name testing? [94, 36] (*cf.* Section 5.5).

4.1.2 What can be observed in the join-calculus

Back to the join-calculus, a natural manner to distinguish processes is to look at their basic interaction with the environment, namely emission on free names. Specifically, the first interaction between a process and an enclosing context only depends on the presence of pending messages sent on free names; in the absence of such messages, the process is chemically inert. We define our predicates accordingly:

Definition 4.2 *The basic observation predicate $\downarrow_x$, also known as the strong barb on x, detects whether a process emits on some nullary free name x:*

$$P \downarrow_x \quad \stackrel{def}{=} \quad \exists P', P \equiv P' \mid x\langle\rangle$$

Barbs are messages in their simplest form; they do not carry any contents; they are just "signals". As we shall see, the nullary restriction on barbs is not very important for the equivalences we use; the important point is that we do not separate messages sent on the same name but with different contents, such as $x\langle y\rangle$ and $x\langle z\rangle$.

Strong barbs may appear on free names as the result of reductions, but once a strong barb is present, it remains stable through subsequent reductions. This simplifying property is specific to the join-calculus; for instance it does not hold for the π-calculus, as discussed in Section 4.6.1.

The barbs detect only the superficial behavior of a process; a natural manner to refine them is to require some congruence property.

4.1.3 Contexts and congruence properties

As we study relations among processes, an important property of related processes is their dependence to the environment. More precisely, we are interested in equivalences that provide a behavioral account of processes: two processes are equivalent when no external observer can tell the difference between the two in any context. In practice, this congruence property is the key to modular proofs, where separate parts of protocols can be treated in separate lemmas. Hence, most of our equivalences are congruences, either by definition, or as an important property.

We recall the standard definitions of contexts and congruences, and we set a few notations. These definitions apply to each process calculus we consider, but they depend on each particular syntax and each notion of guarded process.

Definition 4.3 *A* context *is a function on processes, represented by a term of the grammar for processes extended with the special placeholder process* $[\cdot]$.

The context $C[\cdot]$ *maps every process* P *to the process* $C[P]$ *obtained by substituting* P *for the hole* $[\cdot]$.

An evaluation context *is a context where the hole* $[\cdot]$ *occurs exactly once, and not under a guard.*

Evaluation contexts describe environments that can communicate with the process being observed, can filter its messages, but can neither replicate the process (as the context $\mathsf{repl}\,[\cdot]$) nor prevent its internal reductions (as the context $\mathsf{def}\ x\langle\rangle \triangleright [\cdot]\ \mathsf{in}\ 0$). In the join-calculus, the only guard is the join pattern; evaluation contexts are thus defined by the grammar

$$E[\cdot] \quad ::= \quad [\cdot] \quad \Big| \quad P \,|\, E[\cdot] \quad \Big| \quad E[\cdot] \,|\, P \quad \Big| \quad \mathsf{def}\ D\ \mathsf{in}\ E[\cdot]$$

We use the notation $\mathcal{E}$ for the set of all evaluation contexts in the join-calculus. Since we usually consider processes up to structural equivalence, we can apply Remark 2.1 after application of a context, and thus restrict our attention to contexts of the form

$$E[\cdot] \;\equiv\; \mathsf{def}\ D\ \mathsf{in}\ M \,|[\cdot]$$

where D ranges over definitions and M ranges over parallel compositions of messages.

In CCS and in the π-calculus, evaluation contexts are often named static contexts [97]. Notice that parallel composition plus hiding (which is the evaluation context for CCS) would not be enough here, because in $P\,|\,Q$ the processes P and Q

cannot define names that appear in the other process as free variables: with parallel composition alone, there is no interaction. Later in the chapter (Section 4.8.4, and specifically Lemma 4.39), we show that parallel composition in a single particular context is enough. Besides, parallel composition alone will be shown sufficient in the open join-calculus presented in the next chapter.

We sometimes use more general contexts (e.g., with several distinct placeholders), but this makes little difference in our case, as generalized contexts preserve equivalence that are congruence for regular contexts by transitivity.

Application of contexts to processes is naturally lifted to the application of contexts to relation: for $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$, and for a set of contexts $\mathcal{C}$, we let

$$\mathcal{C}\,[\mathcal{R}] \quad \stackrel{\text{def}}{=} \quad \big\{(C[P], C[Q]) \mid C \in \mathcal{C} \text{ and } P \mathrel{\mathcal{R}} Q\big\}$$

Next we use this notation to define congruence properties:

Definition 4.4 *Let $\mathcal{C}$ be a given set of contexts. A* precongruence *is a preorder $\mathcal{R}$ such that $\mathcal{C}[\mathcal{R}] \subseteq \mathcal{R}$. A* congruence *is a relation that is both a precongruence and an equivalence.*

The congruence of an equivalence relation $\mathcal{R}$ is the largest congruence that is included in $\mathcal{R}$; this congruence consists of all pairs of processes $P, Q \in \mathcal{P}$ such that for all $C \in \mathcal{C}$ we have $C[P] \mathrel{\mathcal{R}} C[Q]$.

For every set of contexts, the congruence operator clearly preserves inclusion of relations and is idempotent as soon as these contexts contain the identity $[\,\cdot\,]$. This provides many simple inclusions in our hierarchy of congruences, and this will be useful in co-inductive definitions.

By default, *all congruence properties are stated for evaluation contexts only.* In particular, we use the notation $\mathcal{R}^\circ$ for the congruence of $\mathcal{R}$ for all $C[\,\cdot\,] \in \mathcal{E}$. Conversely, we explicitly name "full congruence" a congruence for all contexts.

As the usual equivalences are all congruences, in the following we use plain relation symbols ($\simeq$, $\approx$, $\succeq$, …) for them, and dotted relation symbols for non-congruence sibling relations ($\dot{\simeq}$, $\dot{\approx}$, $\dot{\succeq}$, …).

4.1.4 Weak semantics

We are mostly interested in equivalences that are insensitive to the number of internal reduction steps—unless, of course, they induce other visible effects. We say that a semantics is *weak* when it is defined only in terms of sequences of reductions $\rightarrow^*$ instead of single reductions $\rightarrow$. We define weak observation predicates accordingly:

Definition 4.5 *The* may predicate $\Downarrow_x$*—also known as the (weak) barb on x—detects whether a process may satisfy the basic observation predicate $\downarrow_x$, possibly after performing a sequence of internal reductions.*

$$P \Downarrow_x \quad \stackrel{def}{=} \quad \exists P', P \rightarrow^* P' \downarrow_x$$

In general, weak barbs may disappear as the result of some internal choice, but the absence of a weak barb is stable through reduction. For instance, the process $x\langle\rangle \oplus y\langle\rangle \oplus 0$ has two weak barbs $\Downarrow_x$ and $\Downarrow_y$.

Weak equivalences are harder to deal with, but they reflect some natural properties of distributed asynchronous systems, as it makes little sense to count the number of local steps of computation in a distributed system. Said otherwise, an observer should not be given a discriminating power that compares the relative speed of several implementations, because that would assume the use of a global clock.

Weak equivalences abstract over infinite sequences of reduction, and thus equate stable processes and divergent processes. They also raises technical complications such as the issue of fair computations, even for processes equivalent to processes with finite behavior. Hence, weak equivalences are insensitive to termination; again, this choice is consistent with our setting. Distributed termination is usually considered as a global property that is hard to achieve [53], and that cannot easily be detected by an observer. The problem of termination is more relevant for sequential calculi, or for studying single-machine implementations of process calculi [140].

4.1.5 On barbs and contexts

We conclude our preamble on weak reduction-based semantics by a discussion of alternate definitions of observation. So far, we assumed a distinct predicate $\downarrow_x$ for every nullary name, but there are other natural choices. We briefly consider some variations.

In the initial paper on barbed equivalences [101], and in most definitions of testing equivalences, a *single predicate* is used instead of an indexed family. Either there is a single observable action ω—which would correspond in our case to a single observable message $\omega\langle\rangle$—or all barbs are collected by an existential predicate. Accordingly, for every family of observation predicates (e.g., $\Downarrow_x$), we define an existential observation predicate that tests any of these predicates ($P \Downarrow \overset{\text{def}}{=} \exists x. P \Downarrow_x$). This in turn induces existential variants for all our equivalences.

Further choices of tests are also possible. We may wish, for instance, to test for the presence of messages on n-ary channels, or the simultaneous presence of several barbs; for instance, the x-and-y barb $\Downarrow_{x,y} \overset{\text{def}}{=} \rightarrow^* (\downarrow_x \wedge \downarrow_y)$ is more demanding than the conjunction $\Downarrow_x \wedge \Downarrow_y$, as can be seen by comparing the processes $x\langle\rangle \oplus y\langle\rangle$ and $x\langle\rangle \oplus y\langle\rangle \oplus (x\langle\rangle \mid y\langle\rangle)$.

It seems that these variations would introduce a variety of unrelated equivalences, but this is usually not the case for weak congruences. For instance, the congruence property allows the use of contexts that restrict all free names but one, and thus recover $\Downarrow_x$ from $\Downarrow$, and conversely we have $P \Downarrow$ if and only if $P \Downarrow_x$ for some $x \in \mathsf{fv}[P]$. Likewise, simple contexts can be used to transform join predicates into simple barbs: let J be an arbitrary join-pattern, and let $\Downarrow_J$ be the observation predicate that detects whether this join-pattern could be triggered in an enclosing definition. The context

$$T_J[\,\cdot\,] \quad \overset{\text{def}}{=} \quad \mathsf{def}\ J \triangleright t\langle\rangle\ \mathsf{in}\ [\,\cdot\,]$$

converts this elaborate barb into a single barb on $t \in \mathcal{N}_0$, since for every process P that does not have t as free variable we have $P \Downarrow_J$ if and only if $T_J[P] \Downarrow_t$.

That is, the exact form of the barbs seems irrelevant for all weak equivalences that are congruences at least for evaluation contexts, and provided there is at least one discriminating predicate. This remark is treated more abstractly by Honda and Yoshida in [73], where a "soundness" condition precisely ensures that every sound equivalence separate at least two processes.

The situation is more controversial when the definition of equivalence does not directly require both the congruence property and the respect of all barbs. With bisimulations for instance, this may induce significant differences. The problem occurs in the original formulation of barbed bisimulation congruence; it is discussed in Section 4.4.

4.2 Testing semantics

As regards discriminating power alone, it suffices to test the observation predicates under all possible contexts. The resulting *testing semantics* is relevant to many practical issues in distributed programming; it makes sense from a programming point of view, when barbs are interpreted as, e.g., print statements. Note that we detect the presence or the absence of messages after executing the process in context; we are not interested in intermediate states, or in the internal branching structure of processes.

Testing semantics have a long history; they can be traced back to Morris equivalence for the λ-calculus [104], where two λ-terms are equivalent when they have normal forms in the same contexts. Testing semantics have been thoroughly investigated in concurrency; they have been proposed for CCS in [51, 66, 97], with parallel composition with arbitrary processes as observers, and with both axiomatic characterizations and denotational semantics. They have also been extended to the π-calculus in [67, 34], and more recently to the join-calculus [83].

A testing semantics is usually defined as a preorder relation $\sqsubseteq$, the corresponding equivalence being $\sqsubseteq \cap \sqsubseteq^{-1}$. The preorder $\sqsubseteq$ is commonly interpreted as the "correct implementation" relation: the implementation can rule out some traces, but not exhibit traces whose behavior is not captured by the specification. Anticipating on the next sections, this preorder-based definition is an advantage of testing equivalences over bisimulation-based equivalences, where the simulation preorders do not capture bisimulation equivalences. In order to establish testing equivalences, however, one must cope with quantification over both contexts and traces, which makes the proofs particularly difficult [83].

In general, a test is an observer plus a way of observing; here, the set of observers is defined as the set of all evaluation contexts and the way of observing is defined in terms of the barbs $\Downarrow_x$. In a non-deterministic setting, we can decompose testing semantics as the intersection of two coarser equivalences: may testing and must testing. *May testing* detects whether there is a successful interaction between the context and the process. *Must testing* detects whether all interactions are successful.

Definition 4.6 *The* may testing preorder $\sqsubseteq_{may}$ *is the largest precongruence that respects the barbs* $\Downarrow_x$*;* may testing equivalence $\simeq_{may}$ *is the largest congruence that respects the barbs* $\Downarrow_x$*:*

$$\begin{aligned} P \sqsubseteq_{may} Q &\overset{def}{=} \forall C \in \mathcal{E}, x \in \mathcal{N}_0 \,.\, C[P] \Downarrow_x \textit{ implies } C[Q] \Downarrow_x \\ P \simeq_{may} Q &\overset{def}{=} \forall C \in \mathcal{E}, x \in \mathcal{N}_0 \,.\, C[P] \Downarrow_x \textit{ if and only if } C[Q] \Downarrow_x \end{aligned}$$

Typical examples of may-testing properties are, for any process P, $\mathsf{0} \sqsubseteq_{may} P$ (if a deadlocked process passes a test, then all processes pass this test) and $P \oplus \mathsf{0} \simeq_{may} P$

(may-testing is not sensitive to deadlock). Conversely $x\langle u\rangle \not\simeq_{may} x\langle v\rangle$ because the test $(\mathsf{def}\ x\langle u\rangle \triangleright u\langle\rangle\ \mathsf{in}\ [\,\cdot\,]), \Downarrow_u$ distinguishes the two messages.

Since an arbitrary substitution can be rendered by a context that entirely consists of forwarders, the may-testing preorder is clearly preserved by substitution on free names. Hence, it is straightforward to show that may-testing preorder is a full pre-congruence. We prove by induction that for all general context $C[\,\cdot\,]$, if $P \sqsubseteq_{may} Q$ and $C[P] \rightarrow^* \downarrow_x$ using at most n copies of P, then $C[Q] \Downarrow_x$. If $C[P] \rightarrow^* C'[P, P\sigma] \rightarrow^* \Downarrow_x$ where $C'[P,\,\cdot\,]$ is an evaluation context and the last series of derivation triggers at most n further copies of P, then by induction hypothesis applied to the context $C'[\,\cdot\,, P\sigma]$, we have $C'[Q, P\sigma] \Downarrow_x$ and, since $P\sigma \sqsubseteq_{may} Q\sigma$, $C'[Q, Q\sigma] \Downarrow_x$.

May testing is most useful to prove safety requirements: the specification of a program states that bad things should never happen, these bad behaviors are detected by a family of contexts that test for such behaviors and manifest their presence by emitting specific messages, and correctness is captured by may-testing in these contexts. For example, may testing is adequate to specify security properties in cryptographic protocols [3, 5, 4]. Note, however, that this negative usage of barbs does not tell much about the presence of suitable behaviors.

A complementary approach is to look at messages that are always emitted, independently of the internal choices. Indeed, another common family of derived observation predicates discriminates processes according to outputs that appear in every execution.

Definition 4.7 *The* must predicate $\downarrow_{\Box x}$ *detects whether all stable derivatives of a process exhibit the basic observation predicate* $\downarrow_x$.

$$P \downarrow_{\Box x} \quad \overset{def}{=} \quad \forall P',\ \textit{if}\ P \rightarrow^* P' \not\rightarrow,\ \textit{then}\ P' \downarrow_x$$

The must testing equivalence $\simeq_{must}$ *is the largest congruence that respects the must predicates:*

$$P \simeq_{must} Q \quad \overset{def}{=} \quad \forall C \in \mathcal{E}, x \in \mathcal{N}_0\ .\ C[P] \downarrow_{\Box x}\ \textit{iff}\ C[Q] \downarrow_{\Box x}$$

Must testing is not very interesting in our case, because it is sensitive to the presence of diverging computations. This is sometimes referred to as the "catastrophic interpretation" of infinite computation, as indeed the must predicate does not say anything about the partial outcome of such computations. For instance, if R is a diverging process, we immediately have $P \mid R \simeq_{must} Q \mid R$ for all processes P and Q.

A more tempting equivalence would be *Morris equivalence* $\simeq_{must} \cap \simeq_{may}$, the largest congruence that respects both may- and must- predicates, but again the treatment of diverging computations is not very satisfactory.

We refer to [83] for a detailed study of may and must testing in the join-calculus, including examples and stronger context lemmas.

4.3 Fair testing

In order to capture the positive behavior of processes with potentially infinite computations, we strengthen the must predicate of the previous section. The resulting *fair-must* predicate incorporates a notion of "abstract fairness", and induces an interesting tier between may-testing and bisimulations.

Definition 4.8 *The* fair-must predicate $\Downarrow_{\Box x}$ *detects whether a process always retain the possibility of emitting on x.*

$$P \Downarrow_{\Box x} \quad \stackrel{def}{=} \quad \forall P',\ \textit{if } P \rightarrow^* P',\ \textit{then } \exists P'', P' \rightarrow^* P'' \downarrow_x$$

This predicate is also *if $P \rightarrow^* P'$, then $P' \Downarrow_x$*, and it entails the stronger statement *if $P \rightarrow^* P'$, then $P' \Downarrow_{\Box x}$*. Thus, the set $\{x \mid P \Downarrow_{\Box x}\}$ increases as P performs reductions. This property holds for any ARS, while the similar property for strong barbs is proper to the join-calculus.

For processes whose computations are all finite, must predicates and fair-must predicates coincide; intuitively, the latter predicates can be considered as must-testing on all fair traces instead of on finite traces only. We summarize below the relative strengths of our predicates:

- in general, $\downarrow_x$ implies $\Downarrow_x$, and $\Downarrow_{\Box x}$ implies both $\Downarrow_x$ and $\downarrow_{\Box x}$;
- in the join-calculus, $\downarrow_x$ also implies $\Downarrow_{\Box x}$ and $\downarrow_{\Box x}$.

As can be expected, fair testing equivalence is the testing semantics induced by our new observation predicates:

Definition 4.9 *The* fair testing preorder $\sqsubseteq_{fair}$ *and* fair testing equivalence $\simeq_{fair}$ *are the largest precongruence and congruence, respectively, that respect the fair-must predicates $\Downarrow_{\Box x}$:*

$$\begin{array}{lcl} P \sqsubseteq_{fair} Q & \stackrel{def}{=} & \forall C \in \mathcal{E}, x \in \mathcal{N}_0 \,.\, C[P] \Downarrow_{\Box x} \ \textit{implies}\ \ C[Q] \Downarrow_{\Box x} \\ P \simeq_{fair} Q & \stackrel{def}{=} & \forall C \in \mathcal{E}, x \in \mathcal{N}_0 \,.\, C[P] \Downarrow_{\Box x} \ \textit{if and only if}\ \ C[Q] \Downarrow_{\Box x} \end{array}$$

The particular notion of fairness embedded in fair testing deserves further explanations: the fair-must predicate does not describe any reduction strategy; it only says that it is always possible to perform reductions that eventually emit on x. Nonetheless, we can interpret the barb $\Downarrow_{\Box x}$ as a successful observation "the message $x\langle\rangle$ is eventually emitted". As we do so, we consider only infinite traces that emit on x and we discard all other infinite traces. This is precisely a model of *fair traces* for a very strong notion of fairness. For example, we have the fair testing equivalence:

$$\mathsf{def} \begin{array}{rl} & t\langle\rangle \triangleright x\langle\rangle \\ \wedge & t\langle\rangle \triangleright t\langle\rangle \end{array} \mathsf{in}\ t\langle\rangle \quad \simeq_{fair} \quad x\langle\rangle$$

where the first process provides two alternatives in the definition of t: either the message $x\langle\rangle$ is emitted, or the message $t\langle\rangle$ is re-emitted, which reverts the process to its initial state. It is possible to always select the second, stuttering branch of the alternative, and thus there are infinite computations that never emit on $x\langle\rangle$. Nonetheless, the possibility of emitting on x always remains, and any fair evaluation strategy should eventually select the first branch.

Related works This approach to fairness has been previously advocated for several CCS-like process calculi equipped with trace-based semantics [42, 105, 43]. In [42], Brinksma et al. introduce a similar notion of *should-testing*, and compare it to their previous notions of fair testing for CCS. They independently remark that weak bisimulation equivalence incorporates a particular notion of fairness. They identify the problem of gradual commitment, and in general of sensibility to the branching structure, as an undesirable property of bisimulation. They discuss in detail the relation between should-testing and the so-called Koomen's Fair Abstraction Rule found in axiomatizations of CCS, and they establish that (bisimulation-based) observational equivalence is strictly finer than should-testing.

Independently, a fair testing scenario is proposed by Natarajan and Cleaveland in [105] to deal with divergent behaviors. The authors argue that fairness is crucial to deal with "distributed, fault-tolerant systems"; in this setting, they model protocols that use a lossy medium. Informally, the medium may lose some messages, but certainly not all of them. In a join-calculus setting, the lossy medium example is reformulated into the equation:

$$\mathsf{def} \begin{array}{rl} & x\langle\rangle \triangleright R \\ \wedge & x\langle\rangle \triangleright 0 \end{array} \mathsf{in}\ \mathsf{repl}\ x\langle\rangle \quad \simeq_{fair} \quad \mathsf{repl}\ R$$

The process on the right—the specification—repeatedly fires copies of process R; the process on the left—its implementation through a lossy forwarder—attempts to do the same by repeatedly sending messages $x\langle\rangle$ to start R, but every such message can be discarded. There is in fact a much finer *labeled bisimulation* relation between the two processes, whose co-inductive proof is immediate, but nonetheless this equation is best explained in terms of fairness in the choice of the reaction rule being triggered whenever a message on x is consumed. The authors also study in detail the well-known alternating bit protocol in CCS, and they provide several characterizations of fair equivalence on traces. They finally provide a simulation-based sufficient condition to establish fair equivalence (which is hard to establish otherwise), along with the remark that fair equivalence is strictly coarser than bisimulation-based equivalences.

4.3.1 Fair testing versus may testing

Fair testing is not related to must testing; for instance we have, for every processes P, Q, and for every diverging process Ω with no free variables, $P \sqsubseteq_{must} Q \,|\, \Omega$ because the latter process passes all tests; the situation is different with fair testing, where Ω is invisible: in every context, reductions that are internal to Ω commute with all other reductions, and thus $Q \,|\, \Omega \simeq_{fair} Q$.

Fair testing is seemingly unrelated to may testing, but is actually finer ($\sqsubseteq_{fair} \subset \sqsubseteq_{may}^{-1}$); the inclusion is strict, since for instance we have $x\langle\rangle \not\sqsubseteq_{fair} x\langle\rangle \oplus 0$. Said otherwise, fair-must testing and Morris-style testing with abstract fairness coincide:

Lemma 4.10 *In the join-calculus, fair testing equivalence is the largest congruence that refines both may predicates and fair-must predicates.*

This property also holds in CCS, in the asynchronous π-calculus, and for Actors, where a more operational fair testing equivalence is proposed as the main semantics in [12]. The argument is the same for all these calculi: a family of contexts encodes the may predicate $\Downarrow_x$ in terms of fair must predicates.

Proof: We first prove that there is an evaluation context $C[\,\cdot\,]$ such that, for all processes P, we have $P \Downarrow_x$ iff $C[P] \not\Downarrow_{\Box y}$ up to renaming.

For $x, y \in \mathcal{N}_0$, let $C[\,\cdot\,]$ be the context defined as follows:

$$C[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ r\langle z\rangle \mid \mathit{once}\langle\rangle \triangleright z\langle\rangle\ \mathsf{in}\ \big(r\langle y\rangle \mid \mathit{once}\langle\rangle \mid \mathsf{def}\ x\langle\rangle \triangleright r\langle x\rangle\ \mathsf{in}\ [\,\cdot\,]\big)$$

This context binds x and uses non-determinism to transform the presence of a barb $\Downarrow_x$ on this name into the absence of a barb $\Downarrow_{\Box y}$ on another name y. Let P be a process such that $\mathit{once}, r, y \notin \mathsf{fv}[P]$. The process $C[P]$ initially contains a single message $\mathit{once}\langle\rangle$, and never re-emits such messages; thus, the rule $r\langle z\rangle \mid \mathit{once}\langle\rangle \triangleright z\langle\rangle$ is triggered at most once. There are two possibilities: either the context consumes its own message $r\langle y\rangle$ and exhibits a barb on y, or, inasmuch as some message $r\langle x\rangle$ is present, the context consumes $r\langle x\rangle$ and silently becomes inert.

For every process P, we choose names $\mathit{once}, r, y \notin \mathsf{fv}[P]$. If $P \rightarrow^* P' \downarrow_x$, then $C[P] \rightarrow^* C[P'] \rightarrow\rightarrow \not\Downarrow_y$ where the two last reductions first receive the message on x and replace it by $r\langle x\rangle$, then synchronize it with $\mathit{once}\langle\rangle$ and thus withdraw the possibility of sending a message on y. Therefore, $C[P] \not\Downarrow_{\Box y}$. Conversely, if P never emits any message on x, it does not interact with the context; the synchronization $r\langle y\rangle \mid \mathit{once}\langle\rangle$ described above always remains enabled until it occurs, and thus $\Downarrow_{\Box y}$.

For a given pair of processes P and Q such that $P \sqsubseteq_{\mathit{fair}} Q$, we now prove that $Q \sqsubseteq_{\mathit{may}} P$. Let $C'[\,\cdot\,]$ be an evaluation context and x a name. We have $C'[Q] \Downarrow_x$ iff $C[C'[Q]] \not\Downarrow_{\Box x}$, so $C[C'[P]] \not\Downarrow_{\Box x}$ and $C'[P] \Downarrow_x$. We therefore obtain the inclusion $\sqsubseteq_{\mathit{fair}} \subseteq \sqsubseteq_{\mathit{may}}^{-1}$, and thus $\simeq_{\mathit{fair}} \subseteq \simeq_{\mathit{may}}$. $\Box$

Overall, fair testing is adequate to deal with distributed systems; it is stronger than may-testing, detects deadlocks, but remains insensitive to livelocks; in [43], for instance, other distributed communication protocols are studied using the fair-testing preorder as an implementation relation.

Note, however, that "abstract fairness" departs from the liveness properties that are typically guaranteed by implementations (*cf.* for instance the scheduling policy of our prototype [59]). From a programming language point of view, a full implementation of abstract fairness would be able to select a fair trace for any process; in particular, if $P \Downarrow_{\Box x}$ then the execution of P must eventually emit on $x\langle\rangle$. Unfortunately, it is easy to code hard problems as processes that emit the message $x\langle\rangle$ in case of success, and it seems that any evaluation strategy that guarantees abstract fairness would have to explore most—if not all—reduction sequences.

Fair testing suffers from another drawback: the proofs are difficult because of the traces. As we shall see, the redeeming feature of fair testing is that it is coarser than barbed congruence, which provides practical proof techniques for it. More precisely, we provide an alternate characterization of fair testing in terms of coupled simulations in Section 4.5.

4.4 Barbed Congruence

Initially proposed by Park [115] and by Milner for CCS [96] in a labeled setting, bisimulations have somehow become a standard semantics in concurrency theory. Independently of their intrinsic merits, bisimulation-based equivalences offer several technical advantages. Foremost, the proofs are simple and elegant; they proceed by identifying pairs of equivalent states of processes, and a case analysis of their reductions. Hence, one needs to consider only a few reduction steps instead of whole traces. Accordingly, automated verification is significantly simpler than for trace-based semantics thanks to partition-refinement algorithms [65]. Besides, numerous sophisticated techniques lead to smaller candidate bisimulations, and to more modular proofs (see, e.g., [131], and Section 4.7).

The definitions are standard; we briefly recall them. The reader should refer to, e.g., [97, 101] for discussion and examples.

Definition 4.11 (Simulation, bisimulation) *Let $(\mathcal{P}, \xrightarrow{\alpha})$ be a LTS, and $\mathcal{R}$ be a relation on processes. $\mathcal{R}$ is a strong simulation when for all processes P, Q, for every label α, if $P \xrightarrow{\alpha} P'$, then for some Q' we have $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{R} Q'$. $\mathcal{R}$ is a strong bisimulation when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations.*

Let also τ be a particular label associated with the transition $\rightarrow$, which represents silent reductions, and let $\Rightarrow^{\alpha}$ be the τ-saturated transition system induced by $\xrightarrow{\alpha}$ (i.e., the transition system with silent transitions $\rightarrow_{\tau}^{*}$ and transitions $\rightarrow^{*}\xrightarrow{\alpha}\rightarrow^{*}$ for every $\alpha \neq \tau$). $\mathcal{R}$ is a weak simulation (resp. bisimulation) when it is a strong simulation (resp. bisimulation) for the τ-saturated system $(\mathcal{P}, \Rightarrow^{\alpha})$.

We will consider labeled bisimulations for the join-calculus in the next chapter. For the time being, we instantiate this definition to reduction-based systems (no label $\alpha \neq \tau$). In the whole chapter, simulations and bisimulations are therefore weak reduction-based relations when left unspecified.

In our quest for natural equivalences, a first attempt would be to consider the largest reduction-based bisimulation, but this notion is degenerate for lack of basic observations, and relates all pairs of processes. If we also require that the bisimulation refine the barbs, we obtain the classical notion of *barbed bisimulation* [101], initially proposed for CCS but easily adapted to many other calculi [111, 16, 73, 130, 109].

Definition 4.12 (Barbed bisimilarity) *Let $(\mathcal{P}, \rightarrow, \downarrow_x)$ be an ARS and $\mathcal{R}$ be a relation on processes. $\mathcal{R}$ is a barbed simulation when it is a (reduction-based) simulation that refines the barbs: for all $\Downarrow_x$, if $P \mathcal{R} Q$ and $P \Downarrow_x$, then $Q \Downarrow_x$. $\mathcal{R}$ is a barbed bisimulation when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are barbed simulations.*

Barbed bisimilarity $\dot{\approx}$ is the largest barbed bisimulation.

Strong barbed simulation, bisimulation, and bisimilarity are respectively defined by substituting strong simulation for simulation and strong barbs $\downarrow_x$ for weak barbs $\Downarrow_x$ in the above definition. Strong barbed bisimilarity is written $\dot{\sim}$.

As a reduction-based equivalence, barbed bisimilarity can be used to define sensible behavioral equivalences on different process calculi; it is usually considered as the best mix between semantics from functional theory and from concurrency theory. This is the approach chosen in this dissertation.

In contrast with our previous equivalences, it is well-known that bisimulations reveal the internal branching structure of processes, and are in general finer than testing semantics. Internal choices are revealed only inasmuch as they can be separated according to the barbs. For instance, we have $x\langle\rangle \oplus (x\langle\rangle \oplus y\langle\rangle) \mathrel{\dot{\approx}} x\langle\rangle \oplus y\langle\rangle$, which is easily proven by checking that the relation

$$\big\{\{x\langle\rangle\}, \{y\langle\rangle\}, \{x\langle\rangle \oplus y\langle\rangle, x\langle\rangle \oplus (x\langle\rangle \oplus y\langle\rangle)\}\big\}$$

is a bisimulation and respects the barbs. Conversely, we have $x\langle\rangle \oplus (y\langle\rangle \oplus z\langle\rangle) \simeq_{fair} (x\langle\rangle \oplus y\langle\rangle) \oplus z\langle\rangle$ but these processes are not barbed bisimilar, because the choice between the three outputs is not performed atomically. As in [117], we refer to this situation as *gradual commitment.*

Co-inductive definitions and testing equivalences are not entirely unrelated, though. For instance, barbed similarity is the preorder induced by the weak barbs: this preorder obviously respects the barbs; it is also a weak simulation because any reduction is simulated by the absence of reductions on the other side. Therefore, the congruence of barbed similarity is an alternate characterization of the may testing preorder $\simeq_{may}$; this provides a convenient proof technique for may-testing equivalence.

Barbed bisimilarity also respects the fair predicates $\Downarrow_{\Box x}$. Let us assume that P and Q are two processes such that $P \mathrel{\dot{\approx}} Q$ and $P \Downarrow_{\Box x}$. Then every sequence of reductions $Q \rightarrow^* Q'$ can be simulated by $P \rightarrow^* P' \approx Q'$, and for every such sequence $P \Downarrow_{\Box x}$ implies $P' \Downarrow_x$. As $P' \mathrel{\dot{\approx}} Q'$, Q' also has a barb on x, and thus $Q \Downarrow_{\Box x}$. Note that we successively used the barbed simulation properties of $\dot{\approx}$ in both directions, which reflects the alternation of quantifiers in the definition of fair testing. In Section 4.5, we use a coarser, simulation-based equivalence that still allows this round trip (Lemma 4.17).

4.4.1 Diagrams

In order to reason about processes and their transitions, it is convenient to visualize their relations in diagrams, instead of writing formulae with numerous quantifiers. The notation is standard.

A diagram consists of nodes that represent processes, linked by labeled edges that represent relations among processes. Every diagram states a property, with the following conventions: solid edges $\longrightarrow$ stand for universally-quantified relations (i.e. the premises) and dotted edges $\dashrightarrow$ stand for existentially-quantified relations (i.e., the conclusions). Processes on nodes can be left implicit, can be named for reference, and can be given a specific form.

For instance, the two following diagrams express that the relation $\mathcal{R}$ is a (weak, reduction-based) bisimulation:

$$\begin{tikzcd} {} \arrow[r, dash, "\mathcal{R}"] \arrow[d, "*"'] & {} \arrow[d, dotted, "*"] \\ {} \arrow[r, dash, dotted, "\mathcal{R}"] & {} \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} {} \arrow[r, dash, "\mathcal{R}"] \arrow[d, dotted, "*"'] & {} \arrow[d, "*"] \\ {} \arrow[r, dash, dotted, "\mathcal{R}"] & {} \end{tikzcd}$$

4.4.2 About co-inductive definitions

Our diagrams can also be read as the definition of functionals $\mathcal{F}$ on relations among processes: $\mathcal{F}(\mathcal{R})$ is the largest relation such that all diagrams are valid when $\mathcal{F}(\mathcal{R})$

replaces $\mathcal{R}$ on solid edges, and $\mathcal{R}$ still appears on dotted edges. Then, it is easy to check on the shape of the diagrams that $\mathcal{F}$ is monotonic, and that $\mathcal{R}$ matches the diagrams iff it is a pre-fixed-point of $\mathcal{F}$ ($\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$).

Apart from diagram-based properties, the definition of an equivalence may also require congruence properties and observational properties, which can be lifted to the definition of $\mathcal{F}$ as well. For congruence properties, we require that $\mathcal{C}[\mathcal{F}(\mathcal{R})] \subseteq \mathcal{R}$; for predicate-based properties, that $\mathcal{F}(\mathcal{R}) \subseteq \phi$, where ϕ is the partition induced by the predicates under consideration. These properties and their conjunctions still yield monotonic functionals.

Such monotonic functionals have a greatest fix-point, which is the union of all their pre-fixed-points. In order to establish that processes are related by this fix-point, it suffices to build a relation $\mathcal{R}$ and to check that it is a pre-fixed-point of $\mathcal{F}$. For every bisimulation-based equivalence that we define, this uniformly establishes that the corresponding bisimilarity is an equivalence relation—in particular that it is transitive—and this provides a co-inductive proof technique to establish this bisimilarity. We omit this general argument in the following.

4.4.3 The two congruences

Barbed bisimilarity alone lacks some discriminating power; for instance, $x\langle u\rangle \dot{\approx} x\langle v\rangle$ but $x\langle u\rangle \not\simeq_{may} x\langle v\rangle$. Some additional congruence property is called for, but a technical problem arises: there are at least two sensible ways of ensuring congruence:

- we can take the largest congruence $\dot{\approx}^{\circ}$ contained in the largest barbed bisimulation $\dot{\approx}$; this is the two-stage definition usually chosen for CCS and for the π-calculus [101, 130, 16];

- or we can take the largest barbed bisimulation that is also a congruence $\approx$; this is the equivalence chosen for the ν-calculus [71, 73, 72] and for the join-calculus [55, 57, 35, 2]

By definition, congruences of bisimilarities are coarser than congruence-bisimulations: we have $\approx \,\subseteq\, \dot{\approx}$, hence $\approx^{\circ} \,\subseteq\, \dot{\approx}^{\circ}$ and, since $\approx \,=\, \approx^{\circ}$ by definition, $\approx \,\subseteq\, \dot{\approx}^{\circ}$. In the opposite direction, $\dot{\approx}^{\circ}$ is a congruence that respects the barbs, and thus the two relations coincide if and only if $\dot{\approx}^{\circ}$ is a bisimulation, but this is not necessarily the case in general. The two diagram below emphasize the difference between the two definitions. Once the context $C[\,\cdot\,]$ has been applied on the left, the stable relation $\dot{\approx}$ is a bisimulation, and not a congruence. On the contrary, the bisimulation-and-congruence relation $\approx$ on the right retains the congruence property after matching reductions, and allows further application of contexts after reductions.

$$\begin{tikzcd}
P \arrow[r, dash, "\dot{\approx}^{\circ}"] & Q \\
C[P] \arrow[r, dash, "\dot{\approx}"] \arrow[d] & C[Q] \arrow[d, dotted, "*"] \\
T \arrow[r, dotted, dash, "\dot{\approx}"] & T'
\end{tikzcd}$$

is coarser than

$$\begin{tikzcd}
P \arrow[r, dash, "\approx"] & Q \\
C[P] \arrow[r, dash, "\approx"] \arrow[d] & C[Q] \arrow[d, dotted, "*"] \\
T \arrow[r, dotted, dash, "\approx"] & T'
\end{tikzcd}$$

Fortunately, the two definitions yield the same equivalence in our setting. The proof is surprisingly difficult; it is detailed in Section 4.8.

Theorem 3 *In the join-calculus, we have* $\approx = \dot{\approx}^{\circ}$.

Even as the two congruences semantically coincide, the two definitions still provide two different co-inductive proof techniques. For example, when comparing fragments of the same calculus $\dot{\approx}^{\circ}$ may be better because the congruence for contexts of the larger fragment is applied once for all; conversely, $\approx$ often leads to simpler proofs, because interaction with the context is more abstract, and can be decomposed into successive exchanges of messages: after every reduction, the context may change, but remains in the class of the bisimulation-congruence being established. Besides, this definition is more stable than the other one to variations in the definition of barbs. This proof technique is illustrated at the end of this chapter, and in the next chapters. In accordance with our preferred proof technique, we choose the following "main" definition of equivalence:

Definition 4.13 Barbed congruence $\approx$ *is the largest equivalence relation that meets the following requirements:*

1. $\approx$ *is a refinement of the predicates* $\Downarrow_x$*;*
2. $\approx$ *is a congruence;*
3. $\approx$ *is a weak bisimulation.*

This definition corresponds to Honda and Yoshida's definition of sound reduction-based equivalence [73], where soundness is the existential variant of *1.*, and reduction-based means the conjunctions of *2.* and *3.*.

The next lemma states that full congruence is not more discriminating than evaluation congruence (in the absence of name-matching). The important property is closure under substitution on free names; the full congruence property is then guaranteed.

Lemma 4.14 *Barbed congruence* $\approx$ *is preserved by substitution on free names, and is a full congruence.*

Proof: Let $S \subseteq \mathcal{N}$ be a finite set of names, and let σ be a substitution from S to $\mathcal{N} \setminus S$. We use the context

$$C[\,\cdot\,] \quad = \mathsf{def}\ \textstyle\bigwedge_{x \in S} x\langle \widetilde{v} \rangle \triangleright (x\sigma)\langle \widetilde{v} \rangle\ \mathsf{in}\ [\,\cdot\,]$$

and we prove the equivalence $P\sigma \approx C[P]$.

For all $P \approx Q$, and for every substitution σ, we let $S = \mathsf{fv}[P] \cup \mathsf{fv}[Q]$ and we apply the result above for two substitutions σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$ and all intermediate names are fresh. We conclude by transitivity of $\approx$.

To prove the second part of the lemma, we establish that $\approx$ is closed for definitions; we consider the relation

$$\mathcal{R} \quad = \quad \{(\mathsf{def}\ J \triangleright P \wedge D\ \mathsf{in}\ R, \mathsf{def}\ J \triangleright Q \wedge D\ \mathsf{in}\ S) \mid P \approx Q \text{ and } R \approx S\}$$

and we establish that $\mathcal{R} \subseteq \approx$. By definition, $\mathcal{R}$ is a congruence for evaluation contexts. $\mathcal{R}$ preserves the barbs: to exhibit a given barb we use only a finite number of copies of the guarded processes. $\mathcal{R}$ is a bisimulation: for reductions that do not fork a new P, this is obvious; otherwise, we have $R \,|\, P\sigma \approx S \,|\, Q\sigma$. $\square$

4.4.4 Single-barbed bisimulation

We study the variant of bisimilarity $\dot{\approx}_\exists$ defined as the largest bisimulation that respects the single barb $\Downarrow$. This equivalence turns out to be very coarse; it separates only three classes of processes.

Lemma 4.15 *Single-barbed bisimilarity $\dot{\approx}_\exists$ partitions the processes of the join-calculus into the three classes defined by the predicates $\Downarrow_\Box$, $\not\Downarrow$, and $\Downarrow \wedge \not\Downarrow_\Box$.*

The congruence of single-barbed bisimilarity is fair testing ($\dot{\approx}^\circ_\exists = \simeq_{fair}$).

The simple model of $\dot{\approx}_\exists$ is depicted below, with reductions between classes and an example process in each class.

$$\begin{tikzcd} & x\langle\rangle \oplus \mathsf{0} \arrow[dl] \arrow[dr] & \\ x\langle\rangle & & \mathsf{0} \end{tikzcd}$$

This result is in sharp contrast with Theorem 3: the existential barbed congruence $\approx_\exists$ is clearly equal to the barbed congruence $\approx$ (*cf.* Section 4.1.5). Hence, we obtain a first example where both styles of congruence definitions yield distinct relations: $\simeq_{fair} = \dot{\approx}^\circ_\exists \neq \approx_\exists = \approx$. The situation in the π-calculus is far more exotic; it is described in Section 4.6.1.

Proof: The three predicates of the lemma induce a partition on join-calculus processes; let $\mathcal{R}$ be the corresponding equivalence relation.

We prove that $\mathcal{R} \subseteq \dot{\approx}_\exists$ by showing that $\mathcal{R}$ is a single-barbed bisimulation.

$\mathcal{R}$ respects the barb $\Downarrow$ by construction: it refines $\{\Downarrow,\not\Downarrow\}$ by splitting the first class in two, according to the predicate $\Downarrow_\Box$, which always implies $\Downarrow$.

$\mathcal{R}$ is a weak bisimulation: the two classes $\Downarrow_\Box$ and $\not\Downarrow$ are closed by reduction and processes in these classes are trivially bisimilar; besides, processes in the upper class always have series of reductions leading to both lower classes ($P \Downarrow$ is $P \rightarrow^*\downarrow$ and implies $P \rightarrow^*\Downarrow_\Box$; $P \not\Downarrow_\Box$ is $P \rightarrow^* P' \not\Downarrow$), and thus for every pair of processes $P \mathcal{R} Q$, for every reduction $P \rightarrow^* P'$, we exhibit a Q' in the same class of P' such that $Q \rightarrow^* Q'$ and $P' \mathcal{R} Q'$, for each of the three classes.

Conversely, $\dot{\approx}_\exists$ separates the three classes of $\mathcal{R}$, hence $\dot{\approx}_\exists \subseteq \mathcal{R}$.

To obtain the second statement of the lemma, we remark that $\mathcal{R}^\circ$ is exactly fair-must-and may testing, and thus $\mathcal{R}^\circ = \simeq_{fair,\exists}$ by Lemma 4.10. The number of barbs makes no difference in the definition of fair-testing (*cf.* 4.1.5), and thus $\dot{\approx}^\circ_\exists = \mathcal{R}^\circ = \simeq_{fair,\exists} = \simeq_{fair}$. $\Box$

4.5 Coupled simulations

The ability of discriminating processes according to their internal choice points is a mixed blessing, because many intuitively correct protocols do not preserve all these internal choices.

To address this issue, Parrow and Sjödin introduce *coupled simulations*, a simulation based equivalence that slightly relaxes the bisimulation clauses to handle these protocols. We adapt coupled simulations to our reduction-based setting, and we study its basic properties.

Alternatively, we could study these protocols by backtracking from barbed congruence to the lower tier in our hierarchy, namely fair-testing. This makes the relation between fair testing and coupled simulation intriguing, especially as these equivalences have been independently introduced to handle the same issue.

In general, fair testing is coarser than the congruences of coupled simulations, yet these equivalences can be made to coincide in the join-calculus, which somehow reconcile testing semantics and bisimulation-based semantics in an intermediate tier that hosts both kinds of equivalences. Interestingly, we obtain this result by using an explicit model of coupled simulations.

4.5.1 Internal choice and gradual Commitment

Coupled simulations is a device introduced by Parrow and Sjödin to address the issue of gradual commitment in multi-way synchronization protocols [117]. The main advantage of this equivalence is that it retains co-inductive proof techniques: quoting the authors, "coupled simulation equivalence can be established by case analysis over single reduction steps, and yet does not require an exact correspondence between choice points in computation". More generally, coupled simulations is presented as a key equivalence for distributed systems, where atomic steps are implemented as a negotiation between distributed components. This is exactly what happens in the encoding of the choice operator presented in [109].

Their setting, however, is quite different from ours; they are mostly interested in divergence-sensitive equivalences for CCS, with a labeled semantics. They position coupled simulation strictly between may-testing and observational equivalence. In a second paper [118], they provide a complete axiomatization as the one for observational equivalence plus the additional axiom $\tau.(\tau.P + Q) = \tau.P + Q$ (in every context), which exactly says that internal choices can be flattened, and is still weaker than the additional axiom for trace equivalence $\tau.P+Q = P+Q$. Accommodations for diverging computations are considered in [118]; they propose a variant named weakly-coupled equivalences. We adopt its reduction-based counterpart.

Definition 4.16 *A pair of relations $\leqslant, \geqslant$ are* coupled simulations *when $\leqslant$ and $\geqslant^{-1}$ are two simulations that satisfy the following coupling diagrams:*

$$\begin{tikzcd}
P \arrow[r, dash, "\leqslant"] \arrow[dr, dotted, dash, "\geqslant"] & Q \arrow[d, dotted, "*"] \\
& Q'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
P \arrow[r, dash, "\geqslant"] \arrow[d, dotted, "*"'] & Q \arrow[dl, dotted, dash, "\leqslant"] \\
P' &
\end{tikzcd}$$

A relation $\mathcal{R}$ is a barbed coupled equivalence *when there is a pair of coupled simulations $(\leqslant, \eqslantless)$ that respect the barbs $\Downarrow_x$ and such that $\mathcal{R} = \leqslant \cap \geqslant$.*

The union of all barbed coupled equivalences is named barbed coupled similarity, and is denoted $\dot{\lessgtr}$; it is also the coupled equivalence obtained from the largest coupled simulations.

The last statement of the definition—in particular the transitivity of $\dot{\lessgtr}$—can be derived by the argument given in Section 4.4.2.

In the special case where $\leqslant \, = \, \geqslant$, the coupling diagrams are always verified, and we recover the definition of weak bisimulation. In particular we have $\dot{\approx} \subseteq \dot{\lessgtr}$. Typically, the discrepancy between $\leqslant$ and $\geqslant$ is most useful to describe processes that are in a transient state, bisimilar neither to the initial state nor to any final state. For instance, we have the diagram

$$\begin{tikzcd}
(P \oplus Q) \oplus R \arrow[r, dash, "\lessgtr"] \arrow[d] & P \oplus (Q \oplus R) \arrow[dd, dotted] \\
P \oplus Q \arrow[ur, dash, "\leqslant"] \arrow[d] \arrow[dr, dotted, dash, "\geqslant"] & \\
P \arrow[r, dotted, dash, "\lessgtr"] & P
\end{tikzcd}$$

where the dotted relations emphasize the coupling requirement on the simulation between $P \oplus Q$ and $P \oplus (Q \oplus R)$.

In the initial definition of coupled simulations of [117], coupling ($\leqslant \cap \geqslant$) is required only for pairs of stable processes. Our definition is more demanding, since whenever $Q \not\rightarrow$ the coupling requirement on $P \leqslant Q$ becomes $P \geqslant Q$, and thus also $P \,\dot{\lessgtr}\, Q$.

The next lemma states that barbed coupled simulations also refine fair-must barbs. Its proof uses simulation in both directions, which somehow reflects the alternation of quantifiers in the definition of fair-must barbs,

Lemma 4.17 *Let $\leqslant, \geqslant$ be a pair of barbed coupled simulations. For all processes P and Q, for all $x \in \mathcal{N}_0$, if $P \geqslant Q$ and $P \Downarrow_{\Box x}$, then $Q \Downarrow_{\Box x}$.*

In particular, $\geqslant^{\circ} \subseteq \sqsubseteq_{fair}$, and $\dot{\lessgtr}^{\circ} \subseteq \simeq_{fair}$.

Proof: With the hypotheses of the lemma, if $Q \rightarrow^* Q'$, these reductions can be simulated by $P \rightarrow^* P' \geqslant Q'$. By coupling, we also have $P' \rightarrow^* P'' \leqslant Q'$. By definition of the fair-must predicate $\Downarrow_{\Box x}$, we have $P'' \Downarrow_x$. Finally, $\leqslant$ respects the barbs, and thus $Q' \Downarrow_x$. $\Box$

The "upward reduction closure" $\leftarrow^*$ is an example of relation that is both a coupled barbed simulation and a precongruence. More generally, we state a proof technique that we use later on to reduce the size of candidate coupled simulations as we establish coupled similarities.

Lemma 4.18 (coupled simulations up to reductions) *Let $(\leqslant, \geqslant)$ be a pair of relations on processes such that*

1. *if $P \Downarrow_x$ and either $P \leqslant Q$ or $Q \geqslant P$, then $Q \Downarrow_x$;*

2. we have

$$\begin{tikzcd}
\cdot \arrow[r, dash, "\leqslant"] \arrow[d, "*"'] & \cdot \arrow[dr, dotted, "*"] & \\
\cdot \arrow[r, dotted, dash, "\geqslant^{=}"] & \cdot \arrow[r, dotted, "*"] & \cdot
\end{tikzcd}$$

and

$$\begin{tikzcd}
& \cdot \arrow[dl, dotted, "*"'] \arrow[r, dash, "\geqslant"] & \cdot \arrow[d, "*"] \\
\cdot & \cdot \arrow[l, dotted, "*"] & \cdot \arrow[l, dotted, dash, "\leqslant^{=}"']
\end{tikzcd}$$

Then $(\leftarrow^{*}\leqslant^{=}, \geqslant^{=}\rightarrow^{*})$ *is a pair of barbed coupled simulations and we have* $\leqslant \cap \geqslant \subseteq \dot{\lessgtr}$.

Proof: as the statement is symmetric, we only check the properties of $\leftarrow^{*}\leqslant^{=}$.

1. the relation $\leftarrow^{*}$ respects weak barbs, and so does $\leqslant$ by the first hypothesis.

2. we separately obtain the coupling diagrams for $\leftarrow^{*}$ and $\leftarrow^{*}\leqslant$. The coupling from $P \leftarrow^{*} Q$ to $P \rightarrow^{*} Q'$ is immediate, because we can use the same series of reductions to close the diagram for $Q' = P$. The coupling from $\leftarrow^{*}\leqslant$ to $\geqslant^{=}\rightarrow^{*}$ is the first diagram of the lemma.

3. closure by reduction on the left trivially ensures that $\leftarrow^{*}\leqslant$ is a simulation. We compose reductions on the left, and we perform no reduction on the right. □

4.5.2 The two congruences yield distinct equivalences

As for weak bisimulation, we can either add precongruence requirements to the definition of barbed coupled simulations and thus obtain a barbed coupled-simulations congruence (denoted $\lessgtr$ in the following), or take the largest congruence that is contained in barbed coupled similarity (denoted $\dot{\lessgtr}^{\circ}$).

By definition, we have that $\lessgtr \subseteq \dot{\lessgtr}^{\circ}$, but in fact $\dot{\lessgtr}^{\circ}$ is not a coupled simulation, and we have:

Lemma 4.19 $\lessgtr$ *is strictly finer than* $\dot{\lessgtr}^{\circ}$.

Proof: The difference appears as soon as internal choices are spawned *between* visible actions. The counter-example is especially simple in asynchronous CCS, where we compare the processes $a.\overline{b} \oplus a.\overline{c}$ and $a.(\overline{b} \oplus \overline{c})$:

$a.\overline{b} \oplus a.\overline{c} \not\lessgtr a.(\overline{b} \oplus \overline{c})$**:** Let us assume that we had $a.\overline{b} \oplus a.\overline{c} \lessgtr a.(\overline{b} \oplus \overline{c})$. The reduction $a.\overline{b} \oplus a.\overline{c} \rightarrow a.\overline{b}$ is simulated by no reduction on the other side, and as the two processes are stable, they are also coupled: $a.\overline{b} \lessgtr a.(\overline{b} \oplus \overline{c})$. By applying the context $\nu ab.(\overline{a} \,|\, [\cdot\,])$, we obtain

$$\nu ab.(\overline{a} \,|\, a.\overline{b}) \quad \lessgtr \quad \nu ab.(\overline{a} \,|\, a.(\overline{b} \oplus \overline{c}))$$

and this relation is clearly inconsistent: only the process on the right may still emit on a free name by reducing to $\overline{c}$ in two steps.

$a.\overline{b} \oplus a.\overline{c} \;\dot{\lessgtr}^{\circ}\; a.(\overline{b} \oplus \overline{c})$**:** For every particular evaluation context $C[\cdot\,]$, however, the visible action is replaced with potential reduction with the context. In this example, interaction with the context is limited to reception on a; let $C[\cdot\,]$ be an evaluation context; this context may interact with our example if and only if there is another evaluation context $C'[\cdot\,]$ such that $C[0] \rightarrow^{*} C'[\overline{a}]$.

We establish the equivalence above in a mostly co-inductive style by applying Lemma 4.18. We let $\leqslant, \geqslant$ be the relations that contain the following pairs of processes: for every evaluation context $C[\,\cdot\,]$,

$$C[a.\overline{b} \oplus a.\overline{c}] \quad \leqslant \cap \geqslant \quad C[a.(\overline{b} \oplus \overline{c})] \tag{4.1}$$

and additionally, in case $C[0] \not\rightarrow^* C'[\overline{a}]$,

$$C[a.\overline{b}] \quad \geqslant \quad C[a.(\overline{b} \oplus \overline{c})] \tag{4.2}$$

$$C[a.\overline{c}] \quad \geqslant \quad C[a.(\overline{b} \oplus \overline{c})] \tag{4.3}$$

In (4.2,4.3) the condition on $C[\,\cdot\,]$ makes all related processes behave as $C[0]$ (up to strong bisimulation). In particular the two requirements of Lemma 4.18 on barbs and couplings are easily met.

In (4.1), the requirement on barbs can be reformulated as the simple may-testing equation $a.\overline{b} \oplus a.\overline{c} \simeq_{may} a.(\overline{b} \oplus \overline{c})$, but the requirement on coupling diagrams requires some more work:

For the first diagram, let us assume that $C[a.\overline{b} \oplus a.\overline{c}] \rightarrow^* T$. We distinguish several cases:

1. the process $a.\overline{b} \oplus a.\overline{c}$ does not reduce; hence the enclosing context cannot interact with this process and for some other context $C'[\,\cdot\,]$ we have $C[0] \rightarrow^* C'[0]$ and $T \equiv C'[a.\overline{b} \oplus a.\overline{c}]$. The same series of reductions applied on the other side yields the process $C'[a.(\overline{b} \oplus \overline{c})]$. These two resulting processes are still related by (4.1).
2. the process $a.\overline{b} \oplus a.\overline{c}$ reduces to, e.g., $a.\overline{b}$, and
   (a) either the enclosing context does not interact with this $a.\overline{b}$, and instead it reduces to a context that cannot emit on a anymore. For some $C'[\,\cdot\,]$ that has no reduction $C'[0] \rightarrow^* C''[\overline{a}]$, we have $C[0] \rightarrow^* C'[0]$ and $T \equiv C'[a.\overline{b}]$. We perform the same series of reductions from $C[\,\cdot\,]$ to $C'[\,\cdot\,]$ on the other side. The two resulting processes are related by (4.2).
   (b) or the context emits some a that interacts with the resulting process $a.\overline{b}$. In that case, $C[a.(\overline{b} \oplus \overline{c})] \rightarrow^* T$ by using the same series of reductions, except for the internal choice which has to be deferred until communication on a enables it. The two resulting processes are the same.
   (c) or this interaction does not occur, but the context can still emit on a. That is, we have $T \rightarrow^* T'$ where the series $C[a.\overline{b} \oplus a.\overline{c}] \rightarrow^* T'$ is obtained as in case (2b). The two resulting processes T and T' are related by $\rightarrow^*$.

For the second diagram, we perform a similar case analysis, but the situation is simpler.

1. the context does not interact with the process (which is inert in isolation); the two resulting processes are still related by (4.1).

2. the context provides a message $\overline{a}$ that is received by the process, and
   (a) either the internal choice is reduced in the following reductions. We anticipate the right choice in the left process, then apply the same series of reductions and obtain the same processes on both sides.
   (b) or the internal choice is not reduced. We select any branch of the choice in the left process, then perform the same series of reductions. The two resulting processes are related by the reduction that chooses the same branch in the right process.

The same counter-example holds in the join-calculus, for the processes

$$\begin{array}{rl} & \mathsf{def}\ x\langle\rangle \mid a\langle\rangle \triangleright b\langle\rangle \oplus c\langle\rangle\ \mathsf{in}\ x\langle\rangle \mid y\langle a\rangle \\ \text{versus} & \mathsf{def}\ x\langle u\rangle \mid a\langle\rangle \triangleright u\langle\rangle\ \mathsf{in}\ (x\langle b\rangle \oplus x\langle c\rangle) \mid y\langle a\rangle \end{array}$$

the proof is the same, except that extraneous pairs of processes are required to describe how $y\langle a\rangle$ provides access to $a\langle\rangle$ from the context. □

4.5.3 A model of coupled simulations

In the join-calculus, every coupled-barbed simulation that is also a precongruence is included in $\simeq_{fair}$, therefore we have $\dot{\lessgtr}^{\circ} \subseteq \simeq_{fair}$. Surprisingly, these two equivalences coincide, which places coupled simulations and testing equivalences in the same tier, and pragmatically provides useful proof techniques to establish fair-testing equivalences.

To show that $\dot{\lessgtr}^{\circ} = \simeq_{fair}$, we describe coupled simulations in terms of classes of processes that are entirely defined by the barbs $\Downarrow_x$ and $\downarrow_x$. We first consider a particular family of processes whose behavior is especially simple. We say that a process P is *committed* when, for all x, we have $P \Downarrow_x$ iff $P \downarrow_x$. When this is the case, no internal reduction may affect the barbs anymore: let S be the set of names $\{x/P\downarrow_x\} = \{x/P \Downarrow_x\}$; for all P' such that $P \rightarrow^* P'$, P' is still committed to S. In a sense, P has converged to S, which entirely captures its weak behavior.

To every join-calculus process P, we associate the semantics $P^\flat \in \mathbb{P}(\mathbb{P}(\mathcal{N}))$ that collects these sets of names for all the committed derivatives of P:

$$P^\flat \quad \stackrel{\text{def}}{=} \quad \left\{S \subseteq \mathcal{N} \mid \exists P' \cdot P \rightarrow^* P' \text{ and } S = \{x \,|\, P' \downarrow_x\} = \{x \,|\, P' \Downarrow_x\}\right\}$$

For example, $0^\flat$ is the singleton $\{\emptyset\}$ and $(x\langle\rangle \oplus y\langle\rangle)^\flat$ is the pair $\{\{x\},\{y\}\}$. As is the case for weak barbs, $P^\flat$ decreases by reduction. Besides, the predicates $\Downarrow_x$ and $\Downarrow_{\Box x}$ are easily recovered from our semantics:

$$\begin{array}{rll} P \Downarrow_x & \text{iff} & x \in \bigcup P^\flat \\ P \Downarrow_{\Box x} & \text{iff} & x \in \bigcap P^\flat \end{array}$$

Let $\subseteq_\flat$ be the preorder defined as $P \subseteq_\flat Q \stackrel{\text{def}}{=} P^\flat \subseteq Q^\flat$. By definition of may testing and fair testing, we immediately obtain that $\subseteq_\flat{}^\circ \subseteq \sqsubseteq_{may}$ and $\subseteq_\flat{}^\circ \subseteq \sqsubseteq_{fair}^{-1}$. Actually, the last two preorders coincide.

Lemma 4.20 *In the join-calculus,* $\subseteq_\flat{}^\circ = \sqsubseteq_{fair}^{-1}$.

Proof: For all finite sets of names S and N such that $S \subseteq N \subset \mathcal{N} \setminus \{t\}$, we define the context $T_S^N[\cdot]$ as follows:

$$T_S^N[\cdot] \stackrel{\text{def}}{=} \mathsf{def} \begin{array}{ll} & \mathit{once}\langle\rangle \triangleright t\langle\rangle \\ \wedge & \mathit{once}\langle\rangle \mid (\prod_{x \in S} x\langle\rangle) \triangleright 0 \\ \wedge & \bigwedge_{x \in N \setminus S} x\langle\rangle \triangleright t\langle\rangle \end{array} \mathsf{in}\ \mathit{once}\langle\rangle \mid [\cdot]$$

We show that the context $T_S^N[\cdot]$ fair tests exactly one set of names in our semantics, that is, for all P such that $\mathsf{fv}[P] \subseteq N$, we have $T_S^N[P] \Downarrow_{\Box t}$ if and only if $S \notin P^\flat$.

The first two clauses of the definition are competing for the single message $\mathit{once}\langle\rangle$, hence at most one of the two may be triggered. The clause $\mathit{once}\langle\rangle \triangleright t\langle\rangle$ can always be triggered; it releases the message $t\langle\rangle$. The second clause can preempt this reduction by consuming a message for every name in S. Independently, the third clause detects the presence of any barb in $N \setminus S$.

The predicate $T_S^N[P] \Downarrow_{\Box t}$ is true iff all the derivatives of P keep one of the two possibilities to emit the message $t\langle\rangle$, namely either do not have messages on all the names in S, or otherwise can always provide a message on a name outside of S.

Let $P \sqsubseteq_{fair}^{-1} Q$. We check that $P^\flat \subseteq Q^\flat$: we let $N = \mathsf{fv}[P] \cup \mathsf{fv}[Q]$. For every set of names $S \subseteq N$, we have $S \in P^\flat$ iff $T_S^N[P] \not\Downarrow_{\Box t}$, which implies $T_S^N[Q] \not\Downarrow_{\Box t}$ and $S \in Q^\flat$. For every other set of names S, none of $P^\flat$ or $Q^\flat$ may contain S anyway.

Thus, $\sqsubseteq_{fair}^{-1} \subseteq \subseteq_\flat$, by precongruence property of fair-testing $\sqsubseteq_{fair}^{-1} \subseteq \subseteq_\flat{}^\circ$, and, since the converse inclusion follows from the above characterization of fair barbs, $\subseteq_\flat{}^\circ = \sqsubseteq_{fair}^{-1}$. $\Box$

The next remark will be useful to relate $\subseteq_\flat$ to $\dot{\lessgtr}$:

Lemma 4.21 *The set $P^\flat$ is never empty.*

Proof: For every process P, we consider the set that contains all series of derivatives $P = P'_0 \rightarrow^* P'_1 \rightarrow^* \cdots \rightarrow^* P'_n$ such that the set of strong barbs $\{x / P'_i \downarrow_x\}$ is strictly increasing, and we order these series by prefix inclusion.

For every set of names S, we have $S \in P^\flat$ iff there is a maximal series of derivatives whose last process P'_n is such that $S = \{x / P'_n \downarrow_x\}$.

The set of increasing derivations defined above is not empty (it contains the series P) so the length of the series is bounded by the number of free variables in P. Hence, there is a maximal series of derivation and $P^\flat$ contains at least its associated set of names. $\Box$

The next lemma shows that our semantics refines barbed coupled similarity.

Lemma 4.22 $(\subseteq_\flat, \supseteq_\flat)$ *are coupled barbed simulations, and thus* $\subseteq_\flat \cap \supseteq_\flat \subseteq \dot{\lessgtr}$.

Proof: We successively check that $\subseteq_\flat$ preserves the barbs, is a simulation, and meets the coupling diagram. Let us assume $P \subseteq_\flat Q$

1. the barbs can be recovered from the semantics: $P \Downarrow_x$ iff $x \in \bigcup P^\flat$, and if $P \subseteq_\flat Q$ then also $x \in \bigcup Q^\flat$ and $Q \Downarrow_x$. hence $P \Downarrow_x$ implies $Q \Downarrow_x$.

2. weak simulation trivially holds: by definition, $P^\flat$ decreases with reductions, and is stable iff $P^\flat$ is a singleton; for every reduction $P \rightarrow P'$, $P' \subseteq_\flat P \subseteq_\flat Q$, and thus reductions in P are simulated by the absence of reduction in Q.

3. by Lemma 4.21 $P^\flat$ is not empty. Let $S \in P^\flat$. By hypothesis $S \in Q^\flat$, and thus for some process Q'_S we have $Q \rightarrow^* Q'_S$ and ${Q'_S}^\flat = \{S\} \subseteq P^\flat$, which provides the coupling from $\subseteq_\flat$ to $\supseteq_\flat$. □

By combining Lemmas 4.20 and 4.22, we finally obtain the coincidence of the congruence of coupled barbed similarity with fair testing. This result is specific to the join-calculus; a counterexample is given for CCS in Section 4.6.1.

Lemma 4.23 *In the join-calculus,* $\simeq_{fair} = \overset{\bullet}{\lessgtr}{}^{\exists}$

While it is interesting to have an exact characterization of fair testing both as the congruence as single-barbed bisimulation and as the congruence of coupled barbed simulations, it seems that the associated proof techniques are still delicate to use. Pragmatically, the proof technique associated with $\lessgtr$ seems more convenient, when available.

