Contributions à la sémantique de la concurrence

Daniele Varacca

Habilitation à diriger des recherches en Informatique

Laboratoire Preuves Programmes Systèmes
Université Paris Diderot

Soutenue le mercredi 5 décembre 2012
devant le jury composé de :

M.	Roberto Amadio	*Rapporteur*
M.	Matthew Hennessy	*Rapporteur*
M.	Davide Sangiorgi	*Rapporteur*
M.	Paul Gastin	
M.	Daniel Hirschkoff	
Mme	Catuscia Palamidessi	

Contributions à la sémantique de la concurrence

Une dissertation
de
Daniele Varacca
14 décembre 2012

Remerciements

Je vais tout d'abord remercier les rapporteurs, Roberto Amadio, Matthew Hennessy et Davide Sangiorgi, qui ont accepté la tache d'évaluer ma maturité scientifique. Je remercie également les autres membres du jury, Paul Gastin, Daniel Hirschkoff et Catuscia Palamidessi, pour leur disponibilité.

Je dois remercier chaleureusement Séverine Maingaud, Valia Michou, Mihaela Sighireanu et Cécile Bordez, qui ont accepté de relire, améliorer et corriger mon Français un peu boiteux, et qui m'ont fait des remarques précieuses.

Finalement je remercie le directeur de PPS, Thomas Ehrhard pour m'avoir encouragé à entreprendre cette démarche que je croyais au delà de mes possibilités.

Daniele Varacca
Paris, 6 Juillet 2012.

Table des matières

Chapitre 1

Introduction : La modélisation des systèmes concurrents

Cette dissertation présente les principaux travaux scientifiques que j'ai effectués pendant les neuf dernières années, après l'obtention de mon doctorat. Ces travaux portent sur différents aspects de la sémantique de la concurrence. Quatre directions sont identifiées : la sémantique causale de la mobilité, les systèmes de types pour la mobilité, la spécification et vérification des systèmes concurrents et les protocoles de communication.

Pour chacune de ces directions, plusieurs articles ont été publiés dans des conférences et journaux internationaux. Une réorganisation de ces articles est présentée en annexe. Ces annexes, en langue anglaise, contiennent tous les détails techniques, mais également des explications sur le contexte et les motivations des recherches.

Ce texte en langue française loin d'être un simple résumé des articles, a comme but la présentation de mes travaux de façon synthétique, avec le moins de détails techniques possibles, permettant à un chercheur débutant ou non-spécialiste d'apprécier la portée de mes contributions. Il devrait également permettre à un lecteur plus expert de s'orienter, d'avoir une vision globale qui puisse l'aider à se plonger plus aisément dans la lecture des articles.

Avant de décrire les travaux qui constituent l'objet de cette dissertation, je voudrais décrire brièvement le contexte dans lequel ces travaux se situent.

1.1 Le calcul concurrent

On parle de calcul concurrent quand plusieurs unités de calcul exécutent certaines opérations de façon indépendante, tout en partageant des informations. Ces unités de calcul peuvent être physiquement séparées (quand il s'agit de

différents processeurs dans une machine, ou même de différentes machines) ou peuvent être juste virtuellement séparées, comme des threads ou des processus étant exécutés sur un seul processeur. Beaucoup de calculs sont aujourd'hui exécutés de façon concurrente et il est très important de bien comprendre leur fonctionnement.

Par rapport à d'autres sciences, comme la physique et la chimie, on pourrait penser l'étude de l'informatique est plus aisée, car elle est une création originale de l'être humain. Pourtant ce n'est pas plus simple de comprendre un gros programme, avec des millions de lignes de code et des centaines d'auteurs différents, que de comprendre l'écoulement d'un fluide, ou la cristallisation d'un hydrocarbure. Pour cette raison, l'utilisation des mêmes méthodes est nécessaire : on crée des modèles mathématiques de la réalité, en simplifiant les détails qui semblent être d'importance secondaire.

On distingue souvent les entités mathématiques utilisées dans l'étude du calcul (concurrent ou pas) en *syntaxiques* et *sémantiques*. Il s'agit d'une distinction où la ligne de démarcation n'est pas clairement dessinée, mais intuitivement les éléments syntaxiques sont ceux qui nous permettent de *décrire* le calcul : langages de programmation, langages de processus, types, formules logiques. Les objets syntaxiques sont généralement finis et ils peuvent être visualisés sur papier ou sur écran. Les éléments sémantiques sont les objets mathématiques qui nous permettent de *représenter le comportement* d'un calcul. Ils peuvent être infinis et sont des structures mathématiques plus abstraites, comme des graphes, des espaces topologiques, des jeux.

Dans leur travail quotidien, les programmeurs manipulent des objets syntaxiques. Ces éléments syntaxiques sont utilisés pour générer des instructions qui permettent aux processeurs d'exécuter les calculs : par exemple, un programme C est compilé vers des instructions binaires. Les objets sémantiques sont une façon de représenter le calcul lui même, en faisant abstraction de plusieurs petits détails : par exemple, dans un graphe qui représente une machine, les nœuds correspondent aux états de la machine et les arcs correspondent aux possibles changements entre états.

Les modèles mathématiques permettent de prouver formellement des propriétés d'un système qui exécute un calcul. Ils permettent également de vérifier de façon automatique si un système satisfait certaines propriétés. Par exemple, on voudrait pouvoir prouver ou vérifier qu'un programme produit les résultats désirés, ou que la communication entre un client et un serveur se déroule toujours correctement, ou encore que des informations secrètes ne sont pas révélées pendant l'exécution d'un calcul.

Mais la compréhension obtenue à l'aide de modèles mathématiques permet aussi d'avoir des intuitions, de concevoir de nouvelles façons de réaliser des calculs concurrents. Il arrive parfois qu'un nouveau langage de programmation issu des intuitions théoriques soit ensuite utilisé par un grand nombre de programmeurs. On veut citer ici, en tant qu'exemple notable, le langage ESTEREL [BG92], qui a été conçu dans le domaine académique par un chercheur expert en sémantique formelle (Gérard Berry de l'INRIA) et qui est devenu un outils important dans la création de systèmes critiques, dans des domaine

comme l'aérospatiale ou le nucléaire.

1.2 Le non-déterminisme et la probabilité

Plusieurs notions mathématiques sont utilisées dans les modèles de la concurrence. Le premier concept qu'on veut mentionner ici est celui de *non-déterminisme*. Un modèle mathématique possède du non-déterminisme s'il représente la possibilité de faire un choix pendant un calcul, sans donner d'indications sur la manière dont ce choix est résolu. On utilise ce concept pour plusieurs raisons.

D'abord, dans les modèles de concurrence, on modélise la planification (*scheduling* de différentes unités de calcul à l'aide du non-déterminisme : à chaque moment du calcul, il y a un choix parmi toutes les unités qui peuvent exécuter une étape de calcul. Ce choix peut représenter une vraie alternative, quand les unités sont des threads exécutés par un seul processeur, ou bien juste une façon de modéliser une séparation spatio-temporelle entre unités. Dans le deuxième cas, on dit qu'on utilise le non-déterminisme pour représenter l'*entrelacement* des exécutions des unités concurrentes.

Le non-déterminisme est également utilisé pour obtenir des modèles *compositionnels*. On dit qu'un modèle est compositionnel s'il permet de décrire le comportement d'un système à partir des comportements des ses parties. Un modèle compositionnel permet de raisonner de façon *modulaire* : les propriétés satisfaites par les parties d'un système nous permettent de dériver les propriétés du système entier. Pour pouvoir être compositionnel, le modèle d'un système S doit pouvoir représenter toutes les interactions possibles entre S, considéré comme partie d'un système plus grand T, et les autres parties de T. En particulier, on doit pouvoir représenter les choix qui seront résolus par ces autres parties et pour représenter ces choix, le système doit être non-déterministe. Dans ce cas le non-déterminisme représente une partie encore inconnue du système.

Finalement, on veut pouvoir utiliser le même formalisme pour représenter à la fois un système et sa *spécification*. La spécification doit représenter le comportement désiré d'un système. Elle a donc le droit d'être moins détaillée que le système, car elle peut s'abstraire de certains détails qui sont pourtant nécessaires pour une implémentation complète. En particulier, dans une spécification, la façon de résoudre certains choix n'a aucune importance et pour cette raison ces choix peuvent rester non-déterministes.

Un modèle de concurrence qui utilise le non-déterminisme est celui des *systèmes de transitions étiquetées*. Ils l'utilisent à la fois pour représenter l'entrelacement, pour être compositionnel et pour pouvoir décrire des spécifications, Un système de transitions étiquetées est essentiellement un graphe dont les noeuds représentent les *états* du système, tandis que les arcs représentent les *transitions* et sont dotés d'*étiquettes* qui dénotent la nature de la transition. À partir de chaque état, plusieurs transitions sont possibles, mais le modèle ne spécifie pas quel choix sera effectivement pris pendant une exécution. Dans un système de transitions étiquetées, les étiquettes représentent aussi l'interaction possible avec les composantes inconnues et elles sont souvent utilisées pour in-

diquer la manière dont les différentes parties d'un système se composent entre elles.

Si le non-déterminisme est utilisé pour des raisons différentes, il peut aussi avoir plusieurs formes. On parle parfois de non-déterminisme *angélique* si c'est le système qui contrôle la façon dont les choix sont résolus. On parle de non-déterminisme *démoniaque* si les choix ne sont pas sous le contrôle du système, et s'il sont en plus contrôlés par une entité ennemie cherchant à empêcher le système de satisfaire sa spécification. Entre ces deux extrêmes il y plusieurs nuances. En particulier, on décrira (dans le Chapitre 4) une notion de non-déterminisme *équitable*.

Une façon de résoudre les choix qui se présentent pendant une exécution est de choisir "au hasard" c'est-à-dire selon une distribution de probabilité. Il y a plusieurs motivations pour représenter des choix probabilistes dans un modèle de calcul. Une motivation est de pouvoir représenter l'interaction du système avec un *environnement* qui a un comportement aléatoire. On pourrait, par exemple, vouloir représenter dans le modèle le fait que, avec une certaine probabilité, la communication entre serveur et client échoue, ou bien que, avec une certaine probabilité, elle contient des erreurs.

Mais on veut également pouvoir activement faire des choix de façon probabiliste : plusieurs algorithmes utilisent le choix probabiliste pour gagner en efficacité. On utilise également le choix probabiliste dans les protocoles de sécurité : très souvent, un protocole de sécurité complètement déterministe se prête plus facilement à des attaques.

Une partie de mon travail de thèse de doctorat consistait en l'étude d'une façon de combiner dans un même modèle le choix non-déterministe et le choix probabiliste. Le choix probabiliste est encore sujet d'étude dans cette dissertation, dans les Chapitres 2 et 4.

1.3 Modéliser la causalité

La modélisation de la concurrence à l'aide de l'entrelacement non-déterministe a d'un coté l'avantage d'être conceptuellement simple et de permettre de prouver plus facilement certaines propriétés de comportement. D'un autre coté, l'utilisation de l'entrelacement engendre un grand nombre d'exécutions possibles et ne peut pas représenter directement des notions comme la *causalité* entre les événements qui constituent l'exécution. Dans plusieurs applications, il est utile de savoir tracer la cause d'un événement intéressant. Par exemple, dans un protocole de sécurité, on voudrait savoir quelle action a révélé une information secrète.

Une question strictement liée à celle de la causalité est celle de l'*indépendance*. Si deux étapes d'un calcul sont indépendantes, il est en principe possible de les exécuter en parallèle, sur deux processeurs différents, en gagnant en efficacité.

Les modèles mathématiques qui représentent directement les liens de causalité et l'indépendance entre les événements sont appelés *modèles causaux*. (Le terme de *concurrence vraie* est également utilisé. Nous n'aimons pas ce terme qui

suggère que les autres sont des modèles de concurrence fausse.) Un des modèles causaux les plus simples est le modèle des *structures d'événements*, en particulier les structures d'événements *premières* [NPW81], où la causalité n'est qu'une relation d'ordre partiel entre événements.

Les structures d'événements premières ont été à l'origine pensées comme un maillon entre les réseaux de Petri, un modèle des systèmes concurrents des années 60, et la théorie des domaines, développée dans les années 70 pour modéliser les langages de programmation. Comme les réseaux de Petri, elles sont un modèle causal, mais elle proposent aussi la simplicité de définition et le haut niveau d'abstraction de la théorie des domaines. Une structure d'événements première est constituée d'un ensemble d'événements, une relation de causalité entre eux, une relation de conflit et une fonction d'étiquetage qui associe à chaque événement une étiquette qui révèle la nature de l'événement. Les structures d'événements forment également une catégorie (dans le sens de la théorie des catégories [Mac71]).

Un grand nombre de variantes du modèle présenté dans [NPW81] ont été étudiées et nous ne pouvons pas toutes les mentionner. En général, ces variantes permettent d'exprimer plus facilement certains comportements, mais au prix de définitions plus complexes. Nous voulons tout de même évoquer les structures d'événement *stables* [Win87] qui sont très proches des structures premières.

Le travail présenté dans le Chapitre 2 et dans les Annexes A and B porte sur un modèle de structures d'événements premières pour un calcul de processus concurrents.

1.4 Le rôle des types

Il n'existe aucune méthode pour décider si un programme donné a une certaine propriété de comportement. Le problème dans sa généralité est *indécidable*. Pour contourner cet obstacle, une possibilité est de restreindre la classe des programmes, de façon à ce que certaines propriétés de comportement soient garanties "par construction". Une des techniques qui permettent cela est l'utilisation des *types*. Les types "imposent des contraintes qui aident à garantir la correction" [CW85].