4.6 A hierarchy of equivalences

Our main results on equivalences are summarized in Figure 4.1. Each tier gathers several definitions that yield the same equivalence, at least in the join-calculus. We explicitly mention the existential variants of equivalences only when they differ from the equivalences obtained for all barbs. All tiers are distinct—vertical lines represent strict inclusion of equivalences. The gap between two tiers is briefly described in terms of discriminating power. All equivalences are full congruences.

The upper tier anticipates on the results of the next chapter; it contains several variants of weak labeled bisimulations. These equivalences are strictly finer than barbed congruence. In the next chapter, we also study a variant of the join-calculus with primitive name-matching. Then, the hierarchy is unchanged, except that the equivalences are congruences only for evaluation contexts, and that the two upper tier coincide.

4.6.1 The situation in the π-calculus

Next we give a survey of the situation in the π-calculus, which is presented in more details in [56]. The overall hierarchy is the same, with similar arguments; indeed, most of the proofs directly carries over the π-calculus. Still, there are a few significant differences that we discuss below.

The properties of equivalences are seemingly not as simple as in the join-calculus, mostly because strong barbs can be *transient*. For instance, the process $P = \overline{x}\langle\rangle \mid x().\mathsf{0}$ reduces to $\mathsf{0}$, and we have $P \downarrow_x$, $\mathsf{0} \not\Downarrow_x$. While the resulting equivalences are less natural than in the join-calculus, we can still recover our specific results, provided that we alter the basic notion of observation. More precisely, the results obtained for the join-calculus carry over the asynchronous π-calculus provided that we use "committed strong barbs" $\downdownarrows_x$ in place of strong barbs $\downarrow_x$ in every definition.

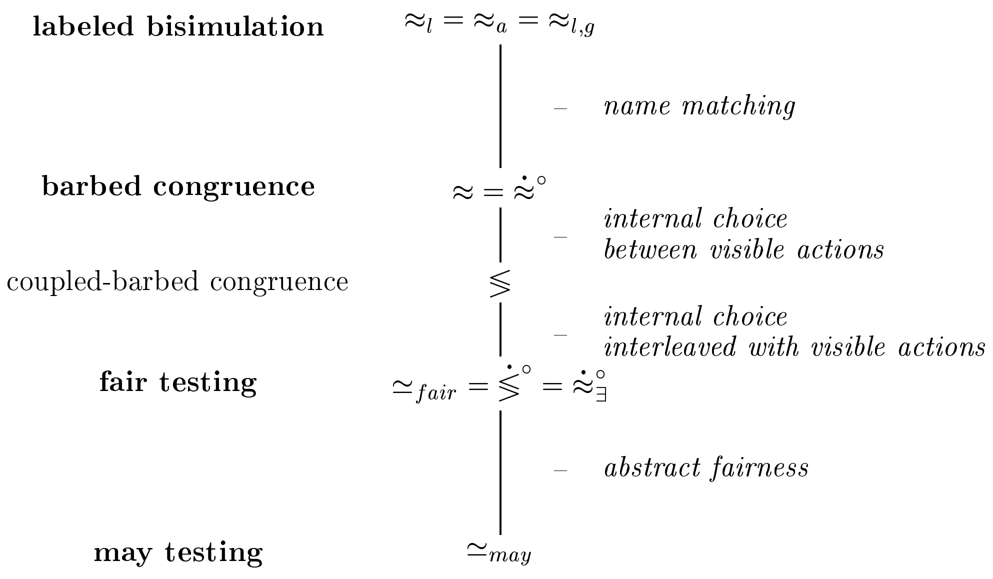

labeled bisimulation
$\approx_l = \approx_a = \approx_{l,g}$
– name matching
barbed congruence
$\approx = \dot{\approx}^\circ$
– internal choice between visible actions
coupled-barbed congruence
$\lesseqgtr$
– internal choice interleaved with visible actions
fair testing
$\simeq_{fair} = \dot{\lesseqgtr}^\circ = \dot{\approx}^\circ_\exists$
– abstract fairness
may testing
$\simeq_{may}$

Figure 4.1: A hierarchy of equivalences for the join-calculus.

Definition 4.24 *The predicate $\downdownarrows_x$—the committed strong barb on x—detects whether P permanently emits on x: $P \downdownarrows_x \overset{def}{=} \forall P', P \rightarrow^* P',\ P' \downarrow_x$*

In the join-calculus, the locality property enforces the identity $\downdownarrows_x = \downarrow_x$ for all free names; in the π-calculus, however, the new predicate $\downdownarrows_x$ is strictly stronger than $\downarrow_x$; for instance we have for the process P above $P \downarrow_x$ and $P \not\downdownarrows_x$.

As is the case with existential variants in this chapter, the underlying choice between $\downdownarrows_x$ and $\downarrow_x$ in the definition of π-calculus equivalences induces variants. We keep the standard notation when transient barbs are used, and index our equivalence with $\downdownarrows$ when committed barbs are used instead.

Fortunately, this variation does not affect the discriminating power of most equivalences in our hierarchy. When present, the congruence property can be used to encode weak committed predicates. It suffices to replace transient barbs with committed barbs that may grab transient messages and relay this detection without further interaction with the process. We use the context

$$T_x[\cdot] \quad \overset{\text{def}}{=} \quad \nu x.(x().\bar{t}\langle\rangle \,|\, [\cdot])$$

and for every P such that $t \notin \mathsf{fv}[P]$ we have $P \Downarrow_x$ iff $T_x[P] \downdownarrows_t$.

Discrepancies between the two calculi appear as we define equivalences in two stages, as the congruence of bisimulation-based equivalences. We survey the few equivalences where committed barbs make a difference; all other results apply unchanged in an asynchronous π-calculus setting. Especially, we retain the important property $\approx = \dot{\approx}^\circ$, with similar techniques as those of Section 4.8.

Single-barbed bisimilarities

With a single committed barb, weak bisimilarity $\dot{\approx}_{\exists,\downdownarrows}$ separates only three classes of processes, but with a single transient barb the situation gets more complicated.

Precisely, the model of existential weak bisimilarity $\dot{\approx}_\exists$ separates the following classes of processes

$$\begin{tikzcd}
\vdots & \\
\oplus \arrow[dd] \arrow[dddr] & \vdots \\
 & \oplus \arrow[dd] \arrow[dddl] \\
\oplus \arrow[dd] \arrow[dddr] & \\
 & \overline{x}\langle\rangle \oplus (\overline{x}\langle\rangle \mid x()) \arrow[dd] \arrow[dddl] \\
\overline{x}\langle\rangle \oplus \mathsf{0} \arrow[dd] \arrow[dddr] & \\
 & \overline{x}\langle\rangle \mid x() \arrow[dd] \\
\overline{x}\langle\rangle & \\
 & \mathsf{0}
\end{tikzcd}$$

plus a single infinite-branching class that can reduce to any finite-branching class described above. Besides the "obvious" terms $T_0 = \mathsf{0}$, $T_1 = \overline{x}\langle\rangle$, we have the term $T_2 = \overline{x}\langle\rangle \mid x().\mathsf{0}$ whose only and peculiar behavior is to rescind its $\downarrow_x$-barb to become $\mathsf{0}$. From these three terms one can construct a quasi-linear sequence of terms, setting $T_{i+3} = T_i \oplus T_{i+1}$. Furthermore, it is possible to construct a context $T[\,\cdot\,]$ such that $T[\mathit{int}\langle\mathtt{n}\rangle] \approx T_n$ along the lines of Lemma 4.33, hence to construct a limit term $T_\omega \approx \bigoplus_{i\in\mathbb{N}} T_i$. These T_i span exactly the equivalence classes of $\dot{\approx}_\exists^\circ$:

Proposition 4.25 *For any π-calculus term P, there is a unique $j \in \mathbb{N} \cup \{\omega\}$ such that $P \dot{\approx}_\exists^\circ T_j$.*

Proof: We partition π-calculus processes as follows. We have $P \dot{\approx}_\exists^\circ T_0$ iff $P \not\Downarrow$. By induction on $n \in \mathbb{N}$, we show that $P \dot{\approx}_\exists^\circ T_{n+1}$ iff $P \Downarrow$ and $P \rightarrow^* \dot{\approx}_\exists^\circ T_i$ for all $i < n$ but not for $i = n$. Finally, $P \dot{\approx}_\exists^\circ T_\omega$ iff $P \rightarrow^* \dot{\approx}_\exists^\circ T_n$ for all $n \in \mathbb{N}$, and this covers all remaining cases. $\square$

The congruence of this bisimilarity yields an hybrid equivalence, in a new tier between fair-testing and barbed congruence: we have the strict inclusions

$$\simeq_{fair} \subset \dot{\approx}_\exists^\circ \subset \dot{\approx}^\circ$$

In [56] we refine this negative result by showing that, rather surprisingly, existential barbed congruence is an *inductive*, or limit, bisimulation.

Fair testing versus coupled simulations

While in general we have the inclusion $\dot{\lessgtr}^\circ \subseteq \simeq_{fair}$, the coincidence between these two equivalences obtained for the join-calculus in Lemma 4.23 specifically relies on committed barbs.

It is tempting to adapt the semantics of Section 4.5.3 to the asynchronous π-calculus, for instance by defining committed processes as processes P such that $\{x \mid P \Downarrow_x\}$

equals $\{x \mid P \Downarrow_{\Box x}\}$. For instance, the semantics of $\overline{x}\langle\rangle \mid !x().\overline{y}\langle\rangle \mid !y().\overline{x}\langle\rangle$ would be the singleton set $\{\{x, y\}\}$. Let $\subseteq_\flat$ be the induced preorder; this preorder does not capture the weak barbs, but nonetheless we obtain the characterization $\leqslant \; = \; \subseteq_\flat \cap \sqsubseteq_{may}$. The main difference is that fair testing is not finer than the congruence induced by $\subseteq_\flat$; in particular, the counterpart of Lemma 4.20 fails, because the context we use cannot test for the presence of several transient messages.

With transient barbs, fair testing is strictly coarser than the congruence of coupled simulation. We give a counter-example for asynchronous CCS. The two processes a and $a \oplus 0$ are fair testing equivalent, while they are separated by coupled simulation in the context $\overline{a} \mid [\cdot]$. As this is the only direct use of fair testing as a proof technique in this dissertation, we detail the proofs of these claims:

Proof of $a \simeq_{fair} a \oplus 0$: By definition of fair testing, we have $\rightarrow\sim \; \subseteq \; \sqsubseteq_{fair}$, and thus $a \oplus 0 \sqsubseteq_{fair} a$. Conversely, let $C[\cdot]$ be a given evaluation context and x be a nullary name such that $C[a] \Downarrow_{\Box x}$. Let us also assume that $C[a \oplus 0] \rightarrow^* T$; in order to prove that $T \Downarrow_x$, we build a series of reductions $C[a] \rightarrow^* T'$ such that T has all the weak barbs of T'. We distinguish several cases, according to the actual use of $a \oplus 0$ in the reductions.

In case a is consumed, we obtain the reductions $C[a] \rightarrow^* T$ simply by deleting the reduction $a \oplus 0 \rightarrow a$ in context.

Otherwise, the context never interacts with its embedded process, and thus for some evaluation context $C'[\cdot]$ we have the reductions $C[a] \rightarrow^* C'[a]$ where $T \approx C'[P]$ for one of the processes 0, a, or $x \oplus 0$. For 0, we have $a \sqsubseteq_{may} 0$; for other processes $T \rightarrow^{=}\sim T'$. ☐

Proof of $\overline{a} \mid a \; \dot{\not\lessgtr} \; \overline{a} \mid a \oplus 0$: By applying the barbed simulation property for the reduction $\overline{a} \mid a \oplus 0 \rightarrow \overline{a}$, for some P we have $\overline{a} \mid a \rightarrow^* P$ and $P \geqslant \overline{a}$; however the only strict derivative of $\overline{a} \mid a$ is 0, which doesn't have a barb on a. Thus, $P = \overline{a} \mid a \geqslant \overline{a}$.

By applying the coupling condition, we must have either $\overline{a} \mid a \leqslant \overline{a}$ or $0 \leqslant \overline{a}$, where the first relation actually implies the second one by simulation.

By coupling again—whenever we have $P \leqslant Q$ and $Q \not\rightarrow$, the coupling condition implies $P \; \dot{\lessgtr} \; Q$—we obtain that $0 \; \dot{\lessgtr} \; \overline{a}$, which is inconsistent because the two processes differ on the barb $\Downarrow_a$. ☐

4.7 Techniques of bisimulation "up to"

In order to prove by co-induction that two processes P and Q are related by some equivalence, a relation $\mathcal{R}$ containing the pair (P, Q) is chosen and proved to meet all the requirements in the definition of the equivalence.

The candidate relation $\mathcal{R}$ should be as small as possible, since this directly affects the size of the proof obligations. Unfortunately, the functional $\mathcal{F}()$ that defines the equivalence often requires that $\mathcal{R}$ be quite large. To circumvent this problem, numerous proof techniques show that it suffices to establish these requirements on much smaller candidate relations $\mathcal{R}' \subset \mathcal{R}$. (See, for instance, [137, 131].) Formally, any coarser monotonous functional $\mathcal{F}'()$ that has the same greatest fix-point as $\mathcal{F}()$ can be preferred in a proof.

These so-called "up to" techniques are routinely used in most bisimulation proofs. Nonetheless, small technical variations are often required, and each new combination of up to techniques has to be carefully established. Soundness can be obtained by tedious diagram-chasing arguments, but it is quite easy to overlook a case, hence to attempt a proof on top of unsound techniques.

In this section, we present a method for gluing diagrams in an systematic manner, and thus automatically derive sound "up-to" techniques. Our approach makes explicit the connection between bisimulation proofs and general confluence results developed for term rewriting systems.

4.7.1 Confluence by decreasing diagrams

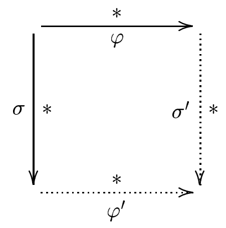
We use a general confluence theorem of Van Oostrom [112]. From an abstract point of view, we can consider that processes are related by a arbitrary family of relations $\rightarrow_i$ indexed by labels $i \in I$ that contains relations of interest such as reductions $\rightarrow$, equivalences $\approx$, and candidate relations $\mathcal{R}$. We then study commutation diagrams between these relations, and recover bisimulation diagrams as special cases.

We order our relations as follows. We let $<$ be a strict partial order on labels in I. Intuitively, the inequation $\mathcal{R}_1 < \mathcal{R}_2$ means that $\mathcal{R}_1$ is negligible in diagrams when it appears after $\mathcal{R}_2$. We measure sequences of reductions by multisets of labels: to every string σ and multiset S of labels, we associate the multiset $[\sigma]S$ inductively defined as follows:

$$\begin{aligned}
[\epsilon]S &\overset{\text{def}}{=} S \\
[i\sigma]S &\overset{\text{def}}{=} \text{if } i < j \text{ for some } j \in S \text{ then } [\sigma]S \text{ else } [\sigma](S \cup \{i\})
\end{aligned}$$

That is, the multiset collects all labels that are not smaller than labels already collected. We also define our measure on strings of labels as $[\sigma] \overset{\text{def}}{=} [\sigma]\emptyset$. For instance, with digits instead of labels, we would have the measures $[1512]=\{1,5\}$ and $[16116983]=\{1,6,6,9\}$.

Definition 4.26 *Let $\sigma, \sigma', \varphi, \varphi' \in I^*$ be strings of labels ; the diagram*



$$\begin{tikzcd}
\cdot \arrow[r, "*", "\varphi"'] \arrow[d, "\sigma"', "*"] & \cdot \arrow[d, dotted, "\sigma'"', "*"] \\
\cdot \arrow[r, dotted, "*", "\varphi'"'] & \cdot
\end{tikzcd}$$

is decreasing *when we have the multiset inclusions $[\varphi\sigma'] \subseteq [\varphi] \cup [\sigma] \supseteq [\sigma\varphi']$.*

Decreasing diagrams can be vertically or horizontally glued, which yields larger, decreasing diagrams; this captures in an abstract manner the use of smaller relations to close diagrams. We recall the main theorem of [112] with our notations:

Theorem 4 (Van Oostrom) *Let $(I, <)$ be a well founded partially ordered set of labels and $(\mathcal{P}, (\rightarrow_i)_{i \in I})$ be a rewriting system.*

Let also $H, V \subseteq I$ be two subsets of labels, and $\rightarrow_h, \rightarrow_v$ be the relations defined as $\rightarrow_h \overset{def}{=} \bigcup_{i \in H} \rightarrow_i$ and $\rightarrow_v \overset{def}{=} \bigcup_{i \in V} \rightarrow_i$.

If for every pair of labels $(i, j) \in H \times V$, there exist sequences of labels $(\varphi, \sigma) \in H^* \times V^*$ such that the first diagram below is decreasing

$$\begin{tikzcd}
{} \arrow[r, "i"'] \arrow[d, "j"'] & {} \arrow[d, dotted, "\sigma"', "*"] \\
{} \arrow[r, dotted, "*", "\varphi"'] & {}
\end{tikzcd}
\qquad \textit{then we have the diagram} \qquad
\begin{tikzcd}
{} \arrow[r, "*", "h"'] \arrow[d, "v"', "*"] & {} \arrow[d, dotted, "v"', "*"] \\
{} \arrow[r, dotted, "*", "h"'] & {}
\end{tikzcd}$$

The reductions $\rightarrow_h$ and $\rightarrow_v$ stand for "horizontal reduction" and "vertical reductions", but the corresponding sets H and V are not necessarily disjoint; for instance the confluence property is obtained when horizontal and vertical reductions are the same.

4.7.2 Weak bisimulation up to bisimilarity

Next we apply this technique for establishing several "up to" techniques. We first derive the simple barbed variant of the weak bisimulation up to bisimilarity lemma presented in [97]. Our horizontal reductions are $\dot{\approx}$, $\mathcal{R}^=$, and $\dot{\sim}$; our vertical reductions are plain reductions $\rightarrow$. We choose the ordering

$$\dot{\sim} < \rightarrow < \dot{\approx} < \mathcal{R}^=$$

The premises of Theorem 4 consist of the three decreasing diagrams

$$\begin{tikzcd}
{} \arrow[r, dash, "\dot{\sim}"] \arrow[d] & {} \arrow[d, dotted] \\
{} \arrow[r, dash, dotted, "\dot{\sim}"] & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[r, dash, "\dot{\approx}"] \arrow[d] & {} \arrow[d, dotted, "*"] \\
{} \arrow[r, dash, dotted, "\dot{\sim}\dot{\approx}\dot{\sim}"] & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[r, dash, "\mathcal{R}^="] \arrow[d] & {} \arrow[d, dotted, "*"] \\
{} \arrow[r, dash, dotted, "\dot{\sim}\mathcal{R}^=\dot{\approx}"] & {}
\end{tikzcd}$$

The first two diagrams hold by definition of $\dot{\sim}$ and $\dot{\approx}$; the first one is strong simulation; the second one is subsumed by weak simulation. Only the last diagram depends on $\mathcal{R}$; moreover, we can substitute $\mathcal{R}$ for $\mathcal{R}^=$ at the top of this diagram.

By applying Theorem 4, we obtain that the relation $\mathcal{R}' = (\dot{\approx} \cup \mathcal{R}^= \cup \dot{\sim})^*$ commutes with $\rightarrow^*$, that is, that this relation is a weak simulation. Besides, we obtain that the inverse relation $(\mathcal{R}')^{-1}$ is a weak bisimulation by applying Theorem 4 again after substituting $\mathcal{R}^{-1}$ for $\mathcal{R}$. In the case we also require that $\mathcal{R}$ preserve all barbs, $\mathcal{R}'$ retains this property, thus $\mathcal{R}'$ is a barbed bisimulation. By definition of $\dot{\approx}$ as the union of all barbed bisimulations, we have $\mathcal{R}' = (\dot{\approx} \cup \mathcal{R} \cup \dot{\sim})^* \subseteq \dot{\approx}$, and in particular $\mathcal{R} \subseteq \dot{\approx}$. Finally, we discard all the diagrams that do not depend on $\mathcal{R}$, and gather our sufficient requirements on $\mathcal{R}$ as follows:

Lemma 4.27 *$\mathcal{R}$ is a weak bisimulation if it respects all barbs and meets the two diagrams*

$$\begin{tikzcd} {} \arrow[r, no head, "\mathcal{R}"] \arrow[d] & {} \arrow[d, dotted, "*"] \\ {} \arrow[r, no head, dotted, "\dot{\sim}\mathcal{R}=\dot{\approx}"] & {} \end{tikzcd}$$

and

$$\begin{tikzcd} {} \arrow[r, no head, "\mathcal{R}"] \arrow[d, dotted, "*"] & {} \arrow[d] \\ {} \arrow[r, no head, dotted, "\dot{\approx}\mathcal{R}=\dot{\sim}"] & {} \end{tikzcd}$$

.

Conversely, we would not be able to complete our programme if we tried to establish the spurious "weak-bisimulation up to weak-bisimilarity" proof technique, whose statement is obtained from the one of Lemma 4.27 by substituting $\dot{\approx}$ for $\dot{\sim}$ at the bottom of the diagrams. These diagrams are not decreasing anymore for our ordering, as it would require that $\dot{\approx} < \rightarrow$. Yet, any other ordering that would contain this inequation would also require a strong bisimulation diagram for $\dot{\approx}$ and $\rightarrow$.

This so-called problem of "weak bisimulation up to" is the starting point of [137], where Milner and Sangiorgi first provide a counter-example for the spurious proof technique—in CCS the processes $\tau.P$ and $\mathsf{0}$ are weakly-bisimilar up to weak bisimilarity, but usually not weakly-bisimilar—then they propose coarser requirements than those of Lemma 4.27 that still preserve soundness. To this end, they introduce the notion of *expansions*, which we discuss next.

4.7.3 Expansions

In most proofs, we need a finer, auxiliary *expansion* relation. This relation was first proposed as a proof technique in [137]. The expansion relation refines bisimulations in an asymmetric manner, which represents the notion of progress, or implementation.

Definition 4.28 *A* barbed expansion *is a relation $\mathcal{R}$ that respects the weak barbs and meets the two diagrams*

$$\begin{tikzcd} {} \arrow[r, no head, "\mathcal{R}"] \arrow[d] & {} \arrow[d, dotted, "="] \\ {} \arrow[r, no head, dotted, "\mathcal{R}"] & {} \end{tikzcd}$$

and

$$\begin{tikzcd} {} \arrow[r, no head, "\mathcal{R}"] \arrow[d, dotted, "*"] & {} \arrow[d] \\ {} \arrow[r, no head, dotted, "\mathcal{R}"] & {} \end{tikzcd}$$

.

We let $\dot{\succeq}$ be the largest expansion, $\succeq$ be the largest expansion that is a precongruence, $\dot{\asymp}$ be the largest expansion whose inverse is an expansion, and $\asymp$ be the largest expansion that is a congruence and whose inverse is an expansion.

All these relations are in-between the corresponding weak and strong bisimilarities. By definition, an expansion is a weak bisimulation with an additional, asymmetric requirement: steps on the "large" side of the expansion must be matched by fewer steps on the small side. This places $\succeq$ strictly between weak and strong bisimulation. Remark that $\dot{\succeq} \cap \dot{\preceq}$ is in general coarser than $\dot{\asymp}$.

The relation $\dot{\asymp}$ is an equivalence that is almost as fine as strong bisimulation, since only "useless" extraneous reductions are allowed on either side. For instance, we have $\Omega \,|\, P \asymp P$ for any process Ω with no free variables, even if this process is diverging. In practice, this relation is always a coarser alternative to strong bisimilarity in up to techniques.

The key technical property of expansions is that they can be used to close weak bisimulation diagrams. Typically, a large proof of weak bisimulation can be made modular by first isolating a few intermediate lemmas that state simplifications as expansion relations, then conducting a main proof of weak bisimulation up to expansion.

As regards barbed congruences, we describe two co-inductive proof techniques, namely barbed congruence up to expansion and expansion up to expansion. Lemmas 4.29 and 4.30 are technical lemmas that state sufficient conditions for relations to be included in the relations of barbed congruence and expansion, respectively; they are the basis for many proofs.

Before stating and proving these proof techniques, we discuss how their bisimulation diagrams can be obtained from Theorem 4. Since expansions are asymmetric, we independently insert pairs of inverse relations $\dot{\preceq}, \dot{\succeq}$ and $\mathcal{R}, \mathcal{R}^{-1}$ in the strict partial ordering.

- This induces two additional decreasing diagram requirements as premises of Theorem 4, for the pairs of labels ($\dot{\succeq}$,$\rightarrow$) and ($\dot{\preceq}$,$\rightarrow$). We easily check that these diagrams are subsumed by those of Definition 4.28 inasmuch as $\rightarrow \; < \; \dot{\preceq}$. In contrast, there is no constraint on $\dot{\succeq}$ so far.

- This also relaxes our proof requirements for the candidate relations $\mathcal{R}$ and $\mathcal{R}^{-1}$. We first derive sufficient diagrams to prove that $\mathcal{R} \subseteq \dot{\approx}$, then to prove that $\mathcal{R} \subseteq \dot{\preceq}$.

  To establish a weak bisimulation requirement for $\mathcal{R}$, we use the refined ordering $\dot{\succeq} < \rightarrow < \dot{\preceq} < \dot{\approx} < \mathcal{R}$ and obtain that

$$\begin{tikzcd} {} \arrow[r, dash, "\mathcal{R}"] \arrow[d] & {} \arrow[d, dotted, "*"] \\ {} \arrow[r, dash, dotted, "\dot{\succeq}\mathcal{R}=\dot{\approx}"] & {} \end{tikzcd}$$

suffices to derive

$$\begin{tikzcd} {} \arrow[r, dash, "(\dot{\approx}\cup\mathcal{R})^*"] \arrow[d, "*"'] & {} \arrow[d, dotted, "*"] \\ {} \arrow[r, dash, dotted, "(\dot{\approx}\cup\mathcal{R})^*"] & {} \end{tikzcd}$$

  To establish an expansion bisimulation requirement for $\mathcal{R}$, we use another ordering $\dot{\asymp} < \dot{\succeq} < \mathcal{R} < \rightarrow$ and obtain that

$$\begin{tikzcd} {} \arrow[r, dash, "\mathcal{R}"] \arrow[d] & {} \arrow[d, dotted, "="] \\ {} \arrow[r, dash, dotted, "\dot{\succeq}\mathcal{R}^*\dot{\succeq}"] & {} \end{tikzcd}$$

suffices to derive

$$\begin{tikzcd} {} \arrow[r, dash, "(\dot{\succeq}\cup\mathcal{R})^*"] \arrow[d] & {} \arrow[d, dotted, "="] \\ {} \arrow[r, dash, dotted, "(\dot{\succeq}\cup\mathcal{R})^*"] & {} \end{tikzcd}$$

Next we wrap these diagrammatic requirements with other requirements on barbs and contexts, to obtain simple proof techniques for later reference.

Lemma 4.29 (barbed congruence up to expansion) *To prove that a relation $\mathcal{R}$ is included in barbed congruence ($\mathcal{R} \subseteq \approx$), it suffices to prove that, for a fixed set of names $\mathcal{N}' \subseteq \mathcal{N}$ with an infinite number of names of every type, and for all processes P and Q such that $P \; \mathcal{R} \; Q$, we have the following properties:*

1. *If $P \downarrow_x$ then $Q \Downarrow_x$; conversely, if $Q \downarrow_x$ then $P \Downarrow_x$.*

2. *$C[P] \approx\mathcal{R}\approx C[Q]$ for every context $C[\cdot]$ of the form* $\mathsf{def}\ D\ \mathsf{in}\ R \,|\, [\cdot]$ *where* $\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ R] \subseteq \mathcal{N}'$.

3. *We have the diagrams*

$$\begin{tikzcd}
P \arrow[r, dash, "\mathcal{R}"] \arrow[d] & Q \arrow[d, dotted, "*"] \\
{} \arrow[r, dotted, dash, "\succeq\mathcal{R}^{=}\approx"] & {}
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
P \arrow[r, dash, "\mathcal{R}"] \arrow[d, dotted, "*"] & Q \arrow[d] \\
{} \arrow[r, dotted, dash, "\approx\mathcal{R}^{=}\preceq"] & {}
\end{tikzcd}\ .$$

Proof: we check that $\approx\mathcal{R}\approx$ is a barbed bisimulation and a congruence, as in the proof of the next lemma. □

Lemma 4.30 (expansion up to expansion) *To prove that a relation $\mathcal{R}$ is included in expansion ($\mathcal{R} \subseteq \succeq$), it suffices to prove that, for a fixed set of names $\mathcal{N}' \subseteq \mathcal{N}$ with an infinite number of names of every type, and for all processes P and Q such that $P \mathrel{\mathcal{R}} Q$, we have the following properties:*

1. *If $P \downarrow_x$ then $Q \Downarrow_x$; conversely, if $Q \downarrow_x$ then $P \Downarrow_x$.*

2. *$C[P] \succeq\mathcal{R}\succeq C[Q]$ for every context $C[\cdot]$ of the form* $\mathsf{def}\ D\ \mathsf{in}\ R \,|\, [\cdot]$ *where* $\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P] \subseteq \mathcal{N}'$.

3. *We have the diagrams*

$$\begin{tikzcd}
P \arrow[r, dash, "\mathcal{R}"] \arrow[d] & Q \arrow[d, dotted, "="] \\
{} \arrow[r, dotted, dash, "\succeq\mathcal{R}^{=}\succeq"] & {}
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
P \arrow[r, dash, "\mathcal{R}"] \arrow[d, dotted, "*"] & Q \arrow[d] \\
{} \arrow[r, dotted, dash, "\succeq\mathcal{R}^{=}\asymp"] & {}
\end{tikzcd}\ .$$

In the special case where $P \mathrel{\mathcal{R}} Q$ implies $P\ (\rightarrow^*\succeq)^*\ Q$, the requirements can be further weakened: property 1 is true as soon as $P \downarrow_x$ implies $Q \Downarrow_x$; property 3 is always true.

Proof: *General case:* we prove that the relation $\succeq\mathcal{R}^{=}\succeq$ matches all the properties required in the definition of $\succeq$ provided that $\mathcal{R}$ matches all the hypotheses of the lemma. This yields the inclusion $(\succeq\mathcal{R}^{=}\succeq) \subseteq \succeq$, and in particular $\mathcal{R} \subseteq \succeq$. Let us assume $P \succeq P_1 \mathrel{\mathcal{R}^{=}} Q_1 \succeq Q$:

- in case $P \Downarrow_x$, we have for some P' $P \rightarrow^* P' \downarrow_x$. By bisimulation of $\succeq$, hypothesis 3, and bisimulation of $\succeq$ again, we obtain for some Q' that $Q \rightarrow^* Q'$ and $P' \succeq\mathcal{R}^{=}\succeq Q'$. By hypothesis 1 we obtain $Q' \Downarrow_x$ and thus $Q \Downarrow_x$. The same argument applies to obtain $P \Downarrow_x$ in case $Q \Downarrow_x$, except that we use hypothesis 3 instead of hypothesis 3;

- the composition of congruences is a congruence;

- the two diagrams of Definition 4.28 are derived from the two diagrams of the lemma as described above.

Special case: we assume $\mathcal{R} \subseteq (\rightarrow^*\succeq)^*$, and we establish the sufficient properties of the general case for the relation $\succeq\mathcal{R}^=$. Let assume $P \succeq P_1\ \mathcal{R}\ Q$ and $P_1(\rightarrow^*\succeq)^n Q$. We build a derivation $P \rightarrow^* P' \succeq Q$ by applying $n+1$ times the bisimulation property of $\succeq$.

In case $Q \downarrow_x$, $P \rightarrow^* P' \Downarrow_x$ and thus $P \Downarrow_x$; in case $Q \rightarrow Q'$, by expansion we have $P' \rightarrow^* P'' \succeq Q'$ and thus $P \rightarrow^*\succeq Q'$; all other properties are immediate. $\square$

Our lemmas mention coarse diagrams that suffice to apply the confluence theorem. In order to establish the requirements of Lemmas 4.29 and 4.30, we can of course use relations stronger than those mentioned. For instance, since we have $\equiv\ \subseteq\ \sim\ \subseteq\ \asymp\ \subseteq\ \succeq\ \subseteq\ \approx$, all proofs can be performed up to structural equivalence and up to strong equivalence.

4.7.4 Accommodating deterministic reductions

Our last example of bisimulation by decreasing diagram involves another kind of “vertical relation”. It is lengthly, but actually used in Section 6.7 to establish a full abstraction result.

So far, the only vertical relation was the reduction relation, but in many proofs this does not suffice: some families of reductions are intuitively negligible in weak bisimulation diagrams, for instance when they are deterministic, or when they are partially confluent independently of any other reductions.

Let $\rightarrow_d$ be a subset of the reduction relation $\rightarrow$. The use of decreasing diagrams makes explicit the trade-off between the additional proof requirements for $\rightarrow_d$ (new commutation diagrams to establish) and the instances of the “small” reduction steps $\rightarrow_d$ that are negligible in larger diagrams to establish bisimulation properties.

By analogy with expansions, we use the notation $\succeq_d$ for the intermediate asymmetric relation that counts large steps $\rightarrow$ but not small steps $\rightarrow_d$. We use the sets of labels $H = \{\rightarrow_d, \leftarrow_d, \succeq_d, \approx, \mathcal{R}\}$ and $V = \{\downarrow_d, \downarrow\}$, and the following partial strict ordering

$$\rightarrow_d\ <\ \downarrow_d\ <\ \leftarrow_d\ <\ \succeq_d\ <\ \downarrow\ <\ \approx\ <\ \mathcal{R}$$

Enumerating the premises of the confluence theorem, we first have a series of simple diagrams that express the partial confluence property of $\rightarrow_d\ \subseteq\ \rightarrow$.

$$\begin{tikzcd}
\cdot \arrow[r, "d"'] \arrow[d, "d"'] & \cdot \arrow[d, dotted, "d"] \arrow[r, phantom, "="] & \cdot \arrow[d, "d"'] & \cdot \arrow[l, "d"] \arrow[d, dotted, "d"] & \cdot \arrow[r, "d"'] \arrow[d] & \cdot \arrow[d, dotted] \arrow[r, phantom, "="] & \cdot \arrow[d] & \cdot \arrow[l, "d"] \arrow[d, dotted] \\
\cdot \arrow[r, dotted, "=", "d"'] & \cdot & \cdot & \cdot \arrow[l, dotted, "d"] & \cdot \arrow[r, dotted, "=", "d"'] & \cdot & \cdot & \cdot \arrow[l, dotted, "d"]
\end{tikzcd}$$

Then we obtain the following sufficient diagrams for the intermediate relation $\succeq_d$:

$$\begin{tikzcd}
\cdot \arrow[r, dash, "\succeq_d"] \arrow[d, "d"'] & \cdot \arrow[d, dotted, "*"] \\
\cdot \arrow[r, dotted, "\rightarrow_d^*\succeq_d^=\leftrightarrow_d^*"] & \cdot
\end{tikzcd}$$

and

$$\begin{tikzcd}
\cdot \arrow[r, dash, "\succeq_d"] \arrow[d] & \cdot \arrow[d, dotted, "*"] \\
\cdot \arrow[r, dotted, "{[\leftrightarrow_d,\succeq_d]^*}"] & \cdot
\end{tikzcd}$$

The diagrams for weak bisimulation $\approx$ are still valid. Finally, the premises on the candidate weak simulation $\mathcal{R}$ can be written

$$\begin{tikzcd}
\cdot \arrow[r, dash, "\mathcal{R}"] \arrow[d, "d"'] & \cdot \arrow[d, dotted, "*"] \\
\cdot \arrow[r, dotted, "\rightarrow_d^* \mathcal{R}^= [\leftrightarrow_d, \succeq_d, \approx]^*"] & \cdot
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\cdot \arrow[r, dash, "\mathcal{R}"] \arrow[d] & \cdot \arrow[d, dotted, "*"] \\
\cdot \arrow[r, dotted, "{[\succeq_d, \leftrightarrow_d]^* \mathcal{R}^= [\leftrightarrow_d, \succeq_d, \approx]^*}"] & \cdot
\end{tikzcd}$$

4.8 Barbed equivalence versus barbed congruence

In this section we relate the two equivalences that are obtained from barbed bisimulation by requiring the congruence property either as part of the bisimulation definition ($\approx$) or after requiring bisimilarity ($\dot{\approx}^\circ$), as explained in Section 4.4.3. The two definition styles can lead to significant differences, as is the case for the relations $\dot{\approx}^\circ_\exists \neq \approx_\exists$ (Section 4.4.4) and $\dot{\lessgtr}^\circ \neq \lessgtr$ (Section 4.5.2).

In this section, we assume encodings for booleans and for integers à la Church inside the join-calculus (see, e.g., [99] for explicit encodings in the π-calculus; formally these encodings use only the deterministic fragment of the join-calculus). To every integer $n \in \mathbb{N}$, we associate the representation $\mathbf{n}$; we also assume that our integers come with operations is_zero$(\cdot) : \langle \mathtt{Int} \rangle \to \langle \mathtt{Bool} \rangle$ and pred$(\cdot) : \langle \mathtt{Int} \rangle \to \langle \mathtt{Int} \rangle$.

4.8.1 Double-barbed bisimulation

We refine our definition of barbed bisimulation according to the number of barbs that are required in the definition. For $n \in \mathbb{N}$, we use the notation $\dot{\approx}_n$ for the bisimilarity that tests for messages on n distinct nullary names, and $\dot{\approx}^\circ_n$ its congruence for evaluation contexts.

By definition, the discriminating power of $\dot{\approx}_n$ and $\dot{\approx}^\circ_n$ increases with n, and these relations remain coarser than $\dot{\approx}$ and $\dot{\approx}^\circ$, respectively. In the join-calculus, we have $\dot{\approx}^\circ_0 = \mathcal{P} \times \mathcal{P}$, $\dot{\approx}^\circ_1 = \dot{\approx}^\circ_\exists$ (this is easily established using the contexts of Section 4.1.5), and we are going to show that, for all $n > 1$, $\dot{\approx}^\circ_n = \approx$. To this end, we focus on $\dot{\approx}_2$, and we assume that $x, y \in \mathcal{N}_0$ are the names associated with the two barbs $\Downarrow_x$, $\Downarrow_y$ that are part of the definition of $\dot{\approx}_2$.

We first need an additional definition and a lemma to build processes that are non-equivalent and retain this property by reduction. We assume given a canonical ordering on all processes, and to every finite set of processes $\mathcal{P}$ we associate the internal sum $\oplus(\mathcal{P}) \stackrel{\text{def}}{=} \bigoplus_{P \in \mathcal{P}} P$ obtained for this ordering. Then, we define an operator $\mathcal{S}(\,\cdot\,)$ that maps every finite set of processes $\mathcal{P}$ to the set of its (strict, partial) internal sums:

$$\mathcal{S}(\mathcal{P}) \quad \stackrel{\text{def}}{=} \quad \left\{ \oplus(\mathcal{P}') \mid \mathcal{P}' \subseteq \mathcal{P} \text{ and } \mathcal{P}' \text{ is not a singleton} \right\}$$

for instance, we have

$$\mathcal{S}(\{\, x\langle\rangle,\ y\langle\rangle,\ \mathsf{0} \,\}) \quad \stackrel{\text{def}}{=} \quad \{\, x\langle\rangle \oplus y\langle\rangle \oplus \mathsf{0},\ x\langle\rangle \oplus y\langle\rangle,\ x\langle\rangle \oplus \mathsf{0},\ y\langle\rangle \oplus \mathsf{0} \,\}$$

The next lemma uses the $\mathcal{S}(\,\cdot\,)$ operator to build classes of non-bisimilar processes in a generic manner:

Lemma 4.31 *Let $\mathcal{R}$ be a weak bisimulation such that $\sim \subseteq \mathcal{R}$, and $\mathcal{P}$ be a set of processes such that, if $P, Q \in \mathcal{P}$ and either $P \rightarrow^* \mathcal{R}\ Q$ or $P\ \mathcal{R} \leftarrow^* Q$, then $P = Q$. Then we have:*

1. *the set $\mathcal{S}(\mathcal{P})$ retains this property;*
2. *the union $\bigcup_{n \geq 0} \mathcal{S}^n(\mathcal{P})$ that contains the iterated results of $\mathcal{S}(\cdot)$ only contains processes that are not related by $\mathcal{R}$.*

Proof: By construction the relation $(\rightarrow^* \mathcal{R})$ is closed by strong equivalence, and by reductions on the right: if $P \rightarrow^* P'\ \mathcal{R}\ Q$ and $Q \rightarrow^* \sim Q'$, then by weak simulation $P \rightarrow^* P' \rightarrow^* P''\ \mathcal{R}\ Q'$. Strong equivalence is required only because summation is a derived construct in the join-calculus.

Let P, Q be a pair of distinct processes in $\mathcal{S}(\mathcal{P})$. We prove that $P \rightarrow^* \mathcal{R}\ Q$ leads to some contradiction. We successively exclude the relations $P \rightarrow\rightarrow^* \mathcal{R}\ Q$ and $P\ \mathcal{R}\ Q$.

In case $P \rightarrow P'' \rightarrow^* \mathcal{R}\ Q$, there exists $P' \in \mathcal{P}$ such that $P' \sim P''$, and for all summands $Q' \in \mathcal{P}$ present in Q we have $Q \rightarrow \sim Q'$. By applying the remark above $P' \rightarrow^* \mathcal{R}\ Q'$, hence by hypothesis on $\mathcal{P}$ we also have $P' = Q'$. This is excluded because Q has at least two different summands.

In case $P\ \mathcal{R}\ Q$, at least one of the sums contains a summand that does not appear in the other. Let us assume that $P \rightarrow P'$ and $Q \not\rightarrow \sim P'$. By weak simulation, we have either $P'\ \mathcal{R}\ Q$—and thus $P \rightarrow \mathcal{R}\ Q$, which is an instance of the first case—or $P'\ \mathcal{R} \leftarrow^* Q' \leftarrow Q$—which contradicts the hypothesis on $\mathcal{P}$ for the $P'', Q'' \in \mathcal{P}$ such that $P'' \sim P'$ and $Q'' \sim Q'$.

To establish the second part of the lemma, we first obtain that all sets of processes $\mathcal{S}^n(\mathcal{P})$ retain the property of $\mathcal{P}$ in the lemma by induction on $n \in \mathbb{N}$. Then for $n, m \in \mathbb{N}$ we consider two processes $P \in \mathcal{S}^n(\mathcal{P})$ and $Q \in \mathcal{S}^{n+m}(\mathcal{P})$. In case $m = 0$, $P\ \mathcal{R}\ Q$ implies $P = Q$. In case $m > 0$, by construction we have the reductions $Q \rightarrow^m \sim Q'$ for some $Q' \in \mathcal{S}^n(\mathcal{P})$ distinct from P. If P could simulate this series of reductions, we would obtain the relation $P \rightarrow^* \mathcal{R}\ Q'$ hence the contradictory equality $P = Q'$. $\square$

Corollary 4.32 *The bisimilarity $\dot{\approx}_2$ separates an infinite number of processes.*

More explicitly, we build a family of unrelated processes, which will be useful in the following. We start from the set of processes

$$\mathcal{P}_0 \quad \stackrel{\text{def}}{=} \quad \{\ x\langle\rangle,\ y\langle\rangle,\ 0\ \}$$

Processes in $\mathcal{P}_0$ have no reduction, and they are separated by $\dot{\approx}_2$ because they differ on at least one of the predicates $\Downarrow_x$, $\Downarrow_y$. The set $\mathcal{P}_0$ meets the hypothesis of Lemma 4.31, as well as all the sets $\mathcal{P}_n \stackrel{\text{def}}{=} \mathcal{S}^n(\mathcal{P}_0)$ for $n \geq 0$. Besides, the cardinality of $\mathcal{P}_n$ grows exponentially. Hence, the disjoint union

$$\mathcal{P} \stackrel{\text{def}}{=} \bigcup_{n \geq 0} \mathcal{P}_n$$

contains infinitely many processes that are not related by $\dot{\approx}_2$. Moreover, every process in $\mathcal{P}$ emits at most one message on a free name. The situation is partially described in the following reduction diagram (most sums are omitted).

$$\begin{tikzcd}
\vdots & \vdots & \vdots \\
\oplus \arrow[d] \arrow[dr] & \oplus \arrow[dl] \arrow[d] \arrow[dr] & \oplus \arrow[dl] \arrow[d] \\
x\langle\rangle \oplus \mathtt{0} \arrow[d] \arrow[dr] & x\langle\rangle \oplus \mathtt{0} \oplus y\langle\rangle \arrow[dl] \arrow[d] \arrow[dr] & \mathtt{0} \oplus y\langle\rangle \arrow[dl] \arrow[d] \\
x\langle\rangle & \mathtt{0} & y\langle\rangle
\end{tikzcd}$$

Of course, this construction captures only processes with finite behaviors (up to our bisimilarity). The bisimilarity $\dot{\approx}_2$ has much more classes than those exhibited here, such as for instance processes that can reach an infinite number of classes, as illustrated below.

Note that the same construction applies for the single-barbed bisimilarity $\dot{\approx}_1$, starting from the set $\{ x\langle\rangle, \ \mathtt{0} \}$, but this construction produces only a third unrelated process $x\langle\rangle \oplus \mathtt{0}$ at rank 1, then stops.

4.8.2 All integers on two exclusive barbs

The next lemma says that, in some sense, a join-calculus process can communicate any integer to its environment by using only the two barbs of $\dot{\approx}_2$ thanks to the discriminating power of bisimulation. To every integer, we associate a particular equivalence class of $\dot{\approx}_2$ in the hierarchy of processes $\mathcal{P}$ described above, then we write a process that receives an integer and expresses it by evolving to its characteristic class.

Lemma 4.33 *There is an evaluation context $N[\,\cdot\,]$ such that, for all integers n and m, the three statements that follow are equivalent:*

1. $n = m$
2. $N[\mathit{int}\langle\mathtt{n}\rangle] \dot{\approx}_2 N[\mathit{int}\langle\mathtt{m}\rangle]$
3. $N[\mathit{int}\langle\mathtt{n}\rangle] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle\mathtt{m}\rangle]$

Moreover, for every process P that has at most int as free variable we have

$$N[P] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle] \quad \mathit{iff} \quad P \rightarrow^* P' \mid \mathit{int}\langle\mathtt{n}\rangle$$

Intuitively, the context $N[\,\cdot\,]$ of the next lemma transforms exclusive integer-indexed barbs $\mathit{int}\langle\mathtt{n}\rangle$ into combinations of the two exclusive barbs $\Downarrow_x$ and $\Downarrow_y$. The first part of the lemma is required to discriminate processes that may emit several different integers. The second part makes the relation between integers and barbs more explicit: it states that integer-encoded barbs $\mathit{int}\langle\mathtt{n}\rangle$ are individually tested by the context $N[\,\cdot\,]$ as if they were regular barbs.

Proof: We define the evaluation context $N[\cdot]$ as follows, and we position the derivatives of $N[\mathit{int}\langle\mathtt{n}\rangle]$ in the hierarchy $\mathcal{P}$ of the previous section.

$$N[\,\cdot\,] \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{def}\ \mathit{int}\langle\mathtt{n}\rangle \mid \mathit{once}\langle\rangle \triangleright \\ \left(\begin{array}{l} \mathsf{def}\ \mathit{iter}\langle\mathtt{n},x,y,z\rangle \triangleright \\ \quad \mathsf{if}\ \mathsf{is\_zero}(\mathtt{n})\ \mathsf{then}\ z\langle\rangle \\ \quad \mathsf{else}\ \mathit{iter}\langle\mathrm{pred}(\mathtt{n}),z,x,y\rangle \oplus \mathit{iter}\langle\mathrm{pred}(\mathtt{n}),y,z,x\rangle\ \mathsf{in} \\ \mathsf{def}\ z\langle\rangle \triangleright \mathsf{0}\ \mathsf{in} \\ \mathit{iter}\langle\mathtt{n},x,y,z\rangle \oplus \mathit{iter}\langle\mathtt{n},y,z,x\rangle \oplus \mathit{iter}\langle\mathtt{n},z,x,y\rangle \end{array}\right) \\ \mathsf{in}\ [\mathit{once}\langle\rangle \mid \cdot\,] \end{array}$$

In the definition above, the name z is used to encode the process $\mathsf{0}$; hence the three processes in $\mathcal{P}_0$ are made symmetric, and the context $N[\,\cdot\,]$ can use permutations of the names x, y, and z to represent them.

Each integer n is associated with a ternary sum of nested binary sums in the $n+1$ layer of $\mathcal{P}$: when an encoded integer is received as $\mathit{int}\langle\mathtt{n}\rangle$, a reduction triggers the definition of int and yields the initial ternary sum; at the same time this reduction consumes the single message $\mathit{once}\langle\rangle$, hence the definition of int becomes inert. Let $P_n \in \mathcal{P}$ be the process such that $N[\mathit{int}\langle\mathtt{n}\rangle] \dot{\approx}_2 P_n$.

To prove the first part of the lemma, we check that *3.* always implies *1.* In case $N[\mathit{int}\langle\mathtt{n}\rangle] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle\mathtt{m}\rangle]$, the reductions of $N[\mathit{int}\langle\mathtt{n}\rangle]$ are simulated by P_n, and thus we have $P_n \rightarrow^* \dot{\approx}_2 P_m$.

For every non-empty series of reduction steps $P_n \rightarrow\rightarrow^* P'$, however, the resulting process P' is (strongly equivalent to) a binary sum in $\mathcal{P}$, and by Lemma 4.31 it cannot be bisimilar to any ternary sum in $\mathcal{P}$ such as P_m. Therefore, the relation $N[\mathit{int}\langle\mathtt{n}\rangle] \dot{\approx}_2 P_n$ above actually is $P_n \dot{\approx}_2 P_m$. By Lemma 4.31 we obtain that $P_n = P_m$, and finally that $n = m$ because the layers $\mathcal{P}_{(n+1)}$ each contain a single P_n and are pairwise disjoint by construction.

To establish the second part of the lemma, let P be a process with at most int as free variable. There is no interaction between P and $N[\,\cdot\,]$ except perhaps for some single message $\mathit{int}\langle\mathtt{m}\rangle$. Besides, there are no reduction in $N[\,\cdot\,]$ before this message is received.

Whenever we have the series of reductions $P \rightarrow^* P' \mid \mathit{int}\langle\mathtt{n}\rangle$, there is also a reduction $N[P' \mid \mathit{int}\langle\mathtt{n}\rangle] \rightarrow T$ that jointly consumes the messages $\mathit{int}\langle\mathtt{n}\rangle$ and $\mathit{once}\langle\rangle$, and after structural rearrangement $T \equiv P_n \mid N'[P']$ where $N'[\,\cdot\,]$ is $N[\,\cdot\,]$ without the message $\mathit{once}\langle\rangle$. In particular $N'[P']$ is inert, we have $N'[P'] \approx \mathsf{0}$ thus $P_n \mid N'[P'] \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle]$, and we obtain the relation $N[P] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle]$.

Conversely, whenever we have $N[P] \rightarrow^* T \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle]$,

- either a message $\mathit{int}\langle\mathtt{m}\rangle$ is received by $N[\,\cdot\,]$ in one of the reductions leading to T, which can be rewritten as

  $$N[P] \rightarrow^* N[P' \mid \mathit{int}\langle\mathtt{m}\rangle] \rightarrow \dot{\approx}_2 N[\mathit{int}\langle\mathtt{m}\rangle] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle]$$

  where $P \rightarrow^* P' \mid \mathit{int}\langle\mathtt{m}\rangle$. Using the first part of the lemma, this ensures that $m = n$ and $P \rightarrow^* P' \mid \mathit{int}\langle\mathtt{n}\rangle$.

- or P and $N[\,\cdot\,]$ did not interact, and we have

  $$N[P] \rightarrow^* N[P'] \dot{\approx}_2 N[\mathit{int}\langle\mathtt{n}\rangle]$$

where $P \rightarrow^* P'$ and P' still has a weak barb on *int*—otherwise we would have $P' \approx \mathsf{0}$, then $N[P'] \approx N[\mathsf{0}] \approx \mathsf{0}$ and we would obtain $\mathsf{0} \dot{\approx}_2 N[\mathit{int}\langle \mathtt{n}\rangle]$; this is excluded because for every $n \in \mathbb{N}$ the process $N[\mathit{int}\langle \mathtt{n}\rangle]$ has the two barbs $\Downarrow_x$ and $\Downarrow_y$.

Therefore we can complete the reductions $P \rightarrow^* P'$ with reductions $P' \rightarrow^* P'' \mid \mathit{int}\langle \mathtt{m}\rangle$. By simulation $N[P'' \mid \mathit{int}\langle \mathtt{n}\rangle] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle \mathtt{m}\rangle]$, and this reverts to the first case for $N[P] \rightarrow^* N[P'' \mid \mathit{int}\langle \mathtt{m}\rangle]$. □

The next lemma applies this remark to restrict the class of contexts in use in congruence properties to contexts with at most two free (nullary) variables. In all the following, we assume given an injective mapping from names to integers, and we use the notation $[\![z]\!]$ to represent the integer associated with $z \in \mathcal{N}$.

Corollary 4.34 *For every finite set of names $S \subset \mathcal{N}$, there is an evaluation context $F_S[\,\cdot\,] \in \mathcal{E}$ such that for all processes P and Q with $\mathsf{fv}[P] \cup \mathsf{fv}[Q] \subseteq S$ we have*

1. *if $N[F_S[P]] \dot{\approx}_2 N[F_S[Q]]$, then $P \dot{\approx} Q$;*
2. *if $N[F_S[P]] \rightarrow^* T$ and $T \rightarrow^* \dot{\approx}_2 N[F_S[Q]]$, then $P \rightarrow^* P'$ and $T = N[F_S[P']]$.*
3. *if $N[F_S[P]] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle \mathtt{n}\rangle]$, then $n \in \{0,1\} \cup [\![S]\!]$.*

Proof: Without loss of generality, we assume that the names *once* and *int* do not appear in S. We set

$$F_S[\,\cdot\,] \stackrel{\text{def}}{=} \begin{array}{lll} & & \mathit{once}\langle\rangle \rhd \mathit{int}\langle \mathtt{0}\rangle \\ \mathsf{def} & \wedge & \mathit{once}\langle\rangle \rhd \mathit{int}\langle \mathtt{1}\rangle \\ & \wedge & \bigwedge_{x \in S} x\langle\rangle \mid \mathit{once}\langle\rangle \rhd \mathit{int}\langle [\![x]\!]\rangle \end{array} \quad \mathsf{in}\ \mathit{once}\langle\rangle \mid [\,\cdot\,]$$

By construction, the process $F_S[P]$ has only *int* as free variable, and we have for any process P and name $z \in S$ the relations

$$\begin{array}{rcl} F_S[P] & \rightarrow\approx & \mathit{int}\langle \mathtt{0}\rangle \\ F_S[P] & \rightarrow\approx & \mathit{int}\langle \mathtt{1}\rangle \\ F_S[P \mid z\langle\rangle] & \rightarrow\approx & \mathit{int}\langle [\![z]\!]\rangle \end{array}$$

Moreover, the first reduction that involves the context $F_S[\,\cdot\,]$ commits the process $F_S[P]$ to a particular integer barb. By applying Lemma 4.33, we use the last above relation to reformulate the barb $\Downarrow_z$

$$\begin{array}{rll} P \Downarrow_z & \text{iff} & F_S[P] \rightarrow^* \dot{\approx}\ \mathit{int}\langle [\![z]\!]\rangle \\ & \text{iff} & N[F_S[P]] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle [\![z]\!]\rangle] \end{array}$$

We prove that the relation $\mathcal{R}$ containing all the pairs P, Q of the lemma is a barbed bisimulation. Let us assume that $P \mathcal{R} Q$, thus in particular that $N[F_S[P]] \dot{\approx}_2 N[F_S[Q]]$.

1. P and Q have the same barbs. In case $P \Downarrow_z$, we have $N[F_S[P]] \rightarrow^* \dot{\approx}_2 N[\mathit{int}\langle [\![z]\!]\rangle]$, these reductions can be simulated by $N[F_S[Q]] \rightarrow^* U$ with $U \dot{\approx}_2 N[\mathit{int}\langle [\![z]\!]\rangle]$, and thus $Q \Downarrow_z$.

2. the weak bisimulation property relies on the presence of the two special integer-indexed barbs $int\langle\mathsf{0}\rangle$ and $int\langle\mathsf{1}\rangle$. In case $P \rightarrow^* P'$, we have $N[F_S[P]] \rightarrow^* N[F_S[P']]$ and the bisimulation hypothesis yields some reductions $N[F_S[Q]] \rightarrow^* T$ with $N[F_S[P']] \dot{\approx}_2 T$. By Lemma 4.33, the two processes $N[F_S[P']$ and T have the same integer-indexed barbs, and in particular T retains the possibility of emitting *any* of the messages $int\langle\mathsf{0}\rangle$ and $int\langle\mathsf{1}\rangle$. By construction of $F_S[\cdot]$, this ensures that the reductions $N[F_S[Q]] \rightarrow^* T$ entirely occur within Q: there is some process Q' with $Q \rightarrow^* Q'$, $T = N[F_S[Q']]$, and finally $P' \mathcal{R} Q'$. □

4.8.3 A join-calculus interpreter for the join-calculus

We define an interpreter ε in the join-calculus that takes as parameters an integer representation $[\![P]\!] \in \mathbb{N}$ and an evaluation environment ρ that binds all the free names $\mathsf{fv}[P]$. Up to barbed congruence, we obtain that the resulting process $\varepsilon\langle[\![P]\!], \rho\rangle$ behaves as process P.

We compile every join process P into some integer representation $[\![P]\!]$, to be evaluated in some environment $\rho = \{[\![x]\!] \mapsto x\}$. The internal representations for processes $[\![P]\!]$, and environment $\rho = \{[\![x]\!] \mapsto x\}$ are omitted; we only specify their interface in a functional syntax (*cf.* Section 3.4.3). Both representations rely on the injective representation of names as integers $[\![x]\!]$ defined above. We assume that $[\![P]\!]$ is an integer encoding of the abstract syntax trees of the join-calculus processes that uses $[\![x]\!]$ to encode names and that is equipped with pattern-matching on the syntax. We also assume that ρ is an association table from integers to names equipped with synchronous names for lookup $\rho(\cdot)$ and for overwriting $+$ (written $\rho + [\![x]\!] \mapsto x$). Without loss of generality, we require that the images of our encodings for processes and names are disjoint ($\forall P \in \mathcal{P}, x \in \mathcal{N}. [\![P]\!] \neq [\![x]\!]$) and that they do not contain the values 0 and 1.

As opposed to most proofs in this chapter, the actual definition of the interpreter is very sensitive to small variations in the calculus, including its type system. We first give an interpreter for join-calculus processes that comply with several limitations, then we rely on preliminary compilation to extend our interpreter to the join-calculus.

For a finite maximal arity $n \in \mathbb{N}$, and for a finite set of definitions $\mathcal{D}_\varepsilon$ that only use names with these arities, we define our join-calculus interpreter D_ε as the conjunction of the following clauses.

$$\begin{array}{lrcl}
\text{for each } m \leq n: & \varepsilon\langle[\![x\langle y_1,\ldots,y_m\rangle]\!], \rho\rangle & \triangleright & \rho([\![x]\!])\langle\rho([\![y_1]\!]),\ldots,\rho([\![y_n]\!])\rangle \\
\text{for each } D \in \mathcal{D}_\varepsilon: & \varepsilon\langle[\![\mathsf{def}\ D\ \mathsf{in}\ Q]\!], \rho\rangle & \triangleright & \mathsf{def}\ [\![D]\!]\ \mathsf{in} \\
 & & & \varepsilon\langle[\![Q]\!], \rho + \uplus_{x\in\mathsf{dv}[D]}([\![x]\!] \mapsto x)\rangle \\
 & \varepsilon\langle[\![P \mid Q]\!], \rho\rangle & \triangleright & \varepsilon\langle[\![P]\!], \rho\rangle \mid \varepsilon\langle[\![P]\!], \rho\rangle \\
 & \varepsilon\langle[\![0]\!], \rho\rangle & \triangleright & 0
\end{array}$$

where the notation $[\![D]\!]$ is recursively replaced according to the structure of D:

$$\begin{array}{rcl}
[\![D_1 \wedge D_2]\!] & \stackrel{\text{def}}{=} & [\![D_1]\!] \wedge [\![D_2]\!] \\
[\![J \triangleright P]\!] & \stackrel{\text{def}}{=} & J \triangleright \varepsilon\langle[\![P]\!], \rho + \uplus_{x\in\mathsf{dv}[D]}([\![x]\!] \mapsto x) + \uplus_{v\in\mathsf{rv}[J]}([\![v]\!] \mapsto v)\rangle \\
[\![\mathsf{T}]\!] & \stackrel{\text{def}}{=} & \mathsf{T}
\end{array}$$

Without loss of generality, we rely on the preliminary use of α-conversion and structural equivalence to replace every definition D in the source process by a definition in $\mathcal{D}_\varepsilon$.

The next lemma relates the source process P to the interpretation of its representation $[\![P]\!]$. As long as the interpreter can be finitely defined, the result is not surprising, since the join-calculus has well enough expressive power; in particular a similar interpreters should be definable for most variants of the join-calculus or the π-calculus.

Lemma 4.35 (Basic interpreter) *Let $\mathcal{D}_\varepsilon$ be a finite set of definitions, and Σ be a finite set of simple recursive types closed by decomposition that suffices to type all definitions in $\mathcal{D}_\varepsilon$. There is a definition D_ϵ such that For every process P whose definitions are all in $\mathcal{D}$, that can be monomorphically typed using only types in Σ, and such that $\mathsf{fv}[P] \cap \{\rho, \varepsilon\} = \emptyset$, for every environment ρ such that $\forall x \in \mathsf{fv}[P], \rho([\![x]\!]) = x$, we have the equivalence*

$$\mathsf{def}\ D_\varepsilon\ \mathsf{in}\ \varepsilon\langle[\![P]\!], \rho\rangle \quad \approx_l \quad P$$

Proof: The translation preserves the structure of the term (same parallel compositions, same binders); the reductions are the same, except for bookkeeping, which is entirely deterministic and can be handled by expansion.

In order to remain closed for every transition, we extend the interpreter to open join processes (by adding optional indices on both the representation of definitions and on their interpreter) then we prove that the relation of the lemma is an asynchronous bisimulation. □

As we try to interpret a richer calculus, we have access to richer constructs in the target language; for instance, name-comparison can trivially be added to the interpreter provided that there is a name-comparison construct (*cf.* Section 5.5), and that all comparisons are performed on names of the same type. The interpreter is extended with the clause

$$\varepsilon\langle[\![\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!], \rho\rangle \quad \triangleright \quad \mathsf{if}\ \rho(x) = \rho(y)\ \mathsf{then}\ \varepsilon\langle[\![P]\!], \rho\rangle\ \mathsf{else}\ \varepsilon\langle[\![Q]\!], \rho\rangle$$

Nonetheless, the interpreter itself must remain finite, which induces limitations at least for our basic approach. For instance, the shapes of join-definitions are infinite, hence some additional encoding is required to circumvent the need for infinitely-many definitions in the interpreter. Anticipating on some results that are established in Chapter 6, we have the compilation scheme:

1. For all monomorphic processes, complex defining processes are compiled into processes that use only single-clause, two-name-join definitions, up to labeled bisimulation $\approx_l$ (Lemma 6.6).

2. For all channel types that do not appear as subtypes of the free names of P, we can use a type-driven translation that recursively substitutes communication on channels of some uniform types for communication on channels of these types (as in Lemma 6.15).

More precisely, we have

Remark 4.36 *Let Σ be a finite set of simple recursive types closed by decomposition. Then there is a finite set of simple recursive types $F(\Sigma)$ such that, for every process P whose free names have types in Σ, there is a process P_Σ such that $P \approx_l P_\Sigma$ and all names in P_Σ have types in $F(\Sigma)$.*

Proof: We say that a type is internal to P when it is used internally in P but does not appear in Σ. We assume given a recursive representation of lists of arguments, where an argument is a tuple of synchronous names that each return a value of a given type, if the argument has this type, or blocks otherwise. The types of arguments range over Σ plus the type of argument lists.

We perform a type-directed transformation of every type, emission and reception in the calculus, where each name whose type does not appear in Σ is replaced with a name that conveys a generic list of arguments.

Every communication on names that have been affected by the rewriting is internal to the process; the extraneous encoding and decoding of the argument list is deterministic, and can be discarded up to labeled expansion. $\square$

By re-defining the run-time representation $[\![P]\!]$ as the representation of the compiled simpler representative of P, we obtain a universal interpreter:

Corollary 4.37 *Let Σ be a finite set of simple recursive types closed by decomposition. There is a definition D_ε such that, for every process P whose interface $\mathsf{fv}[P]$ can be typed using only types in Σ and such that $\mathsf{fv}[P] \cap \{\rho, \varepsilon\} = \emptyset$, for every environment ρ such that $\forall x \in \mathsf{fv}[P], \rho([\![x]\!]) = x$, we have the equivalence*

$$\mathsf{def}\ D_\varepsilon\ \mathsf{in}\ \varepsilon\langle [\![P]\!], \rho\rangle \quad \approx_l \quad P$$

While our compiler-interpreter may be extended to polymorphic types in the absence of name comparison, the correspondence surprisingly breaks in the presence of name comparison. In particular, we have

Remark 4.38 *In the polymorphic join-calculus, labeled bisimulation is strictly finer than the congruence of barbed bisimilarity ($\approx_l \subset \dot{\approx}^\circ$).*

We give a counter-example in the presence of name-testing, in the spirit of Brook's counter-example between limit bisimulations and bisimilarity. For any $n \in \mathbb{N}$, we let P_n be the process that performs a series of tests on a polymorphic functional name $f : \forall\alpha.\langle \mathtt{Int}, \alpha\rangle \rightarrow \langle\langle\alpha\rangle \rightarrow \langle\alpha\rangle\rangle$. The tests check that, for all $i, j < n$, we have $f\langle i, 0\rangle = f\langle j, 0\rangle$ if and only if $i = j$. For instance, we can use the process

$$\begin{array}{rclll} & & \mathsf{def} & x\langle\rangle \mid y\langle\rangle \triangleright 0 \wedge x\langle\rangle \triangleright t\langle\rangle\ \mathsf{in} & \\ P_n & \stackrel{\text{def}}{=} & & \prod_{i,j<n, i\neq j} \mathsf{if}\ f\langle i,0\rangle = f\langle j,0\rangle\ \mathsf{then}\ t\langle\rangle & \\ & & \mid & \prod_{i<n} \quad (x\langle\rangle \mid \mathsf{if}\ f\langle i,0\rangle = f\langle i,0\rangle\ \mathsf{then}\ y\langle\rangle) & \end{array}$$

where the indexed parallel compositions can easily be replaced with internal encodings of loops that takes the representation of n as argument. Moreover, the encoding can be extended to support infinite loops ($n = \omega$). For any $n \in \mathbb{N}$, the process P_n can loose the ability to emit on $t\langle\rangle$ ($P_n \not\Downarrow_{\Box t}$) if and only if f meets its specification at rank n. Conversely, $P_\omega \Downarrow_{\Box t}$.

It is straightforward to write a function f_n that meets the specification at all ranks $m \leq n$ and to embed it in a context $C[\cdot]$. However, in the case $C[P_n]$ passes the test, the context $C[\cdot]$, must have n different values to return. Since every value must be returned twice, these values must be available in f. However, these values cannot be received in a join pattern that defines f, because that would rule out polymorphism according to our generalization criterion. Hence, all values must already be present in $C[\cdot]$, and the size of $C[\cdot]$ grows with n.

We now compare the processes $S_1 = \bigoplus_{n<\omega} P_n$ and $S_2 = \bigoplus_{n\leq\omega} P_n$.

$S_1 \not\approx_l S_2$**:** the reduction $S_2 \rightarrow P_\omega$ cannot be matched by S_1: in the case $S_1 \rightarrow P_n$, P_n has a finite behavior; we can perform transitions that make P_n loose its barb on t, and these transitions cannot be simulated by P_ω. Yet, $S_1 \not\approx_l P_\omega$ either, because no reduction $S_1 \rightarrow^* P_n$ can be matched by P_ω for the same reason.

$S_1 \dot{\approx}^\circ S_2$**:** any given context $C[\cdot]$ can only perform tests at a given depth, hence there is $n \in \mathbb{N}$ such that for any $m \geq n$ we have $C[P_m] \dot{\approx} C[P_\omega]$.

4.8.4 Reducing contexts to integers

A peculiarity of the join-calculus is that processes in parallel may communicate only if they share some common enclosing definitions. In the least, contexts need to define the free variables of the processes to be plugged in. However, this can be done in a uniform manner, using the same common definitions around arbitrary processes instead of arbitrary contexts. The uniform definition simply waits for the "environment" process to send on d_x the name x where messages sent on x by the "internal" process should be forwarded. Let S be a finite set of names; we define the uniform context R_S as follows:

$$R_S \quad \stackrel{\text{def}}{=} \quad \mathsf{def} \bigwedge_{x\in S} x\langle\widetilde{u}\rangle \mid d_x\langle x'\rangle \triangleright x'\langle\widetilde{u}\rangle \mid d_x\langle x'\rangle \;\mathsf{in}\; [\cdot]$$

(where we assume that the names d_x for all $x \in S$ are fresh.) Note that the definition of R_S assumes given the arity of every name in S; moreover, R_S typechecks if and only if all names in S are given a monomorphic type.

The next lemma uses this particular family of contexts for substituting processes for evaluation contexts in weak bisimulation statements.

Lemma 4.39 *For every finite set of names $S \subset \mathcal{N}$ and evaluation context $C[\cdot] \in \mathcal{E}$, there is a process T such that for all processes P we have*

$$\mathsf{fv}[P] \subseteq S \quad \textit{implies} \quad R_S[P \mid T] \approx_l C[P]$$

Proof: As a first attempt, we could use $T = C\left[\prod_{x\in S} d_x\langle x\rangle\right]$; unfortunately, the property of the lemma would not carry over labeled bisimulation, because the process and the context would use two different names instead of one, which would affect the labels where these names appear. The same problem would occur in barbed congruence in case we extend the calculus with a name-comparison operator.

We refine our definition of T accordingly: we set $T = C'\left[\prod_{x\in S} d_x\langle x'\rangle\right]$ where $C'[\cdot]$ is obtained from $C[\cdot]$ by substituting every defined variable $x \in S$ that is a binder for $[\cdot]$ *only in the join-patterns* of $C[\cdot]$. Hence, a free x is used everywhere in both P and $C'[\cdot]$, while x' is only received once then used by the enclosing context to forward any message sent on x.

We prove that the relation of the lemma is contained in a weak labeled bisimulation up to expansion and restriction. Without loss of generality, we assume that every name in S is defined and extruded by $R_S[\cdot]$ and $C[\cdot]$, respectively. The candidate bisimulation contain all pairs of processes of the form $(R_S[C''[P] \mid T], C[P])$ where $C''[P]$ is obtained from $C'[P]$ by substituting messages $x'\langle\widetilde{v}\rangle$ for any message $x\langle\widetilde{v}\rangle$ in evaluation context for $x \in S$.

All transitions are in direct correspondence, except for the additional deterministic reductions on the left to forward new messages $x\langle\widetilde{v}\rangle$ that may appear as the result of intrusions and internal reductions. □

In particular, we can restrict ourselves in congruence proofs to contexts of this particular, process-based form: for all P and Q with $\mathsf{fv}[P] \cup \mathsf{fv}[Q] \subseteq S$ we have

$$\forall C \in \mathcal{E}, C[P] \dot{\approx} C[Q] \quad \text{iff} \quad \forall T \in \mathcal{P}, R_S[P \mid T] \dot{\approx} R_S[Q \mid T]$$

In combination with our compiler, the testing process T can in turn be replaced with an interpreter running its representation $[\![T]\!]$:

Lemma 4.40 (Integers instead of Contexts) *For every finite set of names $S \subset \mathcal{N}$ and evaluation context $C[\cdot] \in \mathcal{E}$, let ρ be the environment with bindings $[\![x]\!] \mapsto x$ for all $x \in \mathsf{fv}[C[\cdot]]$ and bindings $[\![d_z]\!] \mapsto d_z$ for all $z \in S$. There is an integer n such that for all processes P we have*

$$\mathsf{fv}[P] \subseteq S \quad \textit{implies} \quad C[P] \approx R_S\left[P \,|\, \mathsf{def}\ D_\varepsilon\ \mathsf{in}\ \varepsilon\langle n, \rho\rangle\right]$$

Proof: this is the composition of Lemmas 4.39 and 4.37 □

4.8.5 Universal contexts

We are now ready to establish the problematic inclusion $\dot{\approx}_2^\circ \subseteq \approx$. The next lemma is the key of our proof of Theorem 3; it relies on a single context that concentrates the discriminative power of quantification over all contexts. We call this context a universal context.

Lemma 4.41 (Universal Context) *For every finite set of names $S \subset \mathcal{N}$, there is an evaluation context $U_S[\cdot]$ such that the relation $\mathcal{R}$ defined as*

$$\mathcal{R} \quad \stackrel{def}{=} \quad \left\{(P, Q) \mid \mathsf{fv}[P] \cup \mathsf{fv}[Q] \subseteq S \text{ and } U_S[P] \dot{\approx}_2 U_S[Q]\right\}$$

has the following properties:

1. *Let $C[\cdot]$ be an evaluation context such that for any process R with $\mathsf{fv}[R] \subseteq S$ we have $\mathsf{fv}[C[R]] \subseteq \{x, y\}$.*

   For all processes P and Q, if $P \mathrel{\mathcal{R}} Q$, then $C[P] \dot{\approx}_2 C[Q]$.

2. *The union $\bigcup_S \mathcal{R}$ is a barbed congruence ($\mathcal{R} \subseteq \approx$).*

Proof: For a given set $S \subset \mathcal{N}$, we build our universal context as follows:

$$Pick(n)[\,\cdot\,] \stackrel{\text{def}}{=} \begin{array}{ll} \mathsf{def} & pick\langle \mathtt{n}\rangle \triangleright pick\langle \mathtt{n}+\mathtt{1}\rangle \\ \wedge & pick\langle \mathtt{n}\rangle \triangleright [\,\cdot\,] \\ \mathsf{in} & pick\langle \mathtt{n}\rangle \end{array}$$

$$\begin{array}{rcl} T_{\mathtt{n}} & \stackrel{\text{def}}{=} & \mathsf{def}\ D_\varepsilon\ \mathsf{in}\ \varepsilon\langle \mathtt{n}, \{\uplus_{z\in S}(\llbracket d_z\rrbracket \mapsto d_z) + \llbracket x\rrbracket \mapsto x + \llbracket y\rrbracket \mapsto y\}\rangle \\ I_{\mathtt{n}} & \stackrel{\text{def}}{=} & int\langle \mathtt{n}\rangle \oplus T_{\mathtt{n}} \\ B[\,\cdot\,] & \stackrel{\text{def}}{=} & N\,[F_{x,y}\,[\,\cdot\,]] \\ U_S[\,\cdot\,] & \stackrel{\text{def}}{=} & B[R_S\,[\,\cdot \mid Pick(4)[I_{\mathtt{n}}]]] \end{array}$$

The context $Pick(n)[\,\cdot\,]$ is used to choose a particular integer as the result of some infinite internal choice; whenever the first clause is used and consumes $pick\langle \mathtt{n}\rangle$, it gradually commits this internal choice to any $m > n$; when the second clause is used, the guarded process is started with the current integer n, while the remains of $Pick(n)[\,\cdot\,]$ become inert.

The process $T_{\mathtt{n}}$ is the interpretation of some integer-encoded process; it is such that for every evaluation context $C[\,\cdot\,]$ and process P that meet the hypotheses of *(1.)* we have the bisimulation $R_S[P \mid T_{\mathtt{n}}] \mathrel{\dot{\approx}_2} C[P]$ for some $n \in \mathbb{N}$. The process $I_{\mathtt{n}}$ either reveals the choice of n (hence of the context $C[\,\cdot\,]$) by exhibiting the integer barb $int\langle \mathtt{n}\rangle$, or silently reduces to $T_{\mathtt{n}}$.

The universal context $U_S[\,\cdot\,]$ assembles these components to pick the integer representative of any context $C[\,\cdot\,]$, then either reveal this choice or behave as $C[\,\cdot\,]$. We let the contexts $K[\,\cdot\,]$ and $K'[\,\cdot\,]$ represent the derivatives of $U_S[\,\cdot\,]$ at these intermediate stages:

$$\begin{array}{rcl} K[\,\cdot\,] & \sim & B[R_S[\,\cdot \mid I_n]] \\ K'[\,\cdot\,] & \sim & B[R_S[\,\cdot \mid T_n]] \end{array}$$

Let us assume that P, Q, and $C[\,\cdot\,]$ meet the hypotheses of *(1.)*, and let n be the integer associated with $C[\,\cdot\,]$ by Lemma 4.40. For every reduction $C[P] \rightarrow^* T$, we prove that $C[Q] \rightarrow^* R$ with $T \mathrel{\dot{\approx}_2} R$. We build a series of reductions that embeds $C[P] \rightarrow^* T$ within $U_S[\,\cdot\,]$, we use the $\dot{\approx}_2$ bisimulation in the definition of $\mathcal{R}$, then we extract a series of reductions leading from $C[Q]$ to R. The situation is detailed in the diagram on the next page.

The upper square of the diagram deals with the internal choice of the particular integer n that represents $C[\,\cdot\,]$. The top edge holds by definition of $\mathcal{R}$. On the left, we choose reductions $U_S[P] \rightarrow^* K[P]$ that only consist of

$$Pick(4)[I_{\mathtt{n}}] \rightarrow Pick(5)[I_{\mathtt{n}}] \rightarrow \cdots \rightarrow Pick(n)[I_{\mathtt{n}}] \rightarrow\sim I_{\mathtt{n}}$$

in the enclosing context $N[F_{x,y}[R_S[\,\cdot\,]]]$. By weak bisimulation, we obtain a series of reductions $U_S[Q] \rightarrow^* U'$ with $K[P] \mathrel{\dot{\approx}_2} U'$. By lemma 4.33, the processes $K[P]$ and U' have the same integer barbs. In particular, U' has the integer barb $int\langle \mathtt{n}\rangle$ and no integer barb $int\langle \mathtt{n}+\mathtt{1}\rangle$. By Lemma 4.34(3), the integer barbs of $N[F_{x,y}[\,\cdot\,]]$ are disjoint from these barbs, hence the same reductions displayed above must also have taken place from $U_S[Q]$ to U'. Moreover, no reduction from $U_S[Q]$ to U' may have involved $T_{\mathtt{n}}$ (which is still guarded), $R_S[\,\cdot\,]$ (which is blocked until $T_{\mathtt{n}}$ is triggered), or $N[F_S[\,\cdot\,]$ (otherwise the reduction $K[P] \to K'[P]$ would be simulated by some $U' \to^* \dot{\approx}_2 N[F_{x,y}[R]]$, which contradicts Lemma 4.34(2)). As a result, the reductions $U_S[Q] \to^* U'$ only consist of reductions $U_S[Q] \to^* K[Q]$ interleaved with reductions $Q \to^* Q'$, which is easily established by induction on the total number of reductions.

$$\begin{tikzcd}
& U_S[P] \arrow[r, dash, "\dot{\approx}_2"] \arrow[d, "*"] & U_S[Q] \arrow[d, dotted, "*"] & & C[Q] \arrow[d, dotted, "*"] \\
& K[P] \arrow[r, dotted, dash, "\dot{\approx}_2"] \arrow[d] & K[Q'] \arrow[d, dotted, "*"] & & C[Q'] \arrow[d, dotted, "*"] \\
B[C[P]] \arrow[r, dash, "\dot{\approx}_2"] \arrow[d] & K'[P] \arrow[d, dotted, "*"] & K'[Q''] \arrow[r, dotted, dash, "\dot{\approx}_2"] \arrow[d, dotted, "*"] & B[C[Q'']] \arrow[d, dotted, "*"] & C[Q''] \arrow[d, dotted, "*"] \\
B[T] \arrow[r, dotted, dash, "\dot{\approx}_2"] & B[T'] \arrow[r, dotted, dash, "\dot{\approx}_2"] & B[R'] \arrow[r, dotted, dash, "\dot{\approx}_2"] & B[R] & R
\end{tikzcd}$$

Next, the reduction $K[P] \to K'[P]$ discards the integer barb used as a marker for this particular $T_{\mathtt{n}}$; by Lemma 4.37, we have $B[C[P]] \dot{\approx}_2 K'[P]$, and by weak simulation in the bottom left square the reductions $C[P] \to^* T$ in context yields a series of reductions $K'[P] \to^* V$ with $B[T] \dot{\approx}_2 V$. Moreover, by Lemma 4.34(2) we obtain reductions $R_S[P \,|\, T_n] \to^* T'$ such that $V = B[T']$.

In the central square of the diagram, the reductions $K[P] \to^* B[T']$ are simulated by some reductions $K[Q'] \to^* W$, and one of these reductions must discard the marker $int\langle\mathtt{n}\rangle$ by reducing the internal choice $I_{\mathtt{n}}$ to $T_{\mathtt{n}}$. Let U'' be the process obtained immediately after performing this reduction. U'' is of the form $K'[Q'']$ for some reductions $Q' \to^* Q''$ for the same reasons than for U', and thus the reductions starting from $K[Q']$ can be decomposed as successive reductions $K[Q'] \to^* K'[Q''] \to^* W$ with $Q' \to^* Q''$. Moreover, by Lemma 4.342 again, there are reductions $R_S[Q'' \,|\, T_n] \to^* R'$ such that $W = B[R']$. Note, however, that there is no central $\dot{\approx}_2$ edge, hence no direct way at this stage to relate $C[P]$ and $C[Q'']$.

By applying Lemma 4.37 again, we obtain that $K'[Q''] \dot{\approx}_2 B[C[Q'']]$ at the top of the rightmost bisimulation diagram. By weak bisimulation in this diagram, we first transform the series of reductions $K'[Q''] \to^* B[R']$ into a series of reductions starting from $B[C[Q'']]$, then again we remark that none of these reductions may interact with $B[\,\cdot\,]$, hence that the same reductions applies from $C[Q'']$ to R.

We glue the source reductions on the right of the diagram into $C[Q] \to^* R$, and we apply Lemma 4.34(1) on the bottom line to obtain the simpler relation $T \dot{\approx}_2 R$. In the special case of an empty series of reductions ($C[P] = T$), we have in particular $C[Q] \to^* \dot{\approx}_2 C[P]$.

We conclude the proof of the first statement by showing that the relation that contains all pairs $(C[P], C[Q])$ is a double-barbed bisimulation up to bisimilarity (Lemma 4.27). We have just established a sufficient weak bisimulation diagram (if $C[P] \to^* T$, then $C[Q] red^* R$ with $T \dot{\approx}_2 R$ and vice-versa), and the preservation of barbs follows from the special case above ($C[P] \Downarrow_z$ and $C[Q] \rightarrow^* \dot{\approx}_2 C[P]$ implies $C[Q] \Downarrow_z$).

The proof of the second statement mostly relies on the first statement in combination with specific instances of $C[\,\cdot\,]$.

1. for the barbs, we choose the context $C[\,\cdot\,] = B[\,\cdot\,]$. Thus, $P \mathcal{R} Q$ yields in particular $B[P] \dot{\approx}_2 B[Q]$. By applying Lemma 4.34(1), we immediately obtain $P \dot{\approx} Q$, hence that P and Q have the same barbs.

2. for the congruence property, let us assume that $P \mathcal{R} Q$, and let $C'[\,\cdot\,]$ be an evaluation context. We use the particular context $C[\,\cdot\,] = U_{S'}[C'[\,\cdot\,]]$ where $S' = S \cup \mathsf{fv}[C'[\,\cdot\,]]$, and we obtain that $C'[P] \mathcal{R}_{S'} C'[Q]$.

3. as regards weak bisimulation, we close the diagram on the left by extracting a series of reductions $Q \rightarrow^* Q'$ from the diagram on the right.

$$\begin{tikzcd}
P \arrow[r, dash, "\mathcal{R}"] \arrow[d] & Q \arrow[d, dotted, "*"] \\
P' \arrow[r, dotted, dash, "\mathcal{R}"] & Q'
\end{tikzcd}
\quad \text{from the diagram} \quad
\begin{tikzcd}
U_S[P] \arrow[r, dash, "\dot{\approx}_2"] \arrow[d] & U_S[Q] \arrow[d, dotted, "*"] \\
U_S[P'] \arrow[r, dotted, dash, "\dot{\approx}_2"] & U_S[Q']
\end{tikzcd}$$

Let us assume that $P \rightarrow P'$; then $U_S[P] \rightarrow U_S[P']$, and by hypothesis this reduction can be simulated by a series of reductions $U_S[Q] \rightarrow^* T$. Specifically, $U_S[P']$ still has all its integer-encoded barbs on $n > 4$, and so does T by Lemma 4.33. This ensures that all reduction steps in $U_S[Q] \rightarrow^* T$ occurred within Q, that is, that there are reductions $Q \rightarrow^* Q'$ such that $T = U_S[Q']$ and thus $P' \mathcal{R} Q'$. $\square$

Corollary 4.42 $\dot{\approx}_2^\circ = \approx$.

Proof: Let $\mathcal{R}$ be the union of relations of the previous lemma. By construction of $\mathcal{R}$, we immediately obtain that $\dot{\approx}_2^\circ \subseteq \mathcal{R}$. By the previous lemma, we have $\mathcal{R} \subseteq \approx$. By definition, $\approx$ is a doubled-barbed bisimulation and a congruence, hence $\approx \subseteq \dot{\approx}_2^\circ$. By composing these inclusions, we have $\dot{\approx}_2^\circ = \approx$. $\square$

Proof of Theorem 3: We have the circular inclusions $\approx \subseteq \dot{\approx}_2^\circ \subseteq \dot{\approx}^\circ \subseteq \approx$. $\square$

Chapter 5

The Open Join-Calculus

In order to reason about processes in a more extensional manner, we introduce a refined operational model where interaction with the environment is explicitly represented as labeled transitions, instead of potential interactions in context.

Using this auxiliary model, we supplement our family of reduction-based equivalences with a theory of labeled equivalences in the join-calculus. Starting from the usual bisimulation of Milner [96], we propose several formulations of equivalences that enjoy purely co-inductive proof techniques on labeled transitions. All these formulations yield the same equivalence, but they correspond to different proof techniques. We prove that weak bisimulation is a full congruence, and we insert it at the upper tier of our hierarchy of equivalences. Labeled bisimulation is strictly coarser than barbed congruence. If we add a construct for name comparison, however, then we can show that both relations coincide for all processes.

In chapter 2 we introduced the RCHAM, an operational semantics where only the internal evolution of processes is taken into account. In order to shape labeled bisimulation upon the join-calculus, we propose a refined chemical machine— the open RCHAM—that explicitly models interaction with the environment. Via these interactions, the environment can get acquainted with names defined in solution when such names are exported as the contents of messages on free names, and can later use these names to inject messages within the solution. We call these two symmetrical interactions *extrusions* and *intrusions*, respectively. Due to the principle of locality, intrusions and extrusions always occur on disjoint sets of names. To keep track of the environment's acquaintance with defined names, definitions are marked with their extruded names when extrusion occurs for the first time, and intrusions are enabled only on marked names. Hence, the grammar for the resulting *open join-calculus* features processes of the form $\mathsf{def}_S\ D \mathsf{\ in\ } P$, where S is a set of names defined by D and extruded to the environment. Informally, extruded names are handled as constants in the input interface of the process.

Let us illustrate what locality means in an open setting. The process

$$\mathsf{def}_{\{x\}}\ x\langle u\rangle \triangleright P \mathsf{\ in\ } x\langle v\rangle$$

defines a name x such that, whenever a message $x\langle u\rangle$ is received, a fresh copy of process P is started. The name x is marked as extruded; the environment can therefore send messages on x to trigger copies of process P. However, the environment cannot interfere with the definition of x; in particular, the message $x\langle v\rangle$ is invisible from the environment, and every weak equivalence would identify the two processes

$$\mathsf{def}_{\{x\}}\ x\langle u\rangle \triangleright P \ \mathsf{in}\ x\langle v\rangle \quad \text{and} \quad \mathsf{def}_{\{x\}}\ x\langle u\rangle \triangleright P \ \mathsf{in}\ P\{^{v}/_{u}\}$$

Our notion of weak labeled bisimulation is obtained by applying to the open RCHAM the standard definition of Milner (*cf.* Definition 4.11). This equivalence relation is closed under renaming, and is a congruence for all contexts of the open calculus.

In the whole chapter, *all bisimulations are implicitly weak, labeled bisimulations,* unless mentioned otherwise. In equations, we put an index to indicate the labeled nature of a relation. For instance $\approx_l$ will denote our (weak, labeled) bisimilarity.

By definition, the open RCHAM allows intrusion of messages on any name that has been extruded, independently of the contents of the chemical solution. This behavior is consistent with asynchrony, because intrusion transitions correspond to the emission of messages toward the solution, and should not be able to detect the reception of messages. Unfortunately, this behavior induces "useless" intrusions in most states, and typically leads to an infinite model. Consequently, and even as the reaction rules and the current messages that are present in solution entirely determine which intrusions may affect the computation, this information is not directly useful in proofs of weak bisimulation because the candidate relation must be closed for all intrusions anyway.

To reduce the size of our model, we alter the open RCHAM and equip it with an alternative equivalence called *asynchronous bisimulation*. The new chemical machine restricts intrusions to sets of messages that immediately trigger a reaction rule; this technical device significantly augments the blocking capability of processes. Asynchronous bisimulation combines two novel features: it has a *delay clause* that handles asynchrony in the spirit of [16]—messages that are intruded need not be used at once; they can be delayed by putting them in parallel in the simulating process—and it allows the intrusion of several messages at the same time, instead of single ones. We prove that weak bisimulation and asynchronous bisimulation coincide, which validates the use of asynchronous bisimulation in proofs.

We complete our study of labeled equivalences by considering their connection with the equivalences of Chapter 4. Reduction-based equivalences are easily extended to open terms, which naturally raises the issue of their coincidence with labeled equivalences. Our last characterization of weak/asynchronous bisimulation is given in terms of barbed equivalences. The barbed congruence of Section 4.4 is at the upper tier of our hierarchy so far, but it is still coarser than labeled bisimilarity, because contexts in barbed congruence have no way to compare names as such, while labels in weak bisimulations separate distinct names that exhibit the same behavior. This is actually the only difference between the two equivalences. To establish this, we augment the join-calculus with a name-comparison operator and we show that barbed congruence coincides with weak bisimulation in the resulting calculus.

Beyond its use as a proof technique, our labeled semantics yields another point of view on the join-calculus and provides a basis for comparing it with other calculi, usually equipped with such weak bisimulations, and especially with the asynchronous π-calculus [37]. In all these calculi, asynchrony means that message outputs have no continuation, and thus that there is no direct way to detect that a message has been received. Noticeably, the usual weak bisimulation of the π-calculus has too much discriminating power, and separates processes with the same behavior (e.g.,

$0 \not\approx_l x(u).\overline{x}\langle u\rangle$); several remedies are considered in [71, 16]. Our bisimulations for the join-calculus make use of similar accommodations, but they yield simpler semantics, mostly because locality constrains interaction with the environment. This prunes the number of transitions to consider, and rules out processes such as $x(u).\overline{x}\langle u\rangle$ where x is used by the environment both for input and output.

Contents of the chapter

In Section 5.1 we extend our syntax and chemical semantics to obtain an open model of the join-calculus. In Section 5.2 we recall the definition of labeled bisimulation in this setting and we study its basic properties. In Section 5.3 we introduce asynchronous bisimulation. In Section 5.4 we carry over our previous reduction-based equivalences to the open join-calculus, and we compare barbed congruence and labeled bisimulation. In Section 5.5 we supplement our calculus with comparison between names, and we show that barbed congruence now coincides with both labeled bisimulations. In Section 5.6 we compare our labeled equivalences to those that equip the asynchronous π-calculus.

5.1 Opening the calculus

We first define the open join-calculus and its operational semantics as extensions of the join-calculus and the RCHAM that are introduced in Chapter 2.

5.1.1 Open syntax

We define *open processes* $A, B \ldots \in \mathcal{A}$ by the following grammar, where nested occurrences of D, P, J within A denote terms of the grammar for the plain join-calculus (Figure 2.1 on page 59).

$$\begin{array}{rcll} A & ::= & & \text{open process} \\ & & x\langle v_1, \ldots, v_n\rangle & \quad\text{message} \\ & | & \mathsf{def}_S\ D\ \mathsf{in}\ A & \quad\text{open definition} \\ & | & A \,|\, A' & \quad\text{parallel composition} \\ & | & \mathsf{0} & \quad\text{null process} \end{array}$$

An open process A is like a regular join-calculus process P, except that its definitions in evaluation context are decorated with *extruded names*: the open definition $\mathsf{def}_S\ D\ \mathsf{in}\ A$ exhibits the subset S of names defined by D that are visible from the environment. We may omit the set S when it is empty, and identify open definitions $\mathsf{def}_\emptyset\ D\ \mathsf{in}\ P$ with local definitions $\mathsf{def}\ D\ \mathsf{in}\ P$; thus open processes formally extend our previous notion of processes ($\mathcal{P} \subset \mathcal{A}$).

The *interface* of an open process A consists of two disjoint sets of names: free names $\mathsf{fv}[A]$ used in A to send messages out, and extruded names $\mathsf{xv}[A]$ (that collects all index sets S in A) used by the environment to send messages in. We adapt our scoping rule to keep track of extruded names as well as defined and free names:

Definition 5.1 *Names that appear in terms are partitioned as follows:* received names*,* defined-local names*,* extruded names*, and* free names.

Visible names *consist of free names* $\mathsf{fv}[A]$ *and extruded names* $\mathsf{xv}[A]$*. Local names are bound in join-pattern; they consist of received names* $\mathsf{rv}[J]$ *and defined names* $\mathsf{dv}[J]$*. The scoping rules are given in Figure 5.1.*

If we compare the scoping rules for the open join-calculus to those for the plain calculus (Figure 2.2), we easily check that this is indeed a refinement by replacing $\mathsf{xv}[A]$ by $\emptyset$ everywhere.

The rule for free variables in defining processes $\mathsf{def}\ D\ \mathsf{in}\ A$ restricts all defined names, but some of them are kept visible as extruded names. This has an impact on the composition of open definitions and open processes, as the names that are free in one component and extruded in another component are extruded—hence not free—in the compound process.

The above definition induces well-formed conditions on terms. As in the join-calculus, we consider processes modulo α-conversion on *bound names*, i.e. received names and defined non-extruded names, and still assume that all join-patterns are linear. In addition, we require that:

1. sets of names extruded by different open sub-processes be pairwise disjoint—intuitively these names are independently defined. This distinction is enforced by the disjoint union operator $\uplus$ in the definition of $\mathsf{xv}[\,\cdot\,]$;

2. open definitions $\mathsf{def}_S\ D\ \mathsf{in}\ P$ define all their extruded names ($S \subseteq \mathsf{dv}[D]$).

(The last restriction is rather natural, but is not technically needed: in the case $x \notin \mathsf{dv}[D]$, the open process $\mathsf{def}_{\{x\}\cup S}\ D\ \mathsf{in}\ P$ would trivially be encoded as $\mathsf{def}_{\{x\}\cup S}\ D \wedge x\langle\widetilde{v}\rangle \mid y\langle\rangle \triangleright 0\ \mathsf{in}\ P$ where $y \notin \mathsf{fv}[D] \cup \mathsf{fv}[P]$, with the same intended behavior: any message sent on x is inert.)

Next we define a notion of renamings that operate on families of open processes, in a way that preserves locality. In the following, most renamings operate only on free variables, and leave extruded variables unchanged.

Definition 5.2 *A* global renaming σ *is a substitution on the interface of open processes that is injective on extruded names.*

5.1.2 Open chemistry

We introduce the operational semantics of the open calculus through an example that explains the role played by the index S in open definitions. Consider the process $\mathsf{def}_\emptyset\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ z\langle x\rangle$. The interface contains no extruded name so far, and two free names y, z. The message $z\langle x\rangle$ can be consumed by the environment, thus exporting x. This yields the transition

$$\mathsf{def}_\emptyset\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ z\langle x\rangle \xrightarrow{\{x\}\overline{z}\langle x\rangle} \mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ 0$$

Once x is known from the environment, it cannot be considered local anymore—the environment can emit on x—, but is not free either—the environment cannot modify or extend its definition. An intrusion transition is enabled:

$$\mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ 0 \xrightarrow{x\langle\rangle} \mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle\rangle$$

$$\begin{aligned}
\mathsf{xv}[x\langle v_1,\ldots,v_n\rangle] &\overset{\text{def}}{=} \emptyset\\
\mathsf{xv}[\mathsf{def}_S\, D \;\mathsf{in}\, A] &\overset{\text{def}}{=} S \uplus \mathsf{xv}[A]\\
\mathsf{xv}[A \mid A'] &\overset{\text{def}}{=} \mathsf{xv}[A] \uplus \mathsf{xv}[A']\\
\mathsf{xv}[0] &\overset{\text{def}}{=} \emptyset\\
\\
\mathsf{fv}[x\langle v_1,\ldots,v_n\rangle] &\overset{\text{def}}{=} \{x, v_1,\ldots,v_n\}\\
\mathsf{fv}[\mathsf{def}_S\, D \;\mathsf{in}\; A] &\overset{\text{def}}{=} (\mathsf{fv}[A] \cup (\mathsf{fv}[D] \setminus \mathsf{xv}[A])) \setminus \mathsf{dv}[D]\\
\mathsf{fv}[A \mid A'] &\overset{\text{def}}{=} (\mathsf{fv}[A] \setminus \mathsf{xv}[A']) \cup (\mathsf{fv}[A'] \setminus \mathsf{xv}[A])\\
\mathsf{fv}[0] &\overset{\text{def}}{=} \emptyset\\
\\
\mathsf{fv}[J \triangleright P] &\overset{\text{def}}{=} \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J])\\
\mathsf{fv}[D \wedge D'] &\overset{\text{def}}{=} \mathsf{fv}[D] \cup \mathsf{fv}[D']\\
\mathsf{fv}[\mathsf{T}] &\overset{\text{def}}{=} \emptyset
\end{aligned}$$

$\mathsf{rv}[J]$, $\mathsf{dv}[J]$, and $\mathsf{dv}[D]$ are defined as in Figure 2.2; $\mathsf{fv}[P] \overset{\text{def}}{=} \mathsf{fv}[A]$

Figure 5.1: Scopes for the open join-calculus

STR-JOIN	$\vdash_S A \vert B$	$\rightleftharpoons$	$\vdash_S A,\ B$
STR-NULL	$\vdash_S 0$	$\rightleftharpoons$	$\vdash_S$
STR-AND	$D \wedge D' \vdash_S$	$\rightleftharpoons$	$D,\ D' \vdash_S$
STR-NODEF	$\mathsf{T} \vdash_S$	$\rightleftharpoons$	$\vdash_S$
STR-DEF	$\vdash_S \mathsf{def}_{S'}\ D\ \mathsf{in}\ A$	$\rightleftharpoons$	$D\sigma \vdash_{S \uplus S'} A\sigma$
RED	$J \triangleright P \vdash_S J\sigma$	$\longrightarrow$	$J \triangleright P \vdash_S P\sigma$
EXT	$\vdash_S x\langle v_1,\ldots,v_p\rangle$	$\xrightarrow{S'\overline{x}\langle v_1,\ldots,v_p\rangle}$	$\vdash_{S \uplus S'}$
INT	$\vdash_{S \cup \{x\}}$	$\xrightarrow{x\langle v_1,\ldots,v_p\rangle}$	$\vdash_{S \cup \{x\}} x\langle v_1,\ldots,v_p\rangle$

Side conditions on the reacting solution $\mathcal{S} = \mathcal{D} \vdash_S \mathcal{A}$:

STR-DEF	σ replaces $\mathsf{dv}[D] \setminus S'$ with distinct fresh names;
RED	ρ substitutes names for $\mathsf{rv}(J)$;
EXT	$x \in \mathsf{fv}[\mathcal{S}]$, $S' = \{v_1,\ldots,v_p\} \cap (\mathsf{dv}[\mathcal{D}] \setminus S)$;
INT	$\{v_1,\ldots,v_p\} \cap \mathsf{dv}[\mathcal{D}] \subseteq S$.

Figure 5.2: The open reflexive chemical machine

Now the resulting process can input some more messages on x, and independently it can perform two transitions to trigger the message $y\langle\rangle$ and pass it to the context:

$$\begin{array}{rcl} \mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle\rangle & \longrightarrow & \mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ y\langle\rangle \\ & \xrightarrow{\{\}\overline{y}\langle\rangle} & \mathsf{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ 0 \end{array}$$

In order to model interactions with the environment, we now extend the RCHAM model defined in Section 2.3 with two operations: *extrusion* of defined names on free names, and *intrusion* of messages sent on previously extruded names. The explicit bookkeeping of extruded names is performed by suitably augmenting the chemical solutions.

Definition 5.3 Open chemical solutions $\mathcal{S}$, $\mathcal{T}$ *are triples* $(\mathcal{D}, S, \mathcal{A})$, *written* $\mathcal{D} \vdash_S \mathcal{A}$, *where*

- $\mathcal{A}$ *is a multiset of open processes with disjoint sets of extruded names.*
- $\mathcal{D}$ *is a multiset of definitions such that* $\mathsf{dv}[\mathcal{D}] \cap \mathsf{xv}[\mathcal{A}] = \emptyset$
- S *is a subset of* $\mathsf{dv}[\mathcal{D}]$.

As in Section 2.3, we lift the functions $\mathsf{dv}[\cdot], \mathsf{fv}[\cdot], \mathsf{xv}[\cdot]$ from terms to multisets of terms, then to chemical solutions component-wise:

$$\begin{array}{rcl} \mathsf{fv}[\mathcal{D} \vdash_S \mathcal{A}] & \stackrel{\text{def}}{=} & (\mathsf{fv}[\mathcal{A}] \cup (\mathsf{fv}[\mathcal{D}] \setminus \mathsf{xv}[\mathcal{A}])) \setminus \mathsf{dv}[\mathcal{D}] \\ \mathsf{xv}[\mathcal{D} \vdash_S \mathcal{A}] & \stackrel{\text{def}}{=} & S \uplus \mathsf{xv}[\mathcal{A}] \end{array}$$

The chemical rules for the open RCHAM are given in Figure 5.2; they define families of transitions between open solutions $\rightharpoonup$, $\rightharpoondown$, $\longrightarrow$, $\xrightarrow{\alpha}$ where α ranges over labels $x\langle\widetilde{v}\rangle$ and $S\overline{x}\langle\widetilde{v}\rangle$ for all subsets S of $\{\widetilde{v}\}$. Since the chemical rules have slightly different side conditions and operate on more general solutions, we recall the whole chemical machinery. As usual, each chemical rule mentions only the processes and the definitions that take part to the transition, and the rule applies to every chemical solution $\mathcal{S}$ whose multisets contain these processes and definitions.

The first six rules are those of the RCHAM: the structural rules that define the inverse relations $\rightharpoonup$ and $\rightharpoondown$ are unchanged, except for Rule STR-DEF that performs some extra bookkeeping on extruded names; the reduction rule RED is unchanged.

The last two rules model interaction with the context. The extrusion rule EXT consumes messages sent on free names; these messages can contain defined names not previously known to the environment, thus causing the scope of their definitions to be opened; this is made explicit in the set of extruded names S' that appears in front of the transition label. Our rule resembles the OPEN rule for restriction in the π-calculus [100], with an important restriction due to locality: for each message $x\langle\widetilde{v}\rangle$, either x is free and the message is consumed only by rule EXT, or x is defined, and the message may be consumed only by rule RED. The intrusion rule INT can be viewed as a disciplined version of one of the INPUT rule for the asynchronous π-calculus proposed by Honda and Tokoro [71]: the latter allows intrusion of any message, whereas our rule allows intrusion of messages only on defined-extruded names.

Formally, the RCHAM of Section 2.3 is the restriction of the present open RCHAM when it operates on chemical solutions with no extruded variables, and once the two opening rules EXT and INT have been disabled. In particular, we retain the existence of fully-diluted normal forms of the RCHAM:

Remark 5.4 *Every open chemical solution is structurally equivalent*

- *to a fully-heated solution that contains only simple reaction rules and messages (the solution is unique up to α-conversion)*

$$\{\cdots J_j \triangleright P_j, \cdots\} \vdash_S \{\cdots x_i\langle\widetilde{u_i}\rangle, \cdots\}$$

- *and to the corresponding solution that contains a single open process*

$$\emptyset \vdash_\emptyset \left\{\mathsf{def}_S \bigwedge J_j \triangleright P_j \;\mathsf{in}\; \prod x_i\langle\widetilde{u_i}\rangle\right\}$$

The multiset of messages can be further partitioned into three multiset of messages on free names, messages on extruded names, and messages on locally-defined names. Informally, all messages on free names are available to the context, and could be exported at once.

When applied to open solutions, the structural rules capture the intended meaning of extruded names: messages sent on extruded names can be moved inside or outside their defining process. Assuming x is an extruded name ($x \in S \cup S' \cup \mathsf{xv}[A]$), and modulo α-conversion to avoid clashes with the arguments ($\{\widetilde{v}\} \cap \mathsf{dv}[D] \subseteq S'$), we have the structural rearrangement

$$\vdash_S \; x\langle\widetilde{v}\rangle \,|\mathsf{def}_{S'} \; D \;\mathsf{in}\; A \quad \rightleftharpoons^* \quad \vdash_S \; \mathsf{def}_{S'} \; D \;\mathsf{in}\; (x\langle\widetilde{v}\rangle \mid A)$$

Remark 5.5 *In Section 2.5, we presented another auxiliary labeled transition system. The intent, however, was quite different. It was used to give an alternative, syntactic description of chemical reduction steps. In contrast, the open* RCHAM *provides an extensional semantics of the calculus; the labels are largely independent of the definitions, and we do not attempt to define silent steps as matching labeled transitions plus hiding.*

We follow the same convention as before: we use open processes and open solutions interchangeably, and we consider transitions on processes as chemical transitions up to structural equivalence: $\xrightarrow{\alpha}$ between processes is actually $\rightleftharpoons^* \xrightarrow{\alpha} \rightleftharpoons^*$ between solutions.

5.1.3 Tracking transitions in context

We conclude this section by mentioning technical properties of the open RCHAM as regards global renamings and contexts.

We introduce an auxiliary notation: we use a linear restriction operator to represent the output-only scope of extruded names. Given a set of names S and a fully-heated solution $\mathcal{T} = \mathcal{D} \vdash_{S' \uplus S} \mathcal{A}$, the *restriction* of $\mathcal{T}$ on S is written $\mathcal{T} \setminus S$ and defined as follows:

$$(\mathcal{D} \vdash_{S' \uplus S} \mathcal{A}) \setminus S \quad \stackrel{\text{def}}{=} \quad \mathcal{D} \vdash_{S'} \mathcal{A}$$

By extension, the restriction operator $\cdot \setminus S$ is partially defined on all open solutions and open processes by applying the restriction to the structurally-equivalent fully-heated open solution, provided that every name in S is extruded by the solution. (Structural rearrangements may be required in order to gather all the instances of an extruded name within the scope of its definition before restricting it.)

The next lemma relates the transitions in a chemical solution $\mathcal{S}$ to the transitions in the solution $\mathcal{S}\sigma$, where σ is a global renaming.

Lemma 5.6 (Transitions through renamings) *Let $\mathcal{S}$ and $\mathcal{T}$ be open solutions, S be a set of names, and σ be a global renaming for $\mathcal{S}$ that does not operate or range over S.*

$$\begin{array}{rcll}
\mathcal{S} \rightleftharpoons^* \mathcal{T} & \textit{iff} & \mathcal{S}\sigma \rightleftharpoons^* \mathcal{T}\sigma & (5.1)\\
\mathcal{S} \rightarrow \mathcal{T} & \textit{implies} & \mathcal{S}\sigma \rightarrow \mathcal{T}\sigma & (5.2)\\
\mathcal{S} \xrightarrow{x\langle\widetilde{v}\rangle} \mathcal{T} & \textit{implies} & \mathcal{S}\sigma \xrightarrow{(x\langle\widetilde{v}\rangle)\sigma} \mathcal{T}\sigma & (5.3)\\
\mathcal{S} \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \mathcal{T} & \textit{implies} & \textit{either } \sigma(x) \in \mathsf{xv}[\mathcal{S}\sigma], \textit{ or } \xrightarrow{S\overline{x\sigma}\langle\widetilde{v}\sigma\rangle} \mathcal{T}\sigma & (5.4)\\
\mathcal{S}\sigma \rightarrow \mathcal{T} & \textit{implies} & \mathcal{S} \rightleftharpoons^* \xrightarrow{S_1\overline{x_1}\langle\widetilde{v}_1\rangle} \xrightarrow{y_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{S_n\overline{x_n}\langle\widetilde{v}_n\rangle} \xrightarrow{y_n\langle\widetilde{v}_n\rangle} \rightleftharpoons^* \rightarrow \mathcal{S}' & (5.5)\\
 & & \textit{with } n \geq 0, \forall i \leq n. \sigma(x_i) = \sigma(y_i), \mathcal{T} \rightleftharpoons^* \mathcal{S}' \setminus (\bigcup_{i=1\ldots n} S_i) & \\
\mathcal{S}\sigma \xrightarrow{y\langle\widetilde{w}\rangle} \mathcal{T} & \textit{implies} & \mathcal{S} \xrightarrow{x\langle\widetilde{v}\rangle} \mathcal{S}' & (5.6)\\
 & & \textit{with } (x\langle\widetilde{v}\rangle)\sigma = y\langle\widetilde{w}\rangle,\ \mathcal{S}'\sigma = \mathcal{T} & \\
\mathcal{S}\sigma \xrightarrow{S\overline{y}\langle\widetilde{w}\rangle} \mathcal{T} & \textit{implies} & \mathcal{S} \xrightarrow{(S)\overline{x}\langle\widetilde{v}\rangle} \mathcal{S}' & (5.7)\\
 & & \textit{with } (x\langle\widetilde{v}\rangle)\sigma = y\langle\widetilde{w}\rangle,\ \mathcal{S}'\sigma = \mathcal{T} &
\end{array}$$

The statements in the above lemma are natural, except perhaps for statement 5.5 with $n > 0$. This case occurs when an internal step consumes messages on extruded names $\sigma(x_i)$ where x_i are free names in the initial solution $\mathcal{S}$. For every such message that is consumed in the internal step, this is mimicked as an extrusion on x_i followed by an intrusion on y_i.

Proof: **5.1** Each structural rule obviously commutes with the global renaming, except perhaps for the rule STR-DEF, because its side condition requires that the defined-local names be disjoint from the visible names. We use preliminary α-conversion to prevent any clash.

5.2, 5.3, 5.6, 5.7 The transition commutes with the renaming by definition.

5.4 The extrusion remains enabled inasmuch as $\sigma(x)$ is free.

5.5 Let $M\sigma$ be the multiset of emissions that is consumed by the reaction rule used for this silent move. After reordering the messages and performing α-conversion to preclude name-clashes, we assume that

$$M = x_1\langle\widetilde{v}_1\rangle \mid \cdots \mid x_n\langle\widetilde{v}_n\rangle \mid x_{n+1}\langle\widetilde{v}_{n+1}\rangle \mid \cdots \mid x_m\langle\widetilde{v}_m\rangle$$

where the names $x_i, i \leq n$ are free and the names $x_j, j > n$ are defined. In particular we can extrude the first n messages at any time using Rule EXT. For every $i = 1 \dots n$, the possibility of the silent move after substitution ensures that $\sigma(x)$ is an extruded name. This is possible only inasmuch as $\sigma^{-1}(\sigma(x))$ contains exactly one extruded name y_i, which can be used before substitution to re-intrude the same arguments on a different name. Once these $2n$ transitions have been completed, we use structural equivalence to re-assemble a multiset of messages

$$M' = y_1\langle\widetilde{v}_1\rangle \,|\, \cdots \,|\, y_n\langle\widetilde{v}_n\rangle \,|\, x_{n+1}\langle\widetilde{v}_{n+1}\rangle \,|\, \cdots \,|\, x_m\langle\widetilde{v}_m\rangle$$

By construction we have $M'\sigma = M\sigma$ and, since σ is injective on the extruded names y_i M', M' matches the join-pattern of the same reaction rule before substitution. Once the reaction has occurred, we apply σ and obtain the same multisets of processes and definitions in $\mathcal{S}'\sigma$ than in $\mathcal{T}$. However, the transmitted names in each set $S_i \subseteq \widetilde{v}_i$ have been unduly extruded to the context in the round trip, thus the solution $\mathcal{S}'$ is more open than $\mathcal{T}$. $\square$

Remark 5.7 (Transitions through parallel composition) *Let A and B be two open processes such that $A \,|\, B$ is well-formed. Let us assume that we have $A \,|\, B \rightarrow T$, and that this reduction uses a join-pattern in B. Then for some integer $n \geq 0$ there are sets of disjoint fresh names $S_1, \dots, S_n$ and series of transitions*

$$\begin{array}{lcll} A & \xrightarrow{S_1\overline{x_1}\langle\widetilde{y}_1\rangle} \cdots \xrightarrow{S_n\overline{x_n}\langle\widetilde{y}_n\rangle} & A' & \\ B & \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} & B' & \rightarrow \; B'' \end{array}$$

such that the reduction can be rewritten

$$A \,|\, B \;\rightleftharpoons^*\; (A' \,|\, B') \setminus \bigcup_{i=1}^{n} S_i \;\rightarrow\; (A' \,|\, B'') \setminus \bigcup_{i=1}^{n} S_i \;\rightleftharpoons^*\; T$$

Proof: We collect the received messages $x_1\langle\widetilde{y}_1\rangle, \dots, x_n\langle\widetilde{y}_n\rangle$ that are in evaluation context in A. For each message, we have $x_i \in \mathsf{xv}[A] \cap \mathsf{fv}[B]$. Using α-conversion, we repeatedly rename all local variables in $\widetilde{y}_i$ to fresh names, then perform an extrusion. On the other side, intrusions are always enabled, and yield no further renaming. $\square$

5.2 Weak bisimulation

Our main motivation for opening the join-calculus is to develop labeled proof techniques. We now define labeled bisimulation for open join processes and investigate its basic properties.

The next definition specializes our definition of bisimulation (Definition 4.11) to the open join-calculus.

Definition 5.8 *A relation $\mathcal{R}$ over open processes is a* weak simulation *if, for all open processes A and B such that $A \mathrel{\mathcal{R}} B$, we have*

1. *if $A \rightarrow A'$ then $B \rightarrow^* B'$ with $A' \mathrel{\mathcal{R}} B'$;*
2. *if $A \xrightarrow{x\langle\widetilde{v}\rangle} A'$ then $B \rightarrow^* \xrightarrow{x\langle\widetilde{v}\rangle} \rightarrow^* B'$ with $A' \mathrel{\mathcal{R}} B'$;*
3. *if $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} A'$ and $S \cap \mathsf{fv}[B] = \emptyset$, then $B \rightarrow^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \rightarrow^* B'$ with $A' \mathrel{\mathcal{R}} B'$.*

$\mathcal{R}$ is a weak bisimulation *when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations.* Weak bisimilarity $\approx_l$ *is the largest weak bisimulation.*

We omit the formal definition of labeled strong bisimulation $\sim_l$ and labeled expansions $\preceq_l$, $\asymp_l$, which are obtained by restricting the number of internal steps on the right-hand-side of the above requirements (in *1.*, by replacing $\rightarrow^*$ by $\rightarrow$ and $\rightarrow^=$ respectively; in *2.,3.*, by removing $\rightarrow^*$). The proof techniques developed in Section 4.7 also apply to labeled diagrams. Typically, labeled transitions would be placed above all other relations in the relation ordering to retain the benefit of proofs of weak bisimulations up to expansion.

As usual with mobile calculi, the simulation clause for extrusion *(3)* does not consider labels $S\overline{x}\langle\widetilde{v}\rangle$ in which some of the names extruded in S are free in $\mathsf{fv}[B]$. Without this condition, the extrusion of names on one side that are already in the interface on the other side could not be simulated. This standard technicality does not affect the intuitive discriminating power of bisimulation, because we consider terms up to α-conversion.

Weak bisimilarity is a precise equivalence relation which discriminates processes without the need for additional congruence requirements: we can tell whether two processes are bisimilar by reasoning about their synchronization tree. For example, $x\langle u\rangle \not\approx_l x\langle v\rangle$ because the two processes perform emissions with different labels. Likewise, $x\langle y\rangle \not\approx_l \mathsf{def}\ z\langle u\rangle \triangleright y\langle u\rangle\ \mathsf{in}\ x\langle z\rangle$ because the first process emits a free name (label $\overline{x}\langle y\rangle$) while the latter emits a local name that gets extruded (label $\{z\}\overline{x}\langle z\rangle$). Positive examples of processes that are indeed weakly bisimilar are presented in the next chapter; for the time being, we focus on the basic properties of weak bisimilarity.

Weak bisimulation is sensitive to the set of extruded names $\mathsf{xv}[A]$ because the input-interface of the open process A entirely determines the use of rule INT:

Remark 5.9 *$A \approx_l B$ implies $\mathsf{xv}[A] = \mathsf{xv}[B]$.*

Proof: To check this property, we consider the partition of open RCHAMs induced by $\mathsf{xv}[\cdot]$. The resulting classes are closed for all chemical rules except EXT, and in particular, any solution is structurally equivalent to a fully-heated solution with the same extruded variables. By definition of INT, a fully-heated solution can perform a transition with label $x\langle\widetilde{v}\rangle$ if and only if $x \in \mathsf{xv}[\mathcal{S}]$. □

Conversely, rule EXT defines the only transitions that can separate solutions with the same extruded names. Obviously, open solutions whose derivatives never perform EXT are bisimilar if and only if they have the same extruded names to begin with, and this provides some "garbage-collection" properties such as $\mathsf{fv}[P] = \emptyset$ implies $P \approx_l 0$, much in the same manner as in the previous chapter where all processes with no barbs were equated.

5.2.1 Renaming and congruence properties

The rest of the section is devoted to the proof that weak bisimulation is a congruence. This is most useful to position weak bisimulation on top of our hierarchy of congruence relations. From the experience of the π-calculus, the congruence property is not really a surprise because our calculus has no operator for external choice. Nevertheless the proof is not completely standard and some interesting properties are required.

As explained in Section 4.7, it is usually more efficient to prove that processes are bisimilar using relations much smaller than bisimulations, and a whole range of "up to proof techniques" are available to reduce the size of the relation in use. Accordingly, we establish that our definition of weak bisimulation is robust with respect to reasonings up to structural equivalence, restriction of the input interface, global renamings, and weak bisimulation on the right. We derive the definitions of "weak bisimulation up to" from the definition of weak bisimulation and use the resulting definitions as proof techniques.

Lemma 5.10 *Let $\mathcal{R}$ be a relation that satisfies Definition 5.8 where the requirements $A' \mathrel{\mathcal{R}} B'$ in the three clauses of the definition has been replaced with one of the following weaker requirements.*

1. $A' \rightleftharpoons^* \mathcal{R} \rightleftharpoons^* B'$
2. $A' \mathcal{R} \approx_l B'$
3. *There is a set of names S such that $A' = A''\backslash S$, $B' = B''\backslash S$, and $A'' \approx_l B''$.*
4. *There is a global renaming σ such that $A' = A''\sigma$, $B' = B''\sigma$, and $A'' \approx_l B''$.*

Then we have $\mathcal{R} \subseteq \approx_l$.

Proof: In each case, we exhibit a weak bisimulation that contains $\mathcal{R}$:

1. The relation $\rightleftharpoons^* \mathcal{R} \rightleftharpoons^*$ is a weak bisimulation, because weak bisimilarity already takes into account structural rearrangements.
2. The relation $\approx_l \mathcal{R} \approx_l$ is a weak bisimulation; this directly follows from the transitivity of $\approx_l$, or from the orderings of relations of Section 4.7.
3. The following relation
$$\mathcal{R} \setminus S \quad \stackrel{\text{def}}{=} \quad \{(A\backslash S, B\backslash S) \mid A \approx_l B \text{ and both restrictions are defined}\}$$
is a weak bisimulation. All transitions after restriction are allowed before, and thus the bisimulation properties after restriction are direct consequences of the same properties before restriction.
4. The following relation
$$\Sigma(\mathcal{R}) \stackrel{\text{def}}{=} \{(A\sigma, B\sigma) \mid A \mathrel{\mathcal{R}} B \text{ and } \sigma \text{ global renaming}\}$$
is a weak bisimulation up to restriction. The proof relies on our case analysis of the effect of each kind of renaming on every chemical rule. We show in detail only the case of internal reduction, as the other cases are much easier.

   Let $A\sigma \rightleftharpoons^* \rightarrow \rightleftharpoons^* A'\sigma$. By Lemma 5.6(5.5) we obtain a series of extrusion-intrusion pairs followed by a silent move
$$A \quad \xrightarrow{S_1\overline{x_1}\langle\widetilde{v}_1\rangle} \xrightarrow{y_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{S_n\overline{x_n}\langle\widetilde{v}_n\rangle} \xrightarrow{y_n\langle\widetilde{v}_n\rangle} \rightarrow \quad A''$$

We apply the bisimulation property to every transition in turn, and thus we obtain a derivation leading to some B'' such that $A'' \mathcal{R} B''$:

$$B \rightarrow^* \xrightarrow{S_1 \overline{x_1}\langle \widetilde{v}_1 \rangle} B_1 \rightarrow^* B_1' \xrightarrow{y_1\langle \widetilde{v}_1 \rangle} B_1'' \rightarrow^* \cdots \rightarrow^* B_n' \xrightarrow{y_n\langle \widetilde{v}_n \rangle} B_n'' \rightarrow^* B''$$

To reflect this series of transitions after applying the global renaming σ, we study the action of σ on each transition. Silent steps remain silent, but the extrusions cannot be performed anymore, so we keep the message in solution. We use the auxiliary notation $B_i[M]$ to represent the process B_i with one extra message M in evaluation context. For instance we have $B_i'' = B_i'[y_i\langle \widetilde{v}_i \rangle]$.

Let $S \overset{\text{def}}{=} \bigcup S_i$. The transitions after renaming become,

$$\begin{array}{rl} B\sigma \rightarrow^* (B_1[x_1\langle \widetilde{v}_1 \rangle])\sigma \backslash S_1 \rightarrow^* & (B_1'[x_1\langle \widetilde{v}_1 \rangle])\sigma \backslash S_1 \\ = & (B_1'[y_1\langle \widetilde{v}_1 \rangle])\sigma \backslash S_1 \\ = & B_1''\sigma \rightarrow^* \cdots \rightarrow^* \quad (B_n'[x_n\langle \widetilde{v}_n \rangle])\sigma \backslash S \\ & \qquad\qquad\qquad = \; (B_n'[y_n\langle \widetilde{v}_n \rangle])\sigma \backslash S \\ & \qquad\qquad\qquad = \; B_n''\sigma \backslash S \rightarrow^* B''\sigma \backslash S \end{array}$$

that is, after aggregating the internal moves, $B\sigma \rightarrow^* B''\sigma \setminus S$. We thus have $A' = A''\sigma \setminus S$, $B' = B''\sigma \setminus S$, and $(A''\sigma, B''\sigma) \in \Sigma(\mathcal{R})$. □

In particular, if we take $\mathcal{R} = \approx_l$ we directly obtain that weak bisimulation is preserved by structural equivalence, restriction and global renaming.

Corollary 5.11 *Let $A, B \in \mathcal{A}$ with $A \approx_l B$. For every global renaming σ, for every set of names $S \subseteq \mathcal{N}$, we have $A\sigma \setminus S \approx_l B\sigma \setminus S$ when both processes are defined.*

The next lemma recalls a standard proof technique to derive the full congruence property from closure under name substitution. The notion of "bisimilarity" is left unspecified; formally we take $\Phi = \approx_l$, and we recheck the proof for any other notion of bisimilarity.

Lemma 5.12 *Let Φ be a bisimilarity that is a congruence for all evaluation contexts and that is closed by substitution. Then Φ is a full congruence.*

Proof: We prove that Φ is a congruence for guarded processes in definitions, that is, for all $C[\cdot] = \mathsf{def}\ D \wedge J \triangleright [\cdot]\ \mathsf{in}\ A$, if $P\ \Phi\ Q$ then $C[P]\ \Phi\ C[Q]$. We then conclude by induction on the structure of an arbitrary context.

For a given pair of processes $P\ \Phi\ Q$, we let $\mathcal{R}$ relate the pairs of processes $(C[P], C[Q])$ for all D and A. We prove that $\mathcal{R}$ is a Φ-bisimulation up to Φ-bisimilarity on the right (Lemma 5.10(2)).

All transitions are identical on both sides of $\mathcal{R}$, and lead to processes related by $\mathcal{R}$ for some updated D and A, except for the reduction steps that consume messages $J\sigma$ to trigger the reaction rule $J \triangleright Q$ and $J \triangleright R$, respectively. For that case, we use the double-context notation $E_{(\cdot_1)}[\cdot_2] \overset{\text{def}}{=} \mathsf{def}\ D \wedge J \triangleright [\cdot_1]\ \mathsf{in}\ [\cdot_2]$ and we establish the diagram

$$\begin{tikzcd}
E_P[A \mid J\sigma] \arrow[rr, dash, "\mathcal{R}"] \arrow[d] & & E_Q[A \mid J\sigma] \arrow[d, dotted] \\
E_P[A \mid P\sigma] \arrow[r, dotted, dash, "\mathcal{R}"] & E_Q[A \mid P\sigma] \arrow[r, dotted, dash, "\Phi"] & E_Q[A \mid Q\sigma]
\end{tikzcd}$$

where the $\mathcal{R}$ relation holds for some updated context $C[\,\cdot\,]$ with $P\sigma$ instead of $J\sigma$, and where the Φ relation is obtained from $P \approx Q$ in two steps, by applying closure through the substitution σ, then congruence for the evaluation context $E_Q[\,\cdot\,]$. □

The congruence property essentially relies on this proposition; its proof is standard.

Lemma 5.13 *Weak bisimulation is a congruence for evaluation contexts ($\approx_l = \approx_l{}^\circ$).*

Proof: We first establish that for all open processes, $A, B, E \in \mathcal{A}$ such that $A \mid E$ and $B \mid E$ are well-formed, if $A \approx_l B$ then $A \mid E \approx_l B \mid E$.

Let $\mathcal{R}$ be the relation containing all such pairs $(A \mid E, B \mid E)$. We show that $\mathcal{R}$ is a weak bisimulation up to restriction and structural rearrangement.

Any transition that involves only E is clearly the same on both sides, and yields a new pair of related processes. Likewise, any transition that does not involve E is simulated on the other side by hypothesis and yields new related processes. This leaves us with two cases

- an internal reductions in E consumes messages of A,B

  By Remark 5.7—and with the same notations—we have series of transitions

$$\begin{array}{lllll} A & \xrightarrow{S_1\overline{x_1}\langle\widetilde{y}_1\rangle} \cdots \xrightarrow{S_n\overline{x_n}\langle\widetilde{y}_n\rangle} & A' & & \\ E & \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} & E' & \rightarrow & E'' \end{array}$$

  the first series of transitions can be simulated by B, hence for some B with $A' \approx_l B'$ we have

$$B \quad \rightarrow^* \xrightarrow{S_1\overline{x_1}\langle\widetilde{y}_1\rangle} \cdots \rightarrow^* \xrightarrow{S_n\overline{x_n}\langle\widetilde{y}_n\rangle} \rightarrow^* \quad B'$$

  We obtain $A' \mid E'' \mathrel{\mathcal{R}} B' \mid E''$, and a diagram of bisimulation up to restriction for the internal reductions $A \mid E \rightarrow (A' \mid E'') \setminus S$ and $B \mid E \rightarrow^* (B' \mid E'') \setminus S$.

- an internal reduction of A,B consumes messages of E.

  By Remark 5.7 again, we have series of transitions

$$\begin{array}{lllll} A & \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} & A' & \rightarrow & A'' \\ E & \xrightarrow{S_1\overline{x_1}\langle\widetilde{y}_1\rangle} \cdots \xrightarrow{S_n\overline{x_n}\langle\widetilde{y}_n\rangle} & E' & & \end{array}$$

  the first series of reductions can be simulated by B as

$$B \quad \rightarrow^* \xrightarrow{x_1\langle\widetilde{y}_1\rangle} \cdots \rightarrow^* \xrightarrow{x_n\langle\widetilde{y}_n\rangle} \rightarrow^* \quad B''$$

  and again we obtain a diagram of weak bisimulation up to restriction.

We easily extend the congruence property from open parallel composition to any evaluation context. Let $C[\cdot] \overset{\text{def}}{=} \mathsf{def}_S\ D\ \mathsf{in}\ [\cdot]$. For any pair of processes A, B such that $C[A]$ and $C[B]$ are well-formed, we let $S' = (\mathsf{dv}[D] \cap (\mathsf{fv}[A] \cup \mathsf{fv}[B])) \setminus S$, $E = \mathsf{def}_{S \cup S'}\ D\ \mathsf{in}\ M$, and by structural rearrangement we obtain $C[A] = (E \,|\, A) \setminus S'$ and $C[B] = (E \,|\, B) \setminus S'$. If $A \approx_l B$, then we have $E \,|\, A \approx_l E \,|\, B$ and, by Lemma 5.10(3), $(E \,|\, A) \setminus S' \approx_l (E \,|\, B) \setminus S'$, hence $C[A] \approx_l C[B]$. □

Theorem 5 *Weak bisimulation is a full congruence.*

Proof: We combine the lemmas 5.13 and 5.12 to extend the congruence property from evaluation contexts to all contexts of the open join-calculus. □

5.3 Asynchronous bisimulation

In order to prove that two chemical solutions are bisimilar, one has to consider a large number of multiset configurations, which may contrast with the intuitive behavior of the solution; for example, a process with an extruded name already has an infinite set of configurations, even if no "real" computation is ever performed. For instance, the two processes below are obviously equivalent

$$\mathsf{def}_{\{x\}}\ x\langle u\rangle|y\langle v\rangle \triangleright P\ \mathsf{in}\ 0 \quad \approx_l \quad \mathsf{def}_{\{x\}}\ x\langle u\rangle \,|\, y\langle v\rangle \triangleright Q\ \mathsf{in}\ 0$$

(the processes P and Q cannot be triggered, so their contents is irrelevant). Nonetheless, one is confronted to infinite models because the rule Int is enabled on both sides; hence, a candidate bisimulation that relates the two processes above would have to relate at least all pairs of processes obtained by replacing the null process 0 above by identical multisets of messages on x. This is unfortunate, because we introduced open semantics to get rid of quantification over all contexts, and we must still deal with quantification over all intruded messages. This flaw in our definition of open chemical semantics motivates the following alternative formulation.

We refine the open RCHAM by allowing intrusions only when they trigger some guarded process. For instance, the two processes above become inert, and trivially bisimilar.