Dans les langages de programmation, on construit des termes complexes en composant des termes plus simples. Pour restreindre la classe des programmes possibles, on peut restreindre la composition des termes. Pour cela, on associe à chaque terme un type. Si un terme est associé à un certain type, il ne pourra pas être composé avec de termes incompatibles avec ce type. Cette limitation peut donner des garanties en ce qui concerne le comportement.

Différents systèmes de types sont étudiés pour garantir des propriétés comportementales différentes. Le rôle premier du typage est d'éviter certaines erreurs pendant l'exécution. Dans le cas des langages de processus concurrents par exemple, on peut garantir qu'un processus qui veut communiquer une paire de nombres ne puisse pas se synchroniser avec un processus qui s'attend à n'en recevoir qu'un seul [Mil99]. Mais les types sont aussi utilisés pour garantir des pro-

priétés de sécurité [HY02], ou de séquentialité [BHY01], ou encore pour garantir qu'un calcul se termine. Une utilisation plus technique des types concerne les *encodages* : la traduction d'un langage vers un autre langage. Les types peuvent garantir la correction d'un encodage, par rapport à différents critères de correction [SW02].

Le travail présenté dans le Chapitre 3 et dans l'Annexe C porte sur un système de types pour un calcul de processus concurrents, qui garantit l'absence d'erreurs de communication et qui permet un encodage correct d'un langage fonctionnel. Les types jouent aussi un rôle dans les travaux des Chapitres 2 et 5.

1.5 Spécification et vérification

Lorsqu'on a un modèle mathématique d'un système de calcul et une spécification du comportement désiré, on souhaite savoir si le système en question *satisfait* la spécification. Si le système et la spécification sont simples, il est possible de trouver une preuve mathématique qui peut être exécutée manuellement. Mais pour des systèmes et des spécifications plus complexes, le recours à des outils automatiques devient indispensable.

On peut utiliser des programmes qui sont capable soit de nous guider dans la recherche de la preuve, soit de rechercher la preuve à notre place. On a donc deux types d'outils : les *assistants de preuve*, comme Coq [HKP04], et les outils de *vérification automatique* [CES09] qui vérifient l'ensemble des exécutions possibles afin de déceler celles qui ne satisfont pas la spécification. Parfois, système et spécification peuvent être représentés dans le même formalisme : dans ce cas, il faut vérifier que le système *raffine* la spécification, c'est-à-dire qu'il en est une implémentation. Parfois, la spécification est exprimée dans une logique, par exemple une *logique temporelle* [MP92].

Une des logiques temporelles utilisées est la logique temporelle *linéaire* (LTL). Une formule de cette logique exprime une propriété d'une seule exécution du système. On dit que le système satisfait la formule si *toutes* les exécutions possibles du système satisfont la formule. Les études sur la notion de non-déterminisme équitable du Chapitre 4 et dans l'Annexe D sont basées sur une notion de spécification linéaire et sur différentes techniques de vérification automatique.

1.6 Aperçu de mes contributions

Ce document présente quatre contributions que j'ai apporté à la sémantique de la concurrence. Dans le Chapitre 2, je présente mes recherches sur un modèle causal pour un langage de processus concurrents. Les publications concernées sont [VY06, VY07, VY10, CVY07, CVY12]. Dans le Chapitre 3, je présente mes recherches sur un système de types pour un langage de processus concurrents [CDV05, CDV08, CDV06]. Dans le Chapitre 4, je présente mes recherches

sur la formalisation de la notion d'équité pour la vérification [VVK05, VV06, VV12]. Finalement, dans le Chapitre 5, je présente mes recherches sur la modélisation d'un protocole de communication [FV08, FV09]. Pour chaque chapitre, les annexes contiennent une version révisée et réorganisée des articles concernés.

1.7 Reconnaissance de collaboration

Les collaborations scientifiques que j'ai eues pendant mes recherches sont facilement identifiables en lisant les noms des coauteurs des mes articles. Je tiens tout de même à les mentionner explicitement.

J'ai connu Hagen Völzer pendant mes travaux de thèse car nous avions travaillé indépendamment sur un sujet similaire. Notre première collaboration a été un article où l'on joignait nos efforts. Par la suite Hagen a voulu partager avec moi ses idées sur l'équité — idées dont il avait en partie discuté avec Ekkart Kindler. Bien que nos discussions aient été continues, je considère que l'impulsion première de nos articles sur l'équité vient de lui. C'est lui qui avait la vision initiale que nous avons par la suite formalisée et précisée ensemble. Hagen a eu la chance de trouver en Matthias Schmalz un étudiant très fort et très motivé, qui a travaillé sur le coté algorithmique de l'équité. De mon coté, j'ai co-encadré avec Eugène Asarin le Master de Rapahel Chane-Yack-Fa en étendant la définition d'équité à un contexte plus général.

Beppe Castagna m'a offert un poste de post-doc à l'ENS de Paris pour travailler sur le projet européen "MyThs", sur le typage pour la sécurité. En même temps, Beppe dirigeait la thèse de Alain Frisch qui a donné lieu à la notion de sous-typage sémantique. Une visite de Rocco de Nicola à Paris a été la source des discussions qui nous ont menés à l'application du sous-typage sémantique au π-calcul. Une visite de Mariangiola Dezani est également à l'origine de l'article sur le codage de CDuce.

Deux ans plus tard, Nobuko Yoshida a bien voulu m'embaucher comme post-doc à l'Imperial College de Londres. Le sujet sur lequel nous avons collaboré était la sémantique à structure d'événements du π-calcul. Suite à notre travail sur le cas typé, nous avons décidé de discuter aussi avec Silvia Crafa : une collaboration qui a continué pendant des années.

Finalement, sur demande de Pierre-Louis Curien, j'ai co-dirigé les travaux de thèse de Luca Fossati, en me focalisant sur un sujet sur lequel il avait déjà commencé à travailler depuis quelques années. Cela a donné lieu à nos travaux sur les protocoles Handshake.

Chapitre 2

La sémantique à structures d'événements pour le π-calcul

2.1 La sémantique à structures d'événements de CCS

Les structures d'événements premières ont été utilisées par Winskel [Win82] pour donner une sémantique à CCS, un langage de processus concurrents qui se synchronisent à l'aide de canaux de communication. Syntaxiquement, un processus CCS de la forme $a.P$ est prêt à se synchroniser sur le lien (ou canal) a et ensuite continuer comme P. Les synchronisations dans ce langage sont binaires, c'est-à-dire que chaque synchronisation a lieu entre deux processus. (Dans d'autres langages il est possible une synchronisation entre plus que deux processus.) Chaque canal a a donc un seul canal "dual" $\bar{a}$ et le processus $a.P$ se synchronisera avec un processus de la forme $\bar{a}.Q$. Pour que cela soit possible, les deux processus doivent être mis *en parallèle*. Tout cela est représenté par la règle de *réduction* :

$$a.P \mid \bar{a}.Q \rightarrow P \mid Q$$

Cette réduction représente le fait que la synchronisation entre les processus entraîne une modification de l'état du système. La syntaxe des processus est certes plus complexe que cela, mais la composition parallèle et la synchronisation sont le coeur de la sémantique de CCS.

La sémantique de réduction est suffisante pour décrire l'évolution d'un système de processus de CCS, mais elle n'est pas compositionnelle : on ne réussit pas à décrire le comportement d'un processus en fonction du comportement des sous-processus. En particulier, pris individuellement les processus $a.P$ et $\bar{a}.Q$ n'ont

aucun comportement. Dans ce cas, la réduction est engendrée "magiquement" par la rencontre des deux.

Pour pouvoir obtenir une sémantique compositionnelle, il faut pouvoir décrire le comportement "incomplet" de chaque processus. Il faut une notation qui puisse permettre de dire qu'un processus est prêt à se synchroniser. C'est ce que permettent les *systèmes de transitions étiquetées*. Les étiquettes indiquent quelles sont les synchronisations auxquelles un processus est prêt à participer. Dans CCS, ces étiquettes sont précisément les canaux. On a donc des transitions étiquetées de la forme

$$a.P \xrightarrow{a} P$$

et pour la synchronisation on a la règle

$$\text{(Comm)} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

qui définit la transition étiquetée d'une composition parallèle en fonction des transitions des deux sous-processus. Pour que toute transition soit étiquetée, on utilise l'étiquette $\boldsymbol{\tau}$ pour représenter les réductions. On peut aussi voir la synchronisation comme une opération algébrique partielle entre étiquettes. La "multiplication" de a et de $\bar{a}$ donne $\boldsymbol{\tau}$, tandis que la "multiplication" de a et de b est indéfinie.

La sémantique à transitions étiquetées est aussi à la base de la théorie de la *bisimulation* [Mil89] et elle permet d'étudier les propriétés des processus concurrents, notamment à l'aide d'outils automatisés.

Comme indiqué dans l'introduction, la sémantique à systèmes de transitions étiquetées ne peut pas représenter de façon explicite les relations de concurrence et causalité entre transitions. En particulier, la concurrence est représentée par le choix non-déterministe entre les différentes actions concurrentes, ce qui est connu sous le nom d'*entrelacement*. La représentation explicite de la concurrence (ainsi que du conflit et de la causalité) devient possible dans les modèles causaux, comme les structures d'événements.

Dans la sémantique proposée par Winskel, à chaque terme du langage correspond une structure d'événements et cette correspondance est obtenue de façon *compositionnelle* : la structure d'événements d'un terme est construite à partir des structures d'événements de sous-termes. Par exemple, la sémantique d'un processus de la forme $a.P$ a un événement étiqueté par a comme minimum de l'ordre partiel, suivi par la structure d'événements du processus P. Intuitivement, l'événement étiqueté par a est la *cause* des tous les événements de P.

Les deux processus

$$a.a.\mathbf{stop} \qquad\qquad a.\mathbf{stop} \mid a.\mathbf{stop}$$

ont le même système de transitions étiquetées, mais deux structures d'événements différentes. La structure d'événements du premier processus a un événement

minimal étiqueté par a suivi dans l'ordre partiel par un autre événement étiqueté par a. La structure d'événements du deuxième processus a deux événements étiquetés par a qui ne sont pas dans la relation d'ordre : ils sont *concurrents*.

Les constructions nécessaires pour obtenir une sémantique compositionnelle ont toutes une explication canonique et élégante dans la *théorie des catégories*,. En particulier, la sémantique de la composition parallèle est obtenue à partir du *produit cartésien* dans la catégorie des structures d'événements. Les événements du produit cartésien des deux structures d'événements représentent toutes les synchronisations imaginables entre paires d'événements, y compris la possibilité de ne pas se synchroniser, à condition de respecter causalité et conflit. Ce produit contient beaucoup trop de synchronisations. Encore faut-il enlever les synchronisations qui ne respectent pas la discipline du calcul de processus. Par exemple, un événement étiqueté a ne pourra pas se synchroniser avec un événement étiqueté b, mais il pourra se synchroniser avec tout événement étiqueté $\bar{a}$. Toute synchronisation interdite est donc effacée de la structure, tandis que les synchronisations restantes sont réétiquetées en $\boldsymbol{\tau}$. Pour résumer, les trois ingrédients utilisés pour la sémantique de la composition parallèle sont le produit cartésien, l'effacement (appelé aussi *restriction*) et le réétiquetage.

Cette description informelle montre l'importance des étiquettes dans la définition d'une sémantique compositionnelle, dans les systèmes de transitions tout comme dans les structures d'événements. Les étiquettes sont également nécessaires pour établir une correspondance entre les deux sémantiques : on peut facilement définir un système de transitions étiquetées à partir d'une structure d'événements étiquetés et montrer que celui-ci est équivalent (le terme exacte est *bisimilaire*) au système de transition "standard".

2.2 Le passage de CCS au π-calcul

L'élégance de la sémantique de Winskel est en partie due à la simplicité du langage considéré. En particulier, les étiquettes utilisées en CCS sont relativement simples. Cette simplicité est possible au prix d'une expressivité réduite. Comme déjà observé par Milner [Mil99], la topologie des liens de communication en CCS est fixée au départ. Il est impossible pour un processus de créer des liens de façon dynamique.

Pour dépasser ces limites, Engberg et Nielsen [EN00] ont proposé une extension de CCS où les processus peuvent s'échanger les liens de communication, en modifiant dynamiquement la topologie. Ce calcul a évolué jusqu'à devenir ce qu'on appelle actuellement le π-calcul [MPW92]. Dans le π-calcul, comme dans CCS, les processus se synchronisent à l'aide de canaux, mais en plus, pendant la synchronisation, les processus peuvent envoyer ou recevoir des noms de canaux. La sémantique par réduction du π calcul est en conséquence une extension simple de la sémantique par réduction de CCS. Mais comme dans le cas de CCS, le point faible de cette sémantique est qu'elle n'est pas compositionnelle.

La vraie difficulté dans la conception du π calcul a en effet été de donner une sémantique *compositionnelle*, et donc étiquetée. La solution qui fût trouvé

finalement par Nielsen and Engberg, bien que parfaitement adaptée, n'est pas simple à comprendre à un premier regard. D'abord, il y a une distinction entre deux types de sémantique : précoce (*early*) et tardive (*late*), qui génèrent deux notions d'équivalence différentes. Ensuite, on a une distinction entre envoi *ouvert* et envoi *lié* d'un nom de canal. Informellement, un envoi est ouvert si le nom envoyé est déjà connu par le reste du système, tandis que il est lié s'il s'agit d'un nouveau nom, fraîchement créé par le processus. La création d'un nouveau nom est syntaxiquement représentée par l'opérateur de *restriction*. L'envoi lié d'un nouveau nom produit le phénomène appelé *extrusion de portée* : la portée de l'opérateur de restriction est étendue après une communication du nouveau nom. Les étiquettes sont aussi sujettes à l'*α-renommage*, pour garantir qu'un canal frais ne puisse pas être identifié à aucun autre canal. Finalement, la sémantique fait intervenir une notion de *substitution*, qui est la façon de représenter syntaxiquement la possibilité, pour un processus, d'utiliser un nom qu'il reçoit.