If we directly apply our refinement with the same intrusion labels $x\langle\widetilde{v}\rangle$ as before, we obtain a dubious result. The process $\mathsf{def}_{\{x,y\}}\ x\langle\rangle|y\langle\rangle|z\langle\rangle \triangleright P\ \mathsf{in}\ z\langle\rangle$ triggers the guarded process P inasmuch as it first inputs the two messages $x\langle\rangle$ and $y\langle\rangle$, then performs a silent step that jointly consumes them together with the local message $z\langle\rangle$. Yet, neither of the messages $x\langle\rangle$ and $y\langle\rangle$ alone can trigger P, and therefore this solution would becomes inert with our proposed refinement. This suggests the use of *join*-inputs on x and y in new, larger transition steps such as

$$x\langle\rangle \,|\, y\langle\rangle \,|\, z\langle\rangle \triangleright P \vdash_{\{x,y\}} z\langle\rangle \xrightarrow{x\langle\rangle \,|\, y\langle\rangle} x\langle\rangle \,|\, y\langle\rangle \,|\, z\langle\rangle \triangleright P \vdash_{\{x,y\}} P$$

On the other hand, if we remove the message $z\langle\rangle$ from the previous example, the resulting process $\mathsf{def}_{\{x,y\}}\ x\langle\rangle|y\langle\rangle|z\langle\rangle \triangleright P\ \mathsf{in}\ 0$ is truly inert; in this case, join-inputs have a greater blocking ability than atomic inputs, and our refinement suppresses all input transitions.

Rules STR-(JOIN,NULL,AND,NODEF,DEF) and EXT are as in Figure 5.2;

$$\text{INT-J} \qquad J \triangleright P \vdash_S M' \quad \xrightarrow{M} \quad J \triangleright P \vdash_S P\rho$$

Side condition: $J\rho \rightleftharpoons^* M \mid M'$, $\quad \mathit{Dom}(\rho) = \mathsf{rv}[J]$,
$\mathsf{dv}[M] \subseteq S$, $\quad \mathsf{rv}[M]$ free, fresh or extruded.

Figure 5.3: The J-open RCHAM

5.3.1 The J-open RCHAM

The J-open RCHAM is defined in Figure 5.3, as a replacement of the intrusion rule. Unlike the previous rule INT, the new rule INT-J permits intrusion of messages only if they are immediately used to trigger a process in a definition. This is formalized by allowing labels M that represent multisets of messages, and by handling them as partial join-patterns: if the solution can supply a complementary parallel composition of messages M' such that the combination $M \mid M'$ matches the join-pattern in a reaction rule, then the transition occurs and triggers this reaction rule. As for the rule INT, we restrict intrusions in M to messages on extruded names. The side condition $J\rho \rightleftharpoons^* M \mid M'$ makes explicit the commutative-associative property of our join-patterns; it uses only the rules STR-JOIN and STR-NULL. Note that rule RED is now subsumed by rule INT-J in the case $M = \mathsf{0}$; nonetheless, we maintain an informal distinction between internal moves and proper input moves in the following discussion.

The same chemical solution now has two different models: for instance, the solution $x\langle\rangle | y\langle\rangle \triangleright P \vdash_{\{x\}}$ has no transition in the J-open RCHAM, while it has infinitely many transitions in the open RCHAM:

$$x\langle\rangle | y\langle\rangle \triangleright P \vdash_{\{x\}} \xrightarrow{x\langle\rangle} \xrightarrow{x\langle\rangle} \xrightarrow{x\langle\rangle} \cdots$$

In the sequel, we shall keep the symbol $\xrightarrow{\alpha}$ for the open RCHAM and use $\xrightarrow{\alpha}_J$ for the J-open RCHAM; the subscript J is dropped whenever no ambiguity can arise.

As a direct consequence of the definitions of the J-open semantics, we have the following relation between our two models:

Proposition 5.14 *The intrusions of the* J*-open* RCHAM *and those of the open* RCHAM *are related as follows:*

1. $\mathcal{S} \xrightarrow{x_1\langle\widetilde{v}_1\rangle \mid \cdots \mid x_n\langle\widetilde{v}_n\rangle}_J \mathcal{S}'$ *implies* $\mathcal{S} \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} \rightleftharpoons^* \rightarrow \mathcal{S}'$.
2. *For any* A*, if* $A \mid x\langle\widetilde{u}\rangle \rightleftharpoons^* \xrightarrow{M}_J \rightleftharpoons^* B$*, then* *either* $A \rightleftharpoons^* \xrightarrow{M}_J \rightleftharpoons^* A'$ *with* $A' \mid x\langle\widetilde{u}\rangle \rightleftharpoons^* B$*,* *or for some* M' *we have* $M' \rightleftharpoons^* M \mid x\langle\widetilde{u}\rangle$ *and* $A \rightleftharpoons^* \xrightarrow{M'}_J \rightleftharpoons^* B$.

All other transitions are common to both chemical machines.

Proof: 1. The side conditions on names that may appear on intrusion labels are the same for both machines; thus we can use rule INT to intrude each message

$x_i\langle\widetilde{v}_i\rangle$ one at a time. Once this is done, we use structural equivalence to assemble a complete join-pattern, and we perform a RED transition on the reaction rule that is used by the J-open RCHAM to perform the INT-J transition, which yield the same chemical solution.

2. The choice between the two possibilities is directed by the extra internal messages that are consumed by the INT-J transition. We check that if the message in not used, then the first statement hold, and that otherwise it must be part of the extra messages (within M in the definition of Rule INT-J). □

Intrusions and extrusions are naturally dual rules and could be used to define a transition system up to the structural rules of the chemical machine and α-conversion. This approach would suggest the use of "join-extrusions" that aggregate join-patterns on the extrusions labels, as opposites to join-intrusions. Should we adopt multiple extrusions as a proof technique, a special treatment of extrusions would be required because there should be no visible synchronization between components of a multiple extrusion. Also, we would have to deal with labels of arbitrary size, with possibly several messages sent on the same name, or several copies of the same message. As an advantage, this would abstract over the ordering of the extrusions—which is superficially complicated because only the first message that sends a local name outside extrudes this name.

5.3.2 Asynchronous bisimulation

We now try to shape the definition of weak bisimulation (Definition 5.8) to the new J-open RCHAM. Consider for instance the two processes

$$\begin{aligned} P &\stackrel{\text{def}}{=} \mathsf{def}\ x\langle\rangle \triangleright a\langle\rangle \wedge a\langle\rangle \mid y\langle\rangle \triangleright R \ \mathsf{in}\ z\langle x,y\rangle \\ Q &\stackrel{\text{def}}{=} \mathsf{def}\ x\langle\rangle \mid y\langle\rangle \triangleright R \ \mathsf{in}\ z\langle x,y\rangle \end{aligned}$$

(where $a \notin \mathsf{fv}[R]$). Using the underlying model of the open RCHAM, P and Q are weakly bisimilar, but with the new J-open RCHAM this is not true anymore, because after emitting on z, P can input on $x\langle\rangle$ while Q cannot. If we equip the J-open RCHAM with the weak bisimulation of Section 5.2, Q becomes inert after the extrusion on z because join inputs are not considered. But if we consider the weak bisimulation that uses join-input labels instead of single ones, Q can input on $x\langle\rangle|y\langle\rangle$ while P cannot, and P and Q are still separated. It turns out that weak bisimulation discriminates too much in the J-open RCHAM.

In order to avoid detection of the structure of join patterns, weak bisimulation must be weakened further on. At least, we must consider that a process simulates another one even if it does not immediately consume all the messages of a join input.

Next we adapt our definition of weak bisimulation by adding a delay clause, in the spirit of [16].

Definition 5.15 *A relation $\mathcal{R}$ over J-open RCHAMs is a (weak)* asynchronous simulation *if, for all processes A and B such that $A\ \mathcal{R}\ \mathcal{B}$ we have*

1. $\mathsf{xv}[A] = \mathsf{xv}[B]$

2. *if* $A \xrightarrow{M} A'$*, then* $B \,|\, M \rightarrow^* B'$ *with* $A' \mathcal{R} B'$*;*

3. *if* $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} A'$ *and* $S \cap \mathsf{fv}[B] = \emptyset$ *then* $B \rightarrow^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \rightarrow^* B'$ *with* $A' \mathcal{R} B'$*.*

$\mathcal{R}$ *is a* asynchronous bisimulation *when both* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are asynchronous simulations.* Asynchronous bisimilarity $\approx_a$ *is the largest asynchronous bisimulation.*

The relation $\approx_a$ is an equivalence by the general argument of Section 4.4.2. In the definition of asynchronous bisimulation, the restriction on S in the output clause plays the same role as in Definition 5.8. We explicitly require that related solutions have the same extruded names, because otherwise Remark 5.9 would not carry over asynchronous bisimulation. For instance the open deadlocked solution $x\langle\rangle | y\langle\rangle \triangleright P \vdash_{\{y\}}$ would be equivalent to the empty solution $\vdash_{\emptyset}$.

Let us show the equality $P \approx_a Q$, where P and Q are the processes at the beginning of this section. Both P and Q perform the same emission, therefore it suffices to prove that

$$A = \mathsf{def}_{\{x,y\}}\ x\langle\rangle \triangleright a\langle\rangle\ \wedge\ a\langle\rangle | y\langle\rangle \triangleright R \ \mathsf{in}\ 0 \quad \approx_a \quad B = \mathsf{def}_{\{x,y\}}\ x\langle\rangle | y\langle\rangle \triangleright R \ \mathsf{in}\ 0$$

We observe $A \xrightarrow{x\langle\rangle} A'$, while $B \not\xrightarrow{x\langle\rangle}$. Nevertheless, it is possible to prove that $A' \approx_a \mathsf{def}_{\{x,y\}}\ x\langle\rangle \,|\, y\langle\rangle \triangleright R\ \mathsf{in}\ x\langle\rangle$. Conversely, there is B' such that $B \xrightarrow{x\langle\rangle \,|\, y\langle\rangle} B'$ and this move cannot be mimicked by A, but it is possible to prove that $B' \approx_a \mathsf{def}_{\{x,y\}}\ x\langle\rangle \triangleright a\langle\rangle\ \wedge\ a\langle\rangle \mid y\langle\rangle \triangleright R\ \mathsf{in}\ R$, which is a derivative of $A \mid x\langle\rangle \mid y\langle\rangle$, and the process R can be discarded up to context. All other transitions are the same, hence we can conclude that $P \approx_a Q$.

The next lemma establishes some basic congruence property for asynchronous bisimulation.

Lemma 5.16 *$A \approx_a B$ implies $A \,|\, x\langle\widetilde{v}\rangle \approx_a B \,|\, x\langle\widetilde{v}\rangle$.*

Proof: Let $\mathcal{R}$ be the relation defined as follows.

$$\mathcal{R} \quad \stackrel{\text{def}}{=} \quad \approx_a \cup \{(A \,|\, x\langle\widetilde{v}\rangle, B \,|\, x\langle\widetilde{v}\rangle) \text{ such that } A \approx_a B\}$$

We prove that $\mathcal{R}$ is an asynchronous bisimulation. We only consider the case of intrusions, all other cases being immediate. Let $A \,|\, x\langle\widetilde{v}\rangle \xrightarrow{M}_J A'$. By Proposition 5.14, one of the following holds:

1. $A \xrightarrow{M}_J A''$ with $A' = A'' \,|\, x\langle\widetilde{v}\rangle$. If $A \approx_a B$, then we have $B \,|\, M \rightarrow^* B'$ with $A' \approx_a B'$, thus $B \,|\, M \,|\, x\langle\widetilde{v}\rangle \rightarrow^* B' \,|\, x\langle\widetilde{v}\rangle$ with $A'' \,|\, x\langle\widetilde{v}\rangle \ \mathcal{R}\ B' \,|\, x\langle\widetilde{v}\rangle$.

2. $A \xrightarrow{M \,|\, x\langle\widetilde{v}\rangle}_J A'$. If $A \approx_a B$, then we have $B \,|\, M \,|\, x\langle\widetilde{v}\rangle \rightarrow^* B'$ with $A' \approx_a B'$, and in particular $A' \mathcal{R} B'$. $\square$

Our next result states that the two bisimilarities $\approx_l$ over open RCHAMs and $\approx_a$ over J-open RCHAMs coincide:

Theorem 6 $\approx_a = \approx_l$.

Proof: This is a direct consequence of Proposition 5.14. We prove the inclusions $\approx_l \subseteq \approx_a$ and $\approx_a \subseteq \approx_l$; we detail only the intrusions, as all other transitions are identical.

Weak bisimilarity is an asynchronous bisimulation. For every intrusion transition $A \xrightarrow{M} A'$ with $M = x_1\langle\widetilde{v}_1\rangle \mid \ldots \mid x_n\langle\widetilde{v}_n\rangle$ in the J-open machine, we can apply Proposition 5.14(1) and obtain a series of transitions that leads to the same process A' in the open machine:

$$A \quad \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle}\rightarrow \quad A'$$

Assume $A \approx_l B$. By applying weak bisimulation for each intrusion, we obtain a series of transitions that alternates internal moves and the same single intrusions:

$$B \quad \rightarrow^* \xrightarrow{x_1\langle\widetilde{v}_1\rangle}\rightarrow^* \cdots \rightarrow^* \xrightarrow{x_n\langle\widetilde{v}_n\rangle}\rightarrow^* \quad B'$$

In the open machine, however, we can always perform intrusions before internal reductions. After reordering the transitions, we obtain

$$B \quad \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} B \mid M \rightarrow^* \quad B'$$

and thus B' meets all the requirements 5.15*(3.)* in the definition of asynchronous bisimulation.

Asynchronous bisimulation is a weak bisimulation. For every intrusion transition $A \xrightarrow{x\langle\widetilde{v}\rangle} A'$, we have $x \in \mathsf{xv}[\mathcal{S}]$ and $A' = A \mid x\langle\widetilde{v}\rangle$ by definition of Rule INT. In the case $A \approx_a B$, since $\mathsf{xv}[A] = \mathsf{xv}[B]$ we have $x \in \mathsf{xv}[B]$ and $B \xrightarrow{x\langle\widetilde{v}\rangle} B \mid x\langle\widetilde{v}\rangle$. We conclude by Proposition 5.16. □

Remark 5.17 *Another choice for Definition 5.15 would require that every intrusion be matched by a sequence of join-intrusions on the right-hand-side, followed by a parallel composition with the remaining messages, instead of a parallel composition followed by internal reductions. The alternative clause would be:*

3′. if $A \xrightarrow{M} A'$, then for some B, $p \geq 0$, $M_1, \ldots, M_p, M'$, we have $M \rightleftharpoons^* M_1 \mid \cdots \mid M_p \mid M'$, $B \xrightarrow{M_1} \cdots \xrightarrow{M_p} B'$, and $A' \mathcal{R} B' \mid M'$.

This alternate characterization is easily proven equivalent to the previous one, and it is slightly better for automated proofs, because it does not consider unrelated silent moves after join-intrusions. It is necessary to allow several intrusion steps; otherwise the equivalence becomes strictly finer.

5.3.3 Ground bisimulations

As first observed in the π-calculus, asynchrony brings another interesting property as regards the number of reductions to consider: the *ground variants* of bisimulations coincide with the basic ones. In early-style semantics, ground bisimulation is obtained by restricting the intrusions to labels that convey fresh, pairwise-distinct variables. This significantly reduces the size of the models, because these variables can be treated as constants—which explains the name "ground bisimulation". We refer to [16] for a thorough discussion in a π-calculus setting.

This property carries over the join-calculus, thus providing enhanced proof techniques where, for every chemical solution, there is exactly one intrusion message per extruded name when using weak bisimulation, and one intrusion message per active partial join-pattern when using asynchronous bisimulation.

Proposition 5.18 *Let $\approx_{l,g}$ and $\approx_{a,g}$ be the bisimilarities obtained by restricting intrusion labels $x\langle\widetilde{v}\rangle$ in the definition of $\approx_l$ (Definition 5.8) and join-intrusion labels $x_1\langle\widetilde{v_1}\rangle \mid \cdots \mid x_m\langle\widetilde{v_m}\rangle$ in the definition of $\approx_a$ (Definition 5.15) by requiring that all names in $\widetilde{v}$, $\widetilde{v}_i$ be pairwise distinct fresh names. We have:*

$$\approx_{l,g} = \approx_l = \approx_a = \approx_{a,g}$$

Proof: This is an easy corollary of the closure of transitions through global renaming (Lemma 5.6); we prove $\approx_{l,g} = \approx_l$, and we obtain $\approx_{l,g} = \approx_{a,g}$ with the same proof than for Theorem 6. By definition $\approx_l \subseteq \approx_{l,g}$. To establish the converse inclusion, we show that $\approx_{l,g}$ is a weak bisimulation up to renaming.

Let us assume $A \approx_{l,g} B$. The problematic transition is the intrusion $A \xrightarrow{x\langle\widetilde{w}\rangle} A'$, where $\widetilde{w}$ may contain free names and extruded names. The definition of $\approx_{l,g}$ tells nothing about such intrusions. Nonetheless, the label can be written $x\langle\widetilde{v}\rho\rangle$ where the v_i's are distinct, fresh names. The intrusion labeled $x\langle\widetilde{v}\rangle$ is enabled in A; by applying ground bisimulation, we obtain a series of ground transitions $B \rightarrow^* \xrightarrow{x\langle\widetilde{v}\rangle} \rightarrow^* B'$ with $A'' \approx_{l,g} B'$ and $A' = A''\rho$. All these transitions commute with the substitution ρ, so we also have $B\rho \rightarrow^* \xrightarrow{x\langle\widetilde{w}\rangle} \rightarrow^* B'\rho$. We conclude by Lemma 5.10(4). $\square$

Remark 5.19 *Another variant of bisimulation named* branching bisimulation *has been advocated for process calculi [62], mostly because it yields more efficient partition-refinement algorithms for automated verification. Branching bisimulation allows additional reduction steps in bisimulation diagrams only inasmuch as these steps preserve the bisimulation. In the join-calculus, branching bisimulation and weak bisimulation coincide.*

5.4 Reduction-based equivalences on open terms

Intuitively, the extension of chemical machines with intrusion and extrusion should not affect purely reduction-based equivalences. Indeed, our definitions and results of Chapter 4 easily carries over to the open join-calculus. We briefly discuss the interest of extruded names for dealing with reduction-based equivalences, and establish that open processes enable simple statements and proofs, but do not provide genuinely new reduction-based equivalences.

5.4.1 Observation

As can be expected, the basic observation predicates $\downarrow_x$, and thus all the derived notions of observation can easily be defined in terms of extrusion labels:

Remark 5.20 (labels versus barbs) *For every open process A, for every name x that is not extruded by A, we have*

$$\begin{array}{lll} A \downarrow_x & \textit{iff} & \exists \widetilde{v}, \exists S \textit{ such that } A \rightleftharpoons^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \\ A \Downarrow_x & \textit{iff} & \exists \widetilde{v}, \exists S \textit{ such that } A \Rrightarrow^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \end{array}$$

The above remark defines barbs for all free names, but not for extruded names $x \in \mathsf{xv}[A]$. This is consistent with our asynchronous approach, where the intrusion capabilities cannot be detected by the emitter.

5.4.2 Extruded names and congruence properties

As regards the congruence property, the well-formed conditions on open terms constrain the application of contexts. Especially, evaluation contexts cannot bind extruded names. This may be counter-intuitive, as the mere possibility of applying a context seems to reveal the extruded names of an open process, while reduction-based congruences remain entirely insensitive to it. As we try to extend a given reduction-based equivalence Φ for all processes of the open calculus, we therefore have two choices: we can either entirely forget about extruded names, or also require their exact correspondence for every pair of related open processes (i.e., that Φ refine the partition induced by $\mathsf{xv}[\cdot]$). This is a minor difference, because the extruded names cannot vary through reductions. For instance, the technical difference appears for equations of the form

$$x\langle\rangle \quad \Phi \quad \mathsf{def}_{\{x\}}\, x\langle\rangle \triangleright 0 \;\mathsf{in}\; 0$$

We adopt the second, stricter point of view and always assume in the following that related processes have the same extrusion interface, because it makes little sense to compare open processes with different sets of extruded names, and because it is technically convenient in order to compare reduction-based equivalences and labeled equivalences. (Alternatively, we could use a coarser intrusion rule that tolerates intrusions on fresh names.)

Despite this subtlety, the congruence requirements in the open join-calculus can be significantly simplified by structural rearrangements. More precisely,

Remark 5.21 *For any process of the form $C[A]$ where $C[\cdot]$ is an open evaluation context, there is an open process B such that*

$$(B \mid A) \setminus S \quad \rightleftharpoons^* \quad C[A]$$

Remark 5.22 *Let Φ be a reduction-based equivalence that refines structural congruence ($\rightleftharpoons^* \subseteq \Phi$). For any open processes A and B, for any set of names S such that $S \subseteq \mathsf{xv}[A] \cap \mathsf{xv}[B]$, we have*

$$A \,\Phi\, B \quad \textit{implies} \quad A\backslash S \,\Phi\, B\backslash S$$

Hence, in the open join-calculus the parallel composition accounts for every evaluation context, and we can simplify our congruence requirements accordingly. Going in the other direction, quantification over open evaluation contexts and quantification over evaluation contexts with no extruded names yield the same congruence.

5.4.3 Plug-in's versus extruded names

Besides enabling the use of labeled equivalences, open processes provide an useful notation to deal with reduction-based equivalences; in particular, it is convenient to specify some parts of a protocol with extruded names. Then, simple parallel composition with a fixed interface can be used instead of using a single flat process—or fully-diluted chemical soup—with numerous side conditions that enforce the conditions induced by the interface.

In the absence of extruded names, the same approach actually applies, except that the interface must be explicitly encoded by additional, initialization messages sent on conventional names. We call such messages *plug-in's*, as they are meant to establish the connections between the processes and their environment before the actual computation begins.

The next lemma relates different formulations of reduction-based equivalence relations. It shows that, in some sense, plug-in messages are the poor man's extruded names.

Lemma 5.23 *Let Φ be one of the relations $\sim$, $\succeq$, and $\approx$. Let X be a set of names, $\widetilde{X}$ be a set of tuples whose contents partition the names in X, $\{plug_{\widetilde{x}} \mid \widetilde{x} \in \widetilde{X}\}$ be a family of names indexed by $\widetilde{X}$, and $P_{\widetilde{X}}$ abbreviate the process $\prod_{\widetilde{x}\in\widetilde{X}} plug_{\widetilde{x}}\langle\widetilde{x}\rangle$.*

For all processes P_1, P_2 and definitions D_1, D_2 where none of the names $plug_{\widetilde{x}}$ occur free and such that $X \subseteq \mathsf{dv}[D_1] \cap \mathsf{dv}[D_2]$, the three following statements are equivalent:

1. $\mathsf{def}_X\ D_1\ \mathsf{in}\ P_1 \ \ \Phi\ \ \mathsf{def}_X\ D_2\ \mathsf{in}\ P_2$
2. $\mathsf{def}\ D_1\ \mathsf{in}\ P_1 \mid P_{\widetilde{X}} \ \ \Phi\ \ \mathsf{def}\ D_2\ \mathsf{in}\ P_2 \mid P_{\widetilde{X}}$
3. *for all D and P such that* $\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P] \cap (\mathsf{dv}[D_1] \cup \mathsf{dv}[D_2]) \subseteq X$ *and* $\mathsf{dv}[D] \cap (\mathsf{dv}[D_1] \cup \mathsf{dv}[D_2]) = \emptyset$,

$$\mathsf{def}\ D \wedge D_1\ \mathsf{in}\ P \mid P_1 \quad \Phi \quad \mathsf{def}\ D \wedge D_2\ \mathsf{in}\ P \mid P_2$$

The first formulation is the most compact; it relies on open terms. The second formulation makes explicit the communication of extruded names to the environment; the third formulation is stable by application of evaluation contexts, and is used in direct proofs of reduction-based equivalences that are congruences. We refer to [2] for a detailed application of the last two formulations of the lemma, in a slightly extended calculus.

Proof: *(1) implies (2):* We obtain (2) from (1) by congruence property for the context $P_{\widetilde{X}} \mid [\cdot]$, followed by structural rearrangement and restriction (Remark 5.22).

(3) implies (2): We obtain (2) as an instance of (3) by letting $P = P_{\widetilde{X}}$ and $D = \mathsf{T}$.

(2) implies (3): Since Φ is a congruence for all evaluation contexts, the third statement only makes explicit a part of a context.

In case Φ is $\approx$, we use the congruence property of $\mathcal{R}$ with the particular context:

$$C[\cdot] \ \stackrel{\text{def}}{=}\ \mathsf{def}\ \begin{array}{ll} & z\langle\widetilde{x}\rangle \triangleright P \\ \wedge & D + \rho\langle\widetilde{x}\rangle \end{array}\ \mathsf{in}\ \mathsf{def}\ P_{\widetilde{X}} \triangleright z\langle\widetilde{x}\rangle \mid \rho\langle\widetilde{x}\rangle\ \mathsf{in}\ [\cdot]$$

where ρ and z are fresh names, where $\widetilde{x}$ is a single tuple that carries all the names in X, and where $D + \rho\langle\widetilde{x}\rangle$ is inductively defined as follows:

$$\begin{aligned}
(D \wedge D') + \rho\langle\widetilde{x}\rangle &\overset{\text{def}}{=} (D + \rho\langle\widetilde{x}\rangle) \wedge (D' + \rho\langle\widetilde{x}\rangle) \\
(J \triangleright P) + \rho\langle\widetilde{x}\rangle &\overset{\text{def}}{=} J \,|\, \rho\langle\widetilde{x}\rangle \;\triangleright\; P \,|\, \rho\langle\widetilde{x}\rangle \\
D + \rho\langle\widetilde{x}\rangle &\overset{\text{def}}{=} D \qquad\qquad \text{in all other cases}
\end{aligned}$$

For all P, P_i, D, and D_i that meet the requirements of the lemma, we have the relations

$$\begin{aligned}
C\left[\mathsf{def}\ D_i\ \mathsf{in}\ P_i \,|\, P_{\widetilde{X}}\right] &\equiv_\alpha C'\left[\mathsf{def}\ D_i\ \mathsf{in}\ P_i \,|\, P'_{\widetilde{X}}\right] && (5.8)\\
&\succeq C'\left[\mathsf{def}\ D_i\ \mathsf{in}\ P \,|\, P_i \,|\, \rho\langle\widetilde{x}\rangle\right] && (5.9)\\
&\sim \mathsf{def}\ D + \rho\langle\widetilde{x}\rangle \wedge D_i\ \mathsf{in}\ P \,|\, P_i \,|\, \rho\langle\widetilde{x}\rangle && (5.10)\\
&\sim \mathsf{def}\ D \wedge D_i\ \mathsf{in}\ P \,|\, P_i && (5.11)
\end{aligned}$$

The structural equivalence (5.8) performs α-conversion; C', $P'_{\widetilde{X}}$ are C, $P_{\widetilde{X}}$ after substituting names that do not occur in D,D_i,P,P_i for the names $plug_{\widetilde{x}}$ with $\widetilde{x} \in \widetilde{X}$. Next the expansion relation (5.9) is obtained by two successive reduction steps that use the rule $P'_{\widetilde{X}} \triangleright z\langle\widetilde{x}\rangle \,|\, \rho\langle\widetilde{x}\rangle$ then the rule $z\langle\widetilde{x}\rangle \triangleright P$. The expansion relation holds because these two reductions are deterministic—they are enabled until performed, no matter of any other reduction. Afterward, the two reaction rules are inert; they can be discarded up to strong equivalence (5.10). The strong equivalence (5.11) is easily established: the message $\rho\langle\widetilde{x}\rangle$ always remains present, a join-pattern in the definition $D + \rho\langle\widetilde{x}\rangle$ is matched if and only if the corresponding join-pattern in D is matched, and all other reductions are unaffected.

By applying $C[\,\cdot\,]$ to statement (2) of the lemma, we have

$$C\left[\mathsf{def}\ D_1\ \mathsf{in}\ P_1 \,|\, P_{\widetilde{X}}\right] \quad\approx\quad C\left[\mathsf{def}\ D_2\ \mathsf{in}\ P_2 \,|\, P_{\widetilde{X}}\right]$$

We apply twice the series of relations 5.8–5.11 and, since all these relations are finer than $\approx$, we conclude by transitivity of $\approx$.

In case $\mathcal{R}$ is $\sim$ or $\succeq$, we apply the same method with the refined context:

$$C[\,\cdot\,] \overset{\text{def}}{=} \mathsf{def}\ \begin{array}{rl} & z\langle\widetilde{x}\rangle \triangleright P \\ \wedge & z\langle\widetilde{x}\rangle \triangleright t\langle\rangle \\ \wedge & D + \rho\langle\widetilde{x}\rangle \end{array}\ \mathsf{in}\ \mathsf{def}\ P_{\widetilde{X}} \triangleright z\langle\widetilde{x}\rangle \,|\, \rho\langle\widetilde{x}\rangle\ \mathsf{in}\ [\,\cdot\,]$$

where t, z, and ρ are fresh names, and where $D + \rho\langle\widetilde{x}\rangle$ is defined as before. Again, we apply $C[\,\cdot\,]$ to statement (2) of the lemma; the resulting processes $Q_i \overset{\text{def}}{=} C[\mathsf{def}\ D_i\ \mathsf{in}\ P_i \,|\, P_{\widetilde{X}}]$ behave as before, except that the additional reaction rule $z\langle\widetilde{x}\rangle \triangleright t\langle\rangle$ provides visible, indirect information about the reduction that consumes the message $z\langle\widetilde{x}\rangle$ and triggers P.

As above, each process Q_i may perform two reduction steps that use the first rule of $C[\,\cdot\,]$ and yield a process Q'_i that is strongly bisimilar to $\mathsf{def}\ D \wedge D_i\ \mathsf{in}\ P \,|\, P_i$. Since $Q_1\ \mathcal{R}\ Q_2$, there must be a way for Q_2 to match the two reduction steps $Q_1 \rightarrow\rightarrow Q'_1$ (in either exactly two steps or at most two steps), yielding a process Q''_2 such that $Q'_1\ \mathcal{R}\ Q''_2$. Since $Q_2 \Downarrow_t$ and not $Q'_1 \Downarrow_t$, the only possible way is with the reductions

$Q_2 \rightarrow\rightarrow Q'_2$, so $Q''_2 \equiv Q'_2$ and $Q'_1 \mathcal{R} Q'_2$. Therefore, $\mathsf{def}\ D \wedge D_1\ \mathsf{in}\ P \,|\, P_1 \ \sim\mathcal{R}\sim\ \mathsf{def}\ D \wedge D_2\ \mathsf{in}\ P \,|\, P_2$, and hence $\mathsf{def}\ D \wedge D_1\ \mathsf{in}\ P \,|\, P_1\ \mathcal{R}\ \mathsf{def}\ D \wedge D_2\ \mathsf{in}\ P \,|\, P_2$.

(3) and (2) implies (1): We prove that the relation that contains all processes related by (1) when the two other properties hold is a Φ-bisimulation. Since (3) and (2) are equivalent, we only need to prove that one of them is preserved by reduction and by congruence. By using (2), this relation is a bisimulation and respects all barbs. By using (3), this relation meets the congruence property for enough evaluation contexts: for every evaluation context $\mathsf{def}\ D\ \mathsf{in}\ P \,|\, [\cdot]$ that can be applied to (1), the first requirement of (3) holds after α-conversion on the names defined in D_1 and D_2 and not extruded in X; the second requirement is guaranteed by the locality property of extruded names. □

5.4.4 Weak bisimulation versus barbed congruence

In calculi where both reduction-based equivalences and labeled-based equivalences are defined, the problem of their coincidence naturally arise. Here, we focus on the relation between barbed congruence and weak bisimulation.

Provided that (1) labeled bisimulation is a congruence for all contexts under consideration—usually "static contexts", or evaluation contexts in our case—and that (2) the barbs $\Downarrow_x$ can be deduced from the labeled transitions, we obtain the two well-known inclusions $\approx_l \subseteq \approx$ and $\approx_l \subseteq \dot{\approx}^\circ$. In the asynchronous π-calculus, for instance, we have $P \downarrow_x$ if and only if $P \xrightarrow{\overline{x}\langle\rangle}$, and the congruence for all evaluation contexts is established in [16].

In our setting, by combining the requirement on reductions and extrusions in the definition of labeled bisimulation (Definition 5.8) with the reformulation of weak barbs (Remark 5.20), we obtain that weak bisimulation is a barbed bisimulation, and thus the inclusion $\approx_l \subseteq \dot{\approx}$. Moreover, weak bisimulation is also a congruence. Therefore we have the expected inclusion

Remark 5.24 *In the join-calculus, $\approx_l \subset \approx$.*

This remark is technically important, because proofs of labeled bisimulations are easier to conduct than proofs of barbed congruence, where the closure under every possible evaluation context must be explicitly handled in each co-inductive proof. Unfortunately, the gap between the two equivalences is pragmatically quite large. To see that the inclusion is strict, it suffices to consider the paradigmatic example of barbed congruence in the join-calculus:

Remark 5.25 (Silent relay)

$$x\langle z\rangle \ \approx_b\ \mathsf{def}\ u\langle v\rangle \triangleright z\langle v\rangle\ \mathsf{in}\ x\langle u\rangle$$

That is, emitting a free name z is the same as emitting a bound name u that forwards all the messages it receives to z; the two processes are indistinguishable, despite an extra internal move for every use of z. However, these two processes are distinguished by weak bisimulation because their respective extrusion labels reveal that z is free and u is extruded.

5.5 The discriminating power of name comparison

Our comparison with barbed congruence shows that weak bisimulation is more discriminating than barbed congruence because of name comparison. Nevertheless, barbed congruence can easily be equipped with such a discriminating power. In this section, we attempt to reconcile the two equivalences by enriching the class of evaluation contexts that are considered in barbed congruence.

5.5.1 Should the join-calculus provide name comparison?

So far, we carefully avoided the use of any join-calculus construct that would enable the direct comparison of names. While such name-testing capabilities are commonly found in process calculi, we believe that it is a mixed blessing, at least in our setting.

The comparison of names is a standard trick to obtain context-based characterizations of labeled-based equivalences, but it also induces technical complications. For instance, most equivalences are not closed anymore by global renaming. Also, name comparison unveils subtle distinctions in the definition of equivalences, such as, concerning bisimulations in the π-calculus, the early-, late-, ground- and open-variants [134].

From the programmer's point of view, comparison is a common feature for basic values such as integers, but it is seldom available for channels, functions, and abstract types in general, probably because it is delicate to implement and often reveals too much about the implementation, thus hindering common optimizations. When the comparison of functions is available, for instance, it does not usually correspond to the comparison of function names in the source program, but more likely to the identity of their run-time representations.

5.5.2 The join-calculus with name-matching

We extend the syntax of the join-calculus with a new construct equivalent to the standard name-matching prefix of [100].

$$A \stackrel{\text{def}}{=} \ldots | \mathsf{if}\ x = y\ \mathsf{then}\ A \qquad\qquad P \stackrel{\text{def}}{=} \ldots | \mathsf{if}\ x = y\ \mathsf{then}\ P$$

Accordingly, we extend our chemical machines with a new reduction rule.

$$\textsc{Match} \qquad \vdash \mathsf{if}\ x = x\ \mathsf{then}\ A \quad \longrightarrow \quad \vdash A$$

The comparison prefix is of course not an evaluation context. (Alternatively, we could introduce the comparison of names by removing the linearity constraint on received names in join-patterns. While this equivalent approach would induce minimal changes, we choose instead to emphasize the presence of comparison with an explicit construct.)

We define some syntactic sugar for conjunctions of tests as follows. For a finite set of pairs $S = \{(x_i, y_i) \mid i = 1 \ldots n\}$ we let

$$\mathsf{if} \bigwedge_{(x,y)\in S} x = y\ \mathsf{then}\ P \quad \stackrel{\text{def}}{=} \quad \mathsf{if}\ x_1 = y_1\ \mathsf{then}\ \mathsf{if}\ x_2 = y_2\ \mathsf{then} \ldots \mathsf{if}\ x_n = y_n\ \mathsf{then}\ P$$

As expected, global renamings don't preserve weak bisimulation anymore: Corollary 5.11 is false in the extended calculus. For instance we have $\mathsf{0} \approx_l \mathsf{if}\ x = y\ \mathsf{then}\ x\langle\rangle$

while after applying the renaming $\{^x/_y\}$, $\mathsf{0} \not\approx_l \mathsf{if}\ x = x\ \mathsf{then}\ x\langle\rangle$. Accordingly, bisimulation is not a full congruence anymore; it is easy to check that the context $C[\cdot] \overset{\text{def}}{=} \mathsf{def}\ z\langle x,y\rangle \triangleright [\cdot]\ \mathsf{in}\ z\langle u,u\rangle$ separates the two processes above.

The next lemma expresses that weak bisimilarity retains the congruence property for all evaluation contexts:

Lemma 5.26 *In the presence of name matching, weak bisimilarity is a congruence for all evaluation contexts ($\approx_l = \approx_l{}^\circ$).*

Proof: the proof of Lemma 5.13 applies unchanged. $\square$

Weak bisimulation is not a congruence for all contexts, however. In order to obtain the full congruence property, we can consider the coarsest equivalence contained into $\approx_l$ and closed under renaming. By applying Lemma 5.12, we obtain that this equivalence is a full congruence. However, this equivalence cannot be a bisimulation.

5.5.3 Barbed congruence is a weak bisimulation

We resume our comparison between weak bisimulation and barbed congruence in the calculus extended with name comparison. Observe that barbed congruence can now use a comparison in the context $C[\cdot] = \mathsf{def}\ x\langle y\rangle \triangleright \mathsf{if}\ y = z\ \mathsf{then}\ a\langle\rangle\ \mathsf{in}\ [\cdot]$ to separate the processes $x\langle z\rangle$ and $\mathsf{def}\ u\langle v\rangle \triangleright z\langle v\rangle\ \mathsf{in}\ x\langle u\rangle$.

More generally, with name comparison in the syntax each particular label can be tested by a particular context that performs a series of comparisons, and thus one would expect that barbed congruence coincides with (some variant of) weak labeled bisimulation. Still, this coincidence is far from obvious; this is left as an open problem by Milner and Sangiorgi in a similar setting [101]:

In the π-calculus with matching, early bisimulation and barbed congruence ($\dot{\approx}^\circ$) do coincide, but all direct proofs are difficult. To our knowledge, the only proof of this result appears in Sangiorgi's dissertation, for both CCS and the monadic π-calculus [130]; the technique consists of building contexts that test all possible behaviors of a process under bisimulation, and exhibit different barbs accordingly; this is problematic because such contexts are infinite: in particular, they have infinite numbers of free names and of recursive constants. Arguably, these contexts correspond to very demanding tests. Such contexts are otherwise never considered in congruence requirements, and cannot be expressed using constructs such as guarded replication instead of parameterized recursive constants.

In other works, approximate results are obtained for variants of the calculus (the asynchronous π-calculus in [16], the open join-calculus in [35]). The proof technique is similar, but it does not rely on this heavy machinery, which is not available anymore in these variants. This has a cost, though, as the coincidence of barbed congruence with labeled bisimulation is established only for pairs of processes that are *image finite*, and can thus be tested using only finite contexts.

A process is image finite when it has a finite number of derivatives for every labeled transition (for each label α, $\{P' \mid P \xrightarrow{\alpha} P'\}$ is finite). This restriction is reasonable for strong bisimulations because transition systems are usually not infinite branching, but it is more questionable for weak bisimulations, where in particular $\{P' \mid P \rightarrow^* P'\}$ has to be finite too. Many common diverging processes do not have this property.

For instance, a process that makes use of replication is usually not image-finite (e.g., $!\tau.P \rightarrow^* \equiv\ !\tau.P \,|\, \tau.P \,|\, \ldots \,|\, \tau.P$). In the join-calculus, where inputs are all implicitly replicated, the problem is all the more serious.

We may adapt Sangiorgi's proof by replacing all the messages on free variables by a few messages that encode all barbs as integers, in the spirit of the proofs in Section 4.8. In combination with Lemma 4.33, this should probably yield a finite encoding of his infinite contexts. Actually, there is a much simpler proof at hand. In Section 4.4.3 we presented two notions of barbed congruence noted $\approx$ and $\dot{\approx}^{\circ}$, and devoted some efforts to the proof that they actually coincide (Theorem 3) with or without name-testing. No matter of the choice between $\approx$ and $\dot{\approx}^{\circ}$ as the "main" barbed congruence, this suggests an alternative, bisimulation-and-congruence proof that focuses on the missing inclusion $\approx \subseteq \approx_l$.

Our next result states that with the comparison of names, barbed congruence and labeled bisimulation yield the same equivalence.

Theorem 7 *In the open join-calculus with comparison, we have $\approx = \approx_l$.*

The following proof is relatively simple because we can apply a new evaluation context before every reduction step. Hence, it suffices to have a context that "grabs" one label, then disappears up to barbed congruence. The corresponding π-calculus lemmas appear in [56], and it seems that our technique can be adapted to other calculi—the existence of tests that discriminate every single label is necessary anyway. The inclusion $\approx \subseteq \approx_l$ is already stated and proved by Honda and Yoshida for the ν-calculus—a variant of the asynchronous π-calculus [73]. Their proof is more intricate than ours because remnants of applied contexts always remain present.

The problematic case is extrusion, because a context of the join-calculus must define a name in order to detect a label that emits on that name; this case is handled by creating a permanent relay for all other messages on that name. Without additional care, however, this relay can be detected through the comparison of names.

To tackle this problem, we use a family of contexts that separates two aspects of a name. For every name $x \in \mathcal{N}$, we let

$$R_x[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ x\langle\widetilde{y}\rangle \triangleright x'\langle\widetilde{y}\rangle\ \mathsf{in}\ v_x\langle x\rangle \,\|\, [\,\cdot\,]$$

where the length of $\widetilde{y}$ matches the arity of x. For every name x that is free in P, $R_x[P]$ uses x' as a free name instead of x, and forwards all messages from x to x'. The context must still be able to discriminate whether the process sends the name x or not; this extra capability is supplied by an auxiliary message on v_x.

Informally, the contexts R_x above will be the residuals of contexts that test for labels such as $\{x\}\overline{y}\langle x\rangle$ where the name x is extruded. The next lemma captures the essential property of $R_x[\,\cdot\,]$:

Lemma 5.27 (Accommodating the extrusions) *In the join-calculus with or without name-matching, for all processes P, Q such that $x', v_x \notin \mathsf{fv}[P] \cup \mathsf{fv}[Q]$, we have*

$$P \approx Q \quad \textit{iff} \quad R_x[P] \approx R_x[Q]$$

Proof: If $P \approx Q$, then the congruence property of $\approx$ yields $R_x[P] \approx R_x[Q]$.

To prove the converse implication, we let $\mathcal{R}$ be the relation that contains all pairs of processes (P', Q') obtained from the pairs (P, Q) that meets the conditions of the lemma by replacing every message $x\langle\widetilde{y}\rangle$ in evaluation context by the message $x'\langle\widetilde{y}\rangle$. Since $P \rightarrow^* P'$ and $Q \rightarrow^* Q'$ use only deterministic reductions, we still have $R_x[P'] \approx R_x[Q']$.

We prove that $\mathcal{R}$ is a barbed bisimulation congruence up to the expansion that forwards messages from x to x' in $R_x[\,\cdot\,]$.

1. $\mathcal{R}$ is a weak bisimulation for reductions up to expansion; if $P' \rightarrow P''$, then $R_x[P'] \rightarrow R_x[P'']$ and, since we have $R_x[P'] \approx R_x[Q']$, this reduction is simulated by some reductions $R_x[Q'] \rightarrow^* T$. By definition of $R_x[\,\cdot\,]$, T is of the form $R_x[Q'']$ where $Q' \rightarrow^* Q''$. The resulting processes P'' and Q'' may have messages on x in evaluation context, but still we have $P'' \succeq\mathcal{R}\preceq Q''$.

2. $\mathcal{R}$ refines all barbs: for all processes P', the barbs of $R_x[P']$ and those of P' are closely related: we have $R_x[P] \not\Downarrow_x$, $R_x[P] \Downarrow_{v_x}$, $R_x[P] \Downarrow_{x'}$ iff $P \Downarrow_x$ or $P \Downarrow_{x'}$, and for all $y \notin \{x, x', v_x\}$, $R_x[P] \Downarrow_y$ iff $P \Downarrow_y$. If $P' \;\mathcal{R}\; Q'$, then $R_x[P']$ and $R_x[Q']$ have the same barbs, and thus P' and Q' have the same barbs.

3. $\mathcal{R}$ is a congruence: let $C[\,\cdot\,]$ be an evaluation context. We show that

$$R_x[P] \approx R_x[Q] \quad \text{implies} \quad R_x[C[P]] \approx R_x[C[Q]]$$

   by translating $C[\,\cdot\,]$ to another context $[\![C]\!][\,\cdot\,]$ that binds v_x, receives x on v_x, uses x everywhere except for the definition of x', and re-applies R_x on the outside: we assume that $C[\,\cdot\,] \equiv \mathsf{def}\ D\ \mathsf{in}\ M \,|\, [\,\cdot\,]$, and we distinguish two cases according to the scope of x:

   - in case C binds x ($x \in \mathsf{dv}[D]$), we use the translation

$$[\![C]\!][\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad R_x[0] \,|\, \mathsf{def}\ D' + \rho\langle x\rangle \wedge v_x\langle x\rangle \triangleright \rho\langle x\rangle \,|\, M\ \mathsf{in}\ [\,\cdot\,]$$

     where D' is D with x' instead of x in every top-level join-pattern.

   - otherwise, we use the translation

$$[\![C]\!][\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} \mathsf{def}\ w_x\langle\widetilde{y}\rangle \triangleright v_x\langle\widetilde{y}\rangle\ \mathsf{in} \\ \mathsf{def}\ D + \rho\langle x\rangle \wedge v_x\langle x\rangle \triangleright \rho\langle x\rangle \,|\, w_x\langle x\rangle \,|\, M\ \mathsf{in}\ [\,\cdot\,] \end{array}$$

   (where $D + \rho\langle x\rangle$ is defined as in the proof of Lemma 5.23). In each case, we establish the labeled expansion $[\![C]\!][R_x[P]] \succeq_l R_x[C[P]]$. To conclude, we apply the congruence property for $[\![C]\!][\,\cdot\,]$ to the hypothesis $R_x[P] \approx R_x[Q]$, and obtain $C[P] \;\mathcal{R}\; C[Q]$ by transitivity. □

Proof of Theorem 7: The inclusion $\approx_l \subseteq \approx$ is a corollary of Lemma 5.26; its proof does not depend on name comparison, and is the same as in Section 5.4.4.

We prove the converse inclusion $\approx \subseteq \approx_l$ by establishing that $\approx$ is a labeled bisimulation. Let us assume that $A \approx B$. For every kind of transition $A \xrightarrow{\alpha} A'$ we use a context that specifically consumes this transition, then behaves as the trivial context $[\,\cdot\,]$ possibly using $R_x[\,\cdot\,]$ and the above lemma.

internal step $\rightarrow$ This is subsumed by the weak bisimulation property of $\approx$.

intrusion $\xrightarrow{x\langle\widetilde{y}\rangle}$ Independently of the values $\widetilde{y}$, intrusion is enabled on x iff $x \in \mathsf{xv}[A]$, and then we have $A \xrightarrow{x\langle\widetilde{y}\rangle} A' = A \mid x\langle\widetilde{y}\rangle$. We simply use the congruence property of $\approx$ for the context $[\,\cdot\,] \mid x\langle\widetilde{y}\rangle$.

extrusion $\xrightarrow{S\overline{x}\langle y_1,\ldots,y_n\rangle}$ Let $m \in 0 \ldots n$ be the cardinal of S. Without loss of generality, we assume that the freshly extruded names in S are the first arguments of the message ($S = \{y_1, \ldots, y_m\}$). We also assume that $S \cap (\mathsf{fv}[A] \cup \mathsf{fv}[B]) = \emptyset$. We use the congruence property for the context

$$\begin{array}{rcl} T & \stackrel{\text{def}}{=} & \{(y_i, z_i) \mid m < i \leq n\} \\ & \cup & \{(z_i, z_j) \mid 0 \leq i < j \leq m \text{ and } y_i = y_j\} \\ F & \stackrel{\text{def}}{=} & (\mathsf{fv}[A] \cup \mathsf{fv}[B] \cup \mathsf{xv}[A] \cup \mathsf{xv}[B]) \times S \\ & \cup & \{(z_i, z_j) \mid 0 \leq i, j \leq m \text{ and } y_i \neq y_j\} \end{array}$$

$$V \stackrel{\text{def}}{=} \mathsf{if} \bigwedge_{(y,z)\in T} y = z \;\mathsf{then}\; \mathit{done}\langle\rangle \mid \prod_{(y,z)\in F} (\mathsf{if}\; y = z \;\mathsf{then}\; \mathit{test}\langle\rangle)$$

$$E[\,\cdot\,] \stackrel{\text{def}}{=} \begin{array}{ll} \mathsf{def} & x\langle\widetilde{z}\rangle \mid \mathit{grab}\langle\rangle \triangleright V \\ \wedge & x\langle\widetilde{z}\rangle \mid \mathit{done}\langle\rangle \triangleright x'\langle\widetilde{z}\rangle \mid \mathit{done}\langle\rangle \\ \wedge & \mathit{once}\langle\rangle \triangleright \mathit{test}\langle\rangle \\ \wedge & \mathit{once}\langle\rangle \mid \mathit{done}\langle\rangle \triangleright \mathit{done}\langle\rangle \\ \mathsf{in} & \mathit{once}\langle\rangle \mid \mathit{grab}\langle\rangle \mid v_x\langle x\rangle \mid [\,\cdot\,] \end{array}$$

where the names grab, once, done, test, and v_x are fresh. Informally, $E[\,\cdot\,]$ grabs a message on x, binds its arguments to the variables $\widetilde{z}$, checks that these arguments meet all the requirements of the expected label using the process V, then behaves like $R_x[\,\cdot\,]$. The finite set $T \subset \mathcal{N} \times \mathcal{N}$ gathers all pairs of names that must coincide: previously-known names and repeated fresh names. The finite set $F \subset \mathcal{N} \times \mathcal{N}$ gathers all pairs of names that must be different from one another: freshly-extruded names are distinct from any previously visible name, and pairwise distinct unless syntactically the same on the label.

- if $A \xrightarrow{S\overline{x}\langle y_1,\ldots,y_n\rangle} A'$, then $E[A] \rightarrow^* \sim R_x[A']$.

  We use the series of $n-m+2$ reductions that consumes $x\langle\widetilde{y}\rangle \mid \mathit{grab}\langle\rangle$, passes the series of positive tests T in V—which releases the message $\mathit{done}\langle\rangle$—then consumes $\mathit{once}\langle\rangle \mid \mathit{done}\langle\rangle$—which erases the barb $\Downarrow_{\mathit{test}}$. Using structural rearrangement, the remains of the context can be written

  $$\begin{array}{rcl} E'[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathsf{def}\; x\langle\widetilde{z}\rangle \mid \mathit{done}\langle\rangle \triangleright x'\langle\widetilde{z}\rangle \mid \mathit{done}\langle\rangle \wedge D \;\mathsf{in} \\ & & \mathit{done}\langle\rangle \mid v_x\langle x\rangle \mid [\,\cdot\,] \mid \prod_{(y,z)\in F}(\mathsf{if}\; y = z_i \;\mathsf{then}\; \mathit{test}\langle\rangle) \end{array}$$

  where D and $\prod_{(y,z)\in F}(\mathsf{if}\; y = z_i \;\mathsf{then}\; \mathit{test}\langle\rangle)$ are inert. Thus, we have the strong equivalence $E'[A'] \sim R_x[A']$ for any A' where the names grab, once, done, and v_x are fresh.

- if $E[B] \rightarrow^* U \not\Downarrow_{test}$, then $B \rightarrow^* \xrightarrow{S\overline{x}\langle y_1,\ldots,y_n\rangle} B'$ with $R_x[B'] \approx U$.

  The series of derivation detailed above in the only way to emit the message $done\langle\rangle$, hence to get rid of the barb $\Downarrow_{test}$. Let $x\langle\widetilde{z}\rangle$ be the first message received by $E[\cdot]$. We decompose the reductions leading to U as follows. Before the first reception on x, all reductions are internal to B; after this receptions, all reductions are either internal steps in the derivative of B, internal steps to $E[\cdot]$ exactly as described above, or further receptions on x in $E[\cdot]$. These receptions are deterministic; they preserve barbed congruence.

  Since the barb $\Downarrow_{test}$ disappears, every comparison in the series on the first line of V has succeeded, which ensures $y_i = z_i$ for $i = m{+}1 \ldots n$. Besides, no comparison in the parallel composition on the second line of V may succeed, as this would reintroduce a $test\langle\rangle$ message, hence the names $z_1, \ldots, z_m$ are all fresh names and we can perform α-conversion before the reduction to enforce $y_i = z_i$ for $i = 1 \ldots m$.

  We write $B \rightarrow^* C[x\langle\widetilde{y}\rangle]$ for the series of reductions internal to B, and we choose B' such that $C[x\langle\widetilde{y}\rangle] \xrightarrow{S\overline{x}\langle y_1,\ldots,y_n\rangle} B'$.

Let us assume $A \approx B$ and $A \xrightarrow{S\overline{x}\langle y_1,\ldots,y_n\rangle} A'$. By congruence property $E[A] \approx E[B]$. By weak reduction-based bisimulation, the reductions $E[A] \rightarrow^* E'[A']$ must be simulated by some reductions $E[B] \rightarrow^* U$ with $E'[A'] \approx U$. Since $E'[A] \not\Downarrow_{test}$, $U \not\Downarrow_{test}$ and thus $R_x[A'] \approx R_x[B']$. By Lemma 5.27, this entails $A' \approx B'$, which closes the required bisimulation diagram for extrusions. $\square$

5.6 Related equivalences in the π-calculus

As discussed in Chapter 2, the join-calculus can be seen as a disciplined variant of the asynchronous π-calculus, in which locality is enforced by the syntax. Both calculi have a lot in common, and our labeled semantics largely draw upon the bisimulations developed for the π-calculus [100, 101, 137, 71, 16]. In this section, we relate our definitions of labeled equivalences to previous proposals in the literature, and we compare the equivalences obtained by applying similar definitions to both the join-calculus and the π-calculus. We refer to Section 6.6.1 for a more general comparison of the two calculi, and in particular for the grammar of π-calculus processes that we use here.

The treatment of names in the asynchronous π-calculus and the join-calculus are quite different. In the open join-calculus, the visible names of a process are partitioned into names for extrusions (*free names*), which do not have a local definition, and names for intrusions and internal reduction (*extruded names*), which do have a local definition. This structured interface restricts interaction with the environment, and in particular significantly reduces the number of transitions to consider, both for weak bisimulation and asynchronous bisimulation.

In the π-calculus, on the contrary, every free name can be used for intrusion, extrusion and internal reduction. Furthermore, a received name can be used to create new input guards, as in $x(y).y(z).P$. In this respect, the barbed congruence of Remark 5.25 is illuminating. If we try to translate this equation in the asynchronous π-calculus, we obtain the two processes $\overline{x}\langle z\rangle$ and $\nu u.(!u(v).\overline{z}\langle v\rangle \mid \overline{x}\langle u\rangle)$, but these processes are not barbed equivalent. For example, let $C[\cdot]$ be the evaluation context

$$C[\cdot] \quad = \nu x.\nu z.(x(a).a(u).\overline{u}\langle\rangle \mid \overline{z}\langle b\rangle \mid [\cdot])$$

We have $C[\overline{x}\langle z\rangle] \Downarrow_b$ and $C[\nu u.(!u(v).\overline{z}\langle v\rangle \mid \overline{x}\langle u\rangle)] \not\Downarrow_b$, hence the two translated processes are not even may-testing equivalent. The context $C[\cdot]$ invalidates our locality property, as it receives messages on a received name. As could be expected, locality does restricts the discriminating power of observers.

Let us compare the observational semantics for the open join calculus to previous proposals for the asynchronous π-calculus, notably [71] and [16]. Both proposals adapt the semantics inherited from the standard π-calculus [100] to asynchrony. As message-output is not a prefix anymore, emitters in contexts cannot detect whether a message is actually read. Technically, this leads to a special treatment of input actions, either in the definition of transitions or in the definition of bisimulation.

In [71], Honda and Tokoro retain the standard notion of weak bisimulation. As a consequence, they are forced to change the intensional semantics. Take for instance the two π-calculus processes 0 and $x(u).\overline{x}\langle u\rangle$. The process $x(u).\overline{x}\langle u\rangle$ consumes a message $\overline{x}\langle v\rangle$, then immediately releases the same message; in an asynchronous setting, this operation is invisible from the context. Intuitively, 0 can simulate the same behavior by doing nothing, hence the two processes are barbed congruent. Unfortunately, the standard labeled bisimulation obviously discriminates between 0 and $x(u).\overline{x}\langle u\rangle$. To cope with this problem, Honda and Tokoro adopt an operational model where asynchrony of communication with the environment is rendered as the *total receptiveness* of the process: the intrusion of any message is always enabled, which can be rendered by using extended structural equivalence in combination with the rule:

$$\textsc{Input} \qquad\qquad 0 \xrightarrow{x\langle v\rangle} \overline{x}\langle v\rangle$$

According to this semantics, both processes 0 and $x(u).\overline{x}\langle u\rangle$ can input then output any message, and in particular the message $\overline{x}\langle y\rangle$.

The intrusion rule of the open RCHAM is reminiscent of this kind of operational semantics, with two important differences: (1) Our rule INT is enabled only for extruded names, and there are finitely many of them; conversely, their input rule immediately yields an infinite transition system; and (2) intruded messages are not observable from the join-calculus environment; for instance, if communication on an extruded name becomes stuck, then extraneous messages that have been introduced in the solution by the rule INT can be simply discarded; conversely, π-calculus contexts may attempt to read back their emissions.

In [16], Amadio *et al.* take the opposite approach. They keep the standard synchronous semantics, and they modify the notion of bisimulation. For two processes to be bisimilar, they do not require that every input on one side be necessarily simulated by an input of the other side. Rather, they introduce a *delay* clause in bisimulation, by which the input of a message can be simulated by adding this message in parallel on the other side instead of consuming it immediately. Hence, 0 cannot input a message $\overline{x}\langle y\rangle$, but can simulate this input by putting this message in parallel. Their resulting *asynchronous bisimulation* offers three advantages: it eliminates the need for total receptiveness, it is consistent with external sum, and it relies on a standard labeled semantics.

Our asynchronous bisimulation presented in Section 5.3 relies on similar motivations (handling asynchrony in the definition of bisimulation with a relaxed clause for intrusions rather than in the operational semantics) but our clause for input is different from theirs: we deal with multiple intrusions, and we allow the simulating process to perform arbitrary internal moves after parallel composition with the intruded messages—in their terminology, this would place our equivalence between 1-bisimulation and asynchronous bisimulation. Besides, the J-open RCHAM is not meant to be a standard semantics, but rather a technical device to reduce the branching of the underlying transition system. Arguably, the open RCHAM gives a more intuitive meaning to extruded names.

It is worth mentioning that the clause they use for intrusions in asynchronous bisimulation does not suitably carry over to the join-calculus. In our syntax, this clause would be

$$\begin{array}{llll}\text{if } P \rightleftharpoons^* \xrightarrow{x\langle\widetilde{u}\rangle} \rightleftharpoons^* P' \text{ then} & \textit{either} & Q \Longrightarrow^* \xrightarrow{x\langle\widetilde{u}\rangle} \Longrightarrow^* Q' \text{ and } P' \mathcal{R}\, Q' \\ & \textit{or} & Q \Longrightarrow^* Q' \text{ and } P' \mathcal{R}\, (Q' \mid x\langle\widetilde{u}\rangle)\end{array}$$

If we extend it to our J-open RCHAM model by allowing join-intrusions of several messages, a problem arises when only some of the messages are needed on the right-hand-side to perform internal reductions. Let us consider for instance the equivalence:

$$\begin{array}{ll} & \mathsf{def}_{\{x,y\}}\ x\langle z\rangle \mid y\langle\rangle \triangleright z\langle\rangle \ \mathsf{in}\ x\langle a\rangle \\ \approx_l & \mathsf{def}_{\{x,y\}}\ x\langle z\rangle \mid y'\langle\rangle \triangleright z\langle\rangle \wedge y\langle\rangle \triangleright y'\langle\rangle \ \mathsf{in}\ x\langle a\rangle\end{array}$$

The two processes are weakly bisimilar, but the latter cannot simulate a double intrusion $x\langle b\rangle \mid y\langle\rangle$ that would commit on emitting the message $b\langle\rangle$ and not $a\langle\rangle$.

From a technical point of view, we often discuss the same, standard properties (closure through renaming, congruence, relation with barbed bisimulation); as we compare our work to the formal treatment of bisimulation in the π-calculus, we observe that locality makes most proofs simpler: it forbids the detection of messages carried on defined names, and reduces the number of cases to consider in the interaction between transitions and global renamings. Noticeably, it also rules out the subtle problems of input and output occurring on the same name. Most complications stem from closure of equivalence under global renamings, because renamings can create new redexes. In the synchronous π-calculus, bisimulations are not closed under substitutions, unless it is directly taken as part of the definition [134]; in the asynchronous π-calculus, the property holds because silent steps can always be split in two opposite transitions. In the join-calculus, global renamings may create new reductions only by substituting extruded variables for free variables, which cannot occur for plain (non-open) processes.

Weak bisimulation and asynchronous bisimulation coincide in the open join-calculus for simple reasons. In contrast, the correspondence between the bisimulations of [71] and those of [16], discussed in details in the latter, is a delicate issue; it is unclear, for instance, whether both approaches yield the same relation in the weak case.

More generally, other devices have been proposed to reduce the number of transitions to consider when comparing processes. For example, typed bisimulations can lead to smaller synchronization trees. It may also be possible to dynamically prune the synchronization tree from families of useless transitions according to some conservative property [102].

Chapter 6

Encodings

In this chapter, we complete our study of the join-calculus by a series of encodings. We explore several variants of the join-calculus, and relate them to the asynchronous π-calculus. We obtain a family portrait of asynchronous process calculi centered on the join-calculus presented in Chapter 2, with a detailed account of its specific features. Along the way, we state and prove several correctness properties, which illustrates the equivalences and the proof techniques developed in Chapters 4 and 5.

In concurrency, it is customary to introduce variants of existing calculi— or even to design specific process calculi—instead of relying on other, better established but less adequate formalisms. For the π-calculus, for instance, there is a tremendous number of variations, and their connections are rather loose. Each formal result is stated and proven for a specific calculus, and does not automatically carry over to other variants, even when intuitively these variants should make little difference. For the join-calculus, the same problem arises: while we mostly deal with the calculus defined in Chapter 2, some results are sensitive to small variations, such as the addition of name comparison. Besides, some choices in the design of the "main" join-calculus are arbitrary, so it is worth reconsidering them, and assessing their actual impact on the calculus. Finally, the join-calculus is a variant of the asynchronous π-calculus [71, 37], and their connection deserves a formal investigation.

In Chapter 2, we defined a calculus that is convenient as the kernel of a programming language. Nonetheless, it is possible to reduce it further to simpler primitives. To this end, we successively remove recursive scope, definitions with several clauses, join-patterns with more than two messages, and messages with several transmitted values. We replace them by internal encodings, prove that these encodings are fully abstract, and eventually obtain a "minimalist" core join-calculus stripped from any construct that is not essential, and still retaining the expressive power of the join-calculus.

The join-calculus is closely related to the π-calculus; it can be seen as a constrained π-calculus where scope restriction, input, and replicated input are merged into a single (join) input definition. More precisely, our main theorem states that the asynchronous π-calculus and the join-calculus have the same expressive power up to barbed congruence; it is obtained by exhibiting fully abstract encodings in each direction. However, subtle differences between the two calculi make the correct encodings surprisingly complex. We present both simple and accurate encodings and discuss their characteristics, which give some precise insight on the distance between the join-calculus and the π-calculus.

Efficiency is not a primary concern in these encodings—in our implementation, most of the encoded constructs are directly supported in a more efficient manner. Rather, we are interested in theoretical results expressed as weak equivalences; in particular, some encodings induce a large number of bookkeeping reductions, or even diverging computations.

Contents of the chapter

In Section 6.1 we define some terminology and methods for encodings among process calculi. In Section 6.2 we present the "core" join-calculus that is the target of our internal encodings. In Section 6.3 we study the structure of definitions, we give some equivalences on the shape of join-patterns, and we strip the join-calculus of complex definitions. In Section 6.4 we develop auxiliary "firewall" encodings to protect our translations. In Section 6.5 we further strip the join-calculus from communication on polyadic messages. In Section 6.6, we compare the core join-calculus and the asynchronous π-calculus. The last section is devoted to the proofs of the full abstraction result between the π-calculus and the join-calculus.

6.1 On encodings

A standard manner of understanding a new formalism is to reduce it to more familiar or better established formalisms. Since an explicit correspondence is better than the mere intuition that similar notions are involved, it is natural to design cross-encodings and to study their properties.

From a more detached point of view, this correspondence often points out what are the essential differences between these formalism. Ideally, the best encodings would tell only about these differences, and although it is delicate to argue that there are no better encodings, this approach gives some idea of the distance between the two formalisms.

Even if we restrict our attention to name-passing process calculi, there exists a vast body of literature about encodings. We merely survey a few themes; we refer to [106] for a more detailed overview. (We have also discussed encodings of the λ-calculus and of objects in Sections 3.4 and 3.5, respectively.)