La sémantique étiquetée du π-calcul est satisfaisante : elle est compositionnelle et elle induit une bonne notion d'équivalence. Pourtant, la signification des étiquettes n'est plus très intuitive et simple. Les complications syntaxiques produites par l'extrusion de portée, l'α-renommage et la substitution ont toujours rendu difficile l'extension de la sémantique de Winskel au π-calcul. Notre travail est allé dans ce sens et nous allons le présenter par étapes, dans l'ordre dans lequel il a été développé.

2.3 Le π-calcul typé

Le premier pas de notre travail a été l'étude d'un modèle pour le π-calcul *linéairement typé* [VY06, VY10].

Plusieurs systèmes de types ont été proposé pour restreindre et contrôler le comportement des processus du π-calcul. Le typage linéaire ou affine [YHB02], qui contraint l'utilisation des canaux de communication, garantit que les processus sont *déterministes*, c'est-à-dire qu'aucun conflit n'est crée par les synchronisations. Le fragment du π-calcul ainsi typé reste assez expressif pour pouvoir simuler le comportement des langages de programmation fonctionnels (comme PCF [Plo77]).

L'idée du système de types est que chaque canal peut être utilisé soit de façon *linéaire*, soit de façon *répliquée*. Un canal linéaire pourra être utilisé exactement une fois pour l'envoi et une fois pour la réception. Les règles de typage garantissent qu'il ne puisse pas y avoir de conflit entre deux différents envois ou deux différentes réceptions sur un même canal. Un canal répliqué peut être utilisé zéro, une ou plusieurs fois en envoi, tandis qu'en réception il doit apparaitre une seule fois en dessous de l'*opérateur de réplication*. Cet opérateur modélise un serveur qui fournit toujours le même service à un nombre quelconque de requêtes.

De plus, dans le cas typé on peut considérer juste un fragment du π-calcul, sans qu'il y ait perte d'expressivité. Le fragment considéré s'appelle le π-calcul

interne. Dans le π-calcul interne, les processus peuvent seulement envoyer comme messages des canaux qu'ils ont crée et cela une seule fois. Il n'y a donc pas d'envoi ouvert.

Cette restriction simplifie largement la sémantique étiquetée : il s'agit d'une sémantique beaucoup plus proche de celle de CCS. De plus, il n'est pas nécessaire de créer dynamiquement des liens à l'exécution : le système de types nous dit, d'une certaine façon, avec qui on communiquera et tous les canaux qui seront utilisés (peut-être un nombre infini) peuvent être créés au début de l'exécution, *au moment de la compilation* pour ainsi dire.

Le calcul linéairement typé est déterministe dans le sens où aucun conflit n'est engendré pendant l'exécution : d'un côté sur les canaux linéaires il n'y a pas de choix sur la synchronisation à effectuer, de l'autre côté, pour les canaux répliqués, toute requête est satisfaite de la même façon par le serveur.

Pour formaliser cette intuition, on voulait que dans la structure d'événements correspondante à un processus typé, la relation de conflit soit vide. Cela n'est pas automatique, car il faut faire attention à la modélisation des serveurs. Dans la sémantique entrelaçante, le fait que les serveurs actifs sur un canal répondent à toutes les requêtes de la même façon amène facilement à un résultat de déterminisme. Moralement, toutes les requêtes seront finalement satisfaites, peu importe l'ordre. Dans la sémantique à structures d'événements, des conflits peuvent quand même apparaitre.

Pour comprendre la nature de ces conflits, imaginons que le serveur soit le bureau de poste et que les clients soient les personnes qui veulent envoyer des lettres. [1] Le bureau de poste accepte la lettre de la part du client et il procède aux opérations nécessaires pour l'envoyer. La spécification est qu'à tout moment le bureau de poste est prêt à recevoir une lettre et que toutes les lettres auront le même traitement.

Comment cette spécification est-elle implémentée ? On peut imaginer qu'au bureau il n'y ait qu'un employé qui reçoit les lettres. Il va traiter toutes les lettres bien sûr, mais si deux clients veulent envoyer une lettre en même temps, un conflit aura lieu pour décider qui des deux clients sera servi en premier. On peut donc imaginer que le bureau soit une série infinie d'employés, chacun capable d'accepter et envoyer une seule lettre. Cela ne garantit pas l'absence de conflit, car rien ne garantit que deux employés cherchent à servir le même client [2] ou que deux clients cherchent à déposer leur lettre chez le même employé [3].

Tous ces conflits apparaissent effectivement si on utilise naïvement la sémantique de Winskel. Pour résoudre ce problème on utilise à nouveau l'intuition que toute synchronisation est décidée au moment de la compilation.

L'idée est toujours que le bureau de poste est constitué d'un nombre infini d'employés, mais cette fois à chaque client est donné un ticket, un numéro, qui lui signale, avant même qu'il ait envie d'envoyer sa lettre, quel sera l'employé qui va la recevoir. Au moment de l'envoi, il n'y aura donc aucun conflit.

1. Même si au jour d'aujourd'hui en France les bureaux de poste font tout sauf accepter des lettres – pour cela il y a des machines.
2. car ils savent qu'il va laisser un bon pourboire.
3. car ils savent qu'il est très efficace.

Cette intuition est réalisée en pratique avec une extension du système de types linéaires, qui garantit l'unicité de ces "tickets". Le calcul ainsi obtenu est réellement très proche de CCS et cela rend plus facile la définition d'une sémantique en termes de structures d'événements. L'unicité des "tickets" permet aussi de décider avant l'exécution quels processus vont se synchroniser. Cela permet de créer tous les noms de canaux pendant l'exécution (techniquement cela est réalisé à l'aide d'un langage de processus intermédiaire).

Pour garantir que toute structure d'événements d'un processus typé ait une relation de conflit vide, on a créé un système de types pour structures d'événements. La propriété d'avoir une relation de conflit vide n'est pas banale car en général elle n'est pas préservée par composition parallèle : la composition de structures sans conflit n'est pas sans conflit. Le système de types contraint la composition de structures, de façon à préserver cette propriété.

Après avoir préparé le terrain de cette façon, la sémantique est définie essentiellement comme dans le cas de Winskel, en prenant en compte les "tickets" dans la définition de l'algèbre de synchronisation.

La correction de cette sémantique par rapport à la sémantique à transitions est prouvée en utilisant les propriétés des structures typées. Pour pouvoir réaliser cette preuve, on a dû donner une nouvelle caractérisation du produit cartésien dans la catégorie des structures d'événement. En particulier, cette nouvelle définition nous permet d'effectuer des preuves par induction sur la hauteur (dans le sens de l'ordre causal) des événements.

2.4 Le π-calcul probabiliste

Une application assez simple de l'étude décrite ci-dessus à été l'étude d'une sémantique pour une variante *probabiliste* du π-calcul [VY07].

Un petite modification du langage et du système de types permet de garantir une propriété de comportement plus faible que l'absence de conflit. Cette propriété est connue sous le nom d'*absence de confusion*. Intuitivement, dans une structure d'événements sans confusion, tous les choix non-déterministe sont "localisés". Les lieux de ces choix sont appelés *cellules*. Pour résoudre ces choix localisés de façon probabiliste, on associe à chaque cellule une distribution de probabilité. Cela nous donne un cas particulier de *structure d'événement probabiliste*, un modèle que j'ai proposé est étudié dans ma thèse [Var03, VVW06]. Ce que j'avais montré, c'est qu'en associant à chaque cellule une distribution de probabilité, on obtient une mesure de probabilité sur l'ensemble des exécutions. Une mesure de probabilité sur les exécution peut être vue comme une seule exécution probabiliste.

Le système de types qui garantit que la sémantique d'un processus est une structure sans confusion est utilisé pour définir un calcul où tous les choix probabilistes sont localisés dans les cellules. Chaque processus typé a donc une seule exécution causale probabiliste. Cela peut être vue comme une extension du résultat de déterminisme au calcul probabiliste.

2.5 Le π-calcul interne

Dans le suite de notre travail, on s'est penchés sur l'étude d'une variante non typé du π-calcul. La variante choisie est celle qu'on appelle le π-calcul *interne*, ou πI-calcul. Comme on a précédemment évoqué, la sémantique du π-calcul interne est simplifiée par rapport au π-calcul non restreint. Envoi et réception sont complètement duales et symétriques, au point où il n'y a plus beaucoup de sens à parler d'envoi ou de réception : on dit que pendant une synchronisation, les deux processus *partagent* un nom connu seulement par eux. Cette symétrie entraine une simplification du point de vue de la sémantique. Il existe une seule étiquette pour l'envoi et aucune distinction n'existe entre sémantique précoce ou tardive. Finalement, ce n'est pas nécessaire de définir la substitution car tout le travail de communication est fait par l'α-renommage.

Toutes ces caractéristiques rendent le π-calcul interne assez proche de CCS. Cependant, ce calcul reste très expressif et, sous certaines conditions, on peut définir un codage du π-calcul complet dans le π-calcul interne [Bor98], ce qui n'est définitivement pas le cas pour la variante typée linéairement.

La sémantique à structures d'événements du π-calcul interne est également très proche de celle de CCS. Tous les opérateurs ont essentiellement la même sémantique, sauf pour la composition parallèle. On rappelle qu'en CCS, la sémantique de la composition parallèle est obtenue à partir du produit cartésien, suivi par un renommage et une restriction. En CCS, le renommage dépend exclusivement des étiquettes, grâce à la notion d'algèbre de synchronisation. Intuitivement, pour savoir si deux événements peuvent se synchroniser, il suffit de regarder leurs étiquettes. Dans le π-calcul interne, deux événements peuvent se synchroniser aussi parce que leurs étiquettes ont été identifiées par une synchronisation précédente. Par exemple, si un processus P s'est synchronisé avec Q_1 en partageant un nom, il pourra ensuite utiliser ce nom pour continuer à se synchroniser avec Q_1, mais si P avait choisi de se synchroniser plutôt avec Q_2, alors les synchronisations ultérieures avec Q_1 seraient devenues impossibles.

Pour représenter ce phénomène dans les structure d'événements, il a fallu changer la façon de renommer les événements. Pour décider si, dans un produit cartésien, une synchronisation est légale, il n'est plus suffisant de regarder les étiquettes des événements et faire appel à l'algèbre de synchronisation, mais il faut regarder aussi dans l'histoire causale des événements (c'est-à-dire dans l'ensemble des événements au dessous dans l'ordre causale), pour voir si une synchronisation précédente l'a rendue possible.

Grâce toujours à notre caractérisation explicite du produit cartésien dans la catégorie des structures d'événements, de façon très similaire au cas typé, on prouve un résultat de correspondance entre la sémantique à structures d'événements et la sémantique à transitions étiquetées.

2.6 Le π-calcul complet

La dernière étape de notre étude a été de proposer une sémantique à structures d'événements du π calcul complet. Le π calcul complet diffère du π calcul interne en deux aspects :

- les processus peuvent communiquer des noms de canal libres. Cela implique qu'une vraie notion de substitution est nécessaire ;
- les processus peuvent "ouvrir" la portée de l'opérateur de restriction.

Pour gérer la substitution, il faut que, dans une composition parallèle, le réétiquetage des événements soit fait de façon incrémentale, en tenant compte de l'histoire causale des événements. Mais une précaution ultérieure est nécessaire : On ne peut pas effacer immédiatement toutes les synchronisations qui ne respectent pas l'algèbre de synchronisation. Une synchronisation qui n'est pas permise, car les noms des canaux sont différents, pourrait le devenir si une communication identifiait les deux canaux. Ce phénomène est dû à la possibilité de communiquer des noms de canaux libres et ne se présente pas dans le π-calcul interne. Ces synchronisations "inachevées", restent donc dans la structure d'événements. Elles ne sont pas simplement prises en compte dans la correspondance avec la sémantique a systèmes de transitions, car elles n'existent pas dans ce contexte.

Cette utilisation de synchronisations temporairement suspendues peut sembler un peu artificiel aux chercheurs familiers avec la théorie de la concurrence. Pour justifier leur apparition, on s'appuie sur deux arguments. D'abord, on observe que les structures d'événements sont des objets très statiques : tout événement même très éloigné dans le futur doit y être représenté au moment où la structure est créée. Si une synchronisation est rendue possible par une communication de noms, une trace de cette synchronisation doit être toujours présente si on veut obtenir une sémantique compositionnelle. Deuxièmement, notre point de vue est que tout événement étiqueté soit, dans un certain sens, un événement inachevé. Même les événements avec des étiquettes "classiques" représentent des synchronisations partielles, en attente de trouver un partenaire. Les seuls "vrais" événements sont les réductions, c'est-à-dire les transitions étiquetés avec $\boldsymbol{\tau}$. Le fait que d'autres types d'étiquettes soient nécessaires c'est juste une conséquence de la nature des structures d'événements.