There are numerous examples of *internal encodings* of some operators or features of a given process calculus into a fragment of the calculus that does not contain them. In the π-calculus for instance, in [116] all processes can be encoded into processes with simpler, uniform guards; in [109, 107] guarded choice can be explicitly encoded in a calculus without primitive choice operator.

In the same spirit, the basic communication patterns can be restricted, by replacing the more complex communications by runs of explicit communication protocols. In the π-calculus, for instance, polyadic communication can be divided into series of monadic communications [99, 153]; in the π-I-calculus the name-passing mechanism can be limited to *internal mobility* where only fresh names are communicated [33]; in the asynchronous π-calculus output guards can be ruled out, which renders the reception of a message invisible [37, 71, 114]; in this chapter, and in recent works on the π-calculus [94], the general channel-based communication can be reduced to simpler communications where all receivers are static, and only output capabilities are exchanged.

Last but not least, encodings from a "specification calculus" to an "implementation calculus" yield a precise account of compilation and implementation issues, because full abstraction results can be interpreted in terms of actual guarantees at run-time [1]. For example, an optimization is correct when it preserves an equivalence finer than observation in the implementation. This approach also enables simpler reasoning about implementations by "lifting" properties of interest to the source calculus. In [2] for instance, security in an open distributed environment is specified in terms of join-calculus behaviors, but it is implemented by using classical cryptographic primitives instead of abstract channels; still, full abstraction results guarantee that security in the implementation is the same as security in the source calculus.

6.1.1 Formal properties of translations

While informal translations may provide useful examples, the mere existence of a translation is not very informative, and some notion of correctness is called for.

We set some terminology. An *encoding*, or a *translation*, is a function from terms in the *source calculus*—or high-level language—to terms in the *implementation calculus*—or low-level language. We use variants of the notation $[\![\cdot]\!]$ to denote translation functions.

When the translation is internal to a given calculus equipped with some notion of equivalence, *correctness* with regards to that equivalence is the most natural notion:

Definition 6.1 *Let $(\mathcal{P}, \approx)$ be a process calculus equipped with a relation $\approx$. A translation $[\![\cdot]\!] : \mathcal{P} \mapsto \mathcal{P}$ is* correct up to $\approx$ *when, for all processes $P \in \mathcal{P}$, we have $P \approx [\![P]\!]$.*

When the translation maps processes from one calculus to another, each calculus has its own definition of equivalence. We must therefore combine some coarser direct relations between the source and the translation with indirect, full abstraction properties that transport equivalences from pairs of source processes to pairs of translations.

A key technical lemma expresses some *operational correspondence* between a source process and its translation, possibly up to some equivalence: in essence, it states how the translation works. For instance, a translation preserves source weak reductions up to $\approx$ when for all processes P, if $P \rightarrow^* P'$, then $[\![P]\!] \rightarrow^* \approx [\![P']\!]$; a translation reflects reduction steps in the implementation back to the source calculus up to expansion when for all P, if $[\![P]\!] \rightarrow T$, then for some P' we have $P \rightarrow^= P'$ and $T \succeq [\![P]\!]$.

Other properties state what the translation reveals or hides, as compared with the source process. Especially, they are sensitive to the low-level interactions that are not in operational correspondence with high-level interactions, and vice-versa. We recall the definition of full abstraction:

Definition 6.2 *Let $(\mathcal{P}_1, \approx_1)$ and $(\mathcal{P}_2, \approx_2)$ be two process calculi equipped with relations $\approx_1$ and $\approx_2$. A translation $[\![\cdot]\!] : \mathcal{P}_1 \mapsto \mathcal{P}_2$ is fully abstract when, for all processes $P, Q \in \mathcal{P}_1$, we have*

$$P \approx_1 Q \quad \textit{iff} \quad [\![P]\!] \approx_2 [\![Q]\!]$$

Such properties systematically transport results from a calculus to another, at least for these particular equivalences. As opposed to correctness, however, full abstraction for a given pair of equivalences says nothing about coarser equivalences.

In the whole chapter, we assess the relative expressive power of miscellaneous calculi from the existence of fully abstract encodings between them. We say that $(\mathcal{P}_2, \approx_2)$ is *more expressive* than $(\mathcal{P}_1, \approx_1)$ when there is an fully abstract encoding from $\mathcal{P}_1$ to $\mathcal{P}_2$; we say that the two process calculi $(\mathcal{P}_1, \approx_1)$ and $(\mathcal{P}_2, \approx_2)$ have the *same expressive power* when there is a pair of fully abstract encodings between the two. We typically use two instances of the same notion of equivalence at a given tier in the hierarchy described in Chapter 4. Most full abstraction results are stated in terms of barbed congruence, that is, observation of barbs and internal choices in any context.

Overall, we present for each encoding a combination of such properties on some variations of the encoding, with a trade-off between the complexity of the encoding and the precision of its properties. From that point of view, our hierarchy of equivalences paves the way, and enables a variety of results of graded precision.

6.1.2 Contexts and compositionality

We detail the relation between correctness and full abstraction in the case of an internal encoding, e.g., an encoding $[\![\cdot]\!]$ from a calculus to one of its syntactic fragments.

The issue is complicated when the equivalences are congruences: while correctness assumes quantification over the same contexts for the source process and its translation, full abstraction for two instances of a congruence may use different family of contexts. Using the notation $\approx_0$ for the congruence over the contexts of the fragment, we may have the correctness property (and thus $[\![P]\!] \approx [\![Q]\!]$ implies $P \approx Q$) but still not the full abstraction property ($[\![P]\!] \approx_0 [\![Q]\!]$ implies $P \approx Q$). Also, correctness may not hold simply because the interfaces of P and $[\![P]\!]$ are not the same. When this is the case, it is often useful to relate P to $C[[\![P]\!]]$ for some context $C[\cdot]$ that somehow inverts the interfaces, but even when a translation has an inverse up to equivalence, full abstraction remains weaker than correctness.

Another issue that pops up for several encodings has to do with the congruence property in the low-level calculus. Contexts in the low-level calculus are naturally more discriminative than contexts in the high-level calculus, because translations of contexts in the high-level calculus always comply with the protocols of the low-level interface, while low-level contexts may use this interface in any other way. This suggests that we protect the encoding from “hostile” context, by taking adequate counter-measures in the encoding. In several of our encodings, these techniques yield full abstraction after all, at the cost of some additional complexity in the encoding. By analogy with the security mechanisms that filter messages and force them to comply with a given security policy, we name these extensions of our encodings “firewalls”. We refer to [2] for another instance of this problem, in an actual security setting, and postpone to future work a general study of firewall techniques and their interaction with type systems.

6.2 The core join-calculus

The core (recursive) join-calculus is a restriction of the full calculus with simpler definitions, join patterns, and messages. Its syntax is given by the grammar:

P ::=		core processes
	$x\langle u\rangle$	message carrying a single name
$\mid$	$P_1 \mid P_2$	parallel composition
$\mid$	$\mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P_1\ \mathsf{in}\ P_2$	local definition of a single two-way rule

If we use the scoping rules of the join-calculus, the scope of u, v is P_1, whereas the scope of x, y extends to the whole definition (P_1 and P_2). Alternatively, we can define a non-recursive core join-calculus where the scope of x, y only extends to the main process P_2. Also, open variants of the join-calculus are easily adapted to the core join-calculus by opening the syntax of restricted definitions.

The core calculus has much simpler definitions than the full calculus; each definition has a single clause that joins exactly two names. Also, the core calculus is *monadic*, in the sense that every message carries exactly one name. More generally, the monadic variant of any join-calculus is the subset of this calculus where all names have the unique, recursive type $\mu\alpha.\langle\alpha\rangle$.

In combination, the following results state that the core join-calculus has the same expressive power as the full join-calculus. In particular, there is a fully abstract encoding $[\![\ \cdot\ ]\!]_0$ from the full calculus to the core calculus.

Theorem 8 *The core join-calculus retains the expressive power of the join-calculus up to both barbed congruence and labeled bisimulation.*

Proof: Anticipating on the next sections, we compose full abstraction results on successive encodings to get rid of redundant features. The simplification is organized as follows:

1. we first remove recursion by Lemma 6.3;
2. then we remove complex definitions by Lemma 6.6;
3. we remove communication on polyadic messages either by Lemma 6.15 for $\approx_l$, or by Lemma 6.19 for $\approx$.
4. we finally easily replace every single-message rule by a two-message rule, again by Lemma 6.3. □

If we remove other features from the core join-calculus, then the resulting calculi seem to loose expressiveness. For instance, if we remove the join operator in patterns, all reductions commute with one another, and we obtain a calculus that is deterministic. In the resulting calculus, polyadic messages make a big difference: the call-by-name λ-calculus can still be encoded by using messages conveying at most two names, while even branching cannot be expressed in the monadic deterministic fragment of the calculus.

6.3 Simpler definitions

Using structural rearrangements, any process can be rewritten into a flat defining process with only messages in evaluation contexts (Remark 2.1). This makes apparent that the nesting of messages and definitions is irrelevant after α-conversion. The internal structure of rules in definitions deserves a closer examination.

We begin with a series of examples that suggest that the shape of join-patterns is rather flexible; definitions can be simplified and rearranged without affecting our notions of equivalence. We then apply these remarks more systematically to encode complex definitions into simpler two-way definitions.

6.3.1 Binders and internal state

Several programming examples already suggest the use of local messages to convey the internal state of an encoding. Informally, these messages are sent on locally fresh names that are never communicated hence they never interfere with other parts of the calculus.

Lemma 6.3 *We have the strong labeled bisimulation:*

$$\mathsf{def}_S\ J \triangleright P \wedge D \ \mathsf{in}\ A \quad \sim_l \quad \mathsf{def}_S\ J|s\langle\widetilde{v}\rangle \triangleright P|s\langle\widetilde{v}\rangle \wedge D \ \mathsf{in}\ A|s\langle\widetilde{v}\rangle$$

whenever s is a fresh name and $\{\widetilde{v}\} \cap \mathsf{rv}[J] = \emptyset$.

Proof: The proof is simplistic. Since the name s does not appear elsewhere, it always conveys the same, single message $s\langle\widetilde{v}\rangle$ that is initially present in the process on the right; moreover, the additional arguments are received in the same scope as this initial message.

We show that the relation $\mathcal{R}$ that contains all the processes of the lemma is a strong labeled bisimulation. Let us assume that $P_1\ \mathcal{R}\ P_2$. All transitions are in exact correspondence on both sides of $\mathcal{R}$ and the resulting processes are still related by $\mathcal{R}$ for some updated A. We detail the internal reductions that use the differing rules. Such a reduction may consume the messages M in P_1 iff another reduction may consume the messages $M \mid s\langle\widetilde{v}\rangle$ in P_2. Moreover, these transitions replace A with the same A' on both sides of $\mathcal{R}$. $\square$

This strong equivalence suffices to prove interesting properties with regards to our scoping rules:

1. If we take $\widetilde{v} = s$, and in the case J simply is $x\langle\widetilde{u}\rangle$, we derive the obvious correctness of the encoding from a calculus with at most two names in join-patterns to a calculus with exactly two names in join-patterns.

2. If we take $\{\widetilde{v}\} = \mathsf{dv}[J] \cup \{s\}$, then all occurrences of names of $\mathsf{dv}[J \mid s\langle\widetilde{v}\rangle]$ that appear in P—we call them recursive occurrences—are now bound as received variables. Up to bisimulation, we can therefore eliminate recursion from every definition.

3. If we take $\{\widetilde{v}\} = (\mathsf{fv}[P] \cup \{s\}) \setminus \mathsf{rv}[J]$, then all the free names of P are bound as received variables, which is reminiscent of λ-lifting in the λ-calculus. This validates a compilation scheme for the join-calculus that would replace every process by an equivalent process with simpler binders (either receptions or immediate definitions).

6.3.2 Rearranging synchronization patterns

We first mention several easy "garbage collection" rules that are used to simplify programs on the fly in most other proofs, possibly using relevant up to techniques.

In Section 2.4.2 we say that a join-pattern is inert when it cannot be triggered anymore, and we remark that such join patterns are "invisible". We now rephrase these properties in terms of equivalences. For example, a join pattern J is inert as soon as there is a name in $\mathsf{dv}[J]$ that is not extruded and that does not appear anymore within the defining process (except perhaps under the $J \triangleright [\,\cdot\,]$ guard). The clause can be simply removed from the definition, provided that, whenever we remove the last occurrence of a defined name in a pattern, an additional rule is provided to discard further messages. For instance, if $x \notin \mathsf{fv}[D] \cup \mathsf{fv}[P]$, $y \in \mathsf{dv}[D]$, and $z \notin \mathsf{dv}[D]$ we have

$$\mathsf{def}\ x\langle u\rangle \mid y\langle\rangle \mid z\langle v,w\rangle \triangleright Q \wedge D \ \mathsf{in}\ P \quad \asymp_l \quad \mathsf{def}\ z\langle v,w\rangle \triangleright 0 \wedge D \ \mathsf{in}\ P \tag{6.1}$$

Independently, messages sent on join-patterns that are all either stuck or guarding an empty process can be discarded. For instance we have

$$\mathsf{def}\ x\langle\widetilde{v}\rangle \triangleright 0 \ \mathsf{in}\ C[x\langle\widetilde{u}\rangle] \quad \asymp_l \quad \mathsf{def}\ x\langle\widetilde{v}\rangle \triangleright 0 \ \mathsf{in}\ C[0] \tag{6.2}$$

(Provided, of course, that $C[\,\cdot\,]$ does not bind x.) In the same manner, we can simplify redundant definitions that contain several times the same clause, or the composition of the same clauses:

$$\mathsf{def}_S\ D \wedge D' \ \mathsf{in}\ A \quad \sim_l \quad \mathsf{def}_S\ D \wedge D \wedge D' \ \mathsf{in}\ A \tag{6.3}$$

$$\begin{array}{c} \mathsf{def}_S\ J \triangleright P \wedge J' \triangleright P' \\ \wedge\ D' \ \mathsf{in}\ A \end{array} \quad \succeq_l \quad \begin{array}{c} \mathsf{def}_S\ J \triangleright P \wedge J' \triangleright P' \\ \wedge\ J|J' \triangleright P|P' \wedge D' \ \mathsf{in}\ A \end{array} \tag{6.4}$$

Weak bisimulation is insensitive to buffering, as one can expect from an asynchronous semantics [139]. We have the simple but important property

Lemma 6.4 (Relays in definitions) *Let* $\mathsf{def}_S\ D \ \mathsf{in}\ A$ *be an open process and* x,x' *be two names such that* $x \in \mathsf{dv}[D]$ *and* x' *fresh. Let also* D' *be the definition obtained from* D *by substituting* x' *for* x *in defined-name position in every join-pattern of* D. *We have the labeled expansion*

$$\mathsf{def}_S\ x\langle\widetilde{u}\rangle \triangleright x'\langle\widetilde{u}\rangle \wedge D' \ \mathsf{in}\ A \quad \succeq_l \quad \mathsf{def}_S\ D \ \mathsf{in}\ A$$

Proof: The relation that contains all pairs of processes related by the lemma is a weak labeled expansion up to the deterministic reductions that relay messages from x to x'. □

The lemma describes the presence of relays before synchronization, but it is also possible to relay messages after synchronization, by using a continuation. For instance, we also have

$$\mathsf{def}_S\ J \triangleright x\langle\widetilde{v}\rangle \wedge x\langle\widetilde{v}\rangle \triangleright P \wedge D \ \mathsf{in}\ A \quad \succeq_l \quad \mathsf{def}_S\ J \triangleright P \wedge D \ \mathsf{in}\ A \tag{6.5}$$

where x is a fresh name and $\widetilde{v}$ is a tuple that conveys all the name received in J ($\{\widetilde{v}\} = \mathsf{rv}[J]$). While we can add buffers before or after synchronization, the buffering of partial join-patterns does not usually preserve bisimulation, because of the usual problem of gradual commitment. For instance, the commitment to one message on x separates the following processes.

$$\begin{array}{lll} & \mathsf{def}\ x\langle u\rangle|y\langle\rangle|z\langle v\rangle \triangleright \mathit{print}\langle u+v\rangle & \mathsf{in}\ Q \\ \not\approx_l & \mathsf{def}\ x\langle u\rangle|y\langle\rangle \triangleright t\langle u\rangle \wedge t\langle u\rangle|z\langle v\rangle \triangleright \mathit{print}\langle u+v\rangle & \mathsf{in}\ Q \end{array}$$

as can be seen for, e.g., $Q = x\langle 0\rangle \mid x\langle 2\rangle \mid y\langle\rangle \mid z\langle 1\rangle \mid z\langle 2\rangle$. The two processes have the same output traces $\{1,2,3,4\}$, but the former process performs the choice atomically, while the latter one can perform a gradual choice first between $x\langle 0\rangle$ and $x\langle 2\rangle$, then between $z\langle 1\rangle$ and $z\langle 2\rangle$, with an intermediate state with possible output $\{1,2\}$.

As discussed in Section 4.5, coupled barbed simulations are adequate to identify such definitions. We have the following result:

Lemma 6.5 *For every join-pattern $J_1 \mid J_2$, set of names $S \subseteq \mathsf{dv}[J_1 \mid J_2]$ and processes P and A, we have*

$$\begin{array}{lll} & J_1 \triangleright x_1\langle\widetilde{v}_1\rangle & \\ \mathsf{def}_S & \wedge \quad J_2 \triangleright x_2\langle\widetilde{v}_2\rangle & \mathsf{in}\ A \quad \overset{\bullet}{\lessgtr}{}^{\circ} \quad \mathsf{def}_S\ J_1 \mid J_2 \triangleright P\ \mathsf{in}\ A \\ & \wedge \quad x_1\langle\widetilde{v}_1\rangle \mid x_2\langle\widetilde{v}_2\rangle \triangleright P & \end{array}$$

where x_1, x_2 are fresh names, $\{\widetilde{v}_1\} = \mathsf{rv}[J_1]$, and $\{\widetilde{v}_2\} = \mathsf{rv}[J_2]$.

Proof: Without loss of generality, we assume that $S = \mathsf{dv}[J_1] \cup \mathsf{dv}[J_2]$, we let

$$\begin{array}{lllll} & & & J_1 \triangleright x_1\langle\widetilde{v}_1\rangle & \\ Q[\cdot] & \stackrel{\text{def}}{=} & \mathsf{def}_S & \wedge \quad J_2 \triangleright x_2\langle\widetilde{v}_2\rangle & \mathsf{in}\ [\cdot] \\ & & & \wedge \quad x_1\langle\widetilde{v}_1\rangle \mid x_2\langle\widetilde{v}_2\rangle \triangleright x\langle\widetilde{v}_1,\widetilde{v}_2\rangle & \\ R[\cdot] & \stackrel{\text{def}}{=} & \mathsf{def}_S & J_1 \mid J_2 \triangleright x\langle\widetilde{v}_1,\widetilde{v}_2\rangle \quad \mathsf{in}\ [\cdot] & \end{array}$$

and we prove that $Q[0] \overset{\bullet}{\lessgtr}{}^{\circ} R[0]$, which entails the result of the lemma after structural rearrangement and removal of a relay from x to the continuation P (*cf.* equation 6.5).

Our candidate coupled simulations contain more processes than those above; they relate all pairs of processes $(A \mid Q[M], A \mid R[N])$ where M is a parallel composition of messages sent on $S \cup \{x, x_1, x_2\}$, N is a parallel composition of messages sent on $S \cup \{x\}$, and we have $M \leq N$ for the partial order on processes defined by the clauses

1. $x_i\langle\widetilde{v}_i\rangle \leq J_i$ for $i = 1, 2$;

2. $x_i\langle\widetilde{v}_i\rangle \geq J_i$ for $i = 1, 2$, when messages on x_i are stuck;

3. $x_1\langle\widetilde{v}_1\rangle \mid x_2\langle\widetilde{v}_2\rangle \geq x\langle\widetilde{v}_1,\widetilde{v}_2\rangle$;

4. $\leq$ and $\geq$ are closed by parallel composition with identical messages on both sides, and by structural equivalence.

Note that the notion of "being stuck" is a global property that depends on the context A and on other messages in M. As in the proof of Lemma 4.19, we check that the pair of simulations induced by our ordering of messages form coupled barbed simulations. $\square$

This result does not hold for labeled coupled simulations, or for barbed coupled simulation-congruence $\lesseqgtr$, for the same reasons as in Section 4.5.2: without prior knowledge of the context, there is no uniform derivation to overrun partial joins.

While the lemma is easily extended to n-way partial synchronizations, it does not carry over to definitions with additional clauses on the same names, because a partial commitment to an inert join-pattern may prevent some other synchronizations. If we go down to may-testing equivalence, however, we can decompose synchronization further for any definition. For instance, we have the coarse equivalence:

$$\mathsf{def}_S\ D \wedge J \triangleright 0 \ \mathsf{in}\ A \quad \simeq_{may} \quad \mathsf{def}_S\ D \ \mathsf{in}\ A$$

for all processes $\mathsf{def}_S\ D\ \mathsf{in}\ A$ and join-patterns J as soon as $\mathsf{dv}[J] \subseteq \mathsf{dv}[D]$.

Besides, it is possible to trade bisimilarity for divergence. Weak bisimulation is insensitive to divergence, hence the previous partial synchronization of Lemma 6.5 can be made invisible up to labeled bisimulation. It suffices to add extra rules that can roll back their effect at any time. For instance, we can temporarily grab arbitrary messages J within a definition D:

$$\mathsf{def}_S\ D\ \mathsf{in}\ A \quad \approx_l \quad \mathsf{def}_S\ J \triangleright x\langle\widetilde{v}\rangle \wedge x\langle\widetilde{v}\rangle \triangleright J \wedge D\ \mathsf{in}\ A$$

provided that $\mathsf{dv}[J] \subseteq \mathsf{dv}[D]$, x is fresh, and $\{\widetilde{v}\} = \mathsf{rv}[J]$.

6.3.3 Encoding complex definitions

We now compile every complex definition, which may contain n-way join patterns and multiple clauses connected by $\wedge$, into several simpler one-pattern two-message definitions. For that purpose, we implement an invisible layer between the emitters and the guarded processes of the definition that makes explicit an automaton that matches messages and patterns.

Emissions are translated into internal actions for the automaton and sent on a conventional name. The automaton joins actions with its current state, detects patterns, then re-emits its new state. When a pattern is found, it also sends a message containing all the received names to a join-continuation. The only pitfall has to do with partial commitment to some messages as the encoding attempt to assemble join-patterns. As discussed above, these partial synchronizations are correct as long as they are reversible.

We encode a generic definition $D = J_1 \triangleright P_1 \wedge \ldots J_n \triangleright P_n$. After performing a global renaming, we assume that $\mathsf{dv}[D] = \{x_1, \ldots, x_m\}$, and that every join-pattern J_k in D joins formal messages $x_i\langle\widetilde{v_i}\rangle$ with the same received variables $\widetilde{v}_i = v_{i,1}, \ldots, v_{i,l_i}$ in all occurrences of the message. For each join-pattern J_k, we let $\widetilde{s_{J_k}} \subseteq \{1, \ldots, m\}$ be the set of indices of the names in $\mathsf{dv}[J_k]$. For the sake of clarity, we use the syntactic sugar developed for continuations to present the encoding (*cf.* Section 3.4.3). We also omit the easy encoding of subsets of $\{1, \dots, m\}$ and of $v_1, \dots, \widetilde{v}_m\}$ as flat tuples $\widetilde{s}$ and $\widetilde{v}$, respectively; we use the notation $\widetilde{o}$ for the tuple $\widetilde{v}$ with no value so far. With these conventions, the translation of a defining process is

$$[\![\mathsf{def}_S \bigwedge_{k=1}^{n} J \triangleright P \;\mathsf{in}\; A]\!] \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{def}\ \mathrm{get}() \mid set\langle \widetilde{s}, \widetilde{v}\rangle \triangleright \mathsf{reply}\ \widetilde{s}, \widetilde{v}\ \mathsf{to}\ \mathrm{get}\ \mathsf{in} \\ \mathsf{def}_{\{x_1\}\cap S}\ D_1\ \mathsf{in} \\ \vdots \\ \mathsf{def}_{\{x_m\}\cap S}\ D_m\ \mathsf{in} \\ set\langle \emptyset, \widetilde{o}\rangle \mid \prod_{k=1}^{n} R_k \mid [\![A]\!] \end{array}$$

where the names get and *set* are fresh, and where the m rules D_i for each name x_i in $\mathsf{dv}[D]$ and the n processes R_k for each clause $J_k \triangleright P_k$ of D are respectively defined as

$$\begin{array}{lcl} D_i & \stackrel{\text{def}}{=} & x_i\langle \widetilde{u}\rangle \triangleright \begin{array}{l} \mathsf{let}\ \widetilde{s}, \widetilde{v} = \mathrm{get}()\ \mathsf{in} \\ set\langle \widetilde{s} \cup \{i\}, \widetilde{v}\{^{\widetilde{u}}/_{\widetilde{v_i}}\}\rangle \mid \mathsf{if}\ i \in \widetilde{s}\ \mathsf{then}\ x_i\langle \widetilde{v_i}\rangle \end{array} \\ R_k & \stackrel{\text{def}}{=} & \mathsf{repl}\left(\begin{array}{l} \mathsf{let}\ \widetilde{s}, \widetilde{v} = \mathrm{get}()\ \mathsf{in} \\ \mathsf{if}\ \widetilde{s_{J_k}} \subseteq \widetilde{s}\ \mathsf{then}\ [\![P_k]\!] \mid set\langle \widetilde{s} \setminus \widetilde{s_{J_k}}, \widetilde{v}\rangle\ \mathsf{else}\ set\langle \widetilde{s}, \widetilde{v}\rangle \end{array} \right) \end{array}$$

The translation essentially consists of a single two-way-join definition that matches *internal actions* to an *internal state* $\langle \widetilde{s}, \widetilde{v}\rangle$ that "caches" the current pending messages on each of the defined names x_i of D: $\widetilde{s}$ collects the indices for all the names that have pending messages, and $\widetilde{v}$ collects the pending values for one of these messages, if any.

For each J_k, the auxiliary process R_k repeatedly checks whether the current state s contains all the defined variables s_k of J_k, and triggers the guarded process P_k when successful.

For each x_i, the auxiliary definition D_i inserts the values of pending messages in the current state. Notice that if another message is already present, it is removed from the cache and re-sent on x_i; this makes sure that the choice of messages that are present in the cache $\widetilde{v}$ can freely be reconsidered until this choice is committed by a P_k that consumes a few messages.

We extend our encoding $[\![\ \cdot\ ]\!]$ from defining processes to any process, by simple structural induction. The encoding is well-typed in the case the source process is typable with *monomorphic* types. The next lemma states that our implementation is correct up to labeled expansion:

Lemma 6.6 *In the join-calculus with monomorphic types, for all open processes A we have the labeled expansion $[\![A]\!] \succeq_l A$.*

Proof: We conduct the proof for a single complex defining process where all defined names have been extruded ($A = \mathsf{def}_{\mathsf{dv}[D]}\ D\ \mathsf{in}\ 0$). To this end, we consider the variant of our encoding that does not encode guarded processes: P_k is substituted for $[\![P_k]\!]$ in R_k.

We check that, for every set of messages, and for every internal state that can be reached from them, the join-patterns that can be chosen are exactly the same as in the source definition. Except for the access to the state guarded by the join-pattern $\mathrm{get}() \mid set\langle \widetilde{s}, \widetilde{v}\rangle$ all reductions of the encoding are deterministic.

We let M, M' range over parallel compositions of messages sent on $\mathsf{dv}[D]$, and, in case M contains at most one message on each name of $\mathsf{dv}[D]$, we let $C(M)$ be the open process obtained from the translation $[\![A]\!]$ by substituting the message $set\langle\widetilde{s},\widetilde{v}\rangle$ that caches the arguments of all messages in M for the initial message $set\langle\emptyset,\widetilde{o}\rangle$. For all $x_i \in \mathsf{dv}[D]$, for all M with no message on x_i, we have the series of reductions

$$x_i\langle\widetilde{v}_i\rangle \,|\, C(M) \quad \rightarrow^*\sim_l \quad C(x_i\langle\widetilde{v}_i\rangle \,|\, M) \tag{6.6}$$
$$x_i\langle\widetilde{w}_i\rangle \,|\, C(x_i\langle\widetilde{v}_i\rangle \,|\, M) \quad \rightarrow^*\sim_l \quad x_i\langle\widetilde{v}_i\rangle \,|\, C(x_i\langle\widetilde{w}_i\rangle \,|\, M) \tag{6.7}$$

The reductions 6.6 cache a message on x_i for the first time; the reductions 6.7 swap the contents of a message being cached on x_i; in both cases, the series consists of a deterministic reduction to trigger the definition of x_i, a join reduction to grab the state, and a few deterministic reductions to test and update the state. As regards the active join-patterns, we have the series of reductions

$$C(J_k\sigma \,|\, M) \quad \rightarrow^*\sim_l \quad P_k\sigma \,|\, C(M) \tag{6.8}$$

for all M with no message on $\mathsf{dv}[J_k]$; again, all reductions are deterministic except for the one that joins the continuation of get() with the current state of the cache.

Let $\mathcal{R}$ be the relation that contains all pairs of processes

$$M \,|\, C(M'), \qquad M \,|\, M' \,|\, A$$

where M' contains at most one message for each name in $\mathsf{dv}[D]$, and let $\rightarrow_d$ denote all reductions of the encoding that unfold the loop, consumes a message on x_i, or perform some operation on the tuples. We prove that $\mathcal{R}$ is a labeled expansion up to deterministic reductions $\rightarrow_d$ and parallel composition.

Intrusions are the same on both sides, and lead to an extended multiset M. There is no extrusion. Internal reductions on the left are either deterministic reductions, or reductions that extend the cache (6.6), or reductions that start a swap (6.7)—all these cases are simulated by no reduction on the other side—, or reductions that join a request from R_k with a state $M' = J_k\sigma \,|\, M''$ that has enough messages to successfully trigger a process $P_k\sigma$ (6.8)—in this case, the real messages in M' on the right-hand-side are jointly received in one step. Conversely, internal reductions on the right may consume messages in both M and M'; for each message in M, we use one of the two series of reductions 6.6 or 6.7 to obtain a join-pattern entirely in M', then we perform 6.8. On both sides, we have the same additional process $P_k\sigma$ in parallel composition with processes that are related by $\mathcal{R}$ with M'' instead of M' in the cache; the process $P_k\sigma$ is discarded up to parallel composition.

We remark that the encoding commutes with every global renaming, and obtain the general case by repeatedly applying the above expansion in context for every join-definition of an arbitrary open process A. $\Box$

Formally, Lemma 6.6 is very precise because the encoding is entirely local to the translated definition. Yet, the relation $\succeq_l$ does not reflect the absence of diverging computations (on the left). On the contrary, infinite computations immediately appear for two reasons: opposite swaps as soon as two messages $x_i\langle\widetilde{v}\rangle$ and $x_i\langle\widetilde{w}\rangle$ are available on the same name, and free replication in each R_k. As in the preliminary discussion, we could get rid of diverging computations by enabling the reception of a message on x_i only when there is no cached message on x_i so far, with a simple "signal" message in the join-pattern that defines x_i, and by triggering the unfolding of each replicated process R_k with another signal message only when a new message on $x_i \in \mathsf{dv}[J_k]$ is cached. This clearly re-introduce gradual commitment to the messages being cached, but it seems possible to obtain a coupled-simulations congruence relation that generalizes Lemma 6.5.

Remark 6.7 (Non-linear join-patterns) *In this dissertation we always require that a defined name occur at most once in every join-pattern; nonetheless, the same encoding shows that this natural limitation could easily be relaxed. For instance, an explicit encoding of the definition*

$$\mathsf{def}_{\{x\}}\ x\langle u\rangle \mid x\langle v\rangle \triangleright P \ \mathsf{in}\ 0$$

in the spirit of the more general translation above would be

$$\begin{array}{l} \mathsf{def}_{\{x\}}\ x\langle u\rangle \mid \mathit{get}\langle \kappa\rangle \triangleright \kappa\langle u\rangle \ \mathsf{in} \\ \mathsf{def}\ \kappa_0\langle u\rangle \triangleright \\ \quad \mathsf{def}\ \kappa_1\langle v\rangle \triangleright \mathit{get}\langle \kappa_0\rangle \mid (P \oplus (x\langle u\rangle \mid x\langle v\rangle))\ \mathsf{in} \\ \quad \mathit{get}\langle \kappa_1\rangle\ \mathsf{in} \\ \mathit{get}\langle \kappa_0\rangle \end{array}$$

6.4 Relays everywhere

We now present general techniques to simplify the interface of a process. The resulting translations are useful to assemble more complex encodings, and illustrate the use of firewalls to protect these encodings from families of transitions that would invalidate some invariant.

The next lemma states that synchronization in join-patterns and reception of messages from the environment can be separated in two successive stages.

Lemma 6.8 *There is a compositional encoding $[\cdot]^\circ$ such that for all open processes A we have*

1. $A^\circ \succeq_l A$;
2. *if $A^\circ\ (\xrightarrow{\alpha})^*\ B$ and x is extruded ($x \in \mathsf{xv}[B]$), then x is defined by a single rule of the form $x\langle \widetilde{u}\rangle \triangleright x'\langle \widetilde{u}\rangle$.*

The lemma is not affected by the presence of comparison in the join-calculus.

Proof: We use the compositional translation that adds a relay for every defined name, as described in Lemma 6.4.

The first property is obtained by structural induction on the syntax of A. In the case of a defining process, we repeatedly apply Lemma 6.4 on each defined name. In the case of a guard (in the presence of name comparison) we check that the relation is preserved by substitution—the translation does not operate on free names—and apply. All other cases immediately follow from the pre-congruence property of $\succeq_l$.

The image of the translation $\cdot^\circ$ is closed by all transitions; the second property is thus obtained by checking that all names in $\mathsf{xv}[A]$ are defined as relays. $\square$

This first level of encoding has the advantage of separating syntactically the issues of communication and synchronization. It is not unduly expensive, as twice as many reductions are needed to translate a series of reduction steps, in the worst case.

We can constrain some more the structure of output labels by requiring that each defined name be sent on a free name at most once—that is, that each name communicated to the outside be a newly-extruded name.

Lemma 6.9 *In the join-calculus with monomorphic types and no comparison, there is an encoding* $[\,\cdot\,]^{\circ\circ}$ *such that for all open processes A we have*

1. $A^{\circ\circ} \succeq A$ *in the absence of name comparisons;*
2. *if* $A^{\circ\circ}\ (\xrightarrow{\alpha})^*\ B$ *and x is extruded* ($x \in \mathsf{xv}[B]$)*, then x is defined by a single rule of the form* $x\langle\widetilde{u}\rangle \triangleright P$*;*
3. *if* $A^{\circ\circ}(\xrightarrow{\alpha})^* \xrightarrow{Sx\langle\widetilde{v}\rangle}$*, then* $S = \{\widetilde{v}\}$.

This second layer of encoding is more demanding: in the worst case, it may cause a quadratic increase of the number of reduction steps. Also, this result does not carry over to labeled equivalences, or to a calculus with name comparison. The encoding is type-directed, which induces restrictions on the types in use in the calculus—they have to be monomorphic.

We explicitly define the encoding $[\,\cdot\,]^{\circ\circ}$ and prove its properties by using *recursive firewalls* that provide definitions that filters every communication on names in the interface of A:

Definition 6.10 *A* firewall context $\mathcal{F}^{\Sigma}_{X,F}[\,\cdot\,]$ *is parameterized by two partial functions on names X and F with finite domains and by a finite set of types Σ that meet the following requirements:*

1. *The intrusion and extrusion parts of the firewall do not mix, i.e.,*
   $\mathsf{dom}\,(X) \cap \mathsf{dom}\,(F) = \emptyset$, $\mathsf{dom}\,(X) \cap \mathsf{ran}\,(X) = \emptyset$, $\mathsf{dom}\,(F) \cap \mathsf{ran}\,(F) = \emptyset$
2. *the partial function $X \cup F$ is cycle-free.*
3. *The set of types Σ is closed by decomposition, i.e,*
   if $\langle\sigma_1, \ldots, \sigma_n\rangle \in \Sigma$*, then also* $\sigma_1, \ldots, \sigma_n \in \Sigma$.
4. *At least the types of all names in* $\mathsf{dom}\,(X) \cup \mathsf{dom}\,(F)$ *are included in* Σ.

It is defined as an evaluation context

$$\mathcal{F}^{\Sigma}_{X,F}[\,\cdot\,] \quad \stackrel{def}{=} \quad \mathsf{def}\ D_\Sigma\ \mathsf{in}\ \mathsf{def}_{\mathsf{dom}\,(X)} \bigwedge_X D^{\sigma}_{x \mapsto x'} \wedge \bigwedge_F D^{\sigma}_{x' \mapsto x}\ \mathsf{in}\ [\,\cdot\,]$$

where the definitions $D^{\sigma}_{x'\mapsto x}$ and D_Σ stand for

$$\begin{aligned} D^{\langle\sigma_1,\ldots,\sigma_n\rangle}_{x'\mapsto x} \quad &\stackrel{def}{=} \quad x'\langle v_1, \ldots, v_n\rangle \triangleright\ \mathsf{let}\ v'_1 = r_{\sigma_1}(v_1)\ \mathsf{in} \\ &\qquad\qquad\qquad\qquad \vdots \\ &\qquad\qquad\qquad\qquad \mathsf{let}\ v'_n = r_{\sigma_n}(v_n)\ \mathsf{in} \\ &\qquad\qquad\qquad\qquad x\langle v'_1, \ldots, v'_n\rangle \\ D_\Sigma \quad &\stackrel{def}{=} \quad \bigwedge_{\sigma\in\Sigma} r_\sigma(x) \triangleright \mathsf{def}\ D^{\sigma}_{x'\mapsto x}\ \mathsf{in}\ \mathsf{reply}\ x'\ \mathsf{to}\ r_\sigma \end{aligned}$$

In the special case of a monadic calculus, two single rules suffice to build the firewall; moreover, the communication of pairs is purely internal to the firewall, and can be internally encoded as described in the next section.

By construction, if $x' \mapsto x$ is in F and $x \notin \mathsf{dom}\,(X)$, then we have the series of relations

$$\begin{array}{rl} & \mathcal{F}^{\Sigma}_{X,F}\left[x'\langle v_1,\ldots,v_n\rangle\right] \\ \rightarrow (\rightarrow\rightarrow\sim_l)^n \xrightarrow{\{w_1,\ldots,w_n\}\overline{x}\langle w_1,\ldots,w_n\rangle} & \mathcal{F}^{\Sigma}_{X\uplus\{w_1\mapsto v_1,\ldots,w_n\mapsto v_n\},F}[0] \end{array} \tag{6.9}$$

for any choice of fresh names $w_1,\ldots,w_n$. Besides, all the internal steps before the extrusion are deterministic, and the strong equivalence is used only to get rid of the definition of continuations that are passed to each call to $r_\sigma\langle\cdot\rangle$, once the call has returned. We also have the symmetric filtering of incoming messages. If $x \mapsto x'$ in X, then

$$\begin{array}{rl} & \mathcal{F}^{\Sigma}_{X,F}[0] \\ \xrightarrow{x\langle v_1,\ldots,v_n\rangle}\!\!\rightarrow (\rightarrow\rightarrow\sim_l)^n & \mathcal{F}^{\Sigma}_{X,F\uplus\{w_1\mapsto v_1,\ldots,w_n\mapsto v_n\},F}\left[x'\langle w_1,\ldots,w_n\rangle\right] \end{array} \tag{6.10}$$

While the filtering is actually the same for incoming and outcoming messages, the distinction between X and F will be useful to preserve invariants later on.

For all messages sent on names defined in a firewall, all reductions that consume these messages and unfold new components in the firewall are deterministic. Besides, the conditions on X and F guarantee that the rewriting terminates. In the following, we use the notation $\rightarrow_d$ for these reductions; in the proofs, we usually normalize processes by immediately performing these reductions, but we retain the original notation. (Formally, we would define an auxiliary translation that translates all messages in evaluation contexts to the result of their reception in the firewall.)

In the following, we assume given a bijection on names $x \mapsto x'$ that we use to rename the interface of processes within firewalls. For all processes A, we let A' be the process obtained by replacing all names x in $\mathsf{fv}[A] \cup \mathsf{xv}[A]$ by names x', and also F and X be the partial functions defined as $\{x' \mapsto x \mid x \in \mathsf{fv}[A]\}$ and $\{x \mapsto x' \mid x \in \mathsf{xv}[A]\}$. We define the encoding $A^{\circ\circ}$ as $\mathcal{F}^{\Sigma(X,F)}_{X,F}[A'] \setminus \mathsf{xv}[A']$. As an example, the translation of $x\langle x\rangle$ requires a firewall with a signature Σ that contains the monadic recursive type $\mu\alpha.\langle\alpha\rangle$, an empty function X, and a function $F = \{x' \mapsto x\}$. We have the firewall unfolding steps

$$\mathcal{F}^{\Sigma}_{\emptyset,\{x'\mapsto x\}}\left[x'\langle x'\rangle\right] \quad \rightarrow^*_d\sim_l \quad \mathcal{F}^{\Sigma}_{\{y\mapsto x'\},\{x'\mapsto x\}}[x\langle y\rangle]$$

We establish the properties claimed in Lemma 6.9. We begin with an explicit the correspondence between the transitions of a process and the transitions of the same process placed within a firewall.

Lemma 6.11 (Operational correspondence) *Let A be a process in the open join-calculus and $\mathcal{F}^{\Sigma}_{X,F}[A'] \setminus \mathsf{xv}[A']$ be its encoding.*

For all processes in firewalls, we assume that every relay has been unfolded (i.e., there is no more immediate $\rightarrow_d$ step). We have:

Internal step *If $A \rightarrow B$, then $\mathcal{F}^{\Sigma}_{X,F}[A'] \rightarrow\rightarrow^{*}_{d}\sim_{l} \mathcal{F}^{\Sigma}_{X,F}[B']$.*

If $\mathcal{F}^{\Sigma}_{X,F}[A'] \rightarrow T$, then $A \rightarrow B$ with $T \rightarrow^{*}_{d}\sim_{l} \mathcal{F}^{\Sigma}_{X,F}[B']$.

(The source reduction $A \rightarrow B$ may have released messages on free names in evaluation context, and there is a new entry in the extrusion part of the firewall for each argument of each such new messages. With our convention on $\rightarrow_{d}$, these relays are kept implicit hence X is not updated.)

Extrusion *If $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} B$, then for every tuple of fresh names $\widetilde{w}$, for the partial function Y defined as $\{\widetilde{w \mapsto v}\}$, we have $\mathcal{F}^{\Sigma}_{X,F}[A] \xrightarrow{\{\widetilde{w}\}\overline{x}\langle\widetilde{w}\rangle} \mathcal{F}^{\Sigma}_{X\uplus Y,F}[B]$.*

If $\mathcal{F}^{\Sigma}_{X,F}[A] \xrightarrow{S\overline{x}\langle\widetilde{w}\rangle} T$, then $S = \{\widetilde{w}\}$, $x' \mapsto x \in X$, and for the partial function $Y = \{\widetilde{v \mapsto w}\}$ we have $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} B$ and $T \rightarrow^{*}_{d}\sim_{l} \mathcal{F}^{\Sigma}_{X\uplus Y,F}[B]$.

(All extruded names are actually already relays, we can α-convert these names before performing the extrusion and registering them in X.)

Ground intrusion *If $A \xrightarrow{x\langle\widetilde{v}\rangle} B$* and the names $\widetilde{v}$ are all fresh and distinct*, then for the partial function $G \overset{def}{=} \{\widetilde{v}' \mapsto \widetilde{v}\}$ we have $\mathcal{F}^{\Sigma}_{X,F}[A'] \xrightarrow{x\langle\widetilde{v}\rangle} \mathcal{F}^{\Sigma}_{X,F\uplus G}[B']$.*

If $\mathcal{F}^{\Sigma}_{X,F}[A] \xrightarrow{x\langle\widetilde{w}\rangle} T$ then $x \in \mathsf{dom}\, X$ and for all distinct fresh names $\widetilde{v}$, for the partial function $G \overset{def}{=} \{\widetilde{v}' \mapsto \widetilde{w}\}$, we have $A \xrightarrow{x\langle\widetilde{v}\rangle} B$ and $T = \mathcal{F}^{\Sigma}_{X,F\uplus G}[B']$.

In particular, the family of open processes protected by a firewall as defined in the above lemma is closed for all ground transitions up to deterministic reductions and strong equivalence.

The next lemma states that protection by a firewall cannot be observed, up to barbed expansion.

Lemma 6.12 (Correctness) *Let A be an open process with monomorphic types. We have the expansion relation $A^{\circ\circ} \succeq A$.*

Proof: We conduct the proof on join-calculus processes, by distinguishing between external and internal parts. The result of the lemma is then obtained by applying Lemma 5.23. Let $\mathcal{R}$ be the relation that contains the pairs of processes

$$\mathsf{def}\ D_{\Sigma}\ \mathsf{in}\ \mathsf{def} \bigwedge_{X,F} D_{x\mapsto y} \wedge D_i \wedge D_e\ \mathsf{in}\ P_i \,|\, P_e \quad \mathcal{R} \quad \mathsf{def}\ D_i\sigma \wedge D_e\sigma\ \mathsf{in}\ P_i\sigma \,|\, P_e\sigma$$

that meet the following conditions:

1. the set of types Σ contains at least the types of all names in $\mathsf{dom}\,(X)$ and $\mathsf{dom}\,(F)$.
2. the finite sets of names $\mathsf{dom}\,(F)$, $\mathsf{dom}\,(X)$, $\mathsf{dv}[D_e]$, and $\mathsf{dv}[D_i]$ are pairwise disjoint.
3. $\mathsf{fv}[\mathsf{def}\ D_i\ \mathsf{in}\ P_i] \subseteq \mathsf{dom}\,(F)$ and $\mathsf{fv}[\mathsf{def}\ D_e\ \mathsf{in}\ P_e] \cap (\mathsf{dv}[D_i] \cup \mathsf{dom}\,(F)) = \emptyset$.
4. $\mathsf{ran}\,(F) \cap \mathsf{dv}[D_i] = \emptyset$ and $\mathsf{ran}\,(X) \subseteq \mathsf{dom}\,(F) \cup \mathsf{dv}[D_i]$.

5. the partial function $X \cup F$ is cycle-free, and σ is the substitution that maps every name in $\mathsf{dom}\,(X) \cup \mathsf{dom}\,(F)$ to its final image obtained by iterating X and F.

and let $\mathcal{R}' \subseteq \mathcal{R}$ be the relation that meets the additional condition:

6. P_i is a parallel composition of messages sent on names defined in D_i; $P_e = P'_e \mid P''_e$ where P'_e is a parallel composition of messages sent on free names and P''_e is a parallel composition of messages sent on names defined in D_e.

We first establish that $\mathcal{R} \subseteq \equiv\rightarrow_d^*\sim_l\mathcal{R}'\equiv$. By using structural rearrangement, we have $P_i \equiv \mathsf{def}\ D'_i\ \mathsf{in}\ M_i$ and $P_e \equiv \mathsf{def}\ D'_e\ \mathsf{in}\ M_e$ where all names in $\mathsf{dv}[D'_i]$ and $\mathsf{dv}[D'_e]$ are fresh—and in particular not affected by σ. Starting from a pair of processes related by $\mathcal{R}$, we thus have

$$\begin{array}{rl} & \mathsf{def}\ D_\Sigma\ \mathsf{in}\ \mathsf{def}\ \bigwedge_{X,F} D_{x\mapsto y} \wedge (D_i \wedge D'_i) \wedge (D_e \wedge D'_e)\ \mathsf{in}\ M_i \mid M_e \\ \mathcal{R} & \mathsf{def}\ (D_i \wedge D'_i \wedge D_e \wedge D'_e)\sigma\ \mathsf{in}\ M_i\sigma \mid M_e\sigma \end{array}$$

Further, every message $x\langle\widetilde{v}\rangle$ in M_i is sent on a name x that is either defined in $\mathsf{dv}[D_i \wedge D'_i]$ or in $\mathsf{dom}\,(F)$. In the latter case, a series of relations as 6.9 applies to the firewalled process, which extends $\mathsf{dom}\,(X)$ with $\widetilde{w \mapsto v}$ for some fresh names $\widetilde{w}$ and replaces $x\langle\widetilde{v}\rangle$ by $F(x)\langle\widetilde{w}\rangle$. This messages meets the conditions to be placed in M_e. On the other side of $\mathcal{R}$, we have for the extended σ that $(F(x)\langle\widetilde{w}\rangle)\sigma = x\langle\widetilde{v}\rangle\sigma$, hence the message is just moved from M_i to M_e using structural equivalence. In a symmetric manner, every message in M_e that is sent on a name in $\mathsf{dom}\,(X)$ can be transported to M_i by unfolding the firewall on the left and structural rearrangement on the right. While the same message may be shifted from external to internal and back a few times, condition 5 guarantees that the rewriting terminates. The resulting pairs of processes are related by $\mathcal{R}'$.

We are now ready to prove that $\mathcal{R}'$ (hence $\mathcal{R}$) are included in barbed expansion by Lemma 4.30. Let us assume that $Q\ \mathcal{R}'\ R$.

- The strong barbs of Q and R are the same; they correspond to messages sent on free names in M_e and these names are not affected by σ.

- The congruence property is immediate: after applying α-conversion to avoid name clashes, the context is not affected by σ, hence it can be applied to P_e only by using structural rearrangement; the resulting pair of processes is still in $\mathcal{R}$.

- The two bisimulation diagrams of $\mathcal{R}'$ are easily established, as in fact there is a strong bisimulation diagram from $\mathcal{R}'$ to $\mathcal{R}$. By construction of $\mathcal{R}'$ there is no message sent to the firewall in evaluation context, hence every reduction either uses a rule in D_i or in D_e. The situation is symmetric; assuming the rule is in D_i, by definition of $\mathcal{R}'$ the reduction consumes only messages in P_i, triggers a new process that meets all the conditions of $\mathcal{R}$ to be placed in P_i, and since the substitution σ does not operates on names in $\mathsf{dv}[D_i]$ the same reduction applies on the other side of $\mathcal{R}'$. □

The next lemma shows that the firewall encoding $[\,\cdot\,]^{\circ\circ}$ protects a process from name-comparison on its extrusions—anyway all extruded names are pairwise different in the image of the encoding—and thus reconciles barbed congruence and labeled bisimulation.

Lemma 6.13 *The encoding* $[\cdot]^{\circ\circ}$ *from the open join-calculus with monomorphic types equipped with barbed congruence, to the same calculus equipped with labeled bisimulation:*

$$A^{\circ\circ} \approx_l B^{\circ\circ} \quad \textit{iff} \quad A \approx B$$

Proof: We already have $\approx_l \subseteq \approx$ (Lemma 5.13), hence if $A^{\circ\circ} \approx_l B^{\circ\circ}$ then by the previous lemma we obtain $A \preceq A^{\circ\circ} \approx B^{\circ\circ} \succeq B$ and simply $A \approx B$.

Conversely, if $A \approx B$ then by congruence property $A^{\circ\circ} \approx B^{\circ\circ}$, hence it suffices to prove that the relation $\mathcal{R}$ that contains all pairs of processes $A^{\circ\circ}, B^{\circ\circ}$ with $A^{\circ\circ} \approx B^{\circ\circ}$ is a ground labeled bisimulation.

We apply the same proof technique as in the presence of name comparison (Theorem 7), except that the contexts for extrusion are much simpler: by definition of the firewall, every argument of an extrusion is a freshly extruded name, hence no name comparison is necessary in the context that tests for the extrusion. □

6.5 Polyadic messages

In this dissertation we usually consider a polyadic variant of the join-calculus: a message can carry an arbitrary number of values, whereas a type system ensures that the arity of every channel is fixed. In contrast, a message in the core join-calculus always carries exactly one name, and the additional structure of types and arities has disappeared. The same variation arises in other settings; for instance in the π-calculus, most theoretical development are conducted on monadic variants of the calculus, while most applications, type systems, and examples are given in polyadic variants of the calculus [145].

In [99], Milner introduces the polyadic π-calculus as a convenient setting, in particular, to describe application examples and to discuss encodings of the λ-calculus. He also proposes a simple encoding that allocates and sends a private channel to emit a series of monadic messages instead of a single polyadic message. The translation of polyadic output is:

$$[\![\overline{x}\langle y_1, \ldots, y_n\rangle]\!] \quad \stackrel{\text{def}}{=} \quad \nu z.\overline{x}\langle z\rangle.\overline{z}\langle y_1\rangle.\cdots.\overline{z}\langle y_n\rangle$$

The encoding owes its simplicity to the symmetric use of input and output guards, but some variants carry over to the asynchronous π-calculus and, as we shall see, to the join-calculus.

To our knowledge, however, little is known about the formal properties of such encodings. The type information is discarded in the translation of a polyadic process, hence valid monadic contexts can be the translations of ill-typed polyadic contexts; in particular, such contexts can invalidate properties that were enforced by typing—for instance, that two names with incompatible types are different—and thus reveal differences between the translations of equivalent source processes. The only full abstraction results are given by Yoshida in [153]. The author develops a sophisticated type system for the monadic π-calculus in order to reflect there the typing assumptions of the polyadic calculus. Her monadic type system expresses dynamic properties of linear transmission protocols as *graph types*. Hence, well-typed monadic contexts are essentially translations of well-typed polyadic contexts.

Our intent here is to consider the plain, untyped monadic join-calculus as a target language, and to obtain precise full abstraction results only through refined encodings. We believe that a similar study may be conducted in the π-calculus, but that the locality property of the join-calculus makes the situation significantly simpler.

More generally, the encoding of polyadic communication into monadic communication is a good example of the use of some alternative communication protocol between processes, and of techniques to achieve full abstraction. More involved communication protocols are described in [2] and in the next section.

6.5.1 Communicating pairs and lists

As in the π-calculus, we communicate tuples of names over auxiliary private names; we first describe the protocol for pairs: the process $E_t\langle u,v\rangle$ serves the pair u,v at the name t; the context $R^t\langle u,v\rangle[P]$ extracts a pair u,v from the name t, then executes P. On the sending side, a local message $w\langle z\rangle$ conveys the next value to be returned to t. As above, we rely on the syntactic sugar of Section 3.4 to hide local continuations.

$$\begin{array}{rcl}
E_t\langle u,v\rangle & \stackrel{\text{def}}{=} & \mathsf{def}\ w\langle z\rangle \mid \mathrm{r}() \triangleright \mathsf{reply}\ z\ \mathsf{to}\ \mathrm{r}\ \mathsf{in} \\
 & & w\langle u\rangle \mid \mathsf{def}_{\{t\}}\ \mathrm{t}() = \left(\begin{array}{l} \mathsf{let}\ z = \mathrm{r}()\ \mathsf{in} \\ w\langle v\rangle \mid \mathsf{reply}\ z\ \mathsf{to}\ \mathrm{t} \end{array} \right) \mathsf{in}\ 0 \\
R^t\langle u,v\rangle[P] & \stackrel{\text{def}}{=} & \mathsf{let}\ u = \mathrm{t}()\ \mathsf{in} \\
 & & \mathsf{let}\ v = \mathrm{t}()\ \mathsf{in} \\
 & & P
\end{array}$$

In these terms, asynchronous names have one argument, synchronous names have no argument, thus the terms above are monadic (as soon as u and v are) and non-recursive. Provided that $t \notin \mathsf{fv}[P]$ we have the operational correspondence

$$(E_t\langle u,v\rangle \mid R^t\langle u,v\rangle[P]) \setminus t \quad \rightarrow^8 \sim_l \quad P \tag{6.11}$$

where each of the two calls $\mathrm{t}()$ uses four reduction steps and returns first u, then v, and where strong bisimilarity $\sim_l$ is used to get rid of the continuations and the definitions in $E_t\langle u,v\rangle$.

It is straightforward to encode tuples of arbitrary lengths $\langle v_1, v_2, \ldots, v_n\rangle$ into nested pairs $\langle v_1, \langle v_2, \langle \cdots \langle v_{n-1}, v_n\rangle\rangle\rangle\rangle$. We omit the details, and we let $E_t\langle\widetilde{v}\rangle$ and $R^t\langle\widetilde{v}\rangle[\,\cdot\,]$ be the extensions of the two terms above from pairs to tuples of fixed lengths. In particular, we can generalize 6.11 into the labeled expansion

$$(E_t\langle\widetilde{v}\rangle \mid R^t\langle\widetilde{u}\rangle[P]) \setminus t \quad \succeq_l \quad P\{\widetilde{^u/_v}\} \tag{6.12}$$

6.5.2 Compositional translation

Intuitively, our translation from the well-typed polyadic join-calculus to the monadic join-calculus replaces every emission of a polyadic message by a run of the emission protocol and every reception of a polyadic message by the reception of a name t that encodes the tuple, followed by a run of the reception protocol.

The translation of well-typed polyadic processes is defined inductively on processes and definitions, from the two basic translations for messages in open processes and reaction rules in open definitions:

$$\begin{array}{rcl} [\![x\langle\widetilde{v}\rangle]\!] & \stackrel{\text{def}}{=} & (x\langle t\rangle \mid E_t\langle\widetilde{v}\rangle) \setminus t \\ [\![x_1\langle\widetilde{v}_1\rangle \mid \ldots \mid x_n\langle\widetilde{v}_n\rangle \triangleright P]\!] & \stackrel{\text{def}}{=} & x_1\langle t_1\rangle \mid \ldots \mid x_n\langle t_n\rangle \triangleright R^{t_1}\langle\widetilde{v}_1\rangle[\ldots[R^{t_n}\langle\widetilde{v}_n\rangle[[\![P]\!]]]] \end{array}$$

where the names t, t_i and other names in the underlying communication protocols are fresh names.

The next lemma relates reductions in the source polyadic process to reductions in the translated monadic process

Lemma 6.14 (Operational correspondence) *For every open process A,*

1. *if $A \rightarrow A'$ then $[\![A]\!] \rightarrow\succeq_l [\![A']\!]$;*
2. *conversely, if $[\![A]\!] \rightarrow^n T$ then $A \rightarrow^m A'$ with $T \succeq_l [\![A']\!]$, for some $m \leq n$.*

Proof: The initial reductions occur on join-patterns whose defined names are in syntactic correspondence in the source process and in its translation.

(1.) In the translation, the names t_i that encode tuples of arguments each appear in the translation of a single emission, hence after the first reduction step these names t_i can be restricted to the triggered process of the translation $R^{t_1}\langle\widetilde{v}_1\rangle[\ldots[R^{t_n}\langle\widetilde{v}_n\rangle[[\![P]\!]]]]$ along with the n emission protocols. We then repeatedly apply the labeled expansion (6.12) for each received t_i, which eventually triggers the translation of the guarded process P where the formal arguments have been replaced with the received ones.

(2.) By induction on n, we prove that if $[\![A]\!] \rightarrow^n T$ then $A \rightarrow^m A'$ with $T \succeq [\![A']\!]$ and $m \leq n$. The base case $n = 0$ is immediate. For the inductive case, we trace the first reduction in $[\![A]\!] \rightarrow T_1 \rightarrow^n T$ back to the polyadic calculus, and obtain a source reduction $A \rightarrow A'_1$. Moreover, *(1.)* yields $T_1 \succeq [\![A'_1]\!]$. By gluing n instances of the simulation diagram in the definition of expansion, we have $[\![A'_1]\!] (\rightarrow^=)^n T'_1$ with $T' \succeq_l T'_1$. By induction hypothesis, $A'_1 \rightarrow^m A'$ with $m \leq n$, $T' \succeq_l T'_1 \succeq_l [\![A']\!]$, and $A \rightarrow^{(m+1)} A'$. □

6.5.3 Full abstraction

The translation of well-typed polyadic processes is performed in two steps: first we perform the translation of Lemma 6.8 to ensure that every name that is communicated to the context is defined as a relay, then we use the compositional protocol described above for communicating tuples. We obtain full abstraction up to labeled bisimulation:

Lemma 6.15 *The encoding $[\![\ \cdot\ ^\circ]\!]$ is fully abstract up to $\approx_l$.*

The two converse implications of the lemma are separately established in Lemmas 6.17 and 6.18. The systematic use of relays in the encoding may appear redundant, as it forces every pair to be encoded and decoded twice! However, this double encoding guarantees that only valid pairs are involved in reductions that correspond to the source join synchronizations. With only one level of encoding, contexts that do not comply with our protocol may interfere, as is the case for the following processes

$$\begin{array}{rcl} A & \stackrel{\text{def}}{=} & \mathsf{def}_x\ x\langle u,v\rangle \mid y\langle\rangle \triangleright z\langle\rangle \ \mathsf{in}\ x\langle a,a\rangle \mid y\langle\rangle \\ B & \stackrel{\text{def}}{=} & \mathsf{def}_x\ x\langle u,v\rangle \triangleright 0 \ \mathsf{in}\ z\langle\rangle \end{array}$$

We have $A \approx B$ but $[\![A]\!] \not\approx [\![B]\!]$; for instance, a context around the translation $[\![A]\!]$ may send the name t defined as $\mathrm{t}() \triangleright 0$ instead of a name that encodes a pair, hence cause a deadlock as the translation attempts to decode this pair after synchronizing it with the unique message $y\langle\rangle$. On the contrary, the translation $[\![B]\!]$ always emit the message $z\langle\rangle$ no matter of the context. Other subtle differences would appear in the case the context triggers several times the continuations of receiving processes.

Lemma 6.16 *if $[\![A]\!] \approx [\![B]\!]$, then $A \approx B$.*

Proof: Let $\mathcal{R}$ be the relation that contains all pairs of processes (A, B) such that $[\![A]\!] \approx [\![B]\!]$. We apply Lemma 4.29 to establish the inclusion $\mathcal{R} \subseteq\ \approx$.

1. The strong barbs are preserved by the translation, even if the arities of messages on free names are changed.

2. The translation $[\![\,\cdot\,]\!]$ is compositional; hence, if $A \mathrel{\mathcal{R}} B$, and $C[\,\cdot\,]$ is an evaluation context, we have $[\![A]\!] \approx [\![B]\!]$, and, by applying the congruence property in the monadic calculus for the evaluation context $[\![C]\!][\,\cdot\,]$,
$$[\![C[A]]\!] = [\![C]\!][[\![A]\!]] \approx [\![C]\!][[\![B]\!]] = [\![C[B]]\!]$$

3. Let us assume $A \rightarrow A'$ and $A \mathrel{\mathcal{R}} B$. Applying Lemma 6.14(1) we have $[\![A]\!] \rightarrow\succeq [\![A']\!]$. By weak bisimulation in the monadic calculus, $[\![B]\!] \rightarrow^n T$ with $T \approx\succeq [\![A']\!]$ where $n \geq 0$ is the length of the simulating reduction sequence. By Lemma 6.14(2) we obtain for some B' that $B \rightarrow^* B'$ and $T \succeq [\![B']\!]$, hence $[\![A']\!] \preceq\approx\succeq [\![B']\!]$ and thus $A' \mathrel{\mathcal{R}} B'$. □

Since our proof does not depend on the presence of name-matching, we obtain by Theorem 7 the first half of full abstraction.

Corollary 6.17 *if $[\![A]\!] \approx_l [\![B]\!]$, then $A \approx_l B$.*

Lemma 6.18 *For all open processes A and B with relays, if $A \approx_l B$, then $[\![A]\!] \approx_l [\![B]\!]$.*

Proof: We supplement our translation with representations of processes that are receiving tuples from the context. The translation of A is extended into a multi-hole context whose holes are indexed by $x \in \mathsf{xv}[A]$ and appear within the translation, in parallel with the definition of the relay on x. A *receiving session* on a relay from $x \in \mathsf{xv}[A]$ to x' is an open process whose free names are either fresh, or in $\mathsf{xv}[A]$, or are x' in a translation $[\![x'\langle\widetilde{v}\rangle]\!]$ that appear under a guard. We let $[\![A]\!][R_x]$ be the extended translation, in which R_x is a family of receiving sessions indexed by $x \in \mathsf{xv}[A]$.

We let $\mathcal{R}$ be the relation that contains all pairs of open processes

$$(\ [\![A]\!][R_x],\ [\![B]\!][R_x]\ ) \text{ such that } A \approx_l B$$

and we prove that $\mathcal{R}$ is a labeled bisimulation up to expansion and evaluation context. Transitions are partitioned as follows:

1. Extrusion, intrusion or internal reduction in a receiving session $R_x \xrightarrow{\alpha} R'_x$: these transitions are the same on both sides, and always yield a new valid receiving session, unless the transition triggers a guarded process $[\![x'\langle\widetilde{v}\rangle]\!]$. In that case, the messages $x'\langle\widetilde{v}\rangle$ are put in parallel with A and B, and by the congruence property of $\approx_l$ we obtain two new processes related by $\mathcal{R}$.

2. Extrusion $[\![A]\!] \xrightarrow{\{u\}\overline{x}\langle u\rangle}$. This transition corresponds to a source transition of the form $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} A'$, where we have $A \equiv (A' \mid x\langle\widetilde{v}\rangle) \setminus S$; we write the first extrusion of the protocol $[\![x\langle\widetilde{v}\rangle]\!] \xrightarrow{\{u\}\overline{x}\langle u\rangle} [\![x\langle\widetilde{v}\rangle]\!]_u$

   This source transition is simulated by $B \rightarrow^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} B'$ on the other side. Hence we have by Lemma 6.14(1)

$$\begin{array}{lcl} [\![A]\!][R] & \xrightarrow{\{u\}\overline{x}\langle u\rangle} & ([\![A']\!][R] \mid [\![x\langle\widetilde{v}\rangle]\!]_u) \setminus S \\ [\![B]\!][R] & \rightarrow^*\succeq_l \xrightarrow{\{u\}\overline{x}\langle u\rangle} & ([\![B']\!][R] \mid [\![x\langle\widetilde{v}\rangle]\!]_u) \setminus S \end{array}$$

   and, since $[\![A']\!][R] \ \mathcal{R} \ [\![B']\!][R]$, this closes our bisimulation diagram up to the particular restricted context $([\cdot] \mid [\![x\langle\widetilde{v}\rangle]\!]_u) \setminus S$.