Le traitement de l'ouverture de portée a été plus difficile. Plusieurs solutions ont été essayées, qui sont discutées en détails dans l'Annexe B, mais aucune ne semblait être la bonne. Le problème à résoudre était que dans la sémantique à entrelacement, l'opération d'extension de portée est toujours "unique". Un premier envoi "ouvre" cette portée et amène vers un processus où le nom envoyé est maintenant "libre" : tout envoi ultérieure est donc libre. En particulier, cela signifie que deux envois en parallèle d'un même nom frais ne pourraient pas se faire vraiment en parallèle : le premier ouvrira la portée et le deuxième sera libre. Presque toutes les sémantiques causales du π-calcul (basées sur des extensions des systèmes de transitions étiquetées) [BS98, DP99, CS00] ont opté pour garder cette unicité de l'ouverture de portée. Nous voulions qu'il y ait une parfaite symétrie et que deux envois en parallèle ne puissent pas engendrer un

conflit pour décider qui est celui qui ouvre la portée.

La seule solution qu'on a trouvé a été celle de sortir du modèle de structures d'événements premières : un débordement contrôlé, mais qu'on a jugé nécessaire. Pourquoi faut-il sortir des structures premières ? Une des caractéristiques des structures premières est d'être *stables*. La stabilité implique que si un événement e peut être causé par e' ou par e'' alors ces derniers doivent être en conflit, de façon qu'au moment où e arrive on sache toujours si c'était à cause de e' ou de e''. Dans le cas du π-calcul, un envoi ou une réception sur un canal lié y dépendent d'une ouverture de portée. Mais deux ouvertures de portée en parallèle ne devraient pas être en conflit. Quand deux envois du canal lié y sont possibles, ce ne devrait pas être important de savoir lequel était la cause de l'envoi/réception sur y. Cette raison intuitive est confirmée par des exemples plus techniques qui nous ont convaincu qu'il fallait sortir des structures stables.

Une fois cette démarche acceptée, l'idée est assez simple — on associe à une structure d'événement un ensemble de noms qui correspondent aux noms liés d'un processus. Cet ensemble n'a que le rôle d'interdire certaines exécutions de la structure. En particulier, tout événement utilisant en envoi/réception un nom lié ne pourra pas arriver si une ouverture de porté n'a pas eu lieu avant. Avec cette limitation des exécutions, on obtient une correspondance avec la sémantique étiquetée standard.

2.7 Conclusion et perspectives

Cette ligne de recherche avait plusieurs objectifs. Dans le cas du π calcul linéairement typé, on voulait montrer comment un résultat de confluence peut se traduire en absence de conflit dans un modèle causal. Cela est effectivement le cas, mais il a fallu quelques précautions pour éviter l'apparition de conflits "cachés". En généralisant le typage linéaire à un calcul probabiliste, on a montré comment ce typage assure une forme de "confluence probabiliste". Bien que ce résultat soit intuitivement vrai, il est très difficile de le formaliser dans un contexte traditionnel [DHW05]. On a également montré que le π-calcul linéairement typé est très proche de CCS. En particulier, la création des noms de canaux ne doit par forcément avoir lieu de façon dynamique, pendant l'exécution, mais peut être faite statiquement, au début de l'exécution.

Ensuite, on a voulu étudier si la sémantique de Winskel pouvait s'étendre ultérieurement. C'était un défi intellectuel intéressant, en particulier en ce qui concerne le calcul complet, mais qui pourrait aussi ouvrir de nouvelles perspectives de recherche. En particulier, j'envisage deux directions pour la poursuite de mon travail. Une première direction consiste à appliquer l'expérience acquise à la recherche d'une sémantique *réversible* pour le π calcul. Il est connu [PU07] que les modèles qui représentent des exécutions concurrentes réversibles sont liés aux modèles causaux de la concurrence. Il n'est pas forcement possible de transférer directement la sémantique causale vers une sémantique réversible, mais certaines techniques probablement seront les mêmes.

La deuxième direction porte sur une meilleure identification du type de

modèles causaux qu'on a abordé dans la sémantique du π-calcul. On a dit qu'on s'est eloigné des structures d'événement premières et même de toute structure stable. Le tout est maintenant de savoir "combien" s'en éloigne-t-on. Peut-on caractériser de façon plus abstraite les structures qu'on a utilisées dans notre sémantique ?

Chapitre 3

Le sous-typage sémantique et la concurrence

3.1 Introduction au sous-typage sémantique

Les systèmes de types contraignent les langages de programmation pour limiter les erreurs et garantir certaines propriétés de comportement. Il y a toujours une tension entre la force des contraintes imposées par un système de types et l'expressivité du langage. Plusieurs techniques ont été étudiées qui permettent la création de systèmes de types plus expressifs et flexibles. Un de ces mécanismes est la notion de *sous-typage*, une relation d'ordre partiel entre les types.

Normalement, si un terme syntaxique a un certain type, il ne pourra pas être utilisé dans un contexte qui attend un terme de type différent. En présence du sous-typage, on peut utiliser des termes du langage qui ont un certain type s dans un contexte qui attend un terme de type t, dans le cas où s est un sous-type de t. Pour paraphraser Cardelli et Wegner [CW85], on peut dire que "FIAT" est un sous-type de "voiture" et, en conséquence, on peut agir sur une FIAT avec toutes les opérations qu'on peut utiliser sur les voitures.

Les systèmes de types avec sous-typage sont particulièrement fréquents pour les langages de programmation orientés objets comme Java ou Scala, mais ils sont utilisés aussi dans des cadres plus théoriques.

L'interprétation plus intuitive du sous-typage est la suivante : si un type s est un sous-type de t, alors tout terme qui a type s a aussi type t. Cette intuition est souvent formalisé par une règle de *subsomption*

$$\frac{e \triangleright s \quad s \leq t}{e \triangleright t}$$

Une des questions fondamentales pour la conception d'un système avec sous-typage est la définition de la relation d'ordre. En général l'intuition qu'on a sur la signification des types d'un langage peut nous guider à formuler des règles (inductives ou coinductives) pour définir cette relation.

Par exemple, dans un système de types avec un opérateur d'*union* (ou disjonction) de types, on pourrait définir la règle suivante :

$$\frac{}{s \leq s \cup t}$$

Les opérateurs syntaxiques utilisés pour construire les types peuvent en général se classifier en *covariant*, *contravariant* ou *invariant*, selon leur comportement par rapport à la relation de sous-typage. Pour un opérateur contravariant la règle serait :

$$\frac{s \leq t}{op(t) \leq op(s)}$$

Dans un système de types avec plusieurs opérateurs, il est parfois délicat de trouver des règles qui définissent un sous-typage raisonnable. Il est aussi difficile de dire si certaines équations entre types sont satisfaites. Un choix différent est d'utiliser l'intuition décrite plus haut comme *définition* de sous-typage. En particulier, on se focalise sur les types des termes qu'on appelle *valeurs*. Les valeurs ou *formes normales*, sont les termes qui ne peuvent pas être évalués ultérieurement. Ce sont les points terminaux d'un calcul. Souvent, c'est ce qu'on observe d'un calcul : les résultats finaux. C'est pour cela que les valeurs jouent un rôle important.

On dira donc qu'un type s est sous-type de t si toute valeur de type s est aussi de type t. Au sens strict, cette définition crée un cercle vicieux : pour définir le sous-typage, il faut avoir donné des types aux termes. Mais pour donner des types aux termes, il faut utiliser la règle de subsomption et donc connaître la relation de sous-typage. Ce cercle peut être cassé si l'attribution des types aux valeurs est plus simple que l'attribution des types aux termes quelconques.

Dans le langage XDUCE [Hos03], les valeurs sont des documents XML et les types sont (essentiellement) des schémas XML (comme par exemple ceux de la classe DTD). Pour savoir si un document XML appartient à un schéma, il ne faut pas connaître tous les termes du langage. La relation de sous-typage est donc définie comme *inclusion* des ensembles des valeurs. Tous les autres termes du langage (des fonctions qui agissent sur les documents XML) sont typés en utilisant, entre autres, une variante de la règle de subsomption.

XDUCE est ce qu'on appelle un langage du *premier ordre*. Les fonctions agissent sur des valeurs, mais elles ne sont pas des valeurs elles mêmes. Il existe aussi des langages qu'on appelle d'*ordre supérieur*, où les fonctions peuvent agir sur d'autres fonctions et peuvent être considérées comme des valeurs. Parmi ces langages on trouve par exemple le λ-calcul, PCF, OCAML, HASKELL et beaucoup d'autres.

Pour étendre la notion de sous-typage de XDUCE à un langage d'ordre supérieur, il fallait définir la relation pour les types des valeurs fonctionnelles. Par conséquent, on aurait dû considérer les ensembles de fonctions sur documents XML, les ensembles de fonctions sur ces fonctions, et ainsi de suite. Or, l'ensemble de toutes ces fonctions n'existe pas (en conséquence du théorème

de Cantor). Une solution serait de considérer des *domaines* plutôt que des ensembles, comme a fait Scott [Sco72] pour donner un modèle du λ-calcul. Cette solution, qui fait appel à certains sous-ensembles d'un domaine (par exemple les *idéaux*) ne s'applique pas bien à tous les opérateurs (par exemple, la négation d'un idéal n'est pas un idéal).

Frisch, Castagna et Benzaken [FCB08] ont observé qu'il n'est pas important d'avoir un modèle ensembliste qui contienne toutes les fonctions mais qu'il suffit de définir un modèle ensembliste qui, du point de vue de la relation de sous-typage, se comporte *comme si* il contenait toutes les fonctions. Sans entrer dans tous les détails techniques, il suffit de mentionner ici qu'un tel modèle est construit en considérant les fonctions *à graphe fini*. On peut construire un ensemble dénombrable D qui contient (à isomorphisme près) toutes les fonctions finies de D en D. De plus si X, X', Y, Y' sont des sous-ensembles de D, l'ensemble des fonctions finies entre X et Y est contenu dans l'ensemble des fonction finies entre X' et Y' si et seulement si cela est le cas lorsqu'on considère toutes les fonctions au lieu des fonctions finies.

À partir de cette intuition, on peut construire un modèle ensembliste pour les types de CDuce et on définit le sous-typage comme inclusion d'ensembles. Ce modèle n'a rien à voir avec les termes du langage, mais ce n'est pas un problème car le but est de définir la relation de sous-typage et non de donner une sémantique aux termes du langage. On appelle ce modèle, le modèle de *démarrage* (bootstrap en anglais). Ayant défini le sous-typage, on peut maintenant donner des types aux termes en utilisant la règle de subsomption. Ensuite, on regarde les valeurs typées. Pour chaque type, on considère le modèle ensembliste des valeurs : à chaque type correspond l'ensemble des valeurs qui ont ce type. Ce modèle définit également une notion de sous-typage par inclusion d'ensembles.

A priori les deux notions de sous-typage définies par les deux modèles (de démarrage et des valeurs) sont différentes. Mais CDuce a été conçu de façon qu'on puisse prouver a posteriori que les deux notions coïncident.

Un des avantages de la définition sémantique de sous-typage est qu'elle s'étend automatiquement aux opérateurs booléens (intersection, union, complément — ou pour utiliser la terminologie logique, conjonction, disjonction, négation). Bien qu'en XDUCE le système de types ne contienne pas d'opérateurs booléens, l'approche ensembliste permet facilement de les définir. Le système de types de CDuce contient explicitement des opérateurs booléens.

Parmi les équations satisfaites par la relation de sous-typage définie ainsi, on a toutes les lois booléennes comme par exemple :

$$\neg(\tau \wedge \sigma) = \neg\tau \vee \neg\sigma$$

Mais il y a des relations moins évidentes. Par exemple, si $\tau \to \sigma$ est le type des fonctions de τ dans σ, on a que

$$\tau \to (\sigma \vee \sigma') \leq (\tau \to \sigma) \vee (\tau \to \sigma')$$

mais la relation inverse n'est pas toujours satisfaite. Pour comprendre l'intuition qui justifie cette relation, on renvoie le lecteur aux articles cités.

3.2 Le soustypage sémantique pour le π-calcul

L'utilisation du sous-typage dans le π-calcul a été introduite par Pierce et Sangiorgi [PS93]. Les types sont donnés aux canaux et aux variables. Ils indiquent quelles sont les valeurs que un canal peut communiquer. Ils précisent aussi si un processus peut utiliser un canal seulement pour envoyer des messages, ou seulement pour en recevoir, ou bien pour les deux utilisations. Le sous-typage est défini avec des règles syntaxiques qui suivent les intuitions suivantes :

- les types d'envoi sont *contravariants* : un canal qui est utilisé pour envoyer des messages de type τ, peut aussi être placé dans un contexte qui l'utilise pour envoyer des messages d'un type *plus petit* que τ ;
- les types de réception sont *covariants* : un canal qui est utilisé pour recevoir des messages de type τ, peut aussi être placé dans un contexte qui l'utilise pour recevoir des messages d'un type *plus grand* que τ ;
- les types d'envoi et réception sont *invariants*.

Un processus qui décide d'appliquer le sous-typage *limite* sa liberté : dans un cas, en pouvant envoyer moins des messages, dans l'autre, en restreignant l'utilisation qu'il peut faire des messages reçus. En conséquence, les types qui indiquent qu'un canal peut être utilisé en envoi et en réception sont *invariants* car il doivent à la fois être covariants et contravariants.

À partir de ces intuitions, Pierce et Sangiorgi construisent un système de types qui satisfait les propriétés de préservation de typage (*subject reduction*), de sûreté, etc.

Pierce et Sangiorgi observent que le type invariant peut être considéré comme une conjonction des types covariant et contravariant. Il n'est pas questions pourtant dans leur papier d'introduire les combinaisons booléennes de types dans leur généralité. Sans même encore parler de l'opérateur de négation, qui est assez problématique, il est déjà difficile de décider quelles équations les opérateurs booléens devraient satisfaire. Par exemple, si on dénote par $c\overline{h}(\tau)$ le type d'envoi seul, par $ch^{+}(\tau)$ le type de réception seule et par $ch(\tau)$ le type d'envoi et réception, alors l'équation

$$c\overline{h}(\tau \wedge \sigma) = c\overline{h}(\tau) \vee c\overline{h}(\sigma)$$

est-elle correcte ? Autrement dit, est-ce qu'un canal sur lequel on peut envoyer des messages qui sont à la fois de type τ et de type σ est la même chose qu'un canal sur lequel on peut envoyer des messages de type τ ou un canal sur lequel on peut envoyer des messages de type σ ? Et si l'égalité n'est pas satisfaite, y a-t-il une relation d'ordre, dans un sens ou dans l'autre ?

Notre but était d'utiliser le cadre du sous-typage sémantique, pour définir un système de types pour le π-calcul avec tous les opérateurs booléens. Pour cela, il a fallu d'abord chercher une intuition sémantique qui puisse nous guider dans la définition de modèle ensembliste. En particulier, on cherchait à définir

l'interprétation ensembliste des trois types : envoi seul, réception seule et envoi et réception. L'intuition qu'on a décidé de suivre est la suivante : un canal est comme une boîte qui contient des valeurs. La boîte a une certaine forme qui détermine quelles valeurs elle peut contenir. Les canaux ont aussi un nom, mais pour ce qui est des types, deux boîtes avec la même forme sont indistinguables, car elles peuvent transporter les mêmes valeurs. Donc, on peut identifier le type des boîtes (canaux) qui peuvent transporter les valeurs de type t avec la forme de ces boîtes. Ensuite, on peut identifier cette forme avec l'ensemble des valeurs qu'elle peut contenir. Mais l'intuition est que cet ensemble est l'interprétation du type t, dénotée $[\![t]\!]$. En conclusion, la sémantique de $ch(\tau)$ est le singleton $\{[\![t]\!]\}$.

Ensuite, les canaux sur lesquels un processus peut envoyer des valeurs de type t sont tous ceux dont la forme permet de contenir *au moins* les valeurs de type t. Cela inclut donc tous les canaux dont la forme permet de contenir les valeurs d'un type t' plus grand que t. En conclusion, la sémantique de $c\bar{h}(\tau)$ est l'ensemble $\{[\![t']\!] \mid t' \geq t\}$. De façon similaire, si un processus attend en réception des valeurs de type t, pour qu'il n'y ait pas d'erreur, il faut et il suffit que le canal puisse transporter *au plus* des valeurs de type t. On en déduit que la sémantique de $ch^+(\tau)$ est l'ensemble $\{[\![t']\!] \mid t' \leq t\}$. Cette intuition nous permet déjà de dériver plusieurs égalités, en particulier $c\bar{h}(\tau) \wedge ch^+(\tau) = ch(\tau)$.

Pour construire un modèle qui respecte à la lettre l'intuition décrite, il faudrait un ensemble D tel que tous les sous-ensembles de D soient aussi éléments de D. Comme l'énonce le Theorème de Cantor, ceci est impossible. Mais, comme pour le cas de types fonctionnels de CDuce, il suffit de donner une sémantique qui, du point de vue de la définition du sous-typage, se comporte *comme* la sémantique intuitive. La construction d'un tel modèle est possible, bien que techniquement assez complexe. L'idée est que le domaine D ne doit pas contenir tous ses sous-ensembles en tant qu'éléments, mais seulement ces sous-ensembles qui sont dénotation de types. Cela résout le problème de cardinalité. Le modèle ainsi construit définit donc une notion de sous-typage qui s'étend à tous les combinateurs booléens. Cela, en particulier, nous permet de répondre à la question posée plus haut : dans notre modèle on a

$$c\bar{h}(\tau \wedge \sigma) \geq c\bar{h}(\tau) \vee c\bar{h}(\sigma)$$

mais en général pas l'inverse.

On utilise ce sous-typage pour typer les processus d'une variante du π-calcul enrichi avec le filtrage ("pattern matching") en entrée. Les règles sont assez naturelles et ressemblent aux règles de Pierce et Sangiorgi. On peut aussi définir, comme dans le cas de CDuce, un modèle des valeurs. Dans ce cas, les valeurs sont les canaux, non pas les processus. Ce modèle a donc moins d'intérêt que dans le cas fonctionnel. Une question intéressante est de savoir si la relation de sous-typage est *décidable*, c'est-à-dire s'il y a un algorithme qui, étant donnés deux types s, t peut dire si $s \leq t$. La réponse est oui et dans l'Annexe C on montre un tel algorithme, qui suit les lignes générales de l'algorithme de CDuce, mais qui est conceptuellement plus compliqué. En effet, dans son exécution, l'algorithme

doit parfois vérifier si un type donné est un *atome*, c'est-à-dire s'il n'existe aucun autre type entre lui et le type vide dans la relation de sous-typage.

Non seulement cette condition est complexe à vérifier, mais elle a comme conséquence l'impossibilité de définir une sémantique pour les types récursifs : en se basant sur ces types atomiques, on définit un type récursif *paradoxal* qui est vide si et seulement si il n'est pas vide.

On montre une solution possible à ces problèmes : il faut se restreindre à la variante *locale* du π-calcul, qui ne permet pas à un processus d'utiliser en réception un canal reçu d'un autre processus. On montre que dans cette variante, il n'est plus nécessaire d'utiliser les types $ch^+(\tau)$ et $ch(\tau)$. La relation de sous-typage en est grandement simplifiée car les types atomiques n'interviennent plus. Cela permet aussi d'utiliser les types récursifs sans engendrer de paradoxe.

3.3 Le codage des fonctions surchargées

Une des premières applications du π-calcul avait été la définition d'un codage du λ-calcul, en particulier des stratégies d'appel par nom et par valeur. Cela était d'ailleurs une des motivations principales pour la création du π-calcul : on avait pas réussi à encoder le λ-calcul dans le calcul de processus CCS.

Le codage du λ-calcul dans le π-calcul ressemble au codage du λ-calcul en passage par continuations. Une fonction est représentée par un canal (le "nom" de la fonction) qui peut être appelé par l'envoi de la valeur d'entrée et un canal sur lequel la valeur de sortie doit être retournée. Ces deux paramètres sont utilisés par un processus répliqué (le "corps" de la fonction) qui retourne la valeur de sortie à la fin du calcul. Dans le codage de l'application, le codage de la fonction est appelée sur le codage de l'argument et la valeur retournée est retournée comme la valeur de toute l'expression. Le canal sur lequel on envoie la valeur d'une expression représente la continuation du calcul.

Pierce et Sangiorgi [PS93] définissent une version typée de l'encodage : à des termes typés du λ-calcul correspondent des termes typés du π-calcul. En particulier, ils définissent une traduction des types du λ-calcul vers les types du π-calculs :

$$(\!|\sigma \to \tau|\!) \;=\; \overline{ch}((\!|\sigma|\!) \times \overline{ch}((\!|\tau|\!))) \;.$$

Intuitivement on appelle une fonction en lui envoyant deux messages : un message contenant l'argument de la fonction et un message contenant le canal sur lequel la fonction est censée envoyer le résultat.

Une traduction similaire a également été proposée par Turner [Tur95] et une discussion extensive se trouve dans le livre de Sangiorgi et Walker [SW02]. Il est important de noter que ce codage respecte aussi le sous-typage : en effet le type fonctionnel du λ-calcul est contravariant sur le type de l'argument et covariant sur le type du résultat, ce qui est exactement le cas pour le codage dans les types canaux (on observe que le codage de l'argument se trouve en dessous d'un opérateur contravariant, tandis que le type du résultat est en dessous de deux opérateurs contravariants.)

Notre idée a été d'appliquer la même traduction entre les deux formalismes du sous-typage sémantique : CDuce et Cpi. Remarquons ici que la sémantique de CDuce est en appel par valeur. L'intérêt de notre travail demeure dans le fait que le codage de Pierce et Sangiorgi ne respecte plus le sous-typage une fois qu'on ajoute les opérateurs booléens. En effet, il existe deux types de CDuce, σ et τ tels que $\sigma \leq \tau$ et $(\!|\sigma|\!) \not\leq (\!|\tau|\!)$. Plus spécifiquement, en CDuce on a $(\sigma \to \tau) \wedge (\sigma \to \tau') \leq \sigma \to \tau \wedge \tau'$, tandis que, en général $c\bar{h}(s \times c\bar{h}(t)) \wedge c\bar{h}(s \times c\bar{h}(t')) \not\leq c\bar{h}(s \times c\bar{h}(t \wedge t'))$. En conséquence, le codage de Pierce et Sangiorgi ne pourra pas marcher si on veut coder CDuce en Cpi, car il échoue déjà au niveau des types.

A quoi est dû ce problème ? Une des caractéristiques de CDuce est la présence de fonctions surchargées, qui peuvent choisir le code à exécuter en fonction du type de l'argument qu'elles reçoivent. Dans le codage de Milner (et donc de Pierce et Sangiorgi), l'appel d'une fonction commence par l'envoi de deux canaux au processus qui encode la fonction : un canal sur lequel on enverra l'argument et un canal sur lequel le processus devra envoyer la réponse. Ces deux canaux sont bien évidemment typés. Le processus qui encode la fonction connaît donc le type de l'argument, mais aussi le type attendu du résultat. Cette connaissance pourrait être utilisée afin de choisi le code à exécuter, option que n'ont pas les fonctions de CDuce. Naturellement, les processus qui encodent de vraies fonctions CDuce n'utilisent pas cette option, mais l'existence de cette possibilité est perceptible au niveau des types. (Plus de détails techniques se trouvent dans l'Annexe C.)

Notre solution a été de modifier le codage des fonctions de façon à ce que le type du canal de réponse ne puisse pas donner d'indication sur le type attendu du résultat. Le canal de réponse peut transporter toutes les valeurs attendues, mais aussi d'autres valeurs. En conséquence, le processus qui encode une fonction ne peut pas se baser sur le type du canal de réponse pour décider quel code exécuter. Ce nouveau codage respecte toutes les égalités entre les types de CDuce, et est très proche du codage de Milner pour ce qui concerne les termes. Les détails techniques ne sont pas aussi simples que cette description semble le suggérer et se trouvent dans l'Annexe C.

Chapitre 4

L'équité dans les systèmes concurrents

4.1 Une notion intuitive

Dans un modèle non-déterministe d'un système informatique, plusieurs exécutions sont possibles. Chaque exécution correspond à une façon de résoudre les choix non-déterministes. Un tel modèle satisfait une spécification quand toutes les exécutions possibles satisfont à leur tour la spécification. Il arrive parfois qu'un le modèle d'un système ne satisfasse pas la spécification souhaitée, mais que les "mauvaises" exécutions ne puissent pas avoir lieu, en pratique. Dans ce cas, on pourrait dire que le modèle ne représente pas correctement la réalité des choses. En particulier, le fait que toute résolution possible des choix soit prise en compte ne correspond pas forcément à la réalité. Dans la réalité, il pourrait y avoir des contraintes qui limitent la façon dont les choix sont résolus. Également, cela peut dépendre des objectifs de l'entité qui résout les choix. La résolution de certains choix peut être guidée pour obtenir des objectifs spécifiques ; ou bien la résolution pourrait être faite sans connaître complètement l'état du système ; parfois, les choix sont résolus en partie par une entité hostile et en partie par une entité collaborative. Toutes ces différentes nuances de non-déterminisme pourraient être représentées en modélisant explicitement le processus qui fait les choix. Cela pourtant nuirait à la simplicité du modèle et en réduirait le niveau d'abstraction.

Il faut donc utiliser un autre système pour contraindre le non-déterminisme. Une des solutions possibles est d'ajouter une hypothèse d'*équité* (*fairness* en anglais) au modèle. La notion d'équité est basée sur l'observation suivante : si un processus demande l'accès à une ressource et si cet accès dépend d'un choix non-déterministe, il est tout-à-fait acceptable que l'accès à la ressource soit refusé ; mais si le processus continue à demander cet accès encore et encore, alors il semble "équitable" qu'il lui soit accordé à un moment donné. Ce fait est garanti par une hypothèse d'équité. Intuitivement, utiliser une hypothèse

d'équité signifie que l'entité qui résout le non-déterminisme a le droit d'être hostile, mais pas pour toujours.

L'exemple le plus simple d'une situation où on ferait appel à une hypothèse d'équité est celui de deux processus en parallèle. Dans ce cas, chaque processus demande juste de faire une action, n'importe laquelle. Un non-déterminisme sans contrainte pourrait laisser exécuter un seul des deux processus. Cela n'est pas considéré comme équitable. D'ailleurs, dans l'exemple donné, le non-déterminisme est en pratique résolu par un gestionnaire de taches (*scheduler*). Ces gestionnaires sont en général programmés pour donner droit d'exécution à tout processus qui le demande.

Suivant cette intuition, plusieurs notions d'équité ont été définies. En général, on définit une exécution équitable par rapport à une transition du système. Il est souvent plus facile de définir les exécutions qui ne sont *pas* équitables. Une exécution n'est pas *faiblement équitable* par rapport à une transition t, si à partir d'un certain moment dans le temps, dans tous les états dans lesquels se trouve le système, la transition t est possible, mais elle n'est jamais exécutée. Une exécution n'est pas *fortement équitable* si la transition t est possible dans un nombre infini d'états de l'exécution mais n'est jamais exécuté.