3. Intrusion on a name $x \in \mathsf{xv}[A]$ (thus $x \in \mathsf{xv}[B]$). We have

$$\begin{array}{lcl} [\![A]\!][R] & \xrightarrow{x\langle t\rangle}\rightarrow_d & [\![A]\!][R'] \\ [\![B]\!][R] & \xrightarrow{x\langle t\rangle}\rightarrow_d & [\![B]\!][R'] \end{array}$$

   where the deterministic reduction triggers the process $R_{x,t} = R^t\langle\widetilde{v}\rangle[[\![x'\langle\widetilde{v}]\!]]$ of the translated relay on x, and where R' is obtained from R by substituting $R_x \mid R_{x,t}$ for the previous receiving session R_x. The two resulting processes are still related by $\mathcal{R}$.

4. Reduction internal to $[\![A]\!]$. From $[\![A]\!] \rightarrow T$, we obtain by Lemma 6.14(2) that $A \rightarrow^= A' \preceq_l T$, then by weak bisimulation in the polyadic calculus $B \rightarrow^* B' \approx_l A'$ and by Lemma 6.14(1) $[\![B]\!] \rightarrow^*\succeq_l [\![B']\!]$. The presence of receiving sessions does not affect these relations, hence we close the diagram with the relation $\succeq_l\mathcal{R}\preceq_l$. □

We actually do not have full abstraction yet for barbed congruence, as illustrated by the relay equation in the source calculus:

$$x\langle y\rangle \quad \approx \quad \mathsf{def}\ z\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle z\rangle$$

While this barbed congruence is immediate in the polyadic calculus, it is not preserved by the translation $[\![\ \cdot\ ^\circ]\!]$ because the two translated processes can be separated by a context $\mathsf{def}\ x\langle z\rangle \triangleright [\![z\langle 1\rangle]\!]\ \mathsf{in}\ [\cdot]$ that attempts to use another arity for the translation of the free name y, while the relay on z works only for nullary message.

Our next result states that, if we rule out messages that re-exports free names to the context by applying the internal encoding $\cdot^{\circ\circ}$ of Lemma 6.9, we still have a fully abstract translation.

Lemma 6.19 *In the join-calculus with monomorphic types, and in the absence of name comparison, the encoding* $[\![\,\cdot\,{}^{\circ\circ}]\!]$ *is fully abstract for* $\approx$.

Its proof is the combination of Lemma 6.16 and of the next result:

Lemma 6.20 *For all processes* P *and* Q *that never extrude free names, if* $P \approx Q$ *then* $[\![P]\!] \approx [\![Q]\!]$.

Proof: We use a larger relation $\mathcal{R}$ that is closed by reduction up to expansion, and we prove that this relation is a barbed congruence up to expansion. We assume that monadic names are partitioned in two infinite subsets X and F, we let

$$\begin{array}{rcl} Q_i & \stackrel{\text{def}}{=} & \mathsf{def}\ [\![D_i]\!] \wedge D_c\sigma_i \ \mathsf{in}\ [\![P_i]\!] \mid P_c\sigma_i \\ R_i & \stackrel{\text{def}}{=} & \mathsf{def}\ D_i \ \mathsf{in}\ P_i \mid \mathit{plug}\langle\widetilde{z}\sigma_i\rangle \end{array}$$

and we let $\mathcal{R}$ relate all pairs of monadic processes (Q_1, Q_2) such that we have

1. $R_1 \approx R_2$ (in the polyadic join-calculus);
2. $\mathsf{dv}[D_i] \subseteq X$, $\mathsf{dv}[D_c] \subseteq F$;
3. $\mathsf{def}\ D_i\ \mathsf{in}\ P_i$ is a polyadic process that only extrudes names on fresh relays;
4. σ_i is a substitution defined on $X \cap \mathsf{fv}[\mathsf{def}\ D_c\ \mathsf{in}\ P_c]$ that ranges over $\mathsf{dv}[D_i]$;
5. names $x \in X$ whose definition is not deterministic only appear as $[\![x\langle\widetilde{v}\rangle]\!]$ with the same arity as in the definition of $x\sigma_i$.

We use Lemma 4.29 to establish that $\mathcal{R} \subseteq \approx$.

- $\mathcal{R}$ is a congruence for all contexts that do not have names in X as free variables, up to structural equivalence. Each context is merged into enlarged $D_c\sigma_i$ and $P_c\sigma_i$; by hypothesis the substitutions do not affect the names in the new components. We check that every defining properties of $\mathcal{R}$ is preserved.
- $\mathcal{R}$ respects all strong barbs.
- $\mathcal{R}$ is a weak bisimulation. Without loss of generality, we first rearrange the terms using structural equivalence to obtain simpler P_i and P_c that consist of messages only, then we prove the weak simulation diagram by a case analysis on the reduction $Q_1 \rightarrow Q_1'$.

  reduction using a join-pattern $[\![J \triangleright P]\!]$ in $[\![D_1]\!]$: this reduction consumes messages that can be traced back to $[\![M_1]\!]$ in $[\![P_1]\!]$ and messages $[\![M_c\sigma_1]\!]$ from P_c, and this reduction triggers a process $[\![P\sigma]\!]$ that can be added to P_1.

  We consider the processes $C[R_i]$ for the polyadic context

  $$C[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ \mathit{plug}\langle\widetilde{z}\rangle \triangleright M_c \mid \mathit{plug}'\langle\widetilde{z}\rangle\ \mathsf{in}\ [\,\cdot\,]$$

  then use bisimulation in the source process for the two reductions $C[R_1] \rightarrow\rightarrow R_1'$ that consume the plugging message, then triggers the same polyadic process $P\sigma$ by consuming the source messages $M_c\sigma_1 \mid M_1$. This yields a series of reductions

$C[R_2] \rightarrow^* R'_2$ with $R'_1 \approx R'_2$. We discard the reduction that enables the messages $M_c\sigma_2$, and we use Lemma 6.14(1) to report the resulting series of derivations into $Q_2 \rightarrow^* \succeq Q'_2$ with the required properties.

reduction using a join-pattern in $D_c\sigma_1$: by hypothesis, the rule in $D_c\sigma_1$ is of the form $J \triangleright P\sigma_1$, and the reduction consumes some messages $[\![M_1]\!]$ from $[\![P_1]\!]$ and some messages $M_c\sigma_1$ from P_c.

Messages received from P_i may be different on both sides; all these names are in X, however, so we can expand the substitutions σ_i to accommodate this difference between the new processes in $\mathcal{R}$.

Again, we use a specific context in the source, polyadic calculus to extract messages from R_2 on the same names as those used in Q_1. Let $\widetilde{u}$ be a tuple of names in $X \setminus (\mathsf{dv}[D_1] \cup \mathsf{dv}[D_2])$ and M be a parallel composition of messages on names in F such that $\{\widetilde{u}\} = \mathsf{rv}[M]$. Let also σ'_1 be an extension of σ_1 on $\widetilde{u}$ such that $M\sigma'_1 = M_1$. We use the context

$$C[\,\cdot\,] \stackrel{\text{def}}{=} \mathsf{def} \left(\begin{array}{cl} & plug\langle\widetilde{z}\rangle \mid M \triangleright plug'\langle\widetilde{z},\widetilde{u}\rangle \\ \wedge & plug\langle\widetilde{z}\rangle \triangleright t\langle\rangle \end{array} \right) \mathsf{in}\ [\,\cdot\,]$$

(where the disappearance of the barb on the fresh name t indicates that the messages have been grasped.) We have the relation

$$C[\mathsf{def}\ D_1\ \mathsf{in}\ P'_1 \mid M_1 \mid plug\langle\widetilde{z}\sigma_1\rangle] \quad \rightarrow\sim \quad \mathsf{def}\ D_1\ \mathsf{in}\ P'_1 \mid plug'\langle(\widetilde{z},\widetilde{u})\sigma'_1\rangle = R'_1$$

and we obtain by weak bisimulation

$$C[\mathsf{def}\ D_2\ \mathsf{in}\ P_2 \mid plug\langle\widetilde{z}\sigma_2\rangle] \quad \rightarrow^*\sim R'_2$$

with the transitions $R_2 \rightarrow^* \mathsf{def}\ D_2\ \mathsf{in}\ P'_2 \mid M_2 \mid plug\langle\widetilde{z}\sigma_2\rangle$ of the required form. Thus we obtain a new pair of related processes, whose corresponding source processes extrude a potentially larger set of names. □

6.6 Cross-encodings with the π-calculus

Despite their syntactic differences, the join-calculus can be considered as an offspring of the π-calculus, in the asynchronous branch of the family. The latter was introduced independently in [37] as the (mini) asynchronous π-calculus, and in [70] as the ν-calculus. Both authors suppress the guards on emission, and compare the result to the original π-calculus. Going further in that direction, the join-calculus is an asynchronous π-calculus with the strong restrictions:

1. the three binders (scope restriction, reception, replicated reception) are syntactically merged into a single construct: the definition;
2. communication occurs only on defined names;
3. for every name, there is exactly one receiving definition.

There are several reasons to be interested in a formal comparison between the two calculi: the π-calculus has been thoroughly studied; it is a reference calculus in concurrency theory, and many results relate other formalisms or implementations to it, as for instance in [17, 98, 143, 148]. Therefore, it is appealing to "translate" such results automatically to the join-calculus. On the other hand, some issues are best addressed in the join-calculus, as for instance locality, programming and implementation purposes, and distribution.

Our most precise encodings are complex, yet their underlying ideas are simple. In particular, much simpler encodings can be obtained in less general settings, or for coarser equivalence properties. For instance, most programming examples written in PICT [122] can be translated to the join-calculus language with almost no change in the source syntax.

Applying the results of Section 6.14, we consider the recursive, polyadic join-calculus with at most two-way-join definitions as the target calculus to encode the π-calculus, and its monadic variant for the reverse encoding.

We first recall the particular syntax of the π-calculus in use here, then we encode the π-calculus in the join-calculus. The first, naive encoding replaces each channel of the π-calculus by a two-way definition; however, some more work is needed to achieve full abstraction. We present our approach based on "firewalls" in detail, but we defer the presentation of the proof to Section 6.7. In the same manner, we encode the join-calculus in the π-calculus using a simple translation of definitions into scope-restriction and replicated reception. Again, full abstraction requires a refinement of the basic encoding, which relies on the firewall we developed for the encoding of polyadic communication.

6.6.1 The asynchronous π-calculus

We define the variant of the asynchronous π-calculus in use in this dissertation. We use the syntax of Milner in [99], but consider only monadic communication Without loss of generality, we allow only monadic messages, and replicated input instead of more general recursion.

$$\begin{array}{lcll}
P & ::= & & \text{processes} \\
 & & \overline{x}\langle u\rangle & \quad\text{message} \\
 & | & P \,|\, P & \quad\text{parallel composition} \\
 & | & \nu u.P & \quad\text{scope restriction} \\
 & | & x(u).P & \quad\text{input} \\
 & | & !x(u).P & \quad\text{replicated input} \\
 & | & \mathsf{0} & \quad\text{null process}
\end{array}$$

We omit the presentation of a reduction-based semantics for this calculus—we refer to [37] for a formal semantics. We simply recall the basic reduction step, which matches pairs of complementary emission and reception, and substitutes actual names for the formal variables in receiving processes:

$$\overline{x}\langle v\rangle \,|\, x(y).Q \longrightarrow Q\{^v/_y\}$$

We define the (output-only) barbs $\downarrow_{\overline{x}}$ as the presence of a message on x in evaluation context; we write $\Downarrow_{\overline{x}}$ for the weak barbs $\rightarrow^*\downarrow_{\overline{x}}$ We equip the π-calculus with *barbed* *congruence*, as defined in Section 4.4. Our definition is not the one that usually appears in the literature, but the resulting equivalence coincides with the congruence of barbed output-only bisimilarity as defined in [101, 16]. This result is the π-calculus counterpart of Theorem 7; it is exposed in [56].

Definition 6.21 *Barbed congruence in the asynchronous π-calculus ($\approx_\pi$) is the largest weak bisimulation on π-calculus processes that is a congruence for evaluation contexts and respects the output barbs $\Downarrow_{\overline{x}}$.*

The next statement summarizes our comparison between this variant of the π-calculus and the join-calculus; the remainder of this chapter is devoted to its constructive proof.

Theorem 9 *The join-calculus and the asynchronous π-calculus have the same expressive power up to their respective barbed congruences.*

6.6.2 Asynchrony, relays, and equators

Our encodings essentially rely on the properties of asynchrony in process calculi without name testing, as first discussed in [71, 73] for the π-calculus and in Chapters 4 and 5 for the join-calculus. We briefly recall the situation in the π-calculus.

As in the join-calculus, the emitter cannot directly observe whether its message is received. The situation is less clear-cut, because a context that emits a message can try to get its message back by attempting reception on its own, hence seemingly detect whether its message has been received by some other process. In case of success, however, the message may have been received, then re-emitted in-between. For instance we have $x(u).\overline{x}\langle u\rangle \approx_\pi 0$. More generally it is not possible to distinguish between two different names that have the same external behavior. We illustrate this property with the definition of *equators* between names, in Honda and Yoshida's terminology [73]:

$$M^\pi_{x,y} \quad \stackrel{\text{def}}{=} \quad !x(u).\overline{y}\langle u\rangle \,|\, !y(v).\overline{x}\langle v\rangle$$

This process repeatedly receives values from x and forwards them to y and vice-versa, so that no matter which name x or y is used to send a value, it can always be made available for reception on the other name in one internal reduction. As a result, the names x and y become synonyms, hence the equators are somehow the π-calculus counterpart for the relays in the join-calculus.

Remark 6.22 *For all π-processes P, Q where z is fresh, we have*

$$P\{^z/_x, ^z/_y\} \approx_\pi Q\{^z/_x, ^z/_y\} \quad \textit{implies} \quad M^\pi_{x,y} \,|\, P \approx_\pi M^\pi_{x,y} \,|\, Q$$

Proof (sketch): The congruence property is immediate for parallel composition; all strong barbs are the same, except on x and y; barbs on x and y always coincide, and are equivalent to barbs on z after substitution; every reduction is simulated either by no reduction whenever it occurs in the equator $M^\pi_{x,y}$, or by the reduction after substitution otherwise; only reductions on z may require an additional reduction step to be simulated before substitution. □

6.6.3 Encoding the π-calculus

As opposed to the join-calculus, a channel name x of the π-calculus conveys *two* communications capabilities $\overline{x}\langle\cdot\rangle$ for output and $x(\cdot)$ for input. Accordingly, we associate to every π-calculus channel x two join-calculus names x_o for output, x_i for input, and an enclosing definition that joins outputs and inputs. The emitter simply sends values on x_o; the receiver defines a name for its continuation, and sends it as a reception offer on x_i.

We give below the structural translation of the π-calculus based on this simple encoding applied to every π-calculus channel. Later on, we explain why it is not fully abstract and how it can be refined.

Definition 6.23 (Basic structural translation) *To every process P in the π-calculus we associate the process $[\![P]\!]_\pi$ in the join-calculus inductively defined on the syntax as follows*

$$
\begin{aligned}
[\![\overline{x}\langle v\rangle]\!]_\pi &\overset{def}{=} x_o\langle v_o, v_i\rangle \\
[\![P \mid Q]\!]_\pi &\overset{def}{=} [\![P]\!]_\pi \mid [\![Q]\!]_\pi \\
[\![\nu x.P]\!]_\pi &\overset{def}{=} \mathsf{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathsf{in}\ [\![P]\!]_\pi \\
[\![x(v).P]\!]_\pi &\overset{def}{=} \mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright [\![P]\!]_\pi\ \mathsf{in}\ x_i\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &\overset{def}{=} \mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid [\![P]\!]_\pi\ \mathsf{in}\ x_i\langle\kappa\rangle \\
[\![0]\!]_\pi &\overset{def}{=} 0
\end{aligned}
$$

The translation above uses explicit continuations κ to represent reception offers; it translates monadic π-calculus channels to tuples with recursive types $\tau = \langle\, \langle\tau\rangle, \langle\langle\tau\rangle\rangle\,\rangle$; it could also be exposed in a more functional style, e.g.,

$$[\![x(v).P]\!]_\pi = \mathsf{let}\ v = x_i()\ \mathsf{in}\ [\![P]\!]_\pi$$

As an example of operational correspondence, we consider a source reduction between two π-calculus processes and we reflect this reduction as relations between their translations in the join-calculus:

$$\nu x.(\overline{x}\langle a\rangle \mid \overline{x}\langle b\rangle \mid x(u).\overline{y}\langle u\rangle) \;\rightarrow\; \nu x.(\overline{x}\langle a\rangle \mid \overline{y}\langle b\rangle)$$

gets translated into

$$
\begin{aligned}
& \mathsf{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathsf{in} \\
& x_o\langle a_o, a_i\rangle \mid x_o\langle b_o, b_i\rangle \mid \mathsf{def}\ \kappa\langle u_o, u_i\rangle \triangleright y_o\langle u_o, u_i\rangle\ \mathsf{in}\ x_i\langle\kappa\rangle \\
\rightarrow\quad & \mathsf{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathsf{in} \\
& x_o\langle a_o, a_i\rangle \mid \mathsf{def}\ \kappa\langle u_o, u_i\rangle \triangleright y_o\langle u_o, u_i\rangle\ \mathsf{in}\ \kappa\langle b_o, b_i\rangle \\
\rightarrow\sim\quad & \mathsf{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathsf{in} \\
& x_o\langle a_o, a_i\rangle \mid y_o\langle b_o, b_i\rangle
\end{aligned}
$$

The first reduction joins $x_o\langle b_o, b_i\rangle$ and $x_i\langle\kappa\rangle$, and performs the same internal choice as in the π-calculus. The second reduction is deterministic, and triggers the translation of the guarded process $\overline{y}\langle b\rangle$. The strong equivalence gets rid of the definition of κ once its unique message has been consumed. The resulting join-calculus process is the translation of the resulting π-calculus process above.

In the same manner, any reduction on a bound name in the π-calculus can be simulated by a join-reduction followed by a deterministic reduction in the join-calculus, and conversely any reduction in a join-calculus translation falls in either of these two cases, and can be simulated in at most one reduction in the π-calculus. The next lemma directly relates a π-calculus process to its translation.

Lemma 6.24 (Operational correspondence) *The hybrid relation*

$$\succeq_j^\pi \quad \stackrel{def}{=} \quad \{(P,Q) \mid P \sim [\![Q]\!]_\pi \text{ and } Q \text{ has no input barb}\}$$

has the following properties

1. $\succeq_j^\pi$ *is an expansion with regards to reduction steps*
2. $\succeq_j^\pi$ *respects the barbs* ($P \succeq_j^\pi Q$ *implies* $P \Downarrow_{x_o}$ *iff* $Q \Downarrow_{\overline{x}}$);
3. *if* $P \succeq_j^\pi Q$ *and* $C[Q]$ *has no input barb, then* $[\![C]\!]_\pi[P] \succeq_j^\pi C[Q]$

(We say that Q has an input barb on x when $Q \rightarrow^* C[x(\widetilde{v}).Q]$ where $C[\,\cdot\,]$ is an evaluation context that does not bind x.) The "no input barb" condition rules out communication on free names; unfortunately, it does not hold in general in the π-calculus, even if it can be automatically guaranteed in subsets of the π-calculus, for instance by using a type system with input-output polarities.

Proof: We easily check each of the properties above on Definition 6.23:

1. We prove the two expansion diagrams up to expansion. Each source reduction implies an input and an output on some channel x in the π-calculus. By hypothesis on input barbs, there must be an enclosing $\nu x.$ binder; we use its translation to effect the translated reduction in two steps as described above. Conversely, each reduction in the translation can be traced back to the translation of a $\nu x.$ binder, and the trailing deterministic reduction is absorbed by the candidate expansion.

2. We have the correspondence on strong barbs $\downarrow_{x_o}$ and $\downarrow_{\overline{x}}$, which entails the correspondence of weak barbs by gluing the above diagrams.

3. By definition 6.23 we have $[\![C]\!]_\pi[[\![Q]\!]_\pi] = [\![C[Q]]\!]_\pi$. □

Toward a fully abstract translation The encoding $[\![\,\cdot\,]\!]_\pi$ does not reflect the behavior of processes of the π-calculus when placed in an arbitrary join-calculus context: the protocol specifically relies on the presence of the translation of a νx binder for every channel x, while the context may define the names x_o, x_i in some other way when x is free in the source process.

For instance, if we remove the binder νx from our example above, the resulting translation $[\![\overline{x}\langle a\rangle \mid \overline{x}\langle b\rangle \mid x(u).\overline{y}\langle u\rangle]\!]_\pi$ has no reduction in the join-calculus, because there is no definition of x_o and x_i. Besides, the translation $[\![x(u).\overline{x}\langle u\rangle]\!]_\pi$ exhibits a barb on x_i that reveals the presence of an input for x, and allows a join-calculus context to distinguish this process from $[\![0]\!]_\pi$, while a π-calculus context would not able to do so.

This suggests that we strengthen our encoding to ensure that there is a correct definition of x_o, x_i for every translated free name x. Unfortunately, name mobility makes this invariant difficult to maintain. For example, a π-calculus context may send a new free name x to a process that receives on some free names; hence the translation must also be able to receive a new pair x_o, x_i *created by the context*. Yet, nothing ensures that a join-calculus context sends such pairs only when they are correctly defined. A malicious context such as

$$C[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ z\langle v_o, v_i\rangle \triangleright 0 \wedge t\langle\kappa\rangle \triangleright x\langle\rangle\ \mathsf{in}\ x_o\langle z, t\rangle \,|\, [\,\cdot\,]$$

is able to "forge" a message $x_o\langle z,t\rangle$ from some of its own names z, t with arbitrary definitions; here the context $C[\,\cdot\,]$ transforms input messages on t into barbs on x, thus revealing the input requests of the translation. Other contexts could also trigger several times the same continuation. Such observations are excluded by the operational semantics of the π-calculus.

To protect the translation from hostile contexts, the names resulting from the free channels of the π-term must be protected by a firewall context that enforces the protocol, in the spirit of Section 6.4. We refine our first idea: each channel x is now represented as *several* pairs x_o, x_i from the basic encoding that cannot be distinguished from the outside. Two pairs can be merged up to barbed congruence by repeatedly communicating their pending messages to one another, which can be achieved by (translations of) equators. New pairs are dynamically created according to the following secure protocol:

- Whenever a pair of names is received from the outside, the firewall defines a new, correct *proxy pair*, merges it to the external pair, and transmits the new pair to the translation.
- Whenever a pair of names is sent to the outside, a new filter is inserted to set up proxies for future incoming messages on this pair.

As a result, the translation and the context never exchange names from a syntactic point of view. We use the following contexts to build the firewall on top of the naive translation:

$$\begin{array}{rcl}
\mathcal{P}_x[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathsf{def}\ x_l\langle v_o, v_i\rangle \,|\, x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathsf{in}\ \mathsf{def}\ x_o\langle v_o, v_i\rangle \triangleright p\langle v_o, v_i, x_l\rangle\ \mathsf{in}\ [\,\cdot\,] \\
\mathcal{E}_x[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathcal{P}_x[x_e\langle x_o, x_i\rangle \,|\, [\,\cdot\,]] \\
\mathcal{M}[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathsf{def}\ p(x_o, x_i, \kappa) \triangleright \mathcal{P}_y[\kappa\langle y_o, y_i\rangle \,|\, [\![M^\pi_{x,y}]\!]_\pi]\ \mathsf{in}\ [\,\cdot\,]
\end{array}$$

For every free name x, $\mathcal{P}_x[\,\cdot\,]$ encodes the creation of a new proxy for its output. $\mathcal{E}_x[\,\cdot\,]$ does the same, and also exports the proxy on a conventional free name x_e (as *plug* in Section 5.23). Finally, $\mathcal{M}[\,\cdot\,]$ recursively defines the proxy creator p for the whole translation. We introduce more concise notations that gather the previous components in a single wrapper. For a given set of names $S = \{x_1, \ldots, x_n\}$ in the π-calculus, we write

$$\begin{array}{rcl}
\mathcal{E}_{\{x_1,\ldots,x_n\}}[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathcal{M}\Big[\mathcal{E}_{x_1}[\ldots\mathcal{E}_{x_n}[\,\cdot\,]\ldots]\Big] \\
\mathcal{P}_{\{x_1,\ldots,x_n\}}[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathcal{M}\Big[\mathcal{P}_{x_1}[\ldots\mathcal{P}_{x_n}[\,\cdot\,]\ldots]\Big]
\end{array}$$

The ordering of the names in S is irrelevant up to structural rearrangement. Intuitively, S is the set of names free in $[\,\cdot\,]$ and filtered by the firewall, for which the translation guarantees a correct filtering definition. As communication occurs, new names may cross the firewall and lead to a larger set S.

Theorem 10 *Let S be a finite set of names in the π-calculus. For all π-calculus processes Q and R such that $\mathsf{fv}[Q] \cup \mathsf{fv}[R] \subseteq S$, we have*

$$Q \approx_\pi R \quad \textit{if and only if} \quad \mathcal{E}_S[[\![Q]\!]_\pi] \approx \mathcal{E}_S[[\![R]\!]_\pi]$$

Due to the presence of a "wrapper" at the top level of the translation, our translation is not entirely compositional. In the proof, however, we also give an auxiliary translation that is strictly compositional.

6.6.4 Encoding the join-calculus

The reverse translation is simpler because the join-calculus is somehow the π-calculus with restrictions on communication patterns. However, a careful encoding is still required to prevent contexts of the π-calculus from reading messages on the names they receive from the translation.

Definition 6.25 (Basic structural translation) *To every open process A in the core join-calculus that extrude only names defined in single-name rules, we associate the process $[\![A]\!]_j$ in the π-calculus inductively defined as*

$$\begin{aligned}
[\![A \mid B]\!]_j &\stackrel{def}{=} [\![A]\!]_j \mid [\![B]\!]_j \\
[\![x\langle\widetilde{v}\rangle]\!]_j &\stackrel{def}{=} \overline{x}\langle\widetilde{v}\rangle \\
[\![\mathsf{def}_S\ x\langle\widetilde{u}\rangle \triangleright P \ \mathsf{in}\ A]\!]_j &\stackrel{def}{=} \nu(\{x\}\backslash S).(!x(\widetilde{u}).[\![P]\!]_j \mid [\![A]\!]_j) \\
[\![\mathsf{def}\ x\langle\widetilde{u}\rangle \mid y\langle\widetilde{v}\rangle \triangleright P \ \mathsf{in}\ A]\!]_j &\stackrel{def}{=} \nu x, y.(!x(\widetilde{u}).y(\widetilde{v}).[\![P]\!]_j \mid [\![A]\!]_j)
\end{aligned}$$

The limitation on extruded names simplifies the analysis of the encoding; for translating all processes, we can apply the pre-encoding $[\,\cdot\,]^\circ$ that adds relays for all defined names (*cf.* Lemma 6.8).

With this restriction, in the translation of the join-pattern $x\langle u\rangle \mid y\langle v\rangle$ the first reception of a message $x(u)$ is a deterministic reduction because the replicated receiving process introduced by the translation is the only reception on x; the second reception of a message $y(v)$ simulates to the two-way synchronization in the join-calculus. Syntactically, we break the symmetry between x and y and the atomicity of their join-reduction, but it does not matter, because scope restriction and $[\![\,\cdot\,]\!]_j$ guarantee that these details cannot be observed. Note, however, that the same trick would not apply for three-way joins or more complex definitions, because we could not associate the commitment of a source communication to the last reception in the π-calculus; this is yet another instance of the gradual commitment problem.

As an example of operational correspondence, the source relations

$$\mathsf{def}\ x\langle u\rangle \mid \mathit{once}\langle\rangle \triangleright z\langle u\rangle \ \mathsf{in}\ x\langle 1\rangle \mid x\langle 2\rangle \mid \mathit{once}\langle\rangle \quad \rightarrow\sim \quad z\langle 1\rangle$$

can be rendered in the π-calculus with the relations

$$\begin{array}{rl} & \nu x, once.(!x(u).once().\overline{z}\langle u\rangle \mid \overline{x}\langle 1\rangle \mid \overline{x}\langle 2\rangle \mid \overline{once}\langle\rangle) \\ \to\to & \nu x, once.(!x(u).once().\overline{z}\langle u\rangle \mid once().\overline{z}\langle 1\rangle \mid once().\overline{z}\langle 2\rangle \mid \overline{once}\langle\rangle) \\ \to\sim & \overline{z}\langle 1\rangle \end{array}$$

The first two reduction steps may occur at any time because these reductions are deterministic; they trigger two alternative receiving processes on *once* for synchronization with the $\overline{once}\langle\rangle$ message. The third reduction actually corresponds to synchronization in the source process. Strong equivalences in both calculi get rid of the now-deadlocked definition.

More generally, we let $\to_1$ gather all reduction steps in the π-calculus that use the replicated reception of a first message in the translation of a two-way join-pattern. We retain the notation $[\![\,\cdot\,]\!]_j$, but we assume that all such reductions have been performed for messages in evaluation contexts.

With this assumption, we have the following operational correspondence lemma for our translation:

Lemma 6.26 *For all open join-calculus processes A that extrude only single-defined names, we have*

1. *If $A \to B$, then $[\![A]\!]_j \to\to_1^* [\![B]\!]_j$*
2. *If $A \xrightarrow{x\langle\widetilde{v}\rangle} B$, then $[\![A]\!]_j \xrightarrow{x\langle\widetilde{v}\rangle} [\![B]\!]_j$.*
3. *If $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} B$, then $[\![A]\!]_j \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} [\![B]\!]_j$.*
4. *If $[\![A]\!]_j \to T$ then $A \to B$ and $T \to_1^* [\![B]\!]_j$.*
5. *If $[\![A]\!]_j \xrightarrow{x\langle\widetilde{v}\rangle} T$ and $x \in \mathsf{fv}[A]$ then $A \xrightarrow{x\langle\widetilde{v}\rangle} B$ and $T = [\![B]\!]_j$.*
6. *If $[\![A]\!]_j \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} T$ and $x \in \mathsf{xv}[A]$ then $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} B$ and $T = [\![B]\!]_j$.*

Proof: As in the proof of Lemma 6.24 we easily check each of these assertions on the definition of the translation. □

In particular, we can summarize this lemma as a "labeled hybrid expansion" between translations and source processes, for the respective labels of the π-calculus and the join-calculus.

Toward full abstraction While the translation is simpler in this direction, we are still facing the same difficulties to achieve full abstraction: the encoding of a name of the join-calculus reveals too much about the source process. For instance a context of the π-calculus could start reading values on names extruded by the translation of definitions, thus breaking the locality principle that is guaranteed in the join-calculus semantics. Technically, the problem is revealed by the need for side conditions on the interface of the translated process for intrusion and extrusion in the translation, in the clauses 5 and 6 of the above lemma. Note that if we were translating the join-calculus into an asynchronous π-calculus extended with a type system with polarities [120], we could specify write-only types for every channel that is exchanged with the translation, and the (typed) previous encoding would already be fully abstract. This fact is also mentioned in [94] in a similar setting.

As is the case for the converse encoding, we rely on a firewall to achieve full abstraction without imposing constraints on the contexts of the target calculus. Fortunately, we can reuse the firewall translation $[\cdot]^{\circ\circ}$ presented in Section 6.4. Intuitively, interaction between the translation of a firewalled process and its π-calculus context occurs only on relays of the firewall, hence on distinct names for input and output. As regards input, for instance, once incoming messages have been received by the firewall they become invisible from the context, and until this occurs, they do not interact with the translation.

We have the following full-abstraction result for the reverse translation:

Theorem 11 *For all open processes A and B that have monomorphic types, we have*

$$A \approx B \quad \textit{iff} \quad [\![A^{\circ\circ}]\!]_j \approx_\pi [\![B^{\circ\circ}]\!]_j$$

Its proof mostly relies on the stability of the firewalled processes $A^{\circ\circ}$ through all ground transitions, in combination with the above operational correspondence:

Lemma 6.27 *If $A^{\circ\circ} \approx_l B^{\circ\circ}$, then $[\![A^{\circ\circ}]\!] \approx_{\pi l} [\![B^{\circ\circ}]\!]$*

Proof: We prove that the two translations are related by ground asynchronous bisimulation instead of weak labeled bisimulation. These two relations and their coincidence are studied in [16]—in our case, the delay clause is never used because all receivers are replicated relays.

Let $\mathcal{R}$ be the relation that contains all pairs of π-processes of the lemma. We prove that $\mathcal{R}$ is a ground asynchronous bisimulation in the π-calculus. As before, we assume that all messages sent to the firewall have been unfolded, but we retain the basic notation for the resulting processes. Note that as long as these messages are not extruded to the π-calculus environment we can freely rename the contents of all messages after they cross the firewall.

We perform a case analysis on the transitions $[\![A^{\circ\circ}]\!]_j \xrightarrow{\alpha} T$ in the π-calculus, and use the composition of the operational correspondence for the two stages of the encoding (firewall and translation).

Extrusion If $[\![A^{\circ\circ}]\!]_j \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} T$, by construction of the firewall we must have $x \in \mathsf{ran}\,(F)$, and by Lemma 6.26(6) we have a source extrusion $A^{\circ\circ} \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle}$, and by Lemma 6.11 we further have that $S = \widetilde{v}$ and that all the names in $\widetilde{v}$ are distinct.

By labeled bisimulation in the join-calculus, the source extrusion is simulated by internal steps followed by the same source extrusion $\rightarrow^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle}$. By applying Lemma 6.11 in the converse direction, we obtain another process of the form $B'^{\circ\circ}$ such that

$$B^{\circ\circ} \quad \rightarrow^* \quad \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle}\sim_l B'^{\circ\circ}$$

(Where the strong bisimulation only gets rid of inert definitions of continuations.) By Lemma 6.26(1,3), this series of transitions can be carried over to the π-calculus translation $[\![B^{\circ\circ}]\!]_j$, which closes the labeled bisimulation for $\mathcal{R}$ up to strong labeled equivalence.

Intrusion An intrusion in the π-calculus becomes an intrusion in the join-calculus (even if intrusion in π-calculus implies a communication, while intrusion in join-calculus is asynchronous).

The intrusion necessarily occurs on a incoming relay of the firewall; the same intrusion applies on both sides, commutes with $[\,\cdot\,]^{\circ\circ}$ up to $\sim_d$,

Internal reduction this is the composition of the two operational correspondences. $\square$

We establish that the composition of the two encodings also reflects barbed bisimulation:

Lemma 6.28 *If* $[\![A^{\circ\circ}]\!] \approx_{\pi l} [\![B^{\circ\circ}]\!]$ *and* $\mathsf{xv}[A] = \mathsf{xv}[B]$, *then* $A^{\circ\circ} \approx B^{\circ\circ}$.

Proof: Let $\mathcal{R}$ be the relation in the open join-calculus that contains all pairs of processes $(\mathcal{F}^{\Sigma}_{X,F}[A], \mathcal{F}^{\Sigma}_{X,F}[B])$ such that the firewall captures the interfaces of A and B, $\mathsf{xv}[A] = \mathsf{dom}\,(X) = \mathsf{xv}[B]$, and $[\![\mathcal{F}^{\Sigma}_{X,F}[A]]\!]_j \approx_{\pi l} [\![\mathcal{F}^{\Sigma}_{X,F}[B]]\!]_j$.

The translation $A^{\circ\circ}$ differs from $\mathcal{F}^{\Sigma}_{X,F}[A]$ only in their respective output interface. The domain of F contains at least $\mathsf{fv}[A]$, but may also contain names that are fresh in A. Nonetheless, this additional part of the firewall cannot be accessed from A, and is thus inert. The two processes are therefore strongly equivalent in the join-calculus, and their translations are strongly equivalent in the π-calculus.

We show that $\mathcal{R}$ is a labeled bisimulation by a case analysis on source transitions. We detail only the case of intrusion—the cases of internal step and of extrusion are handled in the same manner, except that they involve different clauses of the operational correspondence lemmas. We let $x\langle\widetilde{v}\rangle$ be the intrusion label, and F' be the partial function that extends F with $\{\widetilde{w \mapsto v}\}$ for some fresh names $\widetilde{w}$. We have

$$\mathcal{F}^{\Sigma}_{X,F}[A] \quad \xrightarrow{x\langle\widetilde{v}\rangle} \quad \mathcal{F}^{\Sigma}_{X,F}[A] \mid x\langle\widetilde{v}\rangle \ \rightarrow^{*}_{d}\sim_d \ \mathcal{F}^{\Sigma}_{X,F'}[A \mid x\langle\widetilde{w}\rangle]$$

These transitions can be reported by operational correspondence of the translation $[\![\,\cdot\,]\!]_j$ into π-calculus transitions by Lemma 6.26(2,1):

$$[\![\mathcal{F}^{\Sigma}_{X,F}[A]]\!]_j \quad \xrightarrow{x\langle\widetilde{v}\rangle}\rightarrow^{*}\sim_{\pi l} \quad [\![\mathcal{F}^{\Sigma}_{X,F'}[A \mid x\langle\widetilde{w}\rangle]]\!]_j$$

(The equivalence $\sim_{\pi l}$ is used to discard translations of continuations after use, it is easily established as a strong labeled π-calculus bisimulation.) By definition of $\mathcal{R}$, these transitions can be weakly simulated in the π-calculus by some transitions

$$[\![\mathcal{F}^{\Sigma}_{X,F}[B]]\!]_j \quad \rightarrow^{*}\xrightarrow{x\langle\widetilde{v}\rangle}\rightarrow^{*} \quad T$$

with $[\![\mathcal{F}^{\Sigma}_{X,F'}[A \mid x\langle\widetilde{w}\rangle]]\!]_j \approx_{\pi l} T$. Applying the operational correspondence from the translation to the source calculus we have by Lemma 6.26(4,5)

$$\mathcal{F}^{\Sigma}_{X,F}[B] \quad \rightarrow^{*}\xrightarrow{x\langle\widetilde{v}\rangle}\rightarrow^{*}\sim_l \quad U$$

for some open process U such that $[\![U]\!]_j = T$, then by operational correspondence of the firewall (Lemma 6.11,

$$B \quad \rightarrow^* \xrightarrow{x\langle\widetilde{w}\rangle} \rightarrow^* B'$$

for some process B' such that $\mathcal{F}^{\Sigma}_{X,F'}[A \mid x\langle\widetilde{w}\rangle] \; \mathcal{R} \; \mathcal{F}^{\Sigma}_{X,F'}[B']$ □

Proof of Theorem 11: We obtain that the composition of encodings $[\![\cdot {}^{\circ\circ}]\!]_j$ preserves barbed congruence by composing Lemmas 6.13 and 6.27, and that it reflects barbed congruence by composing Lemmas 6.28 and the counterpart of Lemma 6.13 in the π-calculus. □

6.7 Proof of Theorem 10

We now present the detailed proofs of our full abstraction result between the π-calculus and the join-calculus. We rely on the techniques of bisimulation "up to" developed in Section 4.7.

We first prove the direct implication: $Q \approx_\pi R$ implies $\mathcal{E}_S[\![Q]\!]_\pi \approx \mathcal{E}_S[\![R]\!]_\pi$. To this end, we study how the refined translation interacts with an arbitrary join-calculus context. We begin with a few auxiliary definitions.

We consider translations in contexts of the form $\mathcal{P}_S[\,\cdot\,]$ where S ranges over finite sets of names. We say that a join-calculus process E is *valid* with regards to S when, for every name $x \in S$, the name x_l occurs in E only in evaluation context, in messages of the form $x_l\langle y_o, y_i\rangle$ for some $y \in S$.

We use an auxiliary translation $[\![\cdot]\!]$ that is very similar to $[\![\cdot]\!]_\pi$—Lemma 6.33 says that the two translations are equivalent—but that yields terms with a simpler behavior: every initial reduction in the translation can be associated with a source reduction in the π-process. The translation $[\![\cdot]\!]$ maps π-processes to join-processes using the same clauses as $[\![\cdot]\!]_\pi$ in Definition 6.23, except for scope restriction (so that communication always leads to the creation of a new proxy pair, even if it uses a local channel), and for outputs in evaluation contexts (where the deterministic definitions of x_o and p are unfolded):

$$\begin{array}{lcll} [\![\nu x.P]\!] & \stackrel{\text{def}}{=} & \mathcal{P}_x[[\![P]\!]] & \\ [\![\overline{x}\langle v\rangle]\!] & \stackrel{\text{def}}{=} & \mathcal{P}_z\left[x_l\langle z_o, z_i\rangle \mid [\![M^{\pi}_{v,z}]\!]_\pi\right] & \text{(in evaluation context)} \end{array}$$

In the following *deterministic reductions* $\rightarrow_d$ refer to reductions of messages emitted on p, x_l or κ in the translation, and *hybrid term* are join-calculus processes of the form $\mathcal{P}_S[E \mid [\![Q]\!]]$ where E is valid and $\mathsf{fv}[Q] \subseteq S$ (hence $[\![Q]\!]$ is also valid). The key technical property of hybrid terms is that they are closed by reduction up to deterministic reduction and $\asymp$, and also closed by application of an evaluation context that does not use the names bound in $\mathcal{P}_S[\,\cdot\,]$ as free names. In particular, the processes $\mathcal{E}_S\left[[\![Q]\!]_\pi\right]$ that appear in the theorem are hybrid terms for $E = \prod_{x \in S} x_e\langle x_o, x_i\rangle$.

The next lemma makes explicit the reductions that transform a message $x_o\langle s, t\rangle$ into a message $x_l\langle z_o, z_i\rangle$ after unfolding new relays between the name represented by the pair s, t and a new, correct encoding of a π-calculus name z.

Lemma 6.29 *Let S be a set of names, z be a fresh name, and $E[x_o\langle s,t\rangle]$ be a valid process where $E[\,\cdot\,]$ is an evaluation context that does not bind the names x_o, x_l, x_i, z_o, z_l, z_i, and p. We have the deterministic reductions*

$$\begin{aligned}\mathcal{P}_S\Big[E\big[x_o\langle s,t\rangle\big]\Big] &\rightarrow_d \mathcal{P}_S\Big[E\big[p\langle s,t,x_l\rangle\big]\Big]\\ &\rightarrow_d \mathcal{P}_S\Big[E\big[\mathcal{P}_z[x_l\langle z_o,z_i\rangle \mid [\![M^\pi_{y,z}]\!]\{^s/_{y_o},{}^t/_{y_i}\}]\big]\Big]\\ &\equiv \mathcal{P}_{S\uplus\{z\}}\big[x_l\langle z_o,z_i\rangle \mid E[[\![M^\pi_{y,z}]\!]\{^s/_{y_o},{}^t/_{y_i}\}]\big]\end{aligned}$$

Proof: we simply unfold the definitions of x_o and p in $\mathcal{P}_S[\,\cdot\,]$. □

Since the process $[\![M^\pi_{y,z}]\!]\{^s/_{y_o},{}^t/_{y_i}\}$ is valid with regards to $S\uplus\{z\}$ and has no message in evaluation context, we can iterate Lemma 6.29 to obtain deterministic reductions $\mathcal{P}_S[E] \rightarrow_d^*\equiv \mathcal{P}_{S'}[E']$ for some set S' that contains S and some process E' that is valid with regards to S', and that has no more message on names x_o for $x\in S'$ in evaluation context.

The next lemma relates hybrid terms that differ only in the number of unfoldings of synonyms in the context $\mathcal{P}_S[\,\cdot\,]$. This technical lemma is essential to fold back extraneous relays "on the fly" in the proofs that follow.

Lemma 6.30 *Let S be a set of names with $x\in S$ and $y\notin S$, and let E be a valid process with regards to $S\uplus\{y\}$. We have the expansion up to deterministic reductions*

$$\mathcal{P}_{S\uplus\{y\}}[E \mid [\![M^\pi_{x,y}]\!]] \;\succeq_d\; \mathcal{P}_S[E\{^{x_l}/_{y_l},{}^{x_o}/_{y_o},{}^{x_i}/_{y_i}\}]$$

Proof: We exhibit a relation $\mathcal{R}$ that is closed by reduction up to deterministic reductions and that contains all pairs of processes that appear in the lemma. Due to the recursive behavior of the context $\mathcal{P}_{S\uplus\{y\}}[\,\cdot\,]$, the candidate relation $\mathcal{R}$ is actually much larger: we let $\mathcal{R}$ contain all pairs of processes

$$P_1 = \mathcal{P}_S\Big[E \mid \prod_{(y,z)\in\phi} [\![M^\pi_{y,z}]\!]\Big], \qquad P_2 = \mathcal{P}_{S\sigma}\Big[E\sigma^\sharp\Big]$$

for all S, E, σ, $\sigma^\sharp$, and ϕ that meet the following requirements:

1. E is a valid join-calculus process with regards to S;
2. σ is an idempotent substitution on S
3. $\sigma^\sharp$ substitutes the names y_l, y_o, and y_i for x_l, x_o, and x_i whenever $y = x\sigma$.
4. ϕ is a minimal relation on S such that for all $x\in S$ we have $x\sigma\ \phi^*\ x$;

Intuitively, the covering relation ϕ describes the spanning tree of relays that have been unfolded so far out of the initial context $\mathcal{P}_{S\sigma}[\,\cdot\,]$; the minimality of ϕ is irrelevant in this proof. Also, note that $[\![P\sigma]\!] = [\![P]\!]\sigma^\sharp$. For all pairs of processes (P_1,P_2) for which the lemma claims an expansion up to deterministic reductions, we have $P_1\ \mathcal{R}\ P_2$ for the set $S\uplus\{y\}$, the substitution $\sigma=\{^x/_y\}$ and the relation $\phi=\{(x,y)\}$.

We apply the proof technique of Section 4.7.4 to prove that $\mathcal{R}$ is an expansion up to deterministic reductions. The strong barbs are the same on both sides of $\mathcal{R}$, because the translations $[\![M^\pi_{y,z}]\!]$ have no barbs and the substitution $\sigma^\sharp$ only operate on names that are bound in the enclosing contexts $\mathcal{P}_S[\,\cdot\,]$, $\mathcal{P}_{S\sigma}[\,\cdot\,]$. The congruence property is immediate for all contexts that do not have names defined in $\mathcal{P}_S[\,\cdot\,]$ as free variables.

To establish the bisimulation diagrams for $\mathcal{R}$, we partition reductions $P_1 \to P_1'$ and $P_2 \to P_2'$ as follows: reductions inside E; reductions using the definition of x_o in $\mathcal{P}_S[\,\cdot\,]$ or $x_o\sigma^\sharp$ in $\mathcal{P}_{S\sigma}[\,\cdot\,]$; reductions in P_1 using the definition of x_i, x_l in $\mathcal{P}_S[\,\cdot\,]$ (with x_i either in a relay or in E); reductions in P_2 using the definition of $x_i\sigma^\sharp, y_l\sigma^\sharp$ in $\mathcal{P}_{S\sigma}$ when $x\sigma = y\sigma$.

Reductions internal to E: these reductions use names that are not affected by the substitution $\sigma^\sharp$; they commute with $\sigma^\sharp$ and the resulting processes are related by $\mathcal{R}$.

Reductions using the definition of x_o in $\mathcal{P}_S[\,\cdot\,]$ or $x_o\sigma^\sharp$ in $\mathcal{P}_{S\sigma}[\,\cdot\,]$: these reductions are the first deterministic reductions of Lemma 6.29; we apply the two deterministic reductions and the structural equivalence of Lemma 6.29 on both P_1 and P_2. Let P_1'' and P_2'' be the resulting processes, and let z be the fresh name introduced by the lemma. We check that $P_1''\ \mathcal{R}\ P_2''$ for an updated set $S' = S \cup \{z\}$, an updated valid process E', and for the same σ and ϕ. Both expansion diagrams can thus be closed by the relations

$$P_1 \to_d P_1' \to_d \equiv \mathcal{R} \equiv \leftarrow_d P_2' \leftarrow_d P_2$$

Reductions between $y_l\langle x_o, x_i\rangle$ and $[\![M^\pi_{y,z}]\!]$ in P_1: these reductions correspond to message-passing between synonyms, and they cause a new relay to be unfolded in P_1 only. We have the series of relations

$$\begin{array}{rcl}
P_1 & \equiv & \mathcal{P}_S\left[E' \mid y_l\langle x_o, x_i\rangle \mid \prod_{(y,z)\in\phi} [\![M^\pi_{y,z}]\!]\right] \\
 & \equiv\to\to_d & \mathcal{P}_S\left[E' \mid z_o\langle x_o, x_i\rangle \mid \prod_{(y,z)\in\phi} [\![M^\pi_{y,z}]\!]\right] \\
 & \to_d\to_d\equiv & \mathcal{P}_{S\uplus\{x'\}}\left[E' \mid \left(z_l\langle x_o', x_i'\rangle \mid [\![M^\pi_{x,x'}]\!]\right) \mid \prod_{(y,z)\in\phi} [\![M^\pi_{y,z}]\!]\right] \\
 & = & \mathcal{P}_{S'}\left[E' \mid z_l\langle x_o', x_i'\rangle \mid \prod_{(y,z)\in\phi'} [\![M^\pi_{y,z}]\!]\right] \quad \mathcal{R} \quad P_2
\end{array}$$

The first reduction step of (6.13) is the reduction $P_1 \to P_1'$; it joins the message $y_l\langle x_o, x_i\rangle$ with the message $y_i\langle\kappa\rangle$ present in the replicated receiver $!y(u).\overline{z}\langle u\rangle$ within the relays. Then, the deterministic reduction consumes the continuation $\kappa\langle x_o, x_i\rangle$, which puts back the relay in its initial state, and also releases the message $z_o\langle x_o, x_i\rangle$. Next the relations (6.13) are obtained by applying Lemma 6.29 on the message $z_o\langle x_o, x_i\rangle$.

The identity (6.13) holds for the updated components $S' = S \uplus \{x'\}$, $E'' = E' \mid z_l\langle x_o', x_i'\rangle$, $\sigma' = \sigma \circ \{^{x\sigma}/_{x'}\}$, and $\phi' = \phi \cup \{(x, x')\}$. The resulting process meets all the invariants. Moreover, this process is still related to P_2: we have by construction

$(x, x') \in \phi$, and, since there is a relay from y to z in P_1, either $(y, z) \in \phi$ or $(z, y) \in \phi$. That is, $x\sigma' = x'\sigma'$, $y\sigma' = y\sigma$, and finally $(E' \,|\, y\langle x_o, x_i\rangle)\sigma'^\sharp = (E' \,|\, z\langle x'_o, x'_i\rangle)\sigma'^\sharp$. The reduction $P_1 \rightarrow P'_1$ is therefore simulated by the absence of reduction in P_2:

$$P_1 \rightarrow P'_1 \rightarrow_d \rightarrow_d \rightarrow_d \equiv \mathcal{R}\ P_2$$

Reductions joining $y_l\langle x_o, x_i\rangle$ and $y_i\langle\kappa\rangle$ within P_1: these reductions commute with the application of $\sigma^\sharp$; they are in direct correspondence with reductions $P_2 \rightarrow P'_2$, and yield pairs of processes $P'_1\ \mathcal{R}\ P'_2$ for some updated valid E.

Reductions joining $(y_l\langle x_o, x_i\rangle)\sigma^\sharp$ and $(z_i\langle\kappa\rangle)\sigma^\sharp$ within P_2: we necessarily have $y\sigma = z\sigma$, but the names y and z may be different. By definition of ϕ, however, we have the relations $y\ (\phi^{-1})^*\ y\sigma = z\sigma\ \phi^*\ z$. Hence, there is a chain of relays that can forward the message from y_l to z_l, which unfolds a new relay for every relay that is crossed on the way. Starting from P_1, we successively apply the series of relations (6.13–6.13) for each relay of the chain. This leads to a process $P'_1\ \mathcal{R}\ P_2$ for some larger S, σ and ϕ, and a message $z_l\langle x'_o, x'_i\rangle$ with $x\sigma = x'\sigma$ in place of the original message $y_l\langle x_o, x_i\rangle$ in E.

After these preliminary steps, we are back to the previous case for the pair $P'_1\ \mathcal{R}\ P_2$, with a direct correspondence between the reduction $P'_1 \rightarrow P''_1$ that consumes the two messages on z_l, z_i and the reduction $P_2 \rightarrow P'_2$ that consumes their image by $\sigma^\sharp$. We close the diagram with the relations

$$P_1\ \left(\rightarrow (\rightarrow_d \rightarrow_d \rightarrow_d \equiv)\right)^* \rightarrow \mathcal{R}\ \ P'_2 \leftarrow P_2$$

$\square$

The next lemma is a variation of Lemma 6.24; it states that reductions in the π-calculus can be transported on their translations in hybrid terms.

Lemma 6.31 *Let S be a set of names, Q be a π-calculus process with $\mathsf{fv}[Q] \subseteq S$, and E be a valid join-calculus process.*

$$\text{if } Q \rightarrow Q', \quad \text{then} \quad \mathcal{P}_S\,[E \,|\, [\![Q]\!]] \rightarrow (\rightarrow_d^* \sim \succeq_d)^* \mathcal{P}_S\,[E \,|\, [\![Q']\!]]$$

Proof: There are two cases, depending on whether the receiver is replicated or not. We detail only the case of a non-replicated receiver. Using structural equivalence, we rewrite the source reduction as

$$Q \equiv \nu\widetilde{z}.(\overline{x}\langle v\rangle \,|\, x(u).G \,|\, T) \quad\rightarrow\quad Q' \equiv \nu\widetilde{z}.(G\{^v/_u\} \,|\, T)$$

where the names in $\widetilde{z}$ and their translations are fresh in the hybrid term $\mathcal{P}_S[E]$. In the join-calculus, we have the relations

$$\begin{array}{rll}
\mathcal{P}_S[E \,|\, [\![Q]\!]] \equiv & \mathcal{P}_{S\uplus\{\widetilde{z}\}}[E \,|\, [\![\overline{x}\langle v\rangle]\!] \,|\, [\![x(u).G \,|\, T]\!]] & (6.13)\\
\equiv & \mathcal{P}_{S\uplus\{\widetilde{z}\}\uplus\{v'\}}[E \,|\, x_l\langle v'_o, v'_i\rangle \,|\, [\![M^\pi_{v,v'}]\!] \,|\, [\![x(u).G]\!] \,|\, [\![T]\!]] & (6.14)\\
\rightarrow\rightarrow_d\sim & \mathcal{P}_{S\uplus\{\widetilde{z}\}\uplus\{v'\}}[E \,|\, E' \,|\, [\![M^\pi_{v,v'}]\!] \,|\, [\![T]\!]] & (6.15)\\
(\rightarrow_d\rightarrow_d\equiv)^* & \mathcal{P}_{S\uplus\{\widetilde{z}\}\uplus\{v'\}}[E \,|\, [\![G\{^{v'}/_u\}]\!] \,|\, [\![M^\pi_{v,v'}]\!] \,|\, [\![T]\!]] & (6.16)\\
\succeq_d & \mathcal{P}_{S\uplus\{\widetilde{z}\}}[E \,|\, [\![(G\{^{v'}/_u\})\{^v/_{v'}\}]\!] \,|\, [\![T]\!]] & (6.17)\\
\equiv & \mathcal{P}_S[E \,|\, [\![Q']\!]] & (6.18)
\end{array}$$

(where the name v' is fresh). The second structural equivalence (6.14) is the same as in Lemma 6.29. The series $\rightarrow\rightarrow_d\sim$ (6.15) first consumes the pair of messages $x_l\langle v'_o, v'_i\rangle$ and $x_i\langle\kappa\rangle$, then the continuation message $\kappa\langle v'_o, v'_i\rangle$, and finally it discards the definition of κ. The resulting subprocess E' is the translation $[\![G\{^{v'}/_u\}]\!]$ except for the presence of messages $y_o\langle z_o, z_i\rangle$ instead of $[\![y\langle z\rangle]\!]$ for every π-calculus message of $G\{^{v}/_u\}$ in evaluation context. The series $(\rightarrow_d\rightarrow_d\equiv)^*$ (6.16) repeatedly applies Lemma 6.29 for every such message, thus fixing the difference. The expansion up to deterministic reductions (6.17) is obtained by applying Lemma 6.30; it folds back the new synonym v' of v; since v' appears only in $G\{^{v'}/_u\}$, the substitution $\{^{v}/_{v'}\}^\sharp$ affects only the translation of this term. The final structural equivalence (6.18) restores the initial structure of the term by restricting the scopes of the names $\widetilde{z}$ and their translations.

The case of a replicated receiver is almost the same; the main difference is that we use structural equivalence to restore the initial replicated receiver instead of strong equivalence to discard the definition of the continuation. $\square$

We are now ready to establish the soundness of the modified encoding $[\![\,\cdot\,]\!]$.

Lemma 6.32 *For all sets of names S, valid join-calculus processes E, and pairs of π-calculus processes Q and R with $\mathsf{fv}[Q] \cup \mathsf{fv}[R] \subseteq S$,*

$$\textit{if } Q \approx_\pi R, \quad \textit{then} \quad \mathcal{P}_S\,[E \,|\, [\![Q]\!]] \;\approx\; \mathcal{P}_S\,[E \,|\, [\![R]\!]]$$

Proof: Let $\mathcal{R}$ be the relation that contains all pairs of processes (P_Q, P_R) for which the lemma claims $P_Q \approx P_R$. Let also $\mathcal{R}' \subseteq \mathcal{R}$ be the auxiliary relation that contains only the pairs (P_Q, P_R) that meet the additional requirement: there is no message $x_o\langle s,t\rangle$ in evaluation contexts in E' $(\forall x \in S', E' \not\downarrow_{x_o})$.

We easily obtain the inclusion $\mathcal{R} \subseteq \rightarrow_d^*\equiv\mathcal{R}'\equiv\leftarrow_d^*$: let $P_Q\ \mathcal{R}\ P_R$. By applying Lemma 6.29 for every message $x_o\langle s,t\rangle$ in evaluation context in E, we have the relations $P_Q \rightarrow_d^* P'_Q\ \mathcal{R}\ P'_R \leftarrow_d^* P_R$ where the new related processes are obtained for a larger set S and an updated valid process E that meets the additional requirement.

Thus it suffices to prove that $\mathcal{R}' \subseteq \approx$, and moreover we can close bisimulation diagrams by using $\mathcal{R}$ instead of $\mathcal{R}'$. We apply the proof technique of Section 4.7.4 (barbed congruence up to expansion and deterministic reductions).

The barb property of $\mathcal{R}'$ is easily checked, because processes related by $\mathcal{R}'$ have strong barbs on the same names: by hypothesis $\mathsf{fv}[Q] \cup \mathsf{fv}[R] \subseteq S$, hence all the free names of the translations $[\![Q]\!]$ and $[\![R]\!]$ are bound in the context $\mathcal{P}_S[\,\cdot\,]$. That is, messages on free names may only appear in the valid process E, and they are syntactically the same on both sides of $\mathcal{R}'$.

The congruence property of $\mathcal{R}'$ is immediate for all evaluation contexts $C[\,\cdot\,]$ of the join-calculus with no free names that are bound in $\mathcal{P}_S[\,\cdot\,]$ $C[\mathcal{P}_S[E \,|\, X]] \equiv \mathcal{P}_S[C[E] \,|\, X]$ for all X with $\mathsf{fv}[\mathcal{P}_S[X]] = \emptyset$.

The weak bisimulation properties of $\mathcal{R}'$ are established by a detailed analysis of reductions. We partition the reductions and, for each family of reductions, we close the required bisimulation diagram "up to". Since $\mathcal{R}'$ is symmetric, we focus on the reductions $P_Q \rightarrow P'_Q$, which are partitioned as follows: reductions internal to E; reductions internal to the translation $[\![Q]\!]$; reductions using a rule $x_l\langle y_o, y_i\rangle \,|\, x_i\langle\kappa\rangle \triangleright \kappa\langle y_o, y_i\rangle$ defined within $\mathcal{P}_S[\,\cdot\,]$ for some $x \in S$. The message on x_i represents "input"; the message on x_l represents "output". This latter family of reductions is split into four subcases, according to the position of the input and output messages being consumed, in E or in $\llbracket Q \rrbracket$.

Reduction internal to E; input and output both in E: The same reduction occurs on both sides, in different contexts, and leads to an updated E' that is still valid. The two resulting processes are related by $\mathcal{R}$. We close the bisimulation diagram by using the inclusion $\mathcal{R} \subseteq \rightarrow_d^* \equiv \mathcal{R}' \equiv \leftarrow_d^*$.

Reduction internal to $\llbracket Q \rrbracket$ (reduction on a bound name in Q): This reduction uses the translation of a binder νx in the source process, and joins two messages $x_o\langle v_o, v_i \rangle$ and $x_i\langle \kappa \rangle$ that are translations of an emission and a reception in the π-calculus, respectively. The corresponding reduction in the π-calculus is of the form:

$$Q \equiv \nu \widetilde{z}, x.(\overline{x}\langle v \rangle \mid x(u).G \mid Q'') \quad \rightarrow \quad Q' \equiv \nu \widetilde{z}, x.(G\{^v/_u\} \mid Q'')$$

(or a similar form in case of replicated reception.) By definition of $\mathcal{R}'$ we have $Q \approx_\pi R$, and thus the source reduction in Q can be simulated by a series of reductions starting from R:

$$\begin{tikzcd}
Q \arrow[r, dash, "\approx_\pi"] \arrow[d] & R \arrow[d, dotted, "n"] \\
Q' \arrow[r, dash, dotted, "\approx_\pi"] & R'
\end{tikzcd}$$

We apply Lemma 6.31 for every reduction that appears in the source diagram above, once on the left where the initial reduction in the join-calculus is precisely $P_Q \rightarrow P_Q'$, and n times on the right, where n is the length of the derivation.

$$\begin{array}{rcl}
P_Q \rightarrow P_Q' & (\rightarrow_d^* \sim \succeq_d)^* & \mathcal{P}_S[E \mid \llbracket Q' \rrbracket] \\
P_R & \left(\rightarrow (\rightarrow_d^* \sim \succeq_d)^*\right)^n & \mathcal{P}_S[E \mid \llbracket R' \rrbracket]
\end{array}$$

and we close the bisimulation diagram in the join-calculus as follows:

$$\begin{tikzcd}
P_Q \arrow[rr, dash, "\mathcal{R}'"] \arrow[d] & & P_R \arrow[d, dotted, "\left(\rightarrow (\rightarrow_d^* \sim \succeq_d)^*\right)^n"] \\
P_Q' \arrow[r, dash, dotted, "(\rightarrow_d^* \sim \succeq_d)^*"] & \mathcal{P}_S\left[E \mid \llbracket Q' \rrbracket\right] \arrow[r, dash, dotted, "\mathcal{R}'"] & \mathcal{P}_S\left[E \mid \llbracket R' \rrbracket\right]
\end{tikzcd}$$

Input and output in $\llbracket Q \rrbracket$ (reduction on a free name in Q): This case is handled exactly as the above case, except that there is no binder for x in Q. Again, we report a simple π-calculus bisimulation diagram using Lemma 6.31 to obtain the same bisimulation diagram as above.

Input in E, output in $\llbracket Q \rrbracket$ (extrusion from Q): The translation $\llbracket Q \rrbracket$ contains a message $x_l\langle y_o, y_i \rangle$, which correspond to an emission on x in the source process. Using structural rearrangement, we have $Q \equiv \nu u.(Q' \mid \overline{x}\langle y \rangle)$ where the prefix νu is either nothing or νy, according to the scope of the transmitted channel y.

We extract from the process R a message $\overline{x}\langle y' \rangle$ that is equivalent to $\overline{x}\langle y \rangle$ in this context. To this end, we apply the congruence property in the π-calculus to $Q \approx_\pi R$, for the particular context

$$\mathcal{O}[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \overline{t}\langle t \rangle \mid x(y).t(t).M^\pi_{yz} \mid [\,\cdot\,]$$

where t is a π-calculus names that does not appear in Q or R. Intuitively, this context grabs a single message $\overline{x}\langle y \rangle$ and equates its contents to the name z; the progress can be detected thanks to the barb on t, which disappears inasmuch as the context successfully grabs a message.