Pour que cette description informelle devienne une vraie définition mathématique il faut préciser plusieurs concepts. Il faut formellement définir les notions de système, d'exécution et de transition. Il faut dire ce qu'est être "possible", etc. Plusieurs définitions sont donc imaginables et elles ont été en effet imaginées par les chercheurs. Pourtant, bien que la définition intuitive soit assez générale, elle n'avait jamais été formalisée : qu'est-ce qu'une propriété d'équité en général ? Notre travail a été de répondre à cette question.

4.2 Trois définitions équivalentes

Dans les exemples montrés ci-dessus, il y a une structure commune de la définition d'exécution équitable par rapport à une transition t : si t est souvent possible, alors t doit être souvent exécutée. Cette structure devient pour nous la base d'une définition formelle, en généralisant le plus possible les notions de "transition", "souvent" et "possible". D'abord, il faut fixer la notion de système et d'exécution. Un notion simple de système n'est rien d'autre qu'un graphe, les nœuds en étant les états. Une exécution est une suite finie ou infinie d'états telle que deux états successifs sont reliés par un arc. Comme notion de transition, on est amené à considérer d'abord un arc du graphe. Pour généraliser la notion, on commence par observer qu'on peut identifier une transition (arc) t avec l'ensemble des exécutions finies qui se terminent par t. Il semble assez naturel donc de généraliser la notion de transition à tout ensemble d'exécutions finies.

Ensuite, il faut définir la notion de "possible". La première idée est qu'un arc t (vu comme transition) est possible dans un état a si l'arc t a l'état a comme source. Pour généraliser la notion, on peut dire qu'une transition t est possible après une exécution finie x, si elle est possible dans le dernier état de x. Finalement, dans la version plus généralisée, un ensemble d'exécutions finies Q

est possible dans une exécution finie x, si on peut ajouter un état à la fin de x en obtenant une exécution de Q. Plusieurs exemples concrets, ainsi qu'un souci de généralité, nous amènent à considérer toute extension à un nombre arbitraire d'états. En conclusion, on dit qu'une transition généralisée Q est "possible" après une exécution x, s'il existe une extension de x, de longueur quelconque, qui appartient à Q.

Finalement, la notion la plus faible de "souvent" qui semble raisonnable est "infiniment souvent".

Avec ces trois généralisations, on obtient la définition suivante d'équité (ou l'implication devient une disjonction pour faciliter la lecture) : une exécution x est équitable par rapport à une transition généralisée Q si

un nombre infini de préfixes de x appartient à Q *ou* il existe un préfixe y de x tel qu'aucune exécution qui étend y n'est dans Q.

Cette définition est très raisonnable mais elle a deux défauts : premièrement, on peut montrer que certaines notions d'équité ne sont pas incluses, notamment l'équité forte ne satisfait pas la définition (on note que l'équité forte parle d'extensions avec un seul état supplémentaire) ; deuxièmement la classe des propriétés qui satisfont la définition n'est pas fermée par élargissement (ou affaiblissement), ce qui est aussi désirable pour plus de souplesse. Pour résoudre ces deux problèmes, la solution est la même : imposer la clôture par affaiblissement *par définition*.

Soit $F(Q)$ l'ensemble des exécutions équitables selon la définition ci dessus. On dit qu'un ensemble d'exécutions est une propriété d'équité s'il contient un ensemble $F(Q)$ pour un Q quelconque.

Pour montrer que cette définition est raisonnable, on donne deux caractérisations différentes de la même notion.

D'abord, il y a une version topologique. On note que l'ensemble des exécutions finies et infinies d'un système peut être vu comme une ordre partiel directement complet (DCPO). Sur cet ensemble on peut donc considérer une topologie standard appelée la *topologie de Scott*. Déjà Alpern et Schneider [AS85] avaient observé que certaines notions utilisées en vérification ont une définition topologique. En particulier, les propriétés de sécurité sont les *fermés* de la topologie de Scott, les propriétés de garantie sont les *ouverts* et les propriétés de vivacité (liveness) sont les ensembles *denses*.

Ce qu'est prouvé dans nos travaux, est que les propriétés d'équité sont précisément les ensembles *co-maigres* dans la topologie de Scott. La définition d'ensemble co-maigre étant un peu technique, on renvoie le lecteur à l'Annexe D. Il suffit de rappeler ici l'intuition : les ensembles co-maigres sont les ensembles topologiquement "gros". Le complémentaire d'un ensemble co-maigre est un ensemble *maigre* ou topologiquement petit. L'intuition est que, d'un point de vue topologique, il y a "très peu" d'exécutions qui ne sont pas équitables.

La troisième définition d'équité est très liée à la deuxième, mais elle apporte un nouvel et très intéressant point de vue. On peut définir un jeu sur un graphe (modèle d'un système) qui se joue entre deux joueurs : Banach (le Bon) et Mazur (le Méchant). Le but du jeu est un ensemble G d'exécutions infinies. Mazur

commence en choisissant une exécution finie. Banach continue en choisissant une extension finie de l'exécution choisie par Mazur. Mazur étend l'exécution ainsi obtenue et ainsi de suite. L'alternance des deux joueurs produit une exécution infinie. Si cette exécution appartient au but G, c'est Banach qui gagne sinon c'est Mazur.

Ce jeu avait été proposé par (le vrai) Mazur et ce fut (le vrai) Banach qui prouva que Banach a une stratégie gagnante si et seulement si G est co-maigre dans la topologie de Scott [Mau81]. Une exécution est donc équitable si elle a été obtenue pendant une partie gagnante pour Banach. Ici, l'intuition est que Banach et Mazur s'alternent pour résoudre le non-déterminisme du système. Le joueur méchant peut forcer le système à faire de mauvaises choses de temps à autres, mais il ne peut pas empêcher au bon joueur de lui faire faire aussi de bonnes choses de temps à autres.

4.3 Propriétés

Caractérisée par trois définitions indépendantes, la classe des propriétés d'équité a pareillement plusieurs attributs intéressants :

- Si un ensemble contient une proprieté d'équité, il est une propriété d'équité. En conséquence la classe des propriétés d'équité est fermée par union.
- La classe des propriétés d'équité est fermée par intersection dénombrable.
- Toute propriété d'équité est vivace.
- La plupart des notions d'équité étudiées en littérature définissent une propriété d'équité.

La fermeture par élargissement est intéressante parce qu'elle correspond à la notion logique d'*affaiblissement*. On utilise l'équité comme *hypothèse* pour prouver qu'un système satisfait une spécification et pour renforcer le résultat, on pourrait vouloir affaiblir l'hypothèse. La fermeture par intersection permet de définir les notions d'équité de façon modulaire, par exemple en spécifiant différentes notions par rapport à différentes parties du système. Le fait que toute propriété d'équité soit vivace est une condition proposé par Apt, Francez and Katz [AFK88]. Cette condition s'appelle également *fermeture par machine* (*machine-closure* en anglais). Essentiellement, elle postule qu'un système doit pouvoir avoir toujours la possibilité de se comporter de façon équitable.

Une question qu'on pourrait se poser est : a-t-on été suffisamment général ? Pourrait-on ajouter d'autres propriétés à la classe qu'on a définie ? On ne connait pas de réponse définitive, mais si on se restreint à une classe très raisonnable de propriétés, on peut alors donner une réponse négative. La classe qu'on considère est celle des propriétés *déterminées*, c'est-à-dire les propriétés pour lesquelles soit Banach soit Mazur a une stratégie gagnante dans le jeu de Banach-Mazur. Cette classe est très grande et elle contient tous les ensembles Boreliens [Oxt71]. Le résultat qu'on prouve est que la classe des propriétés d'équité est la plus grande classe de propriétés déterminées qui contient seulement des propriétés de vivacité et qui soit fermée par intersection.

Finalement pour le dernier attribut décrit ci-dessus, il faut mentionner que Kwiatkowska avait proposé une notion d'équité [Kwi91] qui était fermée par intersection et qui était incluse dans la vivacité, mais qui ne capturait pas toutes les propriétés étudiées en littérature. En particulier, on montre que l'équité forte ne satisfait pas la définition de Kwiatkowska.

4.4 Relation avec la probabilité

Une autre façon de restreindre le non-déterminisme est d'imposer que tout choix non-déterministe soit fait de façon aléatoire. Dans un graphe, on peut obtenir cela en ajoutant pour chaque nœud une distribution de probabilité sur les arcs sortants. Ce qu'on obtient s'appelle une *chaîne de Markov*. Une chaîne de Markov définit une distribution de probabilité sur les exécutions du système. En présence d'une chaîne de Markov, on peut donc proposer une autre notion d'équité : une propriété d'*équité probabiliste* est un ensemble qui a probabilité 1. Cette définition a plusieurs analogies avec la définition d'équité décrite ci-dessus. En particulier :

- si un ensemble contient une propriété d'équité probabiliste, il est une propriété d'équité probabiliste ;
- la classe des propriétés d'équité probabiliste est fermée par intersection dénombrable ;
- si la définition d'équité correspond aux ensembles "topologiquement très gros", les ensembles de probabilité 1 peuvent être vus comme "probabilisitiquement très gros".

Mises à part ces similarités, pouvons-nous trouver une correspondance formelle entre ces deux notions ? On peut montrer qu'en général elles sont différentes, car il existe un système pour lequel il y a une propriété d'équité qui n'est pas une propriété d'équité probabiliste et une propriété d'équité probabiliste qui n'est pas une propriété d'équité. Mais on a prouvé que dans certaines conditions assez raisonnables, les deux notions coïncident. Les conditions sont les suivantes : le système doit être *fini*, c'est-à-dire il doit avoir un nombre fini de nœuds ; la propriété doit être ω-régulière, c'est-à-dire acceptée par un automate de parité [Tho90]. La classe des propriétés ω-régulières est très grande : elle inclut, par exemple, toutes les propriétés qu'on peut exprimer à l'aide de la logique temporelle LTL.

Le théorème qu'on a prouvé dit que pour un système probabiliste fini, une propriété ω-régulière est une propriété d'équité probabiliste si et seulement si elle est un propriété d'équité. Ce résultat est assez intéressant en soi, mais il a des conséquences pratiques : cela nous permet en particulier d'appliquer les techniques de vérification probabiliste (qui sont bien développés depuis des années) à la vérification équitable qu'on va introduire ci dessous.

4.5 Vérification

Traditionnellement en vérification, on utilise l'équité comme hypothèse qu'on ajoute à un système pour lui permettre de satisfaire une spécification qui serait violée autrement. Mais il faut souvent exprimer explicitement cette hypothèse d'équité : on dit "sous l'hypothèse d'équité forte, la spécification est satisfaite". Parfois il n'est pas évident de trouver la bonne hypothèse.

Pour vérifier automatiquement une spécification sous une hypothèse d'équité, il y a plusieurs possibilités. Si l'hypothèse peut s'exprimer dans le même formalisme que la spécification, alors on peut vérifier une spécification conditionnelle de la forme "si hypothèse alors spécification". Mais parfois l'équité n'est pas exprimable dans le formalisme de spécification. Dans ce cas, on peut modifier l'algorithme de vérification pour qu'il prenne en compte l'hypothèse.

On propose une autre approche : on propose de vérifier si une spécification est satisfaite sous une quelconque hypothèse d'équité qui ne doit pas forcement être connue. En effet si une spécification est satisfaite sous une hypothèse d'équité, c'est que la spécification (vue comme ensemble d'exécutions) contient l'équité. Mais cela implique que la spécification même peut être vue comme une propriété d'équité au sens de notre définition.

Le problème qu'on se pose est le suivant : étant donné un système et une spécification, est-ce que la spécification est une propriété d'équité du système ? On appelle ça le problème de la vérification équitable. Dans nos articles, on propose plusieurs algorithmes pour répondre à cette question. Pourtant l'algorithme le plus efficace qu'on connait est emprunté à la vérification probabiliste. Il s'agit d'un algorithme proposé par Courcoubetis and Yannakakis [CY95] qui s'exécute en temps exponentiel en la taille de la formule logique qui exprime la spécification et en temps linéaire par rapport à la taille du système.

La correspondance avec l'équité probabiliste nous permet aussi de connaître exactement la complexité du problème de la vérification équitable. Pour une spécification exprimée en LTL, il s'agit d'un problème PSPACE-complet en la taille de la formule logique. Cela correspond exactement à la complexité du problème de la vérification "classique". Matthias Schmalz [Schdf] a trouvé que cette correspondance vaut aussi pour plusieurs fragments intéressants de la logique et que pour un fragment en particulier (appelé "formules de Müller"), la vérification équitable est algorithmiquement plus simple que la vérification classique.

On a aussi montré un algorithme pour la vérification équitable qui utilise la caractérisation de l'équité en termes de jeux de Banach-Mazur, mais cet algorithme est doublement exponentiel en la taille de la formule et donc pas optimal. Il reste encore à trouver un algorithme optimal basé sur le jeu de Banach-Mazur.

4.6 Extensions à d'autres domaines

Ajouter une hypothèse d'équité à un système non-déterministe est une façon de réduire le non-déterminisme. Il s'agit dans ce cas d'un non-déterminisme qu'on ne contrôle pas, ou qu'on ne peut pas connaître exactement, mais dont on sait au moins qu'il ne joue pas contre nous. On pourrait dire qu'on modélise des systèmes fermés, mais dont les composantes ne sont pas complètement spécifiées.