By bisimilarity in this context, we have the source diagram

$$\begin{tikzcd}
\mathcal{O}\left[\nu u.(Q'' \mid \overline{x}\langle y \rangle)\right] \arrow[rr, dash, "\approx_\pi"] \arrow[d, "2"] & & \mathcal{O}\left[R\right] \arrow[d, dotted, "n+2"] \\
Q' = \nu u.(Q'' \mid M^\pi_{yz}) \arrow[rr, dotted, dash, "\approx_\pi"] & & R'
\end{tikzcd}$$

where the two reductions on the left consist of the two receptions in $\mathcal{O}[\,\cdot\,]$. We infer the length of the derivation $R \rightarrow^* R'$ and the shape of R': since the resulting process has lost its barb on t, these two reductions must also occur on the right; moreover, these two reductions commute with any subsequent reduction on the right—except for relaying reductions which are reversible—hence we can reorder the $n + 2$ reductions on the right so that they appear last. Finally, the n first reductions do not depend on $\mathcal{O}[\,\cdot\,]$, so we have for some R'' in the π-calculus

$$\begin{array}{rcl} R & \rightarrow^* & \nu u'.(R'' \mid \overline{x}\langle y' \rangle) \\ R' & \equiv & \nu u'.(R'' \mid M^\pi_{y'z}) \end{array}$$

(where again $\nu u'$ may be empty). We report these n π-calculus reductions to the join-calculus by using Lemma 6.31, then we glue the result with the join-calculus reduction that receives the message $x_l\langle z_o, z_i \rangle$ in E and yields the same updated process E' on both sides. The two resulting processes are in $\mathcal{R}'$ for the source processes Q' and R'. We obtain the diagram:

$$\begin{tikzcd}
\mathcal{P}_S\left[E[x_i\langle\kappa\rangle] \mid [\![Q]\!]\right] \arrow[rr, dash, "\mathcal{R}'"] \arrow[dd, dash, "\equiv"] & & \mathcal{P}_S\left[E[x_i\langle\kappa\rangle] \mid [\![R]\!]\right] \arrow[d, dotted, dash, "\left(\rightarrow(\rightarrow_d^*\sim\succeq_d)^*\right)^n"] \\
& & \mathcal{P}_S\left[E[x_i\langle\kappa\rangle] \mid [\![\nu u'.(R'' \mid \overline{x}\langle y' \rangle)]\!]\right] \arrow[d, dotted, dash, "\equiv"] \\
\mathcal{P}_{S\uplus\{z\}}\left[E[x_i\langle\kappa\rangle] \mid x_l\langle z_o, z_i \rangle \mid [\![Q']\!]\right] \arrow[d] & & \mathcal{P}_{S\uplus\{z\}}\left[E[x_i\langle\kappa\rangle] \mid x_l\langle z_o, z_i \rangle \mid [\![R']\!]\right] \arrow[d, dotted] \\
\mathcal{P}_{S\uplus\{z\}}\left[E[\kappa\langle z_o, z_i \rangle] \mid [\![Q']\!]\right] \arrow[rr, dotted, dash, "\mathcal{R}'"] & & \mathcal{P}_{S\uplus\{z\}}\left[E[\kappa\langle z_o, z_i \rangle] \mid [\![R']\!]\right]
\end{tikzcd}$$

Input in $\llbracket Q \rrbracket$, output $x_l\langle z_o, z_i\rangle$ in E (intrusion in Q): Let us assume that $Q \equiv \nu\widetilde{u}.(x(u).G \mid Q'')$ where the translation of the subprocess $x(u).G$ contains the message on x_i being consumed in the reduction. We use bisimilarity in the π-calculus for the context $\overline{x}\langle z\rangle \mid [\,\cdot\,]$; we obtain the source diagram

$$\begin{tikzcd}
\overline{x}\langle z\rangle \mid \nu\widetilde{u}.(x(u).G \mid Q'') \arrow[rr, dash, "\approx_\pi"] \arrow[d] & & \overline{x}\langle z\rangle \mid R \arrow[d, dotted, "*"] \\
Q' = \nu\widetilde{z}, x.(G\{^z/_u\} \mid Q'') \arrow[rr, dash, dotted, "\approx_\pi"] & & R'
\end{tikzcd}$$

and we report its reductions in the join-calculus. On the left, we have

$$\begin{array}{rl} & P_Q = \mathcal{P}_S[E \mid x_l\langle z_o, z_i\rangle \mid \llbracket \nu\widetilde{u}.(x(u).G \mid Q'')\rrbracket] \\ \rightarrow & P'_Q \rightarrow_d \sim (\rightarrow_d \rightarrow_d \equiv)^* \ \mathcal{P}_S[E \mid \llbracket Q' \rrbracket] \end{array}$$

where the relations are obtained as in the proof of lemma 6.31. Note that we do not remove a relay on z as in that lemma, because z may be relayed to a pair of names that are not indexed by S.

On the right, we first apply Lemma 6.31 to every reduction step in the series starting from $\overline{x}\langle z\rangle \mid R$. We obtain

$$P_R = \mathcal{P}_S[E \mid \llbracket \overline{x}\langle z\rangle \mid R \rrbracket \quad \left(\rightarrow (\rightarrow_d^* \sim \succeq_d)^*\right)^* \quad \mathcal{P}_S[E \mid \llbracket R' \rrbracket$$

and, since $Q' \approx_\pi R'$, we have a pair of processes in $\mathcal{R}'$ for these two π-calculus processes and for an updated E where the message on z has been removed.

Note, however, that $\llbracket \overline{x}\langle z\rangle \rrbracket \not\equiv x_l\langle z_o, z_i\rangle$, even as $x, z \in S$. Before gluing the translated relations on the right, we first apply Lemma 6.30 in the "expanding" direction, to unfold a relay around $x_l\langle z_o, z_i\rangle$:

$$\begin{array}{rcl} P_R = \mathcal{P}_S[E \mid x_l\langle z_o, z_i \mid \llbracket R \rrbracket] & \preceq_d & \mathcal{P}_S\Big[E \mid \mathcal{P}_{z'}\left[x_l\langle z'_o, z'_i \mid \llbracket M^\pi_{z,z'} \rrbracket\right] \mid \llbracket R \rrbracket\Big] \\ & \equiv & \mathcal{P}_S[E \mid \llbracket \overline{x}\langle z\rangle \mid R \rrbracket] \end{array}$$

We finally close the diagram

$$\begin{tikzcd}
P_Q \arrow[r, dash, "\mathcal{R}'"] \arrow[d] & P_R \arrow[r, dash, dotted, "\preceq_d"] & \mathcal{P}_S[E \mid \llbracket \overline{x}\langle z\rangle \mid R \rrbracket] \arrow[d, dash, dotted, "\left(\rightarrow(\rightarrow_d^* \sim \succeq_d)^*\right)^*"] \\
P'_Q \arrow[rr, dash, dotted, "\rightarrow_d \sim (\rightarrow_d \rightarrow_d \equiv)^* \ \mathcal{R}'"] & & {}
\end{tikzcd}$$

□

Next we use previous results to relate the auxiliary translation $\llbracket \cdot \rrbracket$ used in this proof to the original translation that appears in the theorem.

Lemma 6.33 *Let S be a finite set of names and Q be a process of the π-calculus such that $\mathsf{fv}[Q] \subseteq S$. We have the barbed expansion up to deterministic reductions*

$$\mathcal{E}_S\left[[\![Q]\!]\right] \succeq_d \mathcal{E}_S\left[[\![Q]\!]_\pi\right]$$

Proof: Let $\mathcal{R}$ be the relation that contains all pairs of join-calculus processes (P_1, P_2) of the form

$$P_1 = \mathcal{P}_S[E \mid [\![Q]\!]], \qquad P_2 = \mathcal{P}_S[E \mid [\![Q]\!]_\pi]$$

where S is a finite set of names, Q is a π-calculus process with $\mathsf{fv}[Q] \subseteq S$, and E is valid with regards to S. We obtain the statement of the lemma in the case $E = \prod_{x \in S} x_e\langle x_o, x_i\rangle$.

We work up to deterministic reductions: we assume that all reductions on x_o defined in a $\mathcal{P}_x$ (either for $x \in S$ or for a local x in $[\![Q]\!]$) have been performed by repeatedly applying Lemma 6.29. However, we retain the notation $[\![\,\cdot\,]\!]_\pi$ to denote these processes.

We use Lemma 4.7.4 (expansion up to deterministic reductions) to prove the inclusion $\mathcal{R} \subseteq \succeq_d$. The requirements on barbs and contexts are easily checked, as above. The case analysis on reductions is simple because we have the same source process on both sides. Almost all reductions are in direct correspondence in P_1 and in P_2, and the resulting processes are still in $\mathcal{R}$, possibly after performing a few deterministic reductions: reductions that trigger relays in $[\![Q]\!]$ and in E are followed by a deterministic reduction step using the definition of p, then structural rearrangement; other reductions in E are the same and remain in $\mathcal{R}$; reductions using a definition of x_o and x_i for $x \in S$ are followed by a few deterministic steps plus strong equivalence on both sides.

The only family of reductions that differ are reductions that use the definitions of x_l, x_o, x_i for some name x that is bound in Q. After rearranging the source process in the π-calculus, let us assume that $Q \equiv \nu\widetilde{z}.(\overline{x}\langle v\rangle \mid x(u).G \mid R)$ where the translations of the names $\widetilde{z}$, u do not appear in $\mathcal{P}_S[E]$.

Let $P_i \rightarrow\rightarrow_d^{*}\sim P_i'$ for $i = 1, 2$ be the two reductions that consume the corresponding messages in the translations, followed by the deterministic reduction that triggers the continuation and the strong equivalence that discards the definition of the continuation. The processes P_i' are the translations of two updated, distinct source processes Q_i', in the same context (with identical S and E):

$$\begin{array}{rcl} Q_1' & \equiv & \nu\widetilde{z}.(\nu w.(G\{^w/_u\} \mid M^\pi_{vw}) \mid R) \\ Q_2' & \equiv & \nu\widetilde{z}.(G\{^v/_u\} \mid R) \end{array}$$

(where w is a fresh name in P_1 and P_2). However, we can use structural equivalence in the resulting process to lift the translation of the binders $\widetilde{z}, w$ at top-level, then apply Lemma 6.30:

$$\begin{array}{rcl} P_1' & \equiv & \mathcal{P}_{S \uplus \{\widetilde{z}, w\}}[E \mid [\![G\{^w/_u\} \mid R]\!] \mid [\![M^\pi_{vw}]\!]] \\ & \succeq_d & \mathcal{P}_{(S \uplus \{\widetilde{z}, w)\{^v/_w\}}[(E \mid [\![G\{^w/_u\} \mid R]\!])\{^v/_w\} \mid] \\ & = & \mathcal{P}_{S \uplus \{\widetilde{z}\}}[E \mid [\![G\{^v/_u\} \mid R]\!]] \\ & \equiv & \mathcal{P}_S[E \mid [\![Q_2']\!]] \ = l P_2' \end{array}$$

we close the two bisimulation diagrams for expansion up to deterministic reductions by using the relations

$$P_1 \rightarrow (\rightarrow_d^* \sim \succeq_d \mathcal{R} \sim \leftarrow_d^*) \leftarrow P_2$$

$\square$

Proof of soundness: For a given S, we apply Lemma 6.32 for the process $E = \prod_{x \in S} x_e\langle x_o, x_i\rangle$, which is clearly valid, then substitute the translation $[\![\,\cdot\,]\!]_\pi$ of the theorem for the auxiliary translation $[\![\,\cdot\,]\!]$ of the lemma by applying Lemma 6.33 on both sides of the barbed congruence. $\square$

We now prove that the translation reflects barbed congruence; we define yet another auxiliary translation $[\![\![\,\cdot\,]\!]\!]$ that is fully compositional. At each step of the new structural definition, each term is wrapped in a protective context that prevents any sharing of translated names; instead, these distinct names in use in each subterm are made synonyms.

Definition 6.34 *The translation $[\![\![\,\cdot\,]\!]\!]$ maps π-calculus processes to join-calculus processes, and (implicitly) collects free names. It is defined on the structure of π-calculus processes as follows:*

$$\begin{array}{rcl}
[\![\![P \mid Q]\!]\!] & \overset{def}{=} & \mathcal{E}_{\mathsf{fv}[P] \cup \mathsf{fv}[Q]}\left[\mathcal{I}_{\mathsf{fv}[P]}[[\![\![P]\!]\!]] \mid \mathcal{I}_{\mathsf{fv}[Q]}[[\![\![Q]\!]\!]]\right] \\
[\![\![\nu x.P]\!]\!] & \overset{def}{=} & \mathcal{N}_x[[\![\![P]\!]\!]] \\
[\![\![\overline{x}\langle v\rangle]\!]\!] & \overset{def}{=} & \mathcal{E}_{\{x,v\}}\left[x_o\langle v_o, v_i\rangle\right] \\
[\![\![x(v).P]\!]\!] & \overset{def}{=} & \mathcal{E}_{\mathsf{fv}[P] \cup \{x\}}\left[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright \mathcal{I}_{\mathsf{fv}[P]}[[\![\![P]\!]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle\right] \\
[\![\![!x(v).P]\!]\!] & \overset{def}{=} & \mathcal{E}_{\mathsf{fv}[P] \cup \{x\}}\left[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid \mathcal{I}_{\mathsf{fv}[P]}[[\![\![P]\!]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle\right]
\end{array}$$

where the contexts $\mathcal{I}_S$ and $\mathcal{N}_x$ are defined by

$$\begin{array}{rcl}
\mathcal{I}_S\,[\,\cdot\,] & \overset{def}{=} & \mathsf{def}\ \bigwedge_{x \in S} x_e\langle y_o, y_i\rangle \triangleright [\![M^\pi_{x,y}]\!]_\pi\ \mathsf{in}\ [\,\cdot\,] \\
\mathcal{N}_x\,[\,\cdot\,] & \overset{def}{=} & \mathsf{def}\ x_e\langle y_o, y_i\rangle \triangleright 0\ \mathsf{in}\ [\,\cdot\,]
\end{array}$$

Intuitively, $\mathcal{I}_S[\,\cdot\,]$ catches exported names for channels in the scope of the context and equates them to names in the context immediately above; $\mathcal{N}[\,\cdot\,]$ catches and discards exported names, thus providing scope restriction.

The next lemma relates our two auxiliary encodings.

Lemma 6.35 *For all processes Q in the π-calculus, we have $[\![\![Q]\!]\!] \approx \mathcal{E}_{\mathsf{fv}[Q]}[\![Q]\!]$.*

Proof: The wrappers $\mathcal{E}$ and $\mathcal{I}$ are somehow inverse up to expansion: for all valid processes P and Q, nested wrappers can be simplified as follows

$$\begin{array}{rcl}
\mathcal{E}_{\{x\}}\left[P \mid \mathcal{I}_x[\mathcal{E}_x\left[Q\right]]\right] & \equiv & \mathcal{E}_{\{x\}}\left[P \mid \mathcal{I}_{x'}\mathcal{E}_{x'}\left[Q\{^{x'}/_{x}\}\right]\right] \\
& \rightarrow_d\sim & \mathcal{P}_{\{x,x'\}}\left[x_e\langle x_o, x_i\rangle \mid M^\pi_{x,x'} \mid P \mid Q\{^{x'}/_{x}\}\right] \\
& \succeq_d & \mathcal{P}_{\{x\}}\left[x_e\langle x_o, x_i\rangle \mid P \mid Q\right] \equiv \mathcal{E}_{\{x\}}[P \mid Q]
\end{array}$$

(where x' is a fresh name used to avoid name-clashes in the internal definition of x_o, x_l, x_i.) The expansion up to deterministic reductions is obtained by Lemma 6.30.

We prove the lemma by structural induction on the π-calculus process Q. In each case of the grammar, we repeatedly use variants of this simplification. We omit the indices when they are clear from the context.

$$\begin{array}{rcl}
[\![P \mid Q]\!] & \overset{\text{def}}{=} & \mathcal{E}_{\ldots}\,[\mathcal{I}_{\ldots}[\![P]\!] \mid \mathcal{I}_{\ldots}[\![Q]\!]] \\
& \approx & \mathcal{E}_{\ldots}\,[\mathcal{I}_{\ldots}\mathcal{E}_{\ldots}[\![P]\!] \mid \mathcal{I}_{\ldots}\mathcal{E}_{\ldots}[\![Q]\!]] \\
& \approx & \mathcal{E}_{\ldots}[\![P \mid Q]\!] \\
\\
[\![\nu x.P]\!] & \overset{\text{def}}{=} & \mathcal{N}_x[\![P]\!] \\
& \approx & \mathcal{N}_x\mathcal{E}_{\mathsf{fv}[P]}[\![P]\!] \\
& \equiv & \mathcal{E}_{\mathsf{fv}[P]\setminus\{x\}}\mathcal{N}_x\mathcal{E}_x\,[[\![P]\!]] \\
& \rightarrow_d\sim & \mathcal{E}_{\mathsf{fv}[P]\setminus\{x\}}\mathcal{P}_x\,[[\![P]\!]] \\
& \approx & \mathcal{E}_{\mathsf{fv}[\nu x.P]}\,[[\![\nu x.P]\!]] \\
\\
[\![\overline{x}\langle v\rangle]\!] & \overset{\text{def}}{=} & \mathcal{E}_{\{x,v\}}\,[[\![\overline{x}\langle v\rangle]\!]] \\
\\
[\![x(v).P]\!] & \overset{\text{def}}{=} & \mathcal{E}_{\ldots}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright \mathcal{I}_{\ldots}[[\![P]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle] \\
& \approx & \mathcal{E}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright \mathcal{I}_{\ldots}\mathcal{E}\,[[\![P]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle] \\
& \approx & \mathcal{E}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle = [\![P]\!]\ \mathsf{in}\ x_i\langle\kappa\rangle] \\
\\
[\![!x(v).P]\!] & \overset{\text{def}}{=} & \mathcal{E}_{\ldots}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid \mathcal{I}_{\ldots}[[\![P]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle] \\
& \approx & \mathcal{E}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid \mathcal{I}_{\ldots}\mathcal{E}\,[[\![P]\!]]\ \mathsf{in}\ x_i\langle\kappa\rangle] \\
& \approx & \mathcal{E}\,[\mathsf{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid [\![P]\!]\ \mathsf{in}\ x_i\langle\kappa\rangle]
\end{array}$$

□

The next lemma show that the auxiliary encoding encoding $[\![\,\cdot\,]\!]$ reflects all equivalences in the join-calculus.

Lemma 6.36 *Let S be a finite set of names. For all π-calculus processes Q and R, if $\mathcal{E}_S[\![Q]\!] \approx \mathcal{E}_S[\![R]\!]$, then $Q \approx_\pi R$.*

Proof: Let $\mathcal{R}$ be the relation that contains all pairs of processes (Q, R) such that, for some set S, we have $\mathcal{E}_S[[\![Q]\!]] \approx \mathcal{E}_S[[\![R]\!]]$. It is not necessary to require that all free variables of the source processes be present in S, because we can always apply the congruence property in the join-calculus to extend S. Conversely, if a name occurs in S but not in Q, R, then the weak bisimulation also holds for $S \setminus \{x\}$.

We prove that $\mathcal{R} \subseteq \approx_\pi$ by establishing that $\mathcal{R}$ is a barbed congruence in the π-calculus.

1. the congruence for all evaluation contexts is a combination of Lemma 6.35 and of the congruence property of $\approx$. Up to strong bisimilarity, we can assume that $\mathsf{fv}[Q] = \mathsf{fv}[R]$. Let $C[\,\cdot\,]$ be an evaluation context in the π-calculus. We have

$$\mathcal{E}_{\mathsf{fv}[C[Q]]}\,[[\![\mathcal{C}\,[Q]\,]\!]] \approx [\![\mathcal{C}\,[Q]\,]\!] = ([\![\mathcal{C}]\!])\,[[\![Q]\!]] \approx ([\![\mathcal{C}]\!])\,[\mathcal{E}_{\mathsf{fv}[Q]}[\![Q]\!]]$$

where the two barbed congruence are obtained by Lemma 6.35. We conclude by applying the congruence property of $\approx$ in the join-calculus for the context $(\llbracket C \rrbracket)[\,\cdot\,]$.

2. every weak output barb $\Downarrow_{\overline{x}}$ of the π-calculus can be tested on $\mathcal{E}_S\llbracket Q \rrbracket$ as soon as $x \in S$. We use the context

$$C_x[\,\cdot\,] \quad \overset{\text{def}}{=} \quad \mathsf{def}\ \kappa\langle\rangle \triangleright t\langle\rangle \wedge x_e\langle x_o, x_i\rangle \triangleright x_i\langle\kappa\rangle\ \mathsf{in}\ [\,\cdot\,]$$

(where t is fresh) with the property $C_x[\mathcal{E}_S[P]] \Downarrow_t$ iff $P \Downarrow_{\overline{x}}$.

3. weak bisimulation is obtained by "decompiling" reductions in the join-calculus: let S be a set such that $\mathsf{fv}[Q] \cup \mathsf{fv}[R] \subseteq S$, and let us assume that $Q \to Q'$. By Lemma 6.31, we have $\mathcal{E}_S[Q] \to P'_Q \succeq_d \mathcal{E}_S[Q']$, and, by weak bisimulation in the join-calculus,

$$\begin{tikzcd}
 & \mathcal{E}_S\llbracket Q \rrbracket \arrow[r, dash, "\approx"] \arrow[d, "*"] & \mathcal{E}_S\llbracket R \rrbracket \arrow[d, dotted, "*"] & & \text{with} & R \arrow[d, dotted, "*"] \\
\mathcal{E}_S\llbracket Q' \rrbracket \arrow[r, dash, "\preceq_d"] & P'_Q \arrow[r, dotted, dash, "\approx"] & P'_R \arrow[r, dotted, dash, "\succeq_d"] & \mathcal{E}_S\llbracket R' \rrbracket & & R'
\end{tikzcd}$$

on the right-hand-side of the diagram, we obtain the source derivation $R \to^* R'$ and the relation $P'_R \succeq_d \mathcal{E}_S\llbracket R' \rrbracket$ by induction on the length of the join-calculus reductions $\mathcal{E}_S\llbracket R \rrbracket \to^n P'_R$; every initial reduction $\mathcal{E}_S\llbracket R \rrbracket \to T_1$ corresponds to a source reduction $R \to R_1$; moreover, we have by Lemma 6.31 that $\mathcal{E}_S\llbracket R_1 \rrbracket \preceq_d T_1$, hence $\mathcal{E}_S\llbracket R_1 \rrbracket \to^m \preceq_d P'_R$ for some $m < n$. $\square$

Chapter 7

Locality, Migration, and Failures

While distributed programming is our primary subject, the actual distribution of resources has hitherto been kept implicit. We introduced and studied the join-calculus, and argued that this calculus has built-in locality, hence that the miscellaneous components in a chemical solution could be partitioned into different units executed on different machines. Moreover, we claimed that this refinement would preserve a fairly transparent correspondence between the model and its implementation.

This chapter gives a more explicit account of distributed programming. We consider computations distributed on several machines—or sites—that can communicate over an asynchronous network; parts of the computation can migrate from a site to another, while some sites may fail during the computation.

Our model is a refinement of the join-calculus and the RCHAM, and benefits from the techniques we developed so far. In general, it is not easy to match concurrency theory and distributed systems, and indeed the *distributed join-calculus* that we present here is significantly more complex than the plain join-calculus. We extend the join-calculus with *locations* and primitives for mobility. The resulting distributed join-calculus allows us to express mobile agents that can move between physical sites. Agents are not only programs but core images of running processes with their communication capabilities and their internal state. We also describe the subtleties of modeling distributed asynchronous communication among fallible machines, in a precise and yet relatively simple manner.

The novelty of the distributed join-calculus is the introduction of locations to reify locality information. Intuitively, a location resides on a physical site, and contains a group of processes and definitions. We can move atomically a location to another site. We represent mobile agents by locations. Agents can contain mobile sub-agents represented by nested locations. Agents move as a whole with all their current sub-agents, thereby locations have a dynamic tree structure. Our calculus treats location names as first class values with lexical scopes, as is the case for channel names, and the scope of every name may extend over several locations and several machines. A location controls its own moves, and can move towards another location only by providing the name of the target location, which would typically be communicated only to selected processes. While more elaborate primitives may be required in a programming language, this already provides a starting point for static analysis and for secure mobility.

Our calculus provides a simple model of failure. The crash of a physical site causes the permanent failure of all its locations. More generally, any location can halt, with all its sublocations. The failure of a location can be detected at any other running location, allowing programmable error recovery. Other approaches are possible; we briefly suggest how they can be modeled in the join-calculus and discuss their relationship.

Our calculus can express distributed configurations with several machines, which may or may not fail according to a variety of conditions. In the absence of failure, however, the execution of processes is independent of distribution. This location transparency simplifies the design of mobile agents, and is very helpful for checking their properties.

Since we use the distributed join-calculus as the core of a distributed programming language, ease of implementation is a key design issue. More precisely, the definition of atomic reduction steps becomes critical, since it defines the balance between abstract features and realistic concerns. Except for the reliable detection of physical failures, the refined operational semantics has been fully implemented in a distributed setting with failures [59]. In this chapter, however, we only make a few comments on the implementation, and postpone a more systematic account to future work.

This chapter contains numerous examples of distributed processes with mobility and some failure recovery. Most of these examples are extracted from the suite of programs that is part of our distributed implementation of the model. We begin with standard patterns of distributed programming, such as remote procedure calls, and dynamic loading of remote applications—or applets. Unlike Java applets, we download a process with its current state, including its communication capabilities, simply by moving its location. More elaborate examples of agent-based mobility may involve a large number of machines. One of them extends the traditional client-server model with migratory agents to provide a better use of network bandwidth and additional guarantees in case of failure; other examples span a large data structures, or a complex computation, on an arbitrary number of machines. For most of the examples, we also provide some basic protection against partial failures.

Contents of the chapter

In Section 7.1 and 7.2 we gradually extend the join-calculus with explicit distribution. At the same time, we present a series of examples of increasing complexity. In Section 7.1, we introduce our location model as a refinement of the reflexive chemical model and present a first set of new primitives aimed at expressing location management and migration. In Section 7.2, we give our final calculus that copes with partial failure and failure recovery, and discuss various semantical models for failure. In Section 7.3 we explore the formal properties of the distributed join-calculus, apply some of the equivalences introduced in previous chapters, and prove a few equations. In Section 7.4 we finally review related work on distributed mobile programming.

7.1 Computing with locations

We refine the reflexive CHAM to model distributed systems. First, we partition processes and definitions into several *local solutions*. This flat model suffices for representing both local computation on different sites and global communication between them. Then, we introduce some more structure to account for the creation and the migration of local solutions: we attach *location names* to solutions, and we organize them as a tree of nested locations

The complete formal definition of the distributed semantics is deferred until Section 7.2 (figures 7.1, 7.2, and 7.3), once all the new constructs have been added to the join-calculus.

7.1.1 Distributed solutions

We first present a flat and static view of distribution. Processes and definitions are grouped in *locations.* Informally, every location is mapped to a physical machine; in our implementation, each machine executes a program that hosts one or several locations running in parallel.

We refine our chemical framework accordingly. A distributed reflexive chemical machine (DRCHAM) is a multiset of RCHAMs; instead of using nested multiset notations, we write the global state of a DRCHAM as several solutions $\mathcal{D} \vdash_\phi \mathcal{P}$ separated by a commutative-associative operator $\parallel$ that represents global composition. Each local solution is annotated with a distinct label ϕ for reference in the text—in the next section we will describe in more details the role of these labels.

Each solution $\mathcal{D} \vdash_\phi \mathcal{P}$ within a DRCHAM can evolve internally by using the same chemical rules as for the plain join-calculus of Chapter 2 (*cf.* Figure 2.3), both for structural rearrangements and for reduction steps. Technically, the chemical context law is extended to enable computation in any local solution. As usual, inactive parts of the contexts are kept implicit; hence the same chemical rules apply unchanged. Of course, the side condition in STR-DEF now requires that globally fresh names be substituted for defined names.

Local solutions can also interact with one another by using a new chemical reduction rule that operates on pairs of solutions:

$$\textsc{Comm} \quad \vdash_\varphi x\langle\widetilde{v}\rangle \;\parallel\; J \triangleright P \vdash_\psi \;\longrightarrow\; \vdash_\varphi \parallel\; J \triangleright P \vdash_\psi x\langle\widetilde{v}\rangle \qquad (\text{when } x \in \mathsf{dv}[J])$$

This rule models global communication; it states that a message emitted in a given solution φ on a port name x that is remotely defined can be forwarded to the solution ψ that contains the definition of x. Later on, this message can be used within ψ to assemble a parallel composition of messages that matches J, then consumed by using the familiar RED rule. The two-step decomposition of communication reflects what happens at run-time in actual implementations, where message transport and message processing are distinct operations. As we shall see in Section 7.2, this design choice is essential in the presence of failures.

In the following, we consider only well-formed DRCHAMs, where *every name is defined in at most one local solution.* Since the rule STR-DEF affects only local solutions, this condition is clearly closed by chemical reductions; it enforces the locality property discussed in Section 2.2 in our refined semantics. With this condition, the transport (COMM) is deterministic, static, and point-to-point.

The routing of messages is kept implicit in the calculus. Formally, there is a partial (meta) function that maps every name to its defining local solution—in the following this mapping from port names to locations will not be affected by any reduction. In the text, we use the notation *loc*(x) to refer to the location of x. In the implementation, each name is represented by a pointer to a data structure that is either a closure for locally-defined names or a *proxy data structure* that contains shared routing information. (Later on, as the definitions of names migrate to a remote site, their associated closures are changed in-place to such proxies. This technique is standard in distributed object systems.)

In contrast, some recent formalisms that address distributed programming force the explicit routing of messages in the calculus [68, 49, 46]. From a language design viewpoint, we believe that the bookkeeping of routing information is tedious and error-prone, and is best handled at the implementation level. At least in the distributed join-calculus, the locality property makes routing information simple enough to be safely omitted from the language. In short, our computational model hides the details of message routing, and focuses on those of synchronization. A refined implementation model would explicitly attach a location name to each name, and maintain the mapping from locations to machines.

We now give a few basic examples of distributed computations.

Remote message passing As a first example that involve several local solutions, we revisit the print spooler example of Chapter 2 (*cf.* page 56).

We now assume that there are three machines: a server machine s that hosts the spooler, a laser printer p that registers to the server, and a user machine u where some print request has been issued. As in Chapter 2 we let

$$D \quad \stackrel{\text{def}}{=} \quad \mathit{ready}\langle \mathit{printer}\rangle \mid \mathit{job}\langle \mathit{file}\rangle \triangleright \mathit{printer}\langle \mathit{file}\rangle$$

and let P represent the printer code. We have the series of chemical steps:

$$\begin{array}{rllll}
 & D \vdash_s & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{ready}\langle \mathit{laser}\rangle & \parallel \vdash_u \mathit{job}\langle 1\rangle \\
\overset{\textsc{Comm}}{\rightarrow} & D \vdash_s \mathit{job}\langle 1\rangle & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{ready}\langle \mathit{laser}\rangle & \parallel \vdash_u \\
\overset{\textsc{Comm}}{\rightarrow} & D \vdash_s \mathit{job}\langle 1\rangle, \mathit{ready}\langle \mathit{laser}\rangle & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p & \parallel \vdash_u \\
\overset{\textsc{Join,Red}}{\rightharpoondown\!\rightarrow} & D \vdash_s \mathit{laser}\langle 1\rangle & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p & \parallel \vdash_u \\
\overset{\textsc{Comm}}{\rightarrow} & D \vdash_s & \parallel \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{laser}\langle 1\rangle & \parallel \vdash_u
\end{array}$$

The first step forwards the message $\mathit{job}\langle 1\rangle$ from the user machine u to the unique machine that defines *job*, here the spooler s. Likewise, the second step forwards the message $\mathit{ready}\langle \mathit{laser}\rangle$ to the spooler. Next, synchronization occurs within the spooler between these two messages as a local reduction step. As a result, a new message on the spooler is sent to the laser printer, where it can be forwarded, then received.

This example also illustrates the notion of *global lexical scope*; assuming that the name *laser* is initially known only on the printer machine p, a preliminary local, structural step on machine p may be

$$\vdash_p \mathsf{def}\ \mathit{laser}\langle f\rangle \triangleright P\ \mathsf{in}\ \mathit{ready}\langle \mathit{laser}\rangle \quad \overset{\textsc{Str-def}}{\rightleftharpoons} \quad \mathit{laser}\langle f\rangle \triangleright P \vdash_p \mathit{ready}\langle \mathit{laser}\rangle$$

Then, the second Comm step in the series above effectively extends the scope of *laser* to the server, hence the server gains the ability to send messages to the printer as the result of further print requests.

Remote procedure call (RPC) We can easily model the classical RPC primitive by composing the previous example with the syntactic sugar for sequential control and for synchronous names that we introduced in Section 3.4.

For instance, we may assume a synchronous name "job" instead of the asynchronous one above. Along with the file to be printed, an implicit continuation is also passed to the printer, then triggered there upon completion of the job. A new user machine such as $\vdash_u \mathrm{job}(1); \mathrm{job}(2); \ldots$ may now print a series of files.

Each synchronous call to a remote name is implemented as (at least) four chemical reductions: a COMM step that routes the message $job\langle 1, \kappa\rangle$ to the spooler, some steps on the spooler or elsewhere that eventually issue the message $\kappa\langle\rangle$, another COMM step that routes the message $\kappa\langle\rangle$ back to the user machine, and a local step on the user machine that triggers the continuation process—here $\mathrm{job}(2); \ldots$. Again, we rely on network transparency and global lexical scope to make the continuation available for remote invocation.

As is the case for RPCs in distributed systems, the interface of the function—here the CPS encoding of the sequence—does not depend on whether the function is local or remote; this property, however, has an impact on the implementation mechanism that is used at run-time.

Usually, the libraries that implement RPCs also refine the standard function call mechanism to allow some primitive error recovery. We present a refined RPC protocol with an abstraction of timeouts: in case the remote computation takes too long, an error recovery function is triggered. As a starting point, the plain RPC described above can be abstracted into a call to the local library as follows:

$$\vdash_u \mathsf{def}\ \mathrm{rpc}(\mathrm{f}, x) \triangleright \mathsf{reply}\ \mathrm{f}(x)\ \mathsf{to}\ \mathrm{rpc}\ \mathsf{in} \ldots \mathrm{rpc}(\mathrm{job}, \mathit{part1}) \ldots$$

The name job has a global scope, but the name rpc is local to u, and can be considered as part of its communication library. We can also use a more elaborate definition of rpc that handles timeouts:

```
def rpc(f, $x$, $error$) $\triangleright$
    def $in\_call\langle\rangle \mid done\langle r\rangle \triangleright$ reply $r$ to rpc
    $\wedge\ in\_call\langle\rangle \mid timeout\langle\rangle \triangleright error\langle\rangle$
    in $in\_call\langle\rangle \mid done\langle \mathrm{f}(x)\rangle \mid start\_timer\langle timeout, 3\rangle$
in ... rpc(job, $part1$, $error\_handler$) ...
```

The in_call message guarantees mutual exclusion between the normal return from the remote call and the timeout error message, whichever occurs first. Note that the timeout is externally defined, as it makes little sense to define timeouts in the join-calculus: in an asynchronous setting, timeouts are just silent reductions that may occur at any time.

Remark 7.1 (The name-server) *In the examples, we usually assume that the scope of a few names that are defined in a local solution extends over several other local solutions—intuitively several machines. Formally, we rely on the global parallel composition operator, but we have explained neither how locations could be created, nor how these names could be initially communicated from one machine to another.*

In the implementation, programs that are executed on different machines are closed, so there is no such magical mechanism for assembling distributed computations, and *local computations should not be able to interact. Instead, we use an ad hoc "name-server" library that provides an interface to register and look up a few names by key. The interface of the name-server library is connected to any machine that joins a computation by using the IP address of the particular machine that hosts the state of the name server.*

Once a few initial names have been exchanged, the whole computation proceeds without relying on the name server anymore, and uses instead the implicit global scope extrusion mechanism.

Remark 7.2 (Global scope and open syntax) *The open calculus of Chapter 5 could be used to emphasize the "global" interface of each local solution: names that are defined in a local solution and that appear free in another solution are marked as "extruded" in the former solution; the rule* STR-DEF *then applies more liberally to rearrange their definitions; the rule* COMM *performs the same bookkeeping than the extrusion rule* EXT*. The restrictions on extruded names in the composition of open terms then coincide with the well-formed condition on distributed chemical solutions that enforce locality.*

7.1.2 Should locations be nested?

Assuming that every location is mapped to some host machine, agent migration is naturally represented by dynamic changes in the mapping from locations to machines. In order to save network bandwidth, for instance, a location that contains the running code of a mobile agent may migrate to the machine that hosts another location providing a particular service, in preparation to an intense exchange of messages with this location. In the distributed join-calculus, the migration can be expressed using the process $\mathrm{go}(a)$ where a is the name of the target location.

Our model of locations is *hierarchical*, locations being attached to parent locations instead of machines. We first motivate this choice, and explain why a flat machine/location structure is not satisfactory.

Technically, a hierarchical model of locations allows us to represent machines themselves as locations, and to treat uniformly partial failures and migrations. Moreover, this model provides enough expressiveness to describe distributed configurations of machines, without introducing new constructs for them.

As regards distributed programming, there are many situations where several levels of moving resources are useful. For instance, the server itself may, from time to time, move from one machine to another to continue the service while a machine goes down. Also, some agents naturally make use of sub-agents, e.g., to spawn some parts of the computation to other machines. Finally, the termination of a machine and of all its locations can then be modeled using the same mechanisms as a migration. When a mobile agent returns to its client machine, for instance, it may contain running processes and other resources; logically, the contents of the agent should be integrated with the client: later on, if the client moves, or fails, this contents should be carried away or discarded accordingly.

From the implementor's viewpoint, the hierarchical model can be implemented as efficiently as the flat model, because each machine only has to know its own local hierarchy of locations. Nonetheless, the model provides additional expressiveness to the programmer, who can assemble groups of resources that move from one machine to another as a whole. This may explain why most implementations of distributed mobile objects provide a rich dynamic structure for controlling migration, for instance by allowing objects to be temporarily attached to one another (*cf.* [77, 79, 78]).

Let us illustrate the need for nested agents in the case *several* migrations occur in some distributed computation. We describe the general situation of a client that just created an agent to go and get some information on a server. In the next array, two migrations occur in parallel: the agent goes to the server, while the server goes to another machine. We give an execution trace t_1, t_2, t_3, and use the symbol $\leadsto$ to emphasize potential migrations.

	Machine 1	Machine 2	Machine 3
t_1	$\vdash_{client}$ $\vdash_{agent}$ go($server$) $\leadsto$	$\vdash_{server}$ go(3) $\leadsto$	
t_2	$\vdash_{client}$	$\vdash_{server}$ go(3) $\leadsto$ $\vdash_{agent}$	
t_3	$\vdash_{client}$	$\vdash_{agent}$	$\vdash_{server}$

With a flat location structure, the migration from the client to the server must be dynamically replaced with a migration to a particular machine, for instance to the machine that currently hosts the server. In the case the server moves before the agent and the client's machine is aware of that move, then the agent arrives on the same machine as the server (machine 3). However, if the server moves just after the arrival of the agent, as detailed in the above trace, the agent is left behind. That is, the mapping from locations to machines depends on the scheduling of the different migrations, and the migration to the server yields no guarantee of being on the same machine as the server. For instance, machine 2 may crash after t_3, which makes the agent disappear while both the client and the server stay alive; this may be troublesome for the agent programmer, especially if she ignores the migratory behavior of the server.

The problem is delicate to fix. The server could maintain a list of its current sublocations, and ask each sublocation to move along as it moves from one machine to another. Nonetheless, this is cumbersome to program, and still this does not guarantee natural atomicity properties, since machine 2 may crash before a sublocation has a chance to catch up the server.

7.1.3 The location tree

In order to compute with locations, we represent them both as syntactic definitions (when they migrate or fail) and as local chemical solutions (when they interact with one another).

We use *location names* to relate the two structures. We assume given a countable set of location names $\mathcal{L}$. We use the letters $a, b, \ldots \in \mathcal{L}$ to denote location names, and $\varphi, \psi, \phi \ldots \in \mathcal{L}^*$ to denote finite strings of location names. We extend the type systems of Chapter 3 with a basic type for locations. In the following we always require that all processes be well-typed.

Location names are first-class values that statically identify a location. Like port names, they can be created locally, sent and received in messages, and they obey the lexical scoping discipline. To introduce new locations, we extend the syntax of definitions with a new location constructor:

$$D \quad \stackrel{\text{def}}{=} \quad \ldots \mid a\left[D' : P\right]$$

where D' gathers all the definitions of the location that are visible outside the location, where P is the running code of the location, and where a is a defined name that uniquely identifies the location. Informally, the definition $a\,[D' : P]$ corresponds to the local solution $\{D'\} \vdash_{\varphi a} \{P\}$. (As usual, we use explicit singleton multisets to stress that there is no other components in the local solution.)

We define the sublocation relation as: $\vdash_\varphi$ is a sublocation of $\vdash_\psi$ when ψ is a prefix of φ. In the following, DRCHAMs are multisets of labeled solutions whose labels φ are all distinct, prefix-closed, and uniquely identified by their rightmost location name, if any. These conditions ensures that solutions ordered by the sublocation relation form a tree.

This association is represented by new structural rules as follows. In the heating direction, the semantics of this new definition constructor is to create a sublocation of the current location that initially contains a single definition D and a single running process P. We have a new structural rule:

$$\textsc{Str-loc} \quad a\,[D : P] \vdash_\varphi \;\rightleftharpoons\; \vdash_\varphi \,\|\, \{D\} \vdash_{\varphi a} \{P\} \qquad (a \text{ frozen})$$

where the side condition "a frozen" requires that there is no solution of the form $\vdash_{\psi a \phi}$ in the implicit chemical context for any ψ, ϕ in $\mathcal{L}^*$. The definition D can contain frozen sublocation definitions. The side condition guarantees that D syntactically captures the whole subtree of sublocations in location a.

In the cooling direction, Str-loc has thus a freezing effect on location a and all its sublocations, which will be useful later for controlling atomicity during migration. Note that the rule Str-def and its side condition now also apply to defined location names, which guarantees in the heating direction that newly-defined locations are given fresh names, and in the cooling direction that locations that are folded back into defining processes are entirely frozen.

In well-formed DRCHAMs, we have required that all reaction rules defining one port name belong to a single local solution, and that all local solutions have distinct labels. With the addition of frozen locations in solution, we also require that frozen locations in solution all have distinct location names that do not appear in the labels of local solutions. We constrain the syntax of definitions accordingly: in a well-formed definition, for all conjunctions $D \wedge D'$, we require that $\mathsf{dv}[D] \cap \mathsf{dv}[D']$ contain only port names that are not defined in a sublocation of D or D'. For instance, the definitions $a\,[\mathsf{T} : 0] \wedge a\,[\mathsf{T} : 0]$ and $a\,[x\langle\rangle \triangleright P \wedge b\,[x\langle\rangle \triangleright Q : 0] : 0]$ are ruled out.

As a first example of nested locations, we describe a series of structural rearrangements that enable some actual reduction steps:

$$\begin{array}{rl}
& \vdash \mathsf{def}\ c\big[x\langle u\rangle \rhd Q \wedge a\,[D_a : P_a] : P_c\big]\ \mathsf{in}\ y\langle c,x\rangle \mid x\langle a\rangle \\
\overset{\textsc{Str-def}}{\rightleftharpoons} & c\big[x\langle u\rangle \rhd Q \wedge a\,[D_a : P_a] : P_c\big] \vdash y\langle c,x\rangle \mid x\langle a\rangle \\
\overset{\textsc{Str-loc}}{\rightleftharpoons} & x\langle u\rangle \rhd Q \wedge a\,[D_a : P_a] \vdash_c P_c \quad \| \quad \vdash y\langle c,x\rangle \mid x\langle a\rangle \\
\overset{\textsc{Str-def,join}}{\rightleftharpoons\rightleftharpoons} & x\langle u\rangle \rhd Q,\ a\,[D_a : P_a] \vdash_c P_c \quad \| \quad \vdash y\langle c,x\rangle,\ x\langle a\rangle \\
\overset{\textsc{Str-loc}}{\rightleftharpoons} & D_a \vdash_{ca} P_a \quad \| \quad x\langle u\rangle \rhd Q \vdash_c P_c \quad \| \quad \vdash y\langle c,x\rangle, x\langle a\rangle \\
\overset{\textsc{Comm}}{\rightarrow} & D_a \vdash_{ca} P_a \quad \| \quad x\langle u\rangle \rhd Q \vdash_c P_c, x\langle a\rangle \quad \| \quad \vdash y\langle c,x\rangle \\
\overset{\textsc{Red}}{\rightarrow} & D_a \vdash_{ca} P_a \quad \| \quad x\langle u\rangle \rhd Q \vdash_c P_c, Q\{^a/_u\} \quad \| \quad \vdash y\langle c,x\rangle \\
\rightleftharpoons^* & \vdash \mathsf{def}\ c\big[x\langle u\rangle \rhd Q : P_c \,|\mathsf{def}\ a\,[D_a : P_a]\ \mathsf{in}\ Q\{^a/_u\}\big]\ \mathsf{in}\ y\langle c,x\rangle
\end{array}$$

(where we assume that a does not occur in P_c or Q.) From this example, it should be clear that it is more convenient to work on a location structure that is fully diluted, at least for the reduction rules we have seen so far. Technically, all reductions occur on fully-diluted terms—except for the joins of messages that are assembled by the cooling rule Str-join—and to each distributed solution we can associate a unique fully-diluted solution up to the renaming of defined names, as in Remarks 2.1 and 5.4.

7.1.4 Moving locations

Now that the bookkeeping of the location tree has been relegated to structural rearrangement, we can naturally express migration as relocation of branches in the location tree by using the frozen location definition $a\,[D : P]$.

We extend the syntax of processes with a new primitive for migration:

$$P \quad \overset{\text{def}}{=} \quad \ldots \mid go\langle b, \kappa\rangle$$

along with a new reduction rule that operates on two chemical solutions:

$$\textsc{Go} \quad a\,[D : P \mid go\langle b,\kappa\rangle] \vdash_\varphi \| \vdash_{\psi b} \quad \longrightarrow \quad \vdash_\varphi \| \ a\,[D : P \mid \kappa\langle\rangle] \vdash_{\psi b}$$

Informally, the location a moves from its current position φa in the tree to a new position ψba just under the location name b given as argument. The target solution $\vdash_{\psi b}$ is identified by its relative name b. Once a arrives, the continuation $\kappa\langle\rangle$ is released, and typically triggers further local computations. In case the rule Str-loc has been used beforehand to cool down location a into a definition, its side condition (a frozen) forces all the sublocations of a to migrate at the same time, as a whole. As we shall see, this suffices to rule out spurious migration from a location to one of its sublocations.

In the whole chapter, we use the same notation for port names and for primitives like $go\langle\cdot,\cdot\rangle$. We extend the synchronous call convention of Section 3.4 accordingly for go($\cdot$). Notice, however, that primitives are not first-class names: they cannot be sent as values in messages, and their effect depends on the enclosing location.

7.1.5 Examples of agent-based protocols

We give a series of examples that involve migratory agents.

Anonymous locations The syntax of the distributed join-calculus represents locations as definitions because locations bind names; however, some locations can also be represented as processes. In particular, in numerous cases the name of the location does not appear within the scope of the location—we call such locations anonymous locations—and in such cases a lighter syntax is preferable. In the following, we use the syntactic sugar

$$\{P\} \quad \overset{\text{def}}{=} \quad \mathsf{def}\ b\,[\mathsf{T} : P]\ \mathsf{in}\ 0$$

(for some $b \in \mathcal{L} \setminus \mathsf{fv}[P]$) to represent a process P in its own anonymous location.

Objective moves versus subjective moves In [46] Cardelli and Gordon distinguish two kinds of process migration; *objective moves* remotely spawn some explicit process at some other location, while *subjective moves* change the localization of the enclosing computation, including other processes running in parallel. Our migration primitive is very subjective, as it encompasses the whole current location that executes a *go* primitive—including any process, definition and sublocation.

In a setting in which new locations can be freely created, subjective moves are more expressive than objective move, because it is possible to create a new, wrapping location that encodes a subjective move into an objective move; the converse encoding may be much harder. In the distributed join-calculus we define a derived *objective move* operator named "Spawn" as follows: we let

$$\text{Spawn}(a, P) \quad \overset{\text{def}}{=} \quad \{go\langle a\rangle; P\}$$

and we can compose chemical steps to obtain a derived reduction rule

$$\textsc{Spawn} \quad \vdash_\varphi \text{Spawn}(a, P) \ \| \vdash_{\psi a} \quad \overset{\textsc{Def}}{\rightharpoonup}\overset{\textsc{Loc}}{\rightharpoonup}\overset{\textsc{Go}}{\rightarrow} \succeq \quad \vdash_\varphi \| \vdash_{\psi a} \{P\}$$

(Where the expansion relation $\succeq$ defined in Section 4.7.3 abstracts over the deterministic reduction that triggers the implicit continuation and the garbage-collection of the continuation's definition.) The side condition on rule STR-LOC holds because the new location name does not appear anywhere else after applying rule STR-DEF. In Section 7.3, we will provide sufficient conditions on P so that $\{P\}$ and P are equivalent.

While other primitives have been proposed for achieving agent migration, the go primitive and its use to define spawn correspond to our idea of a distributed implementation: while it is possible—and even cheap—to create locally a new sublocation, the migration of the new sublocation to another location must be a computation step, since for instance it is hard to guarantee that a process atomically spawns two processes on two different machines.

"Ping" As a direct application of the spawn construct, we can define a ping construct that performs a round trip between the current location and another location given as an argument, which may be useful to test the status of the latter location. The example is similar to the RPC example, except that the name of the location is provided instead of the name of a port in this location. We use the rule

$$\text{ping}(a) \triangleright \text{Spawn}(a, \mathsf{reply\ to}\ \text{ping})$$

Applets A basic pattern of network-based programming is to download a piece of code from a code server *à la Java* for the computation to take place on the local site. (In Java, this pattern does not change the model of distributed computation, which relies on other, traditional mechanisms such as sockets to the server or remote method invocation. In principle, however, the code can be generated on demand, instead of being statically compiled before the computation begins.)

In our example, we consider an applet whose interface consist of a single synchronous name f. Further, we write the code of the applet as an expression E that evaluates to f (*cf.* Section 3.4). E typically contains a few definitions; for instance, we can let $E = \mathsf{def}\ \mathrm{f}(y) \triangleright \mathrm{print}(x+1); \mathsf{reply\ to}\ \mathrm{f}\ \mathsf{in}\ \mathrm{f}$.

The applet function is wrapped within a new location whenever a client asks for its local copy of the function. Such requests for downloads are sent to the name loader, which can be defined as follows:

$$D_{\mathrm{applet}} \quad \stackrel{\mathrm{def}}{=} \quad \mathrm{loader}(a) \triangleright \mathsf{def}\ b\,[\mathsf{T} : \mathrm{go}(a); \mathsf{reply}\ E\ \mathsf{to}\ \mathrm{loader}]\ \mathsf{in}\ 0$$

where b is a fresh name, or simply, with our new abbreviations,

$$D_{\mathrm{applet}} \quad \stackrel{\mathrm{def}}{=} \quad \mathrm{loader}(a) \triangleright \mathrm{Spawn}(a, \mathsf{reply}\ E\ \mathsf{to}\ \mathrm{loader})$$

The applet can be downloaded, then locally used on a client machine c; the applet server machine s is left unchanged. For instance, with the applet E defined above, we have

$$\begin{array}{rl} & D_{\mathrm{applet}} \vdash_s \ \|\ \vdash_c \mathsf{let}\ \mathrm{f} = \mathrm{loader}(c)\ \mathsf{in}\ \mathrm{f}(1); \mathrm{f}(2); \ldots \\ \rightarrow^*\sim & D_{\mathrm{applet}} \vdash_s \ \|\ \vdash_c \mathsf{def}\ b\,[\mathrm{f}(y) \triangleright \mathrm{print}(x+1); \mathsf{reply\ to}\ \mathrm{f} : 0]\ \mathsf{in}\ \mathrm{f}(1); \mathrm{f}(2); \ldots \end{array}$$

(where the equivalence $\sim$ discards the definition of an implicit continuation.)

Assuming that the applet code E does not include another go primitive, the applet location b remains attached to the client location c and the program behaves as if a fresh copy of the applet had been defined at location c. Later on, we will show that for all processes Q such that $b \notin \mathsf{fv}[Q]$ we have

$$\begin{array}{rl} & \mathsf{def}\ b\,[\mathrm{f}(y) \triangleright \mathrm{print}(x+1); \mathsf{reply\ to}\ \mathrm{f} : 0]\ \mathsf{in}\ Q \\ \approx & \mathsf{def}\ \mathrm{f}(y) \triangleright \mathrm{print}(x+1); \mathsf{reply\ to}\ \mathrm{f}\ \mathsf{in}\ Q \end{array}$$

The client-agent-server architecture (CASA) The opposite of retrieving code is sending computation to a remote server. The client defines the request; the request goes to the server, runs there, and sends the result back to the client. This can be expressed on the client side by the process

$$\mathsf{def}\ \mathrm{f}(x, s) \triangleright \{\mathrm{go}(s); \mathsf{reply}\ \ldots\ \mathsf{to}\ \mathrm{f}\}\ \mathsf{in}\ \ldots \mathrm{f}(3, \mathit{server}) \ldots$$

In the code above, f is a synchronous name, hence the remote computation returns a tuple of values. In general, however, the result might contain arbitrary data allocated during the computation, or even active data (processes with internal state). In the generic CASA, the server cannot just return a pointer to the data; it must also move the data and the code back to the client location. To illustrate this, we consider an agent that allocates and uses a reference cell; new_cell is an applet server that creates a fresh cell and returns its two methods set for updates and get for access.

```
def $c[\mathrm{f}(x,s) \triangleright$
    def $a[\mathsf{T} : \mathrm{go}(s);$
        let $\mathrm{set}, \mathrm{get} = \mathrm{new\_cell}(a)$
        in $\mathrm{set}(\mathrm{computation}(x)); \mathrm{go}(c);$ reply $\mathrm{get}$ to $\mathrm{f}]$
    in $\ldots : 0]$
in $\ldots \mathrm{f}(3, server) \ldots$
```

The reference cell is allocated within the agent at location a, during its stay on the server; it can be modified as part of the agent's computation. When this computation terminates, the agent brings back the data structure to the client by using the $\mathrm{go}(c)$ primitive call.

This pattern is most useful when the client needs to build its result state after questioning several servers. An equivalent, RPC-based program would force a centralized communication between the client and each server and a pre-allocation of the data structures, while the CASA protocol enables us to send an agent that has recorded all the operations it needs to perform and all the sites it needs to visit.

Located data structure and iteration A *located data structure* is a piece of data along with its location; the piece of data is represented by its abstract interface, typically a set of access and update methods. The location is used as an anchor to write agents that "follow the data", typically migrating towards the data before accessing or mutating it. For instance, in the case the data is protected by a lock, this guarantees the absence of deadlocks due to the failure of a remote machine.

In the following example, we consider an uniform data structure whose interface consists of a single "iterator" functional that traverses the data structure and iterates a given function on each component of the structure.

The basic structure simply is an array or a list at a given location, with a local iterator such as `list.iter` or `array.iter` in our implementation. More interestingly, we can write a generic merge function that assembles two located data structures into a single larger one:

```
$\mathrm{merge}(a_1, \mathrm{f}_1, a_2, \mathrm{f}_2) \triangleright$
    def $a[\mathrm{f}(g) \triangleright \mathrm{go}(a_1); \mathrm{f}_1(g); \mathrm{go}(a_2); \mathrm{f}_2(g);$ reply  to $\mathrm{f} : 0]$ in
    reply $a, \mathrm{f}$ to merge
```

The new location a represents the compound data structure, but it does not contains the locations of its components. On the contrary, it becomes a sublocation of each of its subcomponents in turn, carrying with it the location of the agent that accesses the distributed data structure.

On the caller's side of the iterator, we illustrate the traversal of such data structures by a mobile agent that collects statistics on the distributed data structure.

$$\text{def statistics}(\mathit{there}, \mathrm{map}) \triangleright \\ \left\{ \begin{array}{ll} & \mathit{state}\langle n, s, s_2\rangle \mid \mathrm{item}(x) \triangleright \mathit{state}\langle n+1, s+x, s_2 + x * x\rangle \mid \text{reply to item} \\ \wedge & \mathit{state}\langle n, s, s_2\rangle \mid \mathit{done}\langle\rangle \triangleright \text{reply } n, s, s_2 \text{ to statistics in} \\ \multicolumn{2}{l}{\mathit{go}\langle \mathit{there}\rangle; (\mathit{state}\langle 0,0,0\rangle \mid \mathrm{map}(\mathrm{item}); \mathrm{done}())} \end{array} \right\}$$

In that case, the result consists only of a tuple of values; for other agents, we may also consider moving back to the original location after collecting all statistical data.

7.1.6 Circular migration

It is possible to write locations that attempt to migrate to one of their sublocation. We name such attempts "circular migrations". For instance, the process

$$R \overset{\text{def}}{=} \mathsf{def}\ a\,[b\,[\mathsf{T} : \mathsf{0}] : \mathrm{go}(b); P]\ \mathsf{in}\ \mathsf{0}$$

attempts to move the location a inside its sublocation b, which naively would create a cycle between a and b detached from the location tree. Another, simpler example is $\mathsf{def}\ a\,[\mathsf{T} : \mathrm{go}(a); P]\ \mathsf{in}\ \mathsf{0}$.

Chemically, we use the same conventions for representing contexts as in the definition of the DRCHAM, and we say that a process (or a chemical solution) *attempts a circular migration* when it is structurally equivalent to a DRCHAM in which one of the two following predicates holds:

$$\begin{array}{ll} \textsc{Circle-0} & \vdash_{\varphi a} go\langle a, \kappa\rangle \\ \textsc{Circle} & \vdash_{\varphi a} go\langle b, \kappa\rangle \ \parallel \vdash_{\varphi\psi b} \end{array}$$

Intuitively, we consider attempts to perform circular migrations as programming errors. In the implementation, circularities can only be created locally, so it is straightforward to add an occur-check and to report circular migrations as run-time errors. Also, there are simple disciplines of programming with migrations that rule out the possibility of circular migrations, e.g., migrations of anonymous locations cannot create cycles, and more generally if there is a static ordering of location variables such that migration always occur towards a "larger" location then cycles are excluded. Such disciplines may be enforced by a static analysis.

Formally, the distributed RCHAM blocks circular migrations, because it is not possible to use the cooling structural rule STR-LOC to meet the requirements of rule GO. Still, this kind of migration is troublesome, because it may introduce non-determinacy in series of migrations, which are otherwise confluent. For instance, for all processes P and Q in the join-calculus, for all fresh location names a, b in $\mathcal{L}$, we easily establish the equation

$$\mathsf{def}\ a\,[\mathsf{T} : \mathrm{go}(b); P] \wedge b\,[\mathsf{T} : \mathrm{go}(a); Q]\ \mathsf{in}\ \mathsf{0} \ \approx\ P \oplus Q$$

because the first migration triggers either P or Q and blocks the other one.

In the absence of potential circular migrations (and later, of failures), we obtain the simple property that we mentioned at the beginning of this section: all migrations that occur in parallel but in different locations are confluent, hence the final location tree does not depend on their interleaving.

7.1.7 Erasing locality information

So far, we have explained how to keep track of locality information during a distributed run of a program, but the location of a particular resource did not affect the result of the computation. Later, we give some observational semantics to locality by modeling partial failure of the computation. In the absence of failure, though, we can remove the location boundaries.

To this end, we explain how locality information can be erased, and state a simple theorem that expresses this notion of network transparency.

For definitions, we erase the location boundaries. We collect definitions and processes running in locations using two distinct translations $[\![\cdot]\!]^d$ and $[\![\cdot]\!]^p$:

$$\begin{aligned}
[\![J \triangleright P]\!]^d &\stackrel{\text{def}}{=} J \triangleright [\![P]\!] \\
[\![D \wedge D']\!]^d &\stackrel{\text{def}}{=} [\![D]\!]^d \wedge [\![D']\!]^d \\
[\![\mathsf{T}]\!]^d &\stackrel{\text{def}}{=} \mathsf{T} \\
[\![a\,[D : P]\,]\!]^d &\stackrel{\text{def}}{=} a\langle\rangle \triangleright \mathsf{0} \wedge [\![D]\!]^d
\end{aligned}$$

$$\begin{aligned}
[\![J \triangleright P]\!]^p &\stackrel{\text{def}}{=} \mathsf{0} \\
[\![D \wedge D']\!]^p &\stackrel{\text{def}}{=} [\![D]\!]^p \mid [\![D']\!]^p \\
[\![\mathsf{T}]\!]^p &\stackrel{\text{def}}{=} \mathsf{0} \\
[\![a\,[D : P]\,]\!]^p &\stackrel{\text{def}}{=} [\![P]\!]
\end{aligned}$$

For processes, we simply erase the migration primitives, and immediately trigger their continuation.

$$\begin{aligned}
[\![go\langle a, \kappa\rangle]\!] &\stackrel{\text{def}}{=} \kappa\langle\rangle \\
[\![\mathsf{def}\ D\ \mathsf{in}\ P]\!] &\stackrel{\text{def}}{=} \mathsf{def}\ [\![D]\!]^d\ \mathsf{in}\ [\![D]\!]^p \mid [\![P]\!]
\end{aligned}$$

Other cases are omitted; they simply propagate the encoding. The dummy new rule $a\langle\rangle = \mathsf{0}$ maintains the same scope for a as before. Alternatively, we could entirely remove this binder and all parts of messages that convey location names in a type-directed translation, at least in a monomorphic calculus.

Formally, we let the *failure-free* distributed join-calculus be the fragment of the (full) distributed join-calculus defined in Figures 7.1 and 7.2 that do not contain the forthcoming constructs for failures (*halt*, *fail*, and Ω). In this failure-free fragment, we let $\approx_d$ be the largest barbed bisimulation that is a congruence for all evaluation contexts of the join-calculus—thus excluding contexts that create or manipulate locations. In the join-calculus $\approx$ is the barbed congruence of Chapter 4.

Theorem 12 (Location transparency) *Let P and Q be two processes in the failure-free distributed calculus such that*

1. *P and Q never attempt circular migrations in any join-calculus context;*

2. $(\mathsf{fv}[P] \cup \mathsf{fv}[Q]) \cap \mathcal{L} = \emptyset$

Then the erasing translation is fully abstract: $P \approx_d Q$ iff $[\![P]\!] \approx [\![Q]\!]$.

The two conditions of the theorem guarantee that every migration succeeds, no matter of the context. Without condition 1 some circular migrations may cause deadlocks; without condition 2 migrations that takes a free name as target location would be deadlocked.

Proof: The translation $[\![\cdot]\!]$ maps only distributed processes to plain processes, but naturally induces a translation from DRCHAMs to RCHAMs that is also denoted $[\![\cdot]\!]$, and that is used for establishing the bisimulation properties by a case analysis on chemical reductions that operate on diluted solutions.

Structural steps before and after the translation are in simple correspondence: the STR-LOC steps are deleted; all other steps in distributed solutions yield equivalent steps in the translated solution, possibly after α-conversion on location names. Conversely, additional folding steps are available in the translation but they do not lead to additional reduction steps.

We have the following operational correspondence properties for all distributed solution $\mathcal{S}$. If $\mathcal{S} \overset{\text{COMM}}{\rightarrow} \mathcal{S}'$ or $\mathcal{S} \overset{\text{GO}}{\rightarrow} \mathcal{S}'$, then $[\![\mathcal{S}]\!] = [\![\mathcal{S}']\!]$. If $\mathcal{S} \overset{\text{RED}}{\rightarrow} \mathcal{S}'$, then $[\![\mathcal{S}]\!] \rightarrow [\![\mathcal{S}']\!]$. Conversely, if $[\![\mathcal{S}]\!] \rightarrow \mathcal{T}$, then for some distributed solution $\mathcal{S}'$ we have that $\mathcal{S} \left(\rightleftharpoons \cup \overset{\text{COMM}}{\rightarrow} \cup \overset{\text{GO}}{\rightarrow} \right)^* \overset{\text{RED}}{\rightarrow} \mathcal{S}'$ with $[\![\mathcal{S}']\!] = \mathcal{T}$.

In particular, we obtain the simple correspondence between the barbs before and after the translation: $[\![P]\!] \downarrow_x$ iff $P \left(\overset{\text{COMM}}{\rightarrow} \cup \overset{\text{GO}}{\rightarrow} \right)^* \downarrow_x$.

- $P \approx_d Q$ implies $[\![P]\!] \approx [\![Q]\!]$. Let $\mathcal{R}$ be the relation that contains all translations $[\![\mathcal{S}]\!], [\![\mathcal{T}]\!]$ of equivalent fully-diluted distributed solutions $\mathcal{S} \approx_d \mathcal{T}$. We prove that $\mathcal{R}$ is a barbed bisimulation and a congruence in the join-calculus, up to structural rearrangement.

  $\mathcal{R}$ respects the strong barbs in the join-calculus.

  We assume that $[\![\mathcal{S}]\!] \mathcal{R} [\![\mathcal{T}]\!]$, and check that each RED step $[\![\mathcal{S}]\!] \rightarrow [\![\mathcal{S}']\!]$ can be simulated from $[\![\mathcal{T}]\!]$. To this end, we apply the operational correspondence on reductions above, first from the translation to the distributed calculus to obtain a series of arbitrary reductions, then from the distributed calculus to the plain join-calculus to discard all non-RED steps.

  All contexts of the plain join-calculus are left unchanged by the translation, and their applications preserve the two conditions on P and Q in the lemma, hence we obtain the congruence property of $\mathcal{R}$ from te congruence property of $\approx_d$ in the distributed calculus.

- $[\![P]\!] \approx [\![Q]\!]$ implies $P \approx_d Q$. Let $\mathcal{R}$ be the relation that contains all distributed solutions $\mathcal{S}, \mathcal{T}$ whose translations are equivalent ($[\![\mathcal{S}]\!] \approx [\![\mathcal{T}]\!]$). We prove that $\mathcal{R}$ is a barbed bisimulation in the distributed calculus and a congruence for all join-calculus contexts.

  Let us assume that $\mathcal{S} \mathcal{R} \mathcal{T}$. If $\mathcal{S} \downarrow_x$, then $[\![\mathcal{S}]\!] \downarrow_x$, by hypothesis $[\![\mathcal{T}]\!] \rightarrow^* \downarrow_x$, and by operational correspondence $\mathcal{T} \rightarrow^* \downarrow_x$.

  $\mathcal{R}$ is a weak bisimulation property: COMM and GO steps are simulated by the absence of reduction. Other steps are simulated by applying the two operational correspondences above.