Mais il y a d'autres interprétations du non-déterminisme. Les choix non-déterministes peuvent être partagés en ceux qui sont contrôlés par le système et ceux qui sont contrôlés par l'environnement. En ce cas, le formalisme qu'on vient de décrire n'est pas satisfaisant. Pour modéliser cette forme de non-déterminisme il faut utiliser la notion de *jeu*. Les états d'un système sont partagés deux : ceux contrôlés par Adam ("nous") et ceux contrôlés par Eve ("eux"). Les exécutions du système sont obtenues par une partie où chaque joueur choisit l'état suivant quand le système se trouve dans un des états qu'il/elle contrôle. Ce jeu diffère du jeu de Banach-Mazur en plusieurs aspects. Cependant, les deux jeux peuvent être combinés comme on l'a montré récemment [ACV10]

Le cadre qu'on a étudié se restreint aussi par l'utilisation d'exécutions totalement ordonnées. Il serait intéressant d'étudier l'équité aussi dans un contexte d'exécutions partiellement ordonnées, comme par exemple dans le cas des structures d'événements. L'ensemble des exécutions d'une structure d'événement forme aussi un ordre partiel directement complet et en conséquence il est doté d'une topologie de Scott. On aimerait étudier à quoi correspondent les ensembles co-maigre dans ce cas plus général. Est-ce qu'ils capturent une notion d'équité intéressante ? Est-ce qu'ils correspondent à une variation du jeu de Banach-Mazur ?

Chapitre 5

Le Protocole de Handshake

5.1 Introduction

Dans le contexte du calcul concurrent, il est important que les différents sujets qui prennent part au calcul soient en accord sur la façon de communiquer l'un avec l'autre. Il y a plusieurs niveaux d'abstraction sur lesquels les sujets doivent se trouver d'accord. Ils doivent tous utiliser des chaînes de bits, ils doivent être d'accord sur le codage de données, sur la façon de vérifier les erreurs de transmission, etc. Au niveau le plus élevé, ils doivent se trouver d'accord sur *le protocole* de communication : est-ce qu'ils communiquent de façon synchrone ou asynchrone ? Est-ce que le temps de transmission est pris en compte ? Est-ce qu'on doit envoyer un récépissé pour tout message reçu ?

Un des protocoles qui ont été utilisés en pratique pour organiser le calcul concurrent est le *protocole Handshake* [VB93]. Intuitivement, les sujets qui communiquent en respectant ce protocole disposent chacun d'un ensemble de *portes*. Chaque porte est partagée par exactement deux sujets. À chaque instant, chaque porte se trouve en mode d'*envoi* pour un sujet et de *réception* pour l'autre. Le sujet en mode réception doit être prêt à accepter le message qui lui sera envoyé. Le sujet en mode envoi décide de manière autonome si et quand envoyer un message. Les communications sur deux portes différentes sont concurrentes. Après qu'un message a été envoyé sur une porte, le sujet en mode envoi entre en mode réception et dualement.

Bien qu'utilisé en pratique, peu d'études théoriques s'étaient penchées sur ce protocole. Van Berkel [VB93] donne une définition formelle de *langage Handshake* en termes de mots finis sur un alphabet de portes et qui satisfont un certain nombre d'axiomes. Un langage handshake représente un processus qui communique avec l'environment. Un mot de ce langage représente une exécution du protocole.

Deux processus qui suivent le protocole Handshake communiquent en se synchronisant sur les portes communes. Cette intuition amène Van Berkel à définir une notion de composition parallèle de deux langages Handshake. Un mot du

langage composé est obtenu en entrelaçant deux mots des deux langages et par la suite en effaçant tous les caractères qui correspondent aux portes communes. Fossati [Fos07] a pourtant montré que cette définition de composition n'est pas associative. Il a donc entrepris de donner une autre description formelle de ces protocoles dans laquelle une notion de composition plus satisfaisante soit possible.

Cela a d'abord amené à un modèle à jeux des protocoles Handshake [Fos07]. Cependant dans ce modèle, certains protocoles au comportement non-déterministe ne peuvent pas être représentés. Notre collaboration a eu pour but de trouver d'autres modèles qui puissent surmonter cet obstacle.

5.2 Un modèle à réseaux de Petri

Le premier travail qu'on a réalisé a été la proposition d'un modèle à réseaux de Petri des sujets qui communiquent en respectant le protocole Handshake. Les deux ingrédients fondamentaux sont

- la définition d'un sous-réseau qui représente une porte. Il faut que ce réseau alterne entre mode réception et mode envoi et il faut qu'en mode réception il soit toujours prêt à recevoir.
- une notion de composition de réseaux. Deux réseaux se composent en connectant les portes qui ont le même nom (à condition qu'elles soient en mode dual)

Les transitions du réseau se divisent en internes (non observables) et externes (observables). Les transitions externes sont exactement les transitions des portes qui ne sont pas encore connectées. Une observation représente donc la communication avec l'environnement. On définit le langage d'un réseau comme l'ensemble des séquences *de repos* de transitions externes. Une séquence est dite de repos si elle se termine dans un marquage où aucun envoi n'est possible. On montre que le langage d'un réseau ainsi défini est toujours un langage Handshake selon la définition de Van Berkel, mais aussi que tout langage Handshake est le langage d'un certain réseau Handshake, bien que ce réseau soit en général infini. Naturellement, on ne peut pas espérer représenter tous les langages Handshake par des réseaux finis, car il y a des langages non-récursifs.

La composition de réseaux est obtenue en mettant en communication les portes communes. Les transitions de ces portes deviennent internes et elle ne contribuent plus à la définition du langage. À cause de la nature "graphique" de cette définition, la composition de réseaux Handshake est naturellement associative. On pourrait donc vouloir définir la composition des langages à l'aide de la notion de composition de réseaux, mais cela n'est pas possible, car en général pour un langage donné, il y a plusieurs réseaux qui le définissent. Pour un langage donné, il n'y a pas de réseau canonique. L'intuition est que les réseaux expriment mieux certaines relations entre choix et concurrence qui ne peuvent pas être représentées par les langages. Il faut souligner que cela n'est pas a priori une évidence. Il est vrai que la sémantique à traces des langages de processus est moins expressive que les systèmes de transitions étiquetées, mais les langages

Handshake de Van Berkel sont plus expressifs qu'une sémantique de trace : il peuvent en effet définir quelques notions de choix plus complexe.

5.3 Un calcul de processus

Nous avons par la suite proposé une deuxième façon de formaliser la notion de protocole Handshake, à l'aide d'un calcul de processus avec deux formes de communication. Une communication par synchronisation semblable à celle de CCS, qui représente la synchronisation sur les portes Handshake et une synchronisation par ressources partagées, qui modélise le comportement purement interne d'un processus. Les envois de messages sur les portes Handshake consomment des ressources internes, tandis que la réception ajoute des ressources. Un système de types garantit que chaque porte est soit externe et visible, soit partagée par exactement deux processus. Il garantit aussi l'alternance entre les modes d'envoi et de réception. Un processus en mode d'envoi peut ne pas envoyer, car il lui manque les ressources, tandis qu'un processus en mode réception est toujours prêt à recevoir, ce qui est conforme avec la discipline Handshake.

Il y a deux façons de composer les processus. On a une composition qui permet de partager les ressources et qui est utilisée pour construire les "briques" de base du protocole. Ensuite, les processus se composent en se synchronisant sur les portes Handshake communes, comme pour les langages et les réseaux.

Tout comme dans le cas des réseaux de Petri, on définit la notion de trace *de repos* : une trace qui se termine dans un état du processus où aucun envoi n'est possible, parce que toute porte qui n'est pas en mode réception n'a pas assez de ressources pour envoyer son message.

Nous montrons que pour tout processus de ce calcul, l'ensemble de traces de repos constitue un langage Handshake.

Finalement, nous définissons une sémantique du calcul de processus à l'aide des réseaux de Petri. On montre que la correspondance ne se limite pas aux langages des mots de repos : un processus Handshake et le réseau de Petri correspondant sont également faiblement bisimilaires. De plus, pour tout réseau de Petri fini, il existe un processus Handshake qui lui est bisimilaire.

5.4 Questions ouvertes

Avec nos modèles des réseaux de Petri et des processus, on a proposé une formalisation des protocoles Handshake alternative à celle de VanBerkel. Cependant, peut-on définir un autre une autre notion de composition parallèle entre langages Handshake qui soit associative ? Ou au contraire, peut-on prouver qu' une telle composition ne peut pas exister ? La question reste ouverte.

Dans ce contexte, les exécutions infinies jouent un rôle important. Dans la définition de Van Berkel, on obtient un mot du langage composé en synchronisant deux mots finis, mais aussi en prenant la partie visible (finie) d'une synchronisation infinie (c'est-à-dire d'une chaîne infinie de synchronisations). Cette

définition n'étant pas satisfaisante, car non associative, on a cherché à réduire les cas où cette synchronisation infinie était permise, en ajoutant différentes conditions d'équité. Pourtant, aucune définition parmi celles qu'on a essayées n'est associative. Ceci nous amène à penser qu'une définition raisonnable de composition de langages est impossible et on cherche actuellement à le prouver.

Chapitre 6

Quelques mots finaux

Ce mémoire a comme but de convaincre le lecteur que je suis apte à “diriger la recherche”. J'ai voulu exposer mes principales contributions à la recherche en informatique théorique et j'ai également évoqué le fait qu'il y a des questions ouvertes, des résultats entrevus, mais pas encore formalisés, que je me promet d'explorer dans les années à venir.

Mais que signifie vraiment l'expression “diriger la recherche”. Pour pouvoir diriger, il faut connaître la direction, mais si on savait au préalable où se trouve ce nous recherchons, nous aurions presque déjà le résultat. C'est la recherche qui dirige les chercheurs et non pas le contraire. Certes, cette affirmation n'est pas complètement valide dans des domaines où l'expérimentation joue le rôle principal. Faire marcher une sonde martienne ou un accélérateur de particules, tester un médicament sur les souris, excaver dans la cordillère, toutes ces activités impliquent outre l'idée première, une organisation, une programmation définie.

Mais les sciences plus théoriques, telles l'informatique, les mathématiques, la philosophie, la littérature, suivent un parcours de recherche différent. Dans certains cas, la formulation et la rédaction d'un projet de recherche complet constituent déjà un travail quasi-abouti, où le résultat est d'emblée soupçonné. Dans ces conditions, “diriger” la recherche devient presque la même chose que “mener” la recherche. Ce mémoire, je l'espère, montre que je suis capable de le faire.

Toutefois “diriger la recherche” peut aussi signifier diriger le travail des étudiants en thèse, accompagner les recherches des jeunes chercheurs. On peut conseiller, encourager, suggérer, tout en affirmant le travail personnel. Ma participation au travail doctoral portant sur le protocole Handshake, ainsi que le suivi des travaux de Master, démontrent mon aptitude à le faire. Et c'est finalement ainsi, je pense, la seule vraie façon de “diriger la recherche” dans notre domaine.

Bibliographie

[AB05] Samy Abbes and Albert Benveniste. Branching cells as local states for event structures and nets : Probabilistic applications. In *Proceedings of 8th FoSSaCS*, volume 3441 of *LNCS*, pages 95–109. Springer, 2005.

[AB06] Samy Abbes and Albert Benveniste. Probabilistic models for true-concurrency : branching cells and distributed probabilities for event structures. *Information and Computation*, 204(2) :231–274, 2006.

[ABG04] Alessandro Aldini, Mario Bravetti, and Roberto Gorrieri. A process-algebraic approach for the analysis of probabilistic noninterference. *Journal of Computer Security*, 12(2) :191–245, 2004.

[ACV10] Eugene Asarin, Raphaël Chane-Yack-Fa, and Daniele Varacca. Fair adversaries and randomization in two-player games. In *Proceedings of 13th FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2010.

[AFG93] Paul C. Attie, Nissim Francez, and Orna Grumberg. Fairness and hyperfairness in multi-party interactions. *Distributed Computing*, 6 :245–254, 1993.

[AFK88] Krysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2 :226–241, 1988.

[AGN96] Samson Abramsky, Simon Gay, and Raja Nagarajan. Interaction categories and the foundations of types concurrent programming. In *Proc. of the 1994 Marktoberdorf Summer School on Deductive Program Design*, pages 35–113. Springer, 1996.

[AH94] Rajeev Alur and Thomas A. Henzinger. Finitary fairness. In *Proceedings of 9th LICS*, pages 52–61. IEEE Computer Society, 1994.

[AH95] Rajeev Alur and Thomas A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In *Proceedings of 7th CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 1995.

[AKH92] S. Arun-Kumar and Matthew Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29(9) :737–760, 1992.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82 :253–284, 1991.

[ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 1–17, London, UK, 1989. Springer-Verlag.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21 :181–185, October 1985.

[Bar98] Andrew Bardsley. Balsa : an asynchronous circuit synthesis system. master's thesis. Department of Computer Science, University of Manchester, 1998.

[B+08] Christel Baier, Nathalie Bertrand, Patricia Bouyer, Thomas Brihaye, and Marcus Größer. Almost-sure model checking of infinite paths in one-clock timed automata. In *Proceedings of the 22nd LICS*, pages 217–226. IEEE Computer Society, 2008.

[BC82] Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(265–321), 1982.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce : an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[BCM99] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and event structure semantics for graph grammars. In *Proceedings of 2nd FoSSaCS*, volume 1578 of *LNCS*, pages 73–89. Springer, 1999.

[BEM96] Rachel Ben-Eliyahu and Menachem Magidor. A temporal logic for proving properties of topologically general executions. *Inf. Comput.*, 124(2) :127–144, 1996.