  Finally, the encoding is compositional, hence if $P \mathcal{R} Q$ and $C[\cdot]$ meets the restrictions on contexts, then $[\![C[P]]\!] = [\![C]\!][[\![P]\!]] \approx [\![C]\!][[\![Q]\!]] = [\![C[Q]]\!]$, thus $C[P] \mathcal{R} C[Q]$. $\square$

7.2 Failure and recovery

In an asynchronous calculus, the routing of messages is invisible—it is just another layer of buffering [139]—hence the locality information is purely descriptive as soon as we adopt implicit message routing.

Indeed, Theorem 12 states that locations have little semantic importance in the distributed join-calculus, as long as there is no failures or ciruclar migrations.

As noted by Amadio in [17] for a variant of the π-calculus, a simple model of failure suffices to refine the semantics and make it very sensitive to locality information.

In a completely reliable (or completely unreliable) setting, localization is not very important; while it may dramatically affect the overall performance, the asynchronous semantics would be the same than for the non-distributed calculus. All the problems come from partial failures, as we still expect results from the surviving parts of the system. As a result, we consider that failures are essential to model distributed systems.

Modeling failures yields some observational meaning to distributed computation formalism, because the physical failures are imposed from the outside, and tend to reveal many details about the computation. It turns out that the resulting equivalences are very discriminative, as already suggested in previous treatment of (explicit) locality information for labeled semantics, in a CCS setting [40, 41, 47]

In this section we present our failure model, we introduce two primitives for failure management, we show their use in examples, and finally we discuss the choice of our failure model. Equivalence properties in the presence of failures are treated in the next section.

7.2.1 The fail-stop model

What does failure mean in a distributed computation? There are numerous answers that describe a large variety of situations. For instance, the "Byzantine" model of failures makes almost no assumption on the behavior of failed machines, which may emit spurious messages. This is the notion of failure found in works on security, where it is assumed that the failed machine—renamed the attacker—may attempt a large variety of attacks to confuse the correct parts of the computation [2].

In this work, we make simplifying assumptions on failures. We adopt the "fail-stop" model, where every machine works correctly until failure, then becomes permanently inert—in practice, when a machine fails, then restarts, it may re-enter the distributed computation only under a new logical identity. This assumption is reasonably easy to implement for a network of friendly machines that may crash from time to time.

In order to trigger error recovery, we also assume that all failures can be reliably detected by all other machines. Without additional hypotheses, it is not possible to guarantee that all failed machines be detected in a purely asynchronous setting. In theory, though, it is possible to achieve a similar effect with slightly more demanding hypotheses on unreliable detectors [48]. In practice, also, at least *some* definite failure information is available—for instance when a machine restarts after a crash, it can tell for sure that the previous runtime has stopped—and more efficient algorithms can take advantage of timed properties to implement failure detection. The specification and the study of such algorithms is beyond the scope of this dissertation.

P	$::=$		processes
		$x\langle v_1, \ldots, v_n\rangle$	message
	$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	local definition
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\mathsf{0}$	null process
	$\mid$	$go\langle a, \kappa\rangle$	migration of current location
	$\mid$	$halt\langle\rangle$	termination of current location
	$\mid$	$fail\langle a, \kappa\rangle$	detection of location failure
D	$::=$		definitions
		$J \triangleright P$	reaction rule
	$\mid$	$D \wedge D'$	conjunction of definitions
	$\mid$	T	null definition
	$\mid$	$a\,[D : P]$	sub-location
	$\mid$	$\Omega a\,[D : P]$	dead sub-location
J	$::=$		join-pattern
		$x\langle y_1, \ldots, y_n\rangle$	message pattern
	$\mid$	$J \mid J'$	join of patterns

Figure 7.1: Syntax for the distributed-join-calculus

There are several ways to define the semantics of failures and failure recovery. We present our proposal, which we call the "asynchronous strong" model of failures, then we relate it to other possibilities.

7.2.2 Representing failures

We complete the presentation of the distributed reflexive chemical machine, in the presence of failure and failure detection. The syntax, the scopes, and the full chemical machinery are described in Figures 7.1, 7.2 and 7.3, respectively.

We supplement the location tree with partial failure information. To this end, we use a special marker $\Omega \notin \mathcal{L} \cup \mathcal{N}$ to tag failed locations. For every $a \in \mathcal{L}$, εa denotes either a or Ωa, and φ, ψ denote finite strings of such εa's. In the DRCHAM, Ω appears in the location string φ of failed locations $\vdash_\varphi$. We say that φ is *dead* if it contains one or several markers Ω, and *alive* otherwise; the positions of the markers record where the failures were triggered. We extend our well-formed condition accordingly, and require that every distributed solution be consistently marked—for every location name a, either all or none of the occurrences of a in labels are Ω-marked. In the syntax of definitions, failed locations are frozen as marked definitions $\Omega a\,[D : P]$; thus the general shape of a location definition now is $\varepsilon a\,[D : P]$.

In accordance to our interpretation of structural rearrangement as "computation free" chemical steps, our structural rules should not depend on the live/failed status of locations and local solutions, and should remain entirely reversible. Hence, the structural rules in Figure 7.3 are almost unchanged from Chapter 2 and Section 7.1, except for the obvious generalization of STR-LOC to the failed location syntax. In

For processes:

$$\begin{array}{rcl}
\mathsf{fv}[x\langle v_1, \ldots , v_n\rangle] & \overset{\text{def}}{=} & \{x, v_1, \ldots , v_n\} \\
\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P] & \overset{\text{def}}{=} & (\mathsf{fv}[P] \cup \mathsf{fv}[D]) \setminus \mathsf{dv}[D] \\
\mathsf{fv}[P \mid P'] & \overset{\text{def}}{=} & \mathsf{fv}[P] \cup \mathsf{fv}[P'] \\
\mathsf{fv}[\mathsf{0}] & \overset{\text{def}}{=} & \emptyset \\
\mathsf{fv}[go\langle a, \kappa\rangle] & \overset{\text{def}}{=} & \{a, \kappa\} \\
\mathsf{fv}[halt\langle\rangle] & \overset{\text{def}}{=} & \emptyset \\
\mathsf{fv}[fail\langle a, \kappa\rangle] & \overset{\text{def}}{=} & \{a, \kappa\}
\end{array}$$

For definitions:

$$\begin{array}{rcl}
\mathsf{fv}[J \triangleright P] & \overset{\text{def}}{=} & \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J]) \\
\mathsf{fv}[D \wedge D'] & \overset{\text{def}}{=} & \mathsf{fv}[D] \cup \mathsf{fv}[D'] \\
\mathsf{fv}[\mathsf{T}] & \overset{\text{def}}{=} & \emptyset \\
\mathsf{fv}[\varepsilon a\, [D : P]] & \overset{\text{def}}{=} & \{a\} \cup \mathsf{fv}[D] \cup \mathsf{fv}[P]
\end{array}$$

$$\begin{array}{rcl}
\mathsf{dv}[J \triangleright P] & \overset{\text{def}}{=} & \mathsf{dv}[J] \\
\mathsf{dv}[D \wedge D'] & \overset{\text{def}}{=} & \mathsf{dv}[D] \cup \mathsf{dv}[D'] \\
\mathsf{dv}[\mathsf{T}] & \overset{\text{def}}{=} & \emptyset \\
\mathsf{dv}[\varepsilon a\, [D : P]] & \overset{\text{def}}{=} & \{a\} \uplus \mathsf{dv}[D] \\
\mathsf{dv}[a\, [D : P]] & \overset{\text{def}}{=} & \{a\} \uplus \mathsf{dv}[D]
\end{array}$$

For join patterns:

$$\begin{array}{rclcrcl}
\mathsf{dv}[x\langle y_1, \ldots , y_n\rangle] & \overset{\text{def}}{=} & \{x\} & \quad & \mathsf{dv}[J \mid J'] & \overset{\text{def}}{=} & \mathsf{dv}[J] \uplus \mathsf{dv}[J'] \\
\mathsf{rv}[x\langle y_1, \ldots , y_n\rangle] & \overset{\text{def}}{=} & \{y_1, \ldots , y_n\} & \quad & \mathsf{rv}[J \mid J'] & \overset{\text{def}}{=} & \mathsf{rv}[J] \uplus \mathsf{rv}[J']
\end{array}$$

Well formed conditions for every definitions D:

- each location name is defined at most once
- each channel name is defined by join-patterns that all appear in the same location (*cf.* 2.2,7.1.3)

Figure 7.2: Scopes for the distributed-join-calculus

Str-join	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1, P_2$
Str-null	$\vdash \mathsf{0}$	$\rightleftharpoons$	$\vdash$
Str-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
Str-nodef	$\mathsf{T} \vdash$	$\rightleftharpoons$	$\vdash$
Str-def	$\vdash \mathsf{def}\ D\ \mathsf{in}\ P$	$\rightleftharpoons$	$D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}}$
Str-loc	$\varepsilon a\,[D : P] \vdash_{\varphi}$	$\rightleftharpoons$	$\vdash_{\varphi} \parallel \{D\} \vdash_{\varphi\varepsilon a} \{P\}$
Red	$J \triangleright P \vdash_{\varphi} J\sigma_{\mathrm{rv}}$	$\longrightarrow$	$J \triangleright P \vdash_{\varphi} P\sigma_{\mathrm{rv}}$
Comm	$\vdash_{\varphi} x\langle\widetilde{v}\rangle \parallel J \triangleright P \vdash$	$\longrightarrow$	$\vdash_{\varphi} \parallel J \triangleright P \vdash x\langle\widetilde{v}\rangle$
Go	$a\,[D : P \mid go\langle b, \kappa\rangle] \vdash_{\varphi} \parallel \vdash_{\psi\varepsilon b}$	$\longrightarrow$	$\vdash_{\varphi} \parallel a\,[D : P \mid \kappa\langle\rangle] \vdash_{\psi\varepsilon b}$
Halt	$a\,[D : P \mid halt\langle\rangle] \vdash_{\varphi}$	$\longrightarrow$	$\Omega a\,[D : P] \vdash_{\varphi}$
Detect	$\vdash_{\varphi} fail\langle a, \kappa\rangle \parallel \vdash_{\psi\varepsilon a}$	$\longrightarrow$	$\vdash_{\varphi} \kappa\langle\rangle \parallel \vdash_{\psi\varepsilon a}$

side conditions ($\mathcal{S}$ is the distributed solution on the left-hand-side)

Str-def	σ_{dv} instantiates variables in $\mathsf{dv}[D]$ to distinct, fresh names: $Dom(\sigma_{\mathrm{dv}}) \cap \mathsf{fv}[\mathcal{S}] = \emptyset$.
Str-loc	a is frozen: the name a does not occur in the indices ψ of any local solution of $\mathcal{S}$.
Red	σ_{rv} substitutes values for the received variables $\mathsf{rv}[J]$.
Comm	$x \in \mathsf{dv}[J]$.
Detect	a is dead: the string $\psi\varepsilon a$ contains a marker Ω.
Red, Comm, Go, Halt, Detect	φ is alive: the string φ contains no marker Ω.

Figure 7.3: The distributed reflexive chemical machine

particular, the scope of the names defined in a given location is independent of its liveness status.

Naturally, reduction steps are strongly affected by failures. We model the failure of a location by prohibiting reactions inside this location and any of its sublocations. More precisely, in Figure 7.3 we have added a side condition to Red, Comm, and Go, that prevents these rules from taking messages (or handling $go\langle\cdot,\cdot\rangle$ primitives) in a solution with a dead label. Note, however, that we do not prevent messages or even locations from moving to a failed location, as such deadly moves are unavoidable in an asynchronous distributed setting.

7.2.3 Primitives for failure and recovery

We introduce two new primitives $halt\langle\rangle$ and $\mathsf{fail}\langle\cdot,\cdot\rangle$, with two specific chemical reductions rules, as defined in figure 7.3.

The process $halt\langle\rangle$ at location a can make this location permanently inert (rule Halt in Figure 7.3), while the process $\mathsf{fail}\langle a,\kappa\rangle$ in a live location triggers $\kappa\langle\rangle$ after it detects that a has failed, i.e. that a or one of its parent locations has halted (rule Detect). Note that the (φ alive) side condition in rules Go and Comm suffices to prevent any output from a dead location; it is also attached to the rules Red, Halt, and Detect only for uniformity.

A simple example of a fallible location is $\{halt\langle\rangle \mid P\}$, which describes an anonymous location that may fail at any time, and that executes the process P until failure occurs. We have the simple equation

$$\{halt\langle\rangle \mid P\} \quad \xrightarrow{\text{Halt}} \sim \quad 0$$

(Where $\sim$ is the strong barbed congruence for all evaluation contexts.)

We can now precisely relate failure to the migration to a failed location. If $C[\,\cdot\,]$ is a context that binds a to a dead location, and in the absence of circular migrations, we have the strong bisimulation

$$C[go\langle a,\kappa\rangle] \quad \sim \quad C[halt\langle\rangle]$$

That is, moving to a dead location or halting have the same visible effect, namely to make the moving branch of the location tree permanently inert.

The $halt\langle\rangle$ process can be triggered only from within the halting location, which statically identifies the sources of failure in the location tree. Nonetheless, it is possible to provide a relay that enables external processes to trigger the failure by an asynchronous message. For instance, $halt\langle\rangle$ can be used to encode a "kill" operation that can terminate the code of the applet example of Section 7.1:

$$\begin{array}{l} \mathsf{def}\ sandbox\ [kill\langle\rangle \triangleright halt\langle\rangle : start\_timer\langle kill, 5\rangle]\ \mathsf{in} \\ \mathsf{let}\ \mathsf{f} = \mathrm{loader}(sandbox)\ \mathsf{in}\ \mathsf{f}(3); kill\langle\rangle \end{array}$$

The suspicious applet user creates a sublocation *sandbox* that hosts the applet, uses the applet, then discards the applet by sending the $kill\langle\rangle$ message. In parallel, it sends a message on *start_timer* to start monitoring the applet execution and eventually terminates it if it takes too long. A *fail* guard would be needed to guarantee that the applet is halted. We can wrap the resulting safe protocol in a single definition

$$\begin{aligned} \mathrm{run}(\mathrm{loader}, n) \triangleright\ & \mathsf{def}\ sandbox\ [kill\langle\rangle \triangleright halt\langle\rangle : 0]\ \mathsf{in} \\ & \mathsf{let}\ \mathrm{f} = \mathrm{loader}(sandbox)\ \mathsf{in} \\ & \mathsf{let}\ r = \mathrm{f}(n)\ \mathsf{in} \\ & kill\langle\rangle \mid \mathrm{fail}(sandbox); \mathsf{reply}\ r\ \mathsf{to}\ \mathrm{run} \end{aligned}$$

More generally, for many patterns of site failure, it is possible to build a distributed join-calculus context that models the pattern. For instance, we may use internal choice to specify a distributed setting where a single arbitrary machine may fail non-deterministically, and use this setting to validate protocols that should resist only this particular class of failures.

7.2.4 Fault-tolerant protocols

Quite often, agent-based mobility turns out to be useful for writing programs that can resist the partial failure of some machines. Informally, migrations allow one to create larger, simpler units of failures; for instance, two machines that need to interact through a complex protocol may describe their part of the protocol as agents, then agree to run the protocol on a given machine. There, the two agents can interact without special care for partial failures, as long as these two locations do not communicate with the outside. If the machine fails, then the two locations are stopped at the same time, and the whole run of the protocol is invisible from the outside. Independently, each machine that sends its agent may monitor the failure of that agent.

We describe the use of the *fail* primitive, which provides a natural guard for error recovery. This failure-detection primitive gives the *certainty* of failure.

In some programs, this information is necessary for safety, or most useful for efficiency, because some protocols can then be cancelled and restarted. When detected, an asynchronous failure provides some useful information:

1. No message can be caused from further messages sent to the failed location, even indirectly.

2. All present and future live *fail* guards on that failure will be triggered.

Conversely, a message can still arrive from a location that is known to have failed, provided that this message was routed by a COMM step before the failure.

In some other programs, this information is less relevant: a classical model of silent crashes with timeout suffices, and is much easier to implement. Indeed, for RPC-like interactions without side effects, a simpler timeout may be used instead of a *fail* check. Even if the same RPC is issued several times, the caller can filter the answers, delivering the first one and discarding other late answers.

As defined, failure detection is difficult to implement efficiently: the latter property above provides in particular a reliable multicast primitive with guaranteed atomicity—either none or all *fail* guards on a given name are eventually triggered. In the discussion, we provide another definition that is simpler to implement. Our implementation does not currently support the full model of failure detection. A practical approach would provide failure detection only for some machines, and would centralize some of the failure information on presumably robust machines, or achieve the same robustness properties by using external protocols.

We illustrate these points on two examples, a simple extension of the CASA protocol, and a large distributed computation.

Building a robust CASA protocol We can make the CASA more robust by testing against the possible failure of the server location s, in which case a new agent is sent to another server s'. The client code now is:

$$\mathrm{f}(x, s) \triangleright \mathsf{def}\ a[\dots]\ \mathsf{in}\ \big(\mathrm{fail}(a); \mathsf{reply}\ \mathrm{f}(x, s')\ \mathsf{to}\ \mathrm{f}\big)$$

Let us consider the impact of a failure on s during a call to f:

- If the *fail* guard is triggered, then the server s must have failed while hosting agent a. As the first agent cannot return to the server, a new agent is created and sent to another server s'.

- Otherwise, the agent eventually migrates back to the client location, then returns. At that point, the agent can fail only if the client itself fails, hence the *fail* guard in the client is permanently disabled.

Anyway, we are assured that there is at most one agent at large, and that its action is only completed once (which might be quite important, e.g., if the action is "get a plane ticket"). This would still be true if the client did not know the server location, and the agent went through several intermediate sites before reaching the server location. Such properties would be hard to obtain with timeouts only.

Distributed computation on fallible machines Our final example models a simple CPU-intensive computation that can be partitioned into a series of independent jobs, in a data-parallel manner. These chunks can then be distributed over a network of machines willing to participate to the computation—the "clients". The client machines have no prior knowledge of the code to run, and they have poor reliability, so the termination of the computation should not be affected by the failure of some clients. The typical client can simply be written

$$\begin{array}{l}\mathsf{let}\ \mathit{join} = \text{name\_server.lookup("party")}\ \mathsf{in}\\ \mathsf{def}\ \mathit{worker}[\mathsf{T} : \mathsf{0}]\ \mathsf{in}\ \mathit{join}\langle \mathit{worker}\rangle\end{array}$$

(Where name_server.lookup("party") is a primitive call to the name-server, used to obtain the name *join* from a remote machine.) This code would be executed on a number of fallible machines, i.e., in contexts of the form $\{\mathit{halt}\langle\rangle \,|\, [\,\cdot\,]\}$.

The supervision of the computation is kept central, presumably on a reliable machine. Its code is written

```
def $join\langle there\rangle \triangleright$
    { $\mathrm{go}(there)$;
      def $\mathrm{f}(i) \triangleright$ reply $E$ to f in
      $\mathrm{enroll}(\mathrm{f}, there)$}

$\wedge$ $job\langle i\rangle \mid enroll\langle \mathrm{f}, there\rangle \triangleright$
    def   $once\langle\rangle \mid done\langle s\rangle \triangleright update\langle s\rangle \mid enroll\langle \mathrm{f}, there\rangle$
     $\wedge$    $once\langle\rangle \mid failed\langle\rangle \triangleright job\langle i\rangle$ in
    $once\langle\rangle \mid done\langle \mathrm{f}(i)\rangle \mid \mathrm{fail}(there); failed\langle\rangle$

$\wedge$ $status\langle n, s\rangle \mid update\langle ds\rangle \triangleright$
    let $s' = \mathrm{merge}(s, ds)$ in
    if $n > 0$ then $status\langle n-1, s'\rangle$ else $conclude\langle s\rangle$
in
    $name\_server.register\langle$"party"$, join\rangle$
  | $status\langle chunks - 1, 0\rangle$
  | for $i = 0$ to $chunks - 1$ do $job(i)$ done
```

and the following names parameterize the computation:

chunks : `Int` describes the number of chunks that must be processed to complete the computation;

merge : $\langle a, a\rangle \rightarrow \langle a\rangle$ is an associative–commutative function that describes how partial results can be combined;

f : $\langle$`Int`$\rangle \rightarrow \langle a\rangle$ represents some arbitrary computation that evaluates a given chunk and (presumably) does not perform any side effect; and

conclude : $\langle a\rangle$ makes the final result available to the context.

For each location name *there* received on *join*, a fresh location is created that contains the code of the computation. This location migrates with its code to *there*, registers as an active participant using the *enroll* message, then awaits for chunks to compute on f.

The remaining chunks to be processed are represented as messages $job\langle i\rangle$; they are dynamically attributed to the available participants by the join synchronization $job\langle i\rangle \mid enroll\langle \mathrm{f}, there\rangle$. Whenever a chunk has been delegated to a participant, the supervisor waits for two events: either the partial computation completes and sends its result back—then this result is asynchronously merged to the global result while the participant enrolls for another chunk—or the host machine fails—then the message $job\langle i\rangle$ is re-issued. These two events are made mutually exclusive by the single message $once\langle\rangle$.

The final part of the supervisor code simply aggregates the results as they arrive, using a counter to detect the termination of the global computation.

Provided that the central machine does not fail and that, from time to time, a machine remains alive for long enough to complete a chunk of the computation, it should be easy to establish that the computation eventually sends the final result on *conclude*.

Using the same model as for the lossy medium of Section 4.3, the supervision code in parallel with a process P that replicates fallible client machines is bisimular to the completed computation in parallel with the same process P, because at any point there is a sequence of reductions that unfold new clients, make them complete their chunk, and ship the results to the supervisor.

7.2.5 Other models for failure

We now relate our model of partial failure and failure detection—which we name the *asynchronous strong* model—to a few alternatives. For each alternative, we sketch a chemical semantics, and we discuss measures for failure recovery.

Loss of messages and timeouts The most conservative model of failure, in our message-passing setting, is that when a location fails, *some* messages to, at, or from the location are lost. This is for instance the usual assumption for low-level communication protocols such as IP or UDP, and also for object-oriented distributed systems. Unfortunately, this provides little support for developing reliable programs, as for nearly every message the application programmer must figure out what to do if the message is lost. Practical solutions usually involve the replication of messages, and an explicit bookkeeping of acknowledgments.

In our setting, it is straightforward to model the loss of messages in transit. Along with the rule COMM, we would add a similar rule FORGET that discards the message instead of forwarding it. According to the details of failures, this rule would be enabled when either the emitting machine or the receiving machine have failed, thus distinguishing between reliable and unreliable communication. As expected, unreliable communication creates some non-determinism for the routing of each message, and even simple protocols become complicated to study. Technically, it also seems that abstract fairness is inadequate for these failures, as it guarantees that replicated messages on failed machine eventually cross the network (*cf.* Section 4.10), which suggests of coarser equivalences.

Unfortunately, the detection of such failures is not very informative. It can be used to make another attempt, with no guarantees about the previous, dubious one. It can also reflect finer, non-asynchronous information that is present in the implementation, such as timeouts with good heuristics, and lead to more efficient programs.

Synchronous failure Going in the other direction, the *synchronous model* of failures is much simpler to deal with. When a location fails in this model, all the messages that the location emitted and that have not been received yet are immediately discarded on every machine.

In the distributed join-calculus, synchronous failures can be modeled by merging the two rules COMM and RED into a single, global rule GLOBAL-RED that consumes in one step all the messages in the join-pattern, directly from their emitting locations. Hence, messages emitted in one location remain in this location until they are consumed; routing and synchronization are performed all at once, which explain the name of the model.

Assuming that all messages from a failed agent are discarded makes programming much easier. Unfortunately this strong model is not compatible the COMM rule and our asynchronous, distributed setting. It would require that the system track and delete all messages issued by a failing location, which cannot be efficiently implemented on top of an asynchronous network.

Independently of the implementation issue, this choice of a synchronous model of failures for an asynchronous calculus is awkward, because the effect of a synchronous failures depends on the structure of the receiving definitions, which invalidates many properties of definitions (*cf.* Section 6.3). For instance, even relays on reliable machines could then be detected.

Nevertheless, the asynchronous detection of such failures would provide us more information about what cannot happen anymore than in our asynchronous model, and would thus lead to easier error recovery. Let us consider the process

$$P_0 \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ x\langle\rangle \mid y\langle\rangle \triangleright \mathit{test}\langle\rangle \wedge a\,[\mathsf{T} : \mathit{halt}\langle\rangle \mid x\langle\rangle]\ \mathsf{in}\ \mathrm{fail}(a); y\langle\rangle$$

The message $x\langle\rangle$ is available only before the failure, while the message $y\langle\rangle$ is available only after the failure, hence the join synchronization is impossible, and the message $\mathit{test}\langle\rangle$ cannot be emitted. In the synchronous model, we would thus obtain $P_0 \approx \mathsf{0}$. In the asynchronous model, however, the message $x\langle\rangle$ can be routed before the failure, and used after the failure, hence we obtain $P_0 \approx \mathsf{0} \oplus \mathit{test}\langle\rangle$.

To illustrate what separates synchronous and asynchronous models of failure, we describe a global encoding of synchronous communication in our distributed join-calculus. It is possible to recover the synchronous semantics for a given join-pattern as long as the names of the originating locations are communicated as an extra argument in every message. After (local) synchronization occurs, if for each received message the emitting machine answers a ping request, then both messages were emitted on a live location when the join-synchronization occurred, and the guard can be safely triggered. On the other hand, if one of the emitting machines fails before all machines answer the pings, then all other messages must be re-emitted with the same original location information. For instance, the strong encoding of the rule $x\langle u\rangle \mid y\langle v\rangle \triangleright P$ would be

$$\begin{array}{l}
x\langle u, a\rangle \mid y\langle v, b\rangle \triangleright \\
\quad \mathsf{def} \quad \mathit{once}\langle\rangle \mid x'\langle\rangle \mid y'\langle\rangle \triangleright P \\
\quad\ \wedge \quad \mathit{once}\langle\rangle \mid \mathit{failed}\langle\rangle \triangleright x\langle u, a\rangle \mid y\langle v, b\rangle\ \mathsf{in} \\
\quad \mathit{once}\langle\rangle \mid \mathrm{ping}(a); x'\langle\rangle \mid \mathrm{ping}(b); y'\langle\rangle \mid \mathrm{fail}(a); \mathit{failed}\langle\rangle \mid \mathrm{fail}(b); \mathit{failed}\langle\rangle
\end{array}$$

Except for the problem of gradual commitment to a particular synchronization, the encoding accurately implements the synchronous model in the strong asynchronous model with failure detection.

Besides, (1) the treatment of partial failure could be refined to avoid the re-emission of messages whose originator has failed, and (2) the *fail* primitives that guard the message $\mathit{failed}\langle\rangle$ could be removed; the resulting variant would provide an accurate encoding of the strong model in the weaker, timeout model, but also introduces divergence computations, since each synchronization attempt would becomes reversible.

Asynchronous, weak We finally describe a model that is slightly weaker than the one of the distributed join-calculus, but closer to our implementation strategy.

In practice, it is hard to guarantee that a machine has actually stopped, but it is possible to ignore its subsequent messages. In the *asynchronous weak* model of failure, a failed location is a location that cannot be affected by a live location. This does not prevent the failed location sending messages to other machines, which may or may not discard them. Chemically, this can be modeled by modifying the side conditions on the rules COMM and GO, so that communication may succeed only towards a live location, and that migration toward a dead location is replaced with halting. In addition, we may provide a rule FORGET that can discard messages and migrations when they are issued on a failed machine.

The relation between the strong and weak models of asynchronous failures is intriguing. On one hand, the change of model does not preserve bisimulation because the internal choices are not interleaved with the same visible interactions. For instance, the process $a\,[\mathsf{T} : \mathit{halt}\langle\rangle \mid x\langle\rangle \mid x\langle\rangle]$ reduces to $\Omega a\,[\mathsf{T} : x\langle\rangle]$ in the weak semantics, in a state where at most one $x\langle\rangle$ is emitted to the outside, and this partial commitment cannot be mimicked in the strong asynchronous semantics. Fair testing is also broken because the abstract fairness requirement would demand that replicated messages in failed location always be routed to the outside.

On the other hand, the traces are clearly the same. For instance, may testing coincides in the weak asynchronous model and the strong one, at least for processes that have no initially-failed locations: strong execution traces are valid weak execution traces, and conversely weak execution traces can be turned into strong execution traces by delaying the failures until all the required COMM and GO have been performed. In particular, we are justified in using the stricter, simpler model in the calculus, but only implementing the weaker one.

7.3 Proofs for mobile protocols

We lift our hierarchy of equivalences to the distributed join-calculus, and present a few typical equations and properties in the presence of failures.

The primary purpose of our calculus is to provide a foundation for a core language that is expressive enough for distributed and mobile programming. But locations with their primitives can also be used to model fallible distributed environments, as specific contexts within the calculus. As a result, we can use our observational equivalence to relate precisely the distributed implementations with their specification (i.e. simpler programs and contexts without failures or distribution). In combination with the proof methods developed in Chapters 4 and 5, this provide a framework for the design and the proof of distributed programs under realistic assumptions.

We first adapt the definitions of contexts (Definition 4.3) and output barbs to the distributed calculus.

Definition 7.3 *An evaluation context is a linear context whose hole is not situated under a join-pattern guard. A* live context *is an evaluation context whose hole is in a live location. A process P has a strong barb on $x \in \mathcal{N}$ (written $P \downarrow_x$) when we have $P \equiv C[x\langle\rangle]$ for some live context $C[\,\cdot\,]$ that does not bind x.*

In the following, we use the instances of the equivalences defined in Chapter 4 obtained for these definitions of observation and context, and for the distributed join-calculus of Section 7.2. We also use the same notations; for instance, the equivalence relation $\approx$ is the barbed congruence defined in Section 4.4 applied to distributed processes and contexts.

Since by definition failure can occur only in a named location, the top-level solution $\vdash$ provides a "safe haven" where pervasive definitions, such as encodings of functions or data structures, may be put—in a distributed implementation, such immutable components can be efficiently implemented with the same semantics guarantees by using replication. This suggests the use of different notions of congruence properties; for instance, we may distinguish a "static equivalence" that is a congruence for all but the $\varepsilon a\,[\cdot : \cdot]$ constructor, and retain most of the properties of equivalences in the plain join-calculus, and a "mobile equivalence" that is a congruence for the full calculus, as studied here.

As suggested in the previous section, contexts that use location constructors and *halt* (or *go* to fallible locations) increase the discriminative power of our equivalences, because the possibility of performing internal reduction steps disappear in case of failure, which renders such reductions partially visible. For instance, the relay equation of the plain calculus is broken when the relay can be put in a fallible location:

$$x\langle y\rangle \not\approx \mathsf{def}\ z\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle z\rangle$$

These two processes can be separated in the context

$$C[\,\cdot\,] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ x\langle u\rangle \triangleright u\langle\rangle \mid v\langle\rangle\ \mathsf{in}\ \{\mathit{halt}\langle\rangle \mid [\,\cdot\,]\}$$

because only $C[\mathsf{def}\ z\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle z\rangle]$ may reduce to a state with a barb on v and no barb on u:

$$C[\mathsf{def}\ z\langle\rangle \triangleright y\langle\rangle\ \mathsf{in}\ x\langle z\rangle] \quad \rightarrow\rightarrow\rightarrow \quad \mathsf{def}\ x\langle u\rangle \triangleright u\langle\rangle \mid v\langle\rangle \wedge \Omega a\,[z\langle\rangle \triangleright y\langle\rangle : z\langle\rangle]\ \mathsf{in}\ v\langle\rangle$$

7.3.1 A few simplifying equations

The notion of dead location is invariant through reduction; this enables us to discard the contents of dead locations. We first state several "garbage collection" laws which are useful for simplifying processes when some parts of the computation have failed.

Lemma 7.4 *Let $C[\,\cdot\,]$ be an evaluation context that binds a to a dead location. Under the context $C[\,\cdot\,]$, we can rewrite terms up to barbed congruence as follows:*

1. *substitute* $\mathsf{0}$ *for all processes in the dead location a;*
2. *simplify the definitions D in the failed location a as in Section 6.3.2, leaving only empty rules $x\langle\widetilde{u}\rangle \triangleright \mathsf{0}$ for every port name $x \in \mathsf{dv}[D]$ and empty locations $b\,[\mathsf{T} : \mathsf{0}]$ for every sublocation name $b \in \mathsf{dv}[D]$, in the case these names still appear in a live location.*
3. *delete any message sent on a name defined in the dead location a;*
4. *substitute $\kappa\langle\rangle$ for all processes $\mathit{fail}\langle a, \kappa\rangle$;*
5. *substitute $\mathit{halt}\langle\rangle$ for all processes $\mathit{go}\langle a, \kappa\rangle$.*

Each of these simplifications is easily established by a case analysis on reductions, on fully-diluted distributed solutions. For instance we have for all processes P, Q, and R

$$\begin{aligned} & \mathsf{def}\ \Omega a\,[x\langle u\rangle \mid y\langle v\rangle \triangleright P : Q]\ \mathsf{in}\ \{z\langle x\rangle \mid \mathrm{go}(a); R\} \mid \mathrm{fail}(a); (t\langle\rangle \mid y\langle 5\rangle) \\ \approx\ & \mathsf{def}\ \Omega a\,[x\langle u\rangle \triangleright 0 : 0]\ \mathsf{in}\ \{z\langle x\rangle \mid \mathit{halt}\langle\rangle\} \mid t\langle\rangle \\ \approx\ & \mathsf{def}\ x\langle u\rangle \triangleright 0\ \mathsf{in}\ \{z\langle x\rangle \mid \mathit{halt}\langle\rangle\} \mid t\langle\rangle \end{aligned}$$

Second, some basic laws hold for the $go\langle\cdot,\cdot\rangle$, $fail\langle\cdot,\cdot\rangle$, and $halt\langle\rangle$ primitives, independently of the tree structure. For instance, we have for failure detection

$$\mathrm{fail}(a); \mathrm{fail}(b); P \approx \mathrm{fail}(b); \mathrm{fail}(a); P \tag{7.1}$$
$$\mathrm{fail}(a); \mathrm{fail}(a); P \approx \mathrm{fail}(a); P \tag{7.2}$$
$$\mathrm{fail}(a); P \mid \mathrm{fail}(a); Q \approx \mathrm{fail}(a); (P \mid Q) \tag{7.3}$$
$$\mathrm{fail}(a); \mathrm{ping}(a); P \approx 0 \tag{7.4}$$

These primitives are strictly static, thus it is immediate to check whether a location may ever move or halt on its own. Besides, the analysis of the local usage of these primitives may yield simplifications of the location tree. The following results show, for instance, how to get rid of a location once it has reached its final destination; they can be applied to most of our examples with mobile agents.

Lemma 7.5 *Let $a, b \in \mathcal{L}$. Let D,D',D'' and P,P',P'' be respectively definitions and processes in the distributed join-calculus such that all occurrences of the primitives $go\langle\cdot,\cdot\rangle$ and $halt\langle\rangle$ in D, D' and P, P' occur only in (strict) sublocations, and such that $\mathsf{dv}[D] \cap \mathsf{dv}[D'] = \emptyset$.*

1. *For all evaluation contexts for definitions $C[\,\cdot\,]$ such that the first process below is well-formed, we have the barbed congruence*

$$\begin{aligned} & C\Big[\varepsilon a\,\big[b\,[D \wedge D' : P \mid P'] \wedge D'' : P''\big]\Big] \\ \approx\ & C\Big[\varepsilon a\,\big[b\,[D : P] \wedge D' \wedge D'' : P' \mid P''\big]\Big] \end{aligned}$$

2. *Let $\sigma = \{^a/_b\}$, $C[\,\cdot\,]$ be an evaluation context for definitions such that the first process below is well-formed, $C'[\,\cdot\,]$ be the context obtained from $C[\,\cdot\,]$ by applying σ from within $C[\,\cdot\,]$. We have*

$$\begin{aligned} & C\Big[\varepsilon a\,\big[b\,[D : P] \wedge D'' : P''\big]\Big] \\ \approx\ & C'\Big[\varepsilon a\,\big[D\sigma \wedge D''\sigma : P\sigma \wedge P''\sigma\big]\Big] \end{aligned}$$

The first part of the lemma states that any component within a location that does not move or fail of its own can be relocated at the enclosing location, up to barbed congruence. The second part states that, when a location is empty, migrations and failure-detections using its name b or its parent's name a cannot be distinguished. The substitution $\{^a/_b\}$ makes explicit that a can be used instead of b for migration or failure-detection. In particular, these equations show that, after a spawn of a plain join-calculus process, the boundaries of the moving location can be erased up to barbed congruence.

Proof: Let $\mathcal{R}$ be the relation that contains all pairs of processes (P_1, P_2) of the first statement of the lemma. We prove that $\mathcal{R}$ is a congruence and a barbed bisimulation up to structural rearrangements

1. The relation $\mathcal{R}$ is closed by application of any evaluation context.
2. The strong barbs are the same on both sides of $\mathcal{R}$. In particular, if the message $x\langle\rangle$ is in P' or D', then we remark that a and b are either both dead or both alive. Otherwise, the barb is syntactically the same.
3. To establish the weak bisimulation requirement, we perform a case analysis on the chemical reduction rule being used. All reductions that involve only the context are in direct correspondence. Otherwise, let $P_1 \mathcal{R} P_2$;

   COMM Let $x\langle\widetilde{u}\rangle$ be the message being routed. In a few specific cases, no routing is required on the other side of $\mathcal{R}$ to remain in the relation. This is the case when (1) $x \in \mathsf{dv}[D']$ and the message is routed from D'' or P'' in P_1, or from D or P in P_2; (2) $x \in \mathsf{dv}[D'']$ and the message is routed from D' or P' is P_1; (3) $x \in \mathsf{dv}[D]$ and the message is routed from D' or P' in P_2.

   In all other cases the COMM step are in bijection and lead to processes in $\mathcal{R}$.

   RED Let $J \triangleright R$ be the rule used for the reduction. By definition of $\mathcal{R}$, a and b have the same liveness status, hence the rules in D' and P' can be used on both sides of $\mathcal{R}$.

   In a few specific cases, the reduction uses messages that have not been routed yet on the other side of $\mathcal{R}$. This is the case when (1) the rule is in D and uses messages from D', P' in P_1; (2) the rule is in D', and uses messages from D, P in P_1, or from D'', P'' in P_2; (3) the rule is in D'' and uses messages from D', P' in P_2. For all these cases, we have

$$P_2 \overset{\text{RED}}{\rightarrow} P_2' \quad \text{implies} \quad P_1 \overset{\text{COMM}^*}{\rightarrow} \overset{\text{RED}}{\rightarrow} P_1' \mathcal{R}' P_2'$$

   (and vice-versa when $P_2 \overset{\text{RED}}{\rightarrow} P_2'$). All other reductions are in direct correspondence, and remain in $\mathcal{R}$.

   GO By hypothesis, the reduction is not triggered from D, D' or P, P', and a and b have the same liveness status. When a migrates to another location (go in P'') , we remain in the relation for another context $C[\cdot]$. When a location migrates to a or b, it is added to D'' or D, respectively, hence in all cases a GO on the other side leads to a pair of processes in $\mathcal{R}$.

   HALT By hypothesis the halting location cannot be b. The same location thus halts on both sides, and lead to related processes (including the special case of an $halt\langle\rangle$ in P'', by replacing εa by Ωa in P_1 and P_2.)

   DETECT Locations in P_1 and P_2 all have the same status, hence the same *fail*s may reduce on both sides of $\mathcal{R}$.

The second part of the lemma can be established by a similar case analysis on each chemical reduction rule. Again, some COMM steps that are necessary before substitution disappear after substitution. Moreover, the absence of go in P rules out migration to a that would become circular migrations. $\square$

7.3.2 Failures and atomicity

We provide a few examples that illustrate how partial failures can reveal non-atomic steps, and also how locations can be used to recover atomicity.

The distributed join-calculus provides two separate mechanisms for global interaction: global communication and agent-based migration. While these mechanisms are very different in our implementation, global communication would almost be subsumed by agent migration if routing were made explicit: we could use a spawn for each message, and restrict the rule Comm to messages that are in a location immediately under the defining location.

The following equation states this property by substituting a Go step for a Comm step for routing the message $x\langle\widetilde{v}\rangle$. Provided that the name x is defined in location a, we have

$$\{go\langle a\rangle; x\langle\widetilde{v}\rangle\} \approx x\langle\widetilde{v}\rangle$$

Although another Comm step is still needed after the migration to exit the anonymous location, this step cannot be prevented by any partial failure, as long as the receiving definition remains alive.

For more general processes, however, the explicit "spawn" form gives additional guarantee about atomicity. for instance

$$\{go\langle a\rangle; P\} \,|\, \{go\langle a\rangle; Q\} \not\approx \{go\langle a\rangle; (P \,|\, Q)\}$$

The process on the right is more efficient, and has simpler properties, because P and Q cannot fail independently of one another. This suggests a constructive usage of migration to build atomicity, for instance to ensures that a few related messages be either all reliably sent or all discarded, and also that objective moves have much simpler properties than subjective ones; for example, we have the barbed congruence relation

$$\begin{array}{rl} & \mathsf{def}\ a\,[D_a : P_a] \wedge D' \ \mathsf{in}\ \{\{go\langle a\rangle; P\} \,|\, halt\langle\rangle\} \,|\, P' \\ \approx & \mathsf{def}\ a\,[D_a : P_a \,|(\{P\} \oplus 0)] \wedge D' \ \mathsf{in}\ P' \end{array}$$

no matter of the interaction that occurs later between P and D_a, P_a. Of, course, this would not the case if another process was running in parallel with the $halt\langle\rangle$.

When a process is executed in a fallible location, messages in parallel composition are routed independently; unless the messages are local, there is no atomicity in $x\langle\rangle \,|\, y\langle\rangle$ and an enclosing failure may cause any or both of the messages to get lost. Let us investigate the behavior of parallel composition of such messages in more details. For all sets S of messages, we let P_S be the process $\{halt\langle\rangle \,|\, \prod_{M\in S} M\}$.

The bisimulation-based behavior of P_S is quite complicated, because there are many intermediate stages in which some messages have been successfully routed while the others are still subject to immediate failures. Precisely, for $S = \{M_1, \ldots, M_n\}$, we have the strong barbed congruence

$$P_S \quad \sim \quad 0 \oplus (M_1 \,|\, P_{S\setminus M_1}) \oplus \cdots \oplus (M_n \,|\, P_{S\setminus M_n})$$

and by induction we can unfold a synchronization tree with an exponential number of states that are all separated by the weak barbed congruence $\approx$.

Fortunately, coupled simulations—and a fortiori fair testing— yields a much simpler model that does not discriminate according to the possibility of losing all the messages in a single step. In the same setting, we have

$$P_S \quad \lessgtr \quad \prod_{M \in S} (M \oplus \mathsf{0})$$

and in particular, if S and T are two disjoint sets of messages, we have

$$\{P_S \mid \mathit{halt}\langle\rangle\} \mid \{P_T \mid \mathit{halt}\langle\rangle\} \quad \lessgtr \quad \{P_{S \cup T} \mid \mathit{halt}\langle\rangle\}$$

7.4 Related work

The field of mobile computation is new, multiform, and it evolves quickly. We do not attempt to give a comprehensive survey of the area, but rather provide a few points of comparison.

Our calculus aims at simplicity, and focuses on a particular model of distributed communication and partial failures. It does not address numerous important issues often found in more complex designs for distributed systems. For instance, our model is not adequate to deal with replication-based techniques, mobile computing, or intermittent connectivity.

Also, the terms mobility and locality already have other meanings, in particular in process calculi. Mobility in the π-calculus refers to the communication of channel names on channels [99], whereas locality has been used as a tool to capture spatial dependencies among processes in non-interleaving semantics [41, 133].

7.4.1 Applets in Java

Since the beginning of this work, the Java programming language and its widespread implementation in web browsers have popularized the idea of *mobile code* that can be downloaded from a server on demand.

While code mobility is usually a prerequisite for agent-based mobility, the latter is more general and more expressive. In Java, the computation is mostly local, even if the code is dynamically assembled. Besides, the distributed aspects of the computation rely on standard techniques, such as sockets for data exchange between the client machine and the applet server or, more recently, libraries for remote method invocation [150].

On the contrary, locations—and most other forms of mobile agents—can carry running processes, local state, and active communication capabilities from one machine to another. As expected, most of the interesting problems raised by applets in Java, e.g., security, are all the more serious in our setting.

7.4.2 Migration as a programming language feature

Migration has been investigated mostly for object-oriented languages. Initially used in distributed systems to achieve a better load-balancing, migration evolves to a language feature in the Emerald programming language [77, 79, 78, 125]. Objects can be *moved* from one machine to another. Emerald objects also have a nested structures as regards their migratory behavior: objects can be *attached* to one another, an object carrying its attached objects as it moves. At the language level, numerous novel calling conventions such as call-by-move reflect these capabilities, and the use of migration for safety purposes is already advocated.

As a system programming project, the emphasis is on efficiency on local-area-networks, rather than on the precise semantics of distributed objects. Remote method invocation may be silently discarded, and, when a machine stops, it is entirely responsible for error recovery when it restarts; in between, no failure notification is available to the programmer. Also, the localization of objects at run-time can be hard to trace, because numerous migrations may occur as side effects of method calls and object attachments.

7.4.3 Migration as a programming paradigm

The Telescript programming language [149] is *entirely* based on mobile agents for all the distributed aspects of the computation. Agents carry some code and resources; they control their own migration, and dynamically gain access to the local environment of their transient host machine. The language designers advocate the use of mobile agents instead of the traditional client-server architecture, and also suggest that some "meeting hall" machines be used to hosts the different components of a protocol, in a neutral and reliable environment. The system apparently guarantees the correct execution of the distributed computation by check-pointing agents as they move from one site to another; there is no built-in mechanism for failure detection.

7.4.4 Agent-based mobility and network transparency

Independently. several languages have been proposed for large-scale distributed programming, with some support for mobile agents. For instance, Obliq [44] encodes migration as a combination of remote cloning and aliasing, in an object-oriented language with both network transparency and global distributed scope. Examples of applications with large-grain mobility in Obliq can be found in [31].

In a functional setting, FACILE [60] provides process mobility from site to site, as the communication of higher-order values. As in this dissertation, the design choices are discussed in a chemical framework [90]. Implementations issues and numerous examples of high-level agent-based programs can be found in Knabe's dissertation [81].

7.4.5 Locality and failures

In order to model the properties of FACILE, Amadio and Prasad [17] developed a process calculus with locality and failure. Their representation of failures in a refined π-calculus setting with localities is closely related to our distributed join-calculus.

In the π_l-calculus, the authors extend the syntax of the π-calculus with localities. Channels are statically located; a location can fail, preventing further communication on its channels; location status can be tested in the language. Locations in the π_l-calculus have a flat and static structure, which suffices to study failure and failure recovery in the absence of migrations. Observation in the presence of failures becomes quite different from the usual observation, but the authors provide an explicit encoding of the π_l-calculus in the π-calculus and prove its adequacy.

7.4.6 Modeling heterogeneous networks

More recently, several languages and calculi have addressed some aspects of distributed computation in a more explicit manner. In particular, network transparency is ruled out. On the contrary, global interaction is highly constrained, in order to reflect the limitations found in heterogeneous networks of machines [45].

For example, the Ambient calculus of Cardelli and Gordon [46] exposes the details of routing. Ambients and locations are both organized as a tree of named multisets of terms meant to models the physical distribution of resources. However, the boundaries of Ambients are mostly opaque, while the boundaries of locations are mostly transparent.

For instance, our Comm and Go rules may implicitly cause a message—or even an agent—to exit several locations and enter several others, in a single reduction step. Conversely, an ambient that wish to migrate from one enclosing ambient to a remote ambient must explicitly exit a series of nested Ambients, then enter another series of nested Ambients. Moreover, the moving ambient must precisely know the route, including the names of all the intermediate Ambients, and requires the cooperation of each intermediate ambient.

Likewise, Hennessy and Riely provide a much refined account of global, channel-based communication in a typed π-calculus, with a precise control of the capabilities being exchanged [68]. Type information provides a sound basis for specifying security properties. In [129], they also present a simple model of failures similar to ours, and propose symbolic, labeled-based proof techniques to study the properties of π-calculus processes in the presence of failures.

These approaches seem complementary to ours; in particular, these calculi are mostly used for specification purposes—not for general purpose distributed programming. Besides, their underlying communication mechanisms provide strong guarantees of atomicity, which render their distributed implementation at least as difficult as for the π-calculus.

Conclusions

In this dissertation, we developed a formalism that fits the needs of distributed programming, explored its use as the core of a programming language, and studied some of its formal properties. Our approach departs from traditional studies of process calculi, which are more interested in specifications of protocols than in actual programs and implementations.

While our initial goal was to obtain a simple core language whose distributed implementation would be straightforward, our formalism turns out to be adequate also for writing general-purpose parallel programs. The join-calculus can be presented as a natural extension of functional languages, and we believe that this is of pragmatic importance for the programmer.

The join-calculus inherits most of the properties of the asynchronous π-calculus. The essential difference between the two is that in the join-calculus all the receivers at a given channel name are statically known. The synchronization behavior of names is entirely declared as join-patterns when the names are introduced in a defining process, compiled as a whole, and mapped to a single machine at run-time. This so-called locality property provides a lot of static information. It can be used to implement routing in a deterministic manner, to analyze these name definitions, and to optimize their representation. It also facilitates the transfer of techniques developed for other languages. For instance, we could equip the join-calculus with an implicit polymorphic type system, and rely on local data structures similar to those of ML in our prototype implementation.

We validated our model in practice by developing a prototype distributed implementation, and this experiment had a major impact on the join-calculus. Many programming examples are already available, yet more experience is required to assess the merits of the primitives for migration and failures. Also, there is still a gap between what happens in the distributed chemical machine and in its implementation as a series of machines executing the local runtime. Since the underlying protocols are rather delicate, it would be worthwhile to study them in more details, for instance as a chemical refinement that describes runtime representations hidden in the distributed RCHAM.

As we introduced yet another process calculus, there is an obvious drawback: in order to tackle formally the properties of programs written in the join-calculus we first had to build its meta-theory. Our initial plan was to apply known results of concurrency theory, but to our surprise we often had to develop our own tools and techniques to deal with equivalences in the join-calculus, especially as regards observational equivalences and asynchrony. Fortunately, most of these developments are also relevant to other asynchronous calculi. Yet, a fundamental issue is to build a more general framework where such common properties could be established once for all. More specifically, the formal analysis of protocols with migrations and partial failures is far from complete, and probably requires tools more convenient than barbed bisimulation congruences.

From a more general point of view, many recent works on process calculi attempt to bring such formalisms closer to the programming practice—in particular, to the distributed aspects of computation—without compromising precise and simple foundations. Interestingly, some of these models explore more radical choices as regards distributed programming; for instance they substitute lower-level, more dynamic communication behavior to location transparency. Our hope is that the join-calculus contributes to this trend toward practicality, and leads to a better understanding between concurrency theoreticians and distributed programmers.

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In Larsen et al. [84], pages 868–883. Also Digital SRC Research Report 154, April 1998.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In LICS '98 [75], pages 105–116.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, April 1997.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, January 1997. Extended version of both [3] and [5].

[5] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In Mazurkiewicz and Winkowski [93], pages 59–73.

[6] ACM. *Conference record of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, January 1995.

[7] ACM. *Conference record of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, January 1996.

[8] ACM. *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996.

[9] ACM. *Conference record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, January 1997.

[10] ACM. *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, January 1998.

[11] Gul Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1987.

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

[13] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. also Rapport Interne LIM February 1997, and INRIA Rapport de recherche 3109.

[14] Roberto M. Amadio. On modelling mobility. To appear in Theoretical Computer Science, 1998.

[15] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[16] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. Also INRIA Rapport de recherche 2913, June 1996. An extended abstract appeared in [103].

[17] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. In P.S. Thiagarajan, editor, *Proceedings of the 14th Foundations of Software Technology and Theoretical Computer Science Conference (FST-TCS '94)*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1994.

[18] Jean-Marc Andreoli, Lone Leth, Remo Pareschi, and Bent Thomsen. On the chemistry of broadcasting. ECRC, Munich, 1992.

[19] Jean-Marc Andreoli, Lone Leth, Remo Pareschi, and Bent Thomsen. True concurrency semantics for a linear logic programming language with broadcast communication. In *Proceedings TAPSOFT '93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[20] Jean-Marc Andreoli and Remo Pareschi. Communication as fair distribution of knowledge. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 212–229, November 1991.

[21] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[22] Andrea Asperti and Nadia Busi. Mobile petri nets. Technical report, Department of Computer Science, University of Bologna, May 1996.

[23] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. Distributed system structuring using multi-functions. Rapport de recherche 694, Institut National de Recherche en Informatique et Automatisme Rennes, June 1987.

[24] Jean-Pierre Banâtre, A. Coutant, and Daniel Le Métayer. A parallel machine for multiset transformation an its programming style. *Future Generation Computing Systems*, 4:133–144, 1988.

[25] Jean-Pierre Banâtre and Daniel Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[26] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36:98–111, 1993.

[27] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model: ten years after. Rapport de recherche 984, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, February 1996.

[28] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings POPL '90*, pages 81–94, San Francisco, January 1990.

[29] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[30] Eike Best, editor. *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, volume 715 of *Lecture Notes in Computer Science*, Hildesheim, Germany, 1993. Springer-Verlag.

[31] Krishna A. Bharat and Luca Cardelli. Migratory applications. Research Report 138, Digital SRC, February 1996.

[32] Andrew Birell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Research Report 115, Digital SRC, 1994.

[33] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998. An extended abstract appeared in [103], pages 163–178.

[34] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, August 1995.

[35] Michele Boreale, Cédric Fournet, and Cosimo Laneve. Bisimulations in the join-calculus. In *Proceedings of PROCOMET '98*. IFIP, Chapman and Hall, June 1998. To appear.

[36] Michele Boreale and Davide Sangiorgi. Bisimulation in name-passing calculi without matching. In LICS '98 [75], pages 165–175.

[37] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de recherche 1702, INRIA Sophia-Antipolis, May 1992.

[38] Gérard Boudol. Some chemical abstract machines. In J. W. de Bakker, W.-P. de Roever, and Grzegorz Rozenberg, editors, *A Decade of concurrency: reflections and perspectives: REX school/symposium, Noordwijkerhout, the Netherlands, June 1–4, 1993: proceedings*, volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer-Verlag, 1994.

[39] Gérard Boudol. The π-calculus in direct style. In POPL '97 [9], pages 228–241.

[40] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. Observing localities. *Theoretical Computer Science*, 114, 1993.

[41] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994. A shorter version appeared in Proceedings of CONCUR '92, Lecture Notes in Computer Science 630, pages 108–123.

[42] Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In Lee and Smolka [89], pages 313–327.

[43] Ed Brinksma, Arend Rensink, and Walter Vogler. Applications of fair testing. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX: Theory, Applications and Tools*, volume IX. Chapman and Hall, 1996.

[44] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. A preliminary version appeared in [6].

[45] Luca Cardelli. Global computation. *ACM Sigplan Notices*, 32:1:66–68, 1997.

[46] Luca Cardelli and Andrew Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[47] Ilaria Castellani. Observing distribution in processes: Static and dynamic localities. *International Journal of Foundations of Computer Science*, 6(4):353–393, 1995.

[48] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[49] Silvano Dal-Zilio. Quiet and bouncing objects: Two migration abstractions in a simple distributed blue calculus. In Hans Hüttel and Uwe Nestmann, editors, *Proceedings of the Worshop on Semantics of Objects as Proceedings (SOAP '98), Aalborg, Denmark*, number NS-98-5 in BRICS Notes Series, pages 35–42, June 1998.

[50] L. Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings on Principles of Programmining Languages*, pages 207–212, 1982.

[51] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[52] Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors. *24th Colloquium on Automata, Languages and Programming (ICALP '97)*, volume 1256 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[53] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[54] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The design of a parallel functional language. In ICFP '96 [8], pages 169–179.

[55] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In POPL '96 [7], pages 372–385.

[56] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi (extended abstract). In Larsen et al. [84], pages 844–855.

[57] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Montanari and Sassone [103], pages 406–421.

[58] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the join-calculus. In Mazurkiewicz and Winkowski [93], pages 196–212.

[59] Cédric Fournet and Luc Maranget. The join-calculus language (version 1.03 beta). Source distribution and documentation available from `http://join.inria.fr/`, June 1997.

[60] A. Giacalone, P. Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989. Also in TAPSOFT '89, pages 184-209, Springer-Verlag, Lecture Notes in Computer Science 352 (1989).

[61] Rob J. van Glabbeek. The linear time—branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In Best [30], pages 66–81. Also Manuscript, preliminary version available at `ftp://Boole.stanford.edu/pub/spectrum.ps.gz`.

[62] Rob J. van Glabbeek and Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[63] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: reduction and typing. In Nestmann and Pierce [110]. To appear.

[64] James Gosling, Bill Joy, and Guy Steele. Java language specification, version 1.0. August 1996.

[65] David Harel, Orna Kupferman, and Moshe Vardi. On the complexity of verifying concurrent transition systems. In Degano et al. [52].

[66] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[67] Matthew Hennessy. A model for the pi-calculus. Computer Science Technical Report 91:08, COGS, University of Sussex, 1991. To appear in Acta Informatica.

[68] Matthew Hennessy and James Riely. A typed language for distributed mobile processes. In POPL '98 [10], pages 378–390.

[69] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[70] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag.

[71] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In P. Wegner, M. Tokoro, and O. Nierstrasz, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, volume 612 of *Lecture Notes in Computer Science*, pages 21–51. Springer-Verlag, 1992.

[72] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *Proceedings POPL '94*, pages 348–360, 1994.

[73] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[74] Hans Hüttel and Josva Kleist. Objects as mobile processes. Research Series RS-96-38, BRICS, October 1996. Presented at MFPS '96.

[75] IEEE. *Thirteenth Symposium on Logic in Computer Science (LICS '98, Indianapolis)*, June 1998.

[76] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In Best [30], pages 158–172.

[77] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, Computer Science Department, December 1988.

[78] Eric Jul. Migration of light-weight processes in emerald. *IEEE Operating Sys. Technical Committee Newsletter, Special Issue on Process Migration*, 3(1):20, 1989.

[79] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 62–74, November 1987.

[80] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[81] Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995. CMU-CS-95-223; also published as Technical Report ECRC-95-36.

[82] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In POPL '96 [7], pages 358–371.

[83] Cosimo Laneve. May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, March 1996. Revised: May 1996.

[84] Kim Larsen, Sven Skyum, and Glynn Winskel, editors. *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *Lecture Notes in Computer Science*, Aalborg, Denmark, July 1998. Springer-Verlag.

[85] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[86] Fabrice Le Fessant. The JoCAML system prototype. Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.

[87] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. In Nestmann and Pierce [110]. To appear.

[88] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Conference on Programming Language Design and Implementation (PLDI '98)*, Montreal (Canada), June 1998. ACM SIGPLAN.

[89] Insup Lee and Scott A. Smolka, editors. *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95, Philadelphia)*, volume 962 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[90] Lone Leth and Bent Thomsen. Some facile chemistry. Technical Report ECRC-92-14, European Computer-Industry Research Centre, Munich, May 1992.

[91] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. Thèse d'état, Université Paris VII, January 1978.

[92] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.

[93] Antoni Mazurkiewicz and Jòzef Winkowski, editors. *Proceedings of the 8th International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.

[94] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Larsen et al. [84], pages 856–867.

[95] Daniel Le Métayer. Higher-order multiset programming. In *Proceedings of the DIMACS worshop on specification of parallel algorithms*, volume 18 of *Dimacs series on Discrete Mathematics*. AMS, 1994.

[96] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[97] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[98] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Preliminary versions appeared in ICALP '90, Lecture Notes in Computer Science 443, pages 167–180, and as INRIA Rapport de recherche 1154, 1990.

[99] Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993. Also appeared as technical report ECS–LFCS–91–180, University of Edinburgh, UK, 1991.

[100] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[101] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695, Vienna, 1992. Springer-Verlag.

[102] Ugo Montanari and Marco Pistore. Checking bisimilarity for finitary π-calculus. In Lee and Smolka [89], pages 42–56.

[103] Ugo Montanari and Vladimiro Sassone, editors. *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*, Pisa, Italy, August 1996. Springer-Verlag.

[104] James H. Morris, Jr. *Lambda-Calculus Models of Programming Languages.* Ph. D. dissertation, MIT, December 1968. Report No. MAC–TR–57.

[105] V. Natarajan and Rance Cleaveland. Divergence and fair testing. In *Proceedings of ICALP '95*, volume 944 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[106] Uwe Nestmann. *On Determinacy and Nondeterminacy in Concurrent Programming.* PhD thesis, Technische Fakultät, Universität Erlangen, November 1996. Arbeitsbericht IMMD-29(14).

[107] Uwe Nestmann. What is a 'good' encoding of guarded choice? In Catuscia Palamidessi and Joachim Parrow, editors, *Proceedings of EXPRESS '97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1997. Full version as report BRICS-RS-97-45, Universities of Aalborg and Århus, Denmark, 1997.

[108] Uwe Nestmann. On the expressive power of joint input. To appear, 1998.

[109] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Montanari and Sassone [103], pages 179–194. Revised full version as report ERCIM-10/97-R051, European Research Consortium for Informatics and Mathematics, 1997.

[110] Uwe Nestmann and Benjamin C. Pierce, editors. *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*, Nice, France, September 1998. Elsevier Science Publishers. To appear.

[111] Martin Odersky. Applying π: Towards a basis for concurrent imperative programming. In *Proc. 2nd ACM SIGPLAN Workshop on State in Programming Languages*, pages 95–108, January 1995.

[112] Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126:259–280, 1994.

[113] Luca Padovani. The Bologna join system., 1997. Software and documentation (in italian) available electronically at `ftp://ftp.cs.unibo.it/pub/asperti/jcb01.tar.gz`.

[114] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In POPL '97 [9], pages 256–265.

[115] D. M. R. Park. *Concurrency and Automata on Infinite Sequences*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[116] Joachim Parrow. Trios in concert. In Plotkin et al. [123]. To appear.

[117] Joachim Parrow and Peter Sjödin. Multiway synchronization verified with coupled simulation. In Rance Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 518–533. Springer-Verlag, 1992.

[118] Joachim Parrow and Peter Sjödin. The complete axiomatization of cs-congruence. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *STACS '94*, volume 775 of *Lecture Notes in Computer Science*, pages 557–568. Springer-Verlag, 1994.

[119] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[120] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, October 1996. A summary was presented at LICS '93, pages 187–215.

[121] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, April 1995.

[122] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Plotkin et al. [123]. To appear.

[123] Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 1998. To appear.

[124] Sanjiva Prasad, A. Giacalone, and P. Mishra. Operational and algebraic semantics of facile: A symmetric integration of concurrent and functional programming. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, July 1990.

[125] R. Raj, E. Tempero, H. Levy, Andrew Black, N. Hutchinson, and Eric Jul. EMERALD: A general-purpose programming language. *Software Practice and Experience*, 21(1), January 1991.

[126] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[127] Didier Rémy and Jerôme Vouillon. Objective ML: A simple object-oriented extension to ML. In POPL '97 [9], pages 40–53.

[128] John H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer-Verlag, 1992.

[129] James Riely and Matthew Hennessy. Distributed processes and location failures. In Degano et al. [52], pages 471–481. Also Report 2/97, University of Sussex, Brighton, April 1997.

[130] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* Ph.D. thesis, University of Edinburgh, May 1993. Available as Technical Report CST–99–93, Computer Science Department, University of Edinburgh.

[131] Davide Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract appears in the Proceedings of MFCS'95, LNCS 969, 1994.

[132] Davide Sangiorgi. Lazy functions and mobile processes. Rapport de recherche 2515, INRIA Sophia-Antipolis, 1995.

[133] Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155, 1996. Also Report ECS–LFCS–94–282, University of Edinburgh, 1994. An extended abstract appeared in *Proceedings of TACS'94*, Lecture Notes in Computer Science 789.

[134] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in [30].

[135] Davide Sangiorgi. The name discipline of uniform receptiveness. In Degano et al. [52], pages 303–313. Also INRIA Rapport de recherche, December 1996.

[136] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998. Also INRIA Rapport de recherche 3000, 1996.

[137] Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In W. R. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[138] Peter Selinger. A compiler for the join-calculus. Available electronically at `http://www.math.lsa.umich.edu/~selinger/join.html`, 1996.

[139] Peter Selinger. First-order axioms for asynchrony. In Mazurkiewicz and Winkowski [93], pages 376–390.

[140] Peter Sewell. On implementations and semantics of a concurrent programming language. In Mazurkiewicz and Winkowski [93], pages 391–405.

[141] Peter Sewell. From rewrite rules to bisimulation congruences. In Robert De Simone and Davide Sangiorgi, editors, *Proceedings of the 9th International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 269–284, Nice, France, September 1998. Springer-Verlag.

[142] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In Larsen et al. [84], pages 695–706. Full version as Technical Report 435, Computer Laboratory, University of Cambridge.

[143] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72, Munchen, Germany, September 1994. Springer-Verlag.

[144] Bent Thomsen. Polymorphic sorts and types for concurrent functional programs. Technical Report ECRC-93-10, European Computer-Industry Research Center, Munich, Germany, 1993.

[145] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1995.

[146] Vasco T. Vasconcelos. Predicative polymorphism in the π-calculus. In *Proceedings of 5th Conference on Parallel Architectures and Languages, Europe (PARLE'94)*, volume 917 of *Lecture Notes in Computer Science*, pages 425–437. Springer-Verlag, 1994.

[147] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[148] David J. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.

[149] J.E. White. Telescript technology: the foundation for the electronic marketplace. Technical report, General Magic, 1994.

[150] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[151] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

[152] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[153] Nobuko Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science (Hyderabad, India, December 18–20, 1996)*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.