[Bes84] Eike Best. Fairness and conspiracies. *Information Processing Letters*, 18 :215–220, 1984. Erratum ibidem 19 :162.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[BG95] Nadia Busi and Roberto Gorrieri. A petri net semantics for pi-calculus. In *Proceedings of 6th CONCUR*, volume 962 of *LNCS*, pages 145–159. Springer, 1995.

[BGK03] Dietmar Berwanger, Erich Grädel, and Stephan Kreutzer. Once upon a time in a west - determinacy, definability, and complexity of path games. In *Proceedings of 10th LPAR*, volume 2850 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2003.

[BGZ03] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *Proceedings of 30th ICALP*, volume 2719 of *LNCS*, pages 133–144. Springer, 2003.

[BHY01] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the π-calculus. In *Proceedings of TLCA'01*, volume 2044 of *LNCS*, pages 29–45. Springer, 2001.

[BMM06] Roberto Bruni, Hernan Melgratti, and Ugo Montanari. Event structure semantics for nominal calculi. In *Proc.of CONCUR*, volume 4137 of *LNCS*, pages 295–309. Springer, 2006.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *TCS*, 195(2) :205–226, 1998.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, 1992.

[BS98] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Inf.*, 35(5) :353–400, 1998.

[BS01] Maria Grazia Buscemi and Vladimiro Sassone. High-level petri nets as type theories in the join calculus. In *Proceedings of 4th FOSSACS*, volume 2030 of *LNCS*, pages 104–120. Springer, 2001.

[BSV03] Falk Bartels, Ana Sokolova, and Erik de Vink. A hierarchy of probabilistic system types. In *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[Cas05] Giuseppe Castagna. Semantic subtyping : Challenges, perspectives, and open problems. In *ICTCS*, pages 1–20, 2005.

[CDV06] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Daniele Varacca. Encoding cduce in the cpi-calculus. In *Proceedings of 17th CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 310–326. Springer, 2006.

[CDV05] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the π-calculus. In *LICS '05, 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2005.

[CDV08] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. *Theor. Comput. Sci.*, 398(1-3) :217–242, 2008.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking : algorithmic verification and debugging. *Commun. ACM*, 52(11) :74–84, 2009.

[CF05a] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *PPDP '05 : Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–199, New York, NY, USA, 2005. ACM.

[CF05b] Pierre-Louis Curien and Claudia Faggian. L-nets, strategies and proof-nets. In *Proceedings of CSL*, volume 3634 of *LNCS*, pages 167–183. Springer, 2005.

[CP05] Kostas Chatzikokolakis and Catuscia Palamidessi. A framework to analyze probabilistic protocols and its application to the partial secrets exchange. In *Proceedings of Symposium on Trustworthy Global Computing, 2005*, volume 3705 of *LNCS*. Springer, 2005.

[CS00] Gian Luca Cattani and Peter Sewell. Models for name-passing processes : Interleaving and causal. In *Proceedings of 15th LICS*, pages 322–332. IEEE, 2000.

[CS72a] Computation Structures Group. Progress report 1969-70. Technical Report MIT-MAC-memo-53, Massachussets Institute of Technology, Project MAC, Cambridge, MA, USA, January 1972.

[CS72b] Computation Structures Group. Progress report 1970-71. Technical Report MIT-MAC-memo-64, Massachussets Institute of Technology, Project MAC, Cambridge, MA, USA, January 1972.

[CVY07] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Compositional event structure semantics for the internal pi -calculus. In *Proceedings of 18th*

CONCUR, volume 4703 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[CVY12] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Event structure semantics of the parallel extrusion in the pi -calculus. In *Proceedings of 23rd CONCUR*, Lecture Notes in Computer Science. Springer, 2012. accepted.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4) :471–522, 1985.

[CY95] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4) :857–907, 1995.

[DDM88] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. On the consistency of "truly concurrent" operational and denotational semantics (extended abstract). In *Proceedings of 3rd LICS*, pages 133–141, 1988.

[DE95] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[DEP02] Josée Desharnais, Abbas Edalat, and Prakash Panangaden. Bisimulation for labelled markov processes. *Information and Computation*, 179(2) :163–193, 2002.

[DH02] Vincent Danos and Russell S. Harmer. Probabilistic game semantics. *ACM Transactions on Computational Logic*, 3(3) :359–382, 2002.

[DK04] Vincent Danos and Jean Krivine. Reversible communicating systems. In *Proceedings of 15th CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[D+92] Philippe Darondeau, Doris Nolte, Lutz Priese, and Serge Yoccoz. Fairness, distances and degrees. *Theoretical Computer Science*, 97(1) :131–142, 1992.

[DP99] Pierpaolo Degano and Corrado Priami. Non-interleaving semantics for mobile processes. *Theor. Comp. Sci.*, 216(1-2) :237–270, 1999.

[DHW05] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic lambda calculus and quantitative program analysis. *Journal of Logic and Computation*, 2005. To appear.

[Dug66] James Dugundji. *Topology*. Allyn and Bacon, 1966.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 995–1072. MIT Press and Elsevier, 1990.

[EN00] Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing - ten years after. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 599–622. The MIT Press, 2000.

[Eng96] Joost Engelfriet. A multiset semantics for the pi-calculus with replication. *Theoretical Computer Science*, 153(1&2) :65–94, 1996.

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping : Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[F+96] Cédric Fournet, Georges Gonthier, Jean-Jacques. Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR '96, 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.

[FM05] Claudia Faggian and François Maurel. Ludics nets, a game model of concurrent interaction. In *Proceedings of 20th LICS*, pages 376–385, 2005.

[FMW02] Harald Fecher, Mila E. Majster-Cederbaum, and Jinzhao Wu. Action refinement for probabilistic processes with true concurrency models. In *PAPM-PROBMIV*, volume 2399 of *LNCS*, pages 77–94. Springer, 2002.

[Fos07] Luca Fossati. Handshake games. In *Second International Workshop on Developments in Computational Models, DCM'06*, volume 171-3, pages 21–41. ENTCS, Elsevier, 2007.

[Fos09] Luca Fossati. *Modeling the Handshake Protocol for Asynchrony.* PhD thesis, Dip. di Informatica, Univ. di Torino & Lab. Preuves Programmes et Systèmes (PPS), Univ. Paris 7, 2009.

[FP07a] Claudia Faggian and Mauro Piccolo. A graph abstract machine describing event structure composition. In *GT-VC workshop*, ENTCS, 2007.

[FP07b] Claudia Faggian and Mauro Piccolo. A graph abstract machine describing event structure composition. In *Proceedings of the workshop GT-VC 06*, volume 175 :4 of *Electr. Notes Theor. Comput. Sci.*, pages 21–36, 2007.

[FP07c] Claudia Faggian and Mauro Piccolo. Ludics is a model for the (finitary) linear pi-calculus. In *Proceedings of TLCA'07*, LNCS. Springer, 2007.

[FR03] Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3) :121–163, 2003.

[Fra86] Nissim Francez. *Fairness.* Springer, 1986.

[Fri04] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML.* PhD thesis, Université Paris 7, December 2004.

[FV08] Luca Fossati and Daniele Varacca. A Petri net model of handshake circuits. In *First International Workshop on Interactive Concurrency Experience, ICE'08.* ENTCS, Elsevier, 2008.

[FV09] Luca Fossati and Daniele Varacca. The calculus of handshake configurations. In *Proceedings of 12th FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2009.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1) :80–112, 1985.

[Ghi07] Dan R. Ghica. Geometry of synthesis : a structured approach to VLSI design. In *POPL '07 : Proc. of the 34th annual ACM Symposium on Principles of Programming Languages*, pages 363–375. ACM Press, 2007.

[Gra08] Erich Grädel. Banach-Mazur games on graphs. In *Proceedings of the 28th FSTTCS.* LIPIcs 2 Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2008.

[GSV04] Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In *Proceedings of 7th FoSSaCS*, volume 2987 of *LNCS*, pages 226–240. Springer, 2004.

[Han91] Hans Hansson. *Time and Probability in Formal Design of Distributed systems.* PhD thesis, Uppsala University, 1991.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.

[HKP04] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004.

[HL10] Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.*, 411(22-24) :2223–2238, 2010.

[HO00] J. Martin E. Hyland and C.-H. Luke Ong. On full abstraction for PCF : I, II, and III. *Information and Computation*, 163(2) :285–408, 2000.

[Hos03] Haruo Hosoya and Benjamin C. Pierce Xduce : A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2) :117–148, 2003.

[HP00] Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In *Proceedings of 3rd FoSSaCS*, volume 1784 of *LNCS*, pages 146–160. Springer, 2000.

[HR02] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Inf. and Comp.*, 173 :82–120, 2002.

[Hso] http ://www.handshakesolutions.com/.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. ECOOP 91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[HY95] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *TCS*, 151(2) :437–486, 1995.

[HY02] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002. Full version available at www.doc.ic.ac.uk/˜yoshida.

[Jae09] Manfred Jaeger. On fairness and randomness. *Information and Computation*, 207(9) :909–922, 2009.

[JJ95] Lalita Jategaonkar Jagadeesan and Radha Jagadeesan. Causality and true concurrency : A data-flow analysis of the pi-calculus (extended abstract). In *Proceedings of 4th AMAST*, volume 936 of *LNCS*, pages 277–291. Springer, 1995.

[BLM05] Allen L. Brown Jr., Cosimo Laneve, and L. Gregory Meredith. Piduce : A process calculus with native xml datatypes. In *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2005.

[Jou99] Yuh-Jzer Joung. Localizability of fairness constraints and their distributed implementations. In *Proceedings of 10th CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 1999.

[Jou01] Yuh-Jzer Joung. On fairness notions in distributed systems, part I : A characterization of implementability. *Information and Computation*, 166 :1–34, 2001.

[JUY93] M.B. Josephs, J.T. Udding, and Y. Yantchev. Handshake algebra. Technical Report SBU-CISM-93-1, School of Computing, Information Systems and Mathematics, South Bank University, London, 1993.

[Kat96] Joost-Pieter Katoen. *Quantitative and Qualitative Extensions of Event Structures*. PhD thesis, University of Twente, 1996.

[KP93] Gilles Kahn and Gordon D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2) :187–277, 1993.

[KP95] Orna Kupferman and Amir Pnueli. Once and for all. In *Proceedings of 10th LICS*, pages 25–35. IEEE Computer Society, 1995.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5) :914–947, 1999.

[Kwi89] Marta Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7) :371–386, 1989.

[Kwi91] Marta Z. Kwiatkowska. On topological characterization of behavioural properties. In G. Reed, A. Roscoe, and R. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 153–177. Oxford University Press, 1991.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2) :125–143, 1977.

[Lam00] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4) :239–245, 2000.

[Lan69] Lawrence H. Landweber. Decision problems for omega-automata. *Mathematical Systems Theory*, 3(4) :376–384, 1969.

[LMS10] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing higher-order pi. In *Proceedings of 21st CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2010.

[LPS81] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness : The ethics of concurrent termination. In *Proceedings of 8th ICALP*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 1981.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[LS82] Daniel Lehmann and Saharon Shelah. Reasoning with time and chance. *Information and Control*, 53(3) :165–198, 1982.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1) :1–28, 1991.

[Mac71] Saunders MacLane. Categories for the working mathematician, 1971.

[Mac95] Ian Mackie. The geometry of interaction machine. In *POPL '95 : Proc. of the 22nd ACM Symposium on Principles Of Programming Languages*, pages 198–208. ACM Press, 1995.

[Mau81] R. Daniel Mauldin. *The Scottish Book : Mathematics from the Scottish Cafe*. Birkhäuser, 1981.

[Maz86] Antoni Mazurkiewicz. Trace theory. In *Petri Nets : Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer, 1986.

[Mel04] Paul-André Melliès. Asynchronous games 2 : The true concurrency of innocence. In *Proceedings of 15th CONCUR*, pages 448–465, 2004.

[Mel05] Paul-André Melliès. Asynchronous games 4 : A fully complete model of propositional linear logic. In *Proceedings of 20th LICS*, pages 386–395, 2005.

[Mer04] Massimo Merro. *Locality in the pi-calculus and applications to distributed objects*. PhD thesis, École des Mines de Paris, 2004.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2) :119–141, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems : The Pi Calculus*. Cambridge University Press, 1999.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM, 1990.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, 1992.

[MP95] Ugo Montanari and Marco Pistore. Concurrent semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 1 :411–429, 1995. Proceedings of MFPS '95.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1) :1–77, 1992.

[MS] Robin Milner and Davide Sangiorgi. Techniques for "weak bisimulation upto". Revised version of a paper appeared in *Proc. of* CONCUR '92, LNCS 630. Available on Sangiorgi's webpage.

[MS04] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *MSCS*, 14 :715–767, 2004.

[Mur89] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[D$^+$00] Rocco De Nicola, Gian Luigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theor. Comput. Sci.*, 240(1) :215–254, 2000.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1) :85–108, 1981.

[Oxt57] John C. Oxtoby. The Banach-Mazur game and Banach category theorem. In *Contributions to the theory of games, Vol. III*, volume 39 of *Annals of Mathematical Studies*, pages 159–163. Princeton University Press, 1957.

[Oxt71] John C. Oxtoby. *Measure and Category. A Survey of the Analogies between Topological and Measure Spaces*. Springer, 1971.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5) :685–719, 2003.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5 :223–257, 1977.

[Pnu83] Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the 15th STOC*, pages 278–290. ACM, 1983.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of 8th LICS*, pages 376–385, 1993.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[PU07] Iain Phillips and Irek Ulidowski. Reversibility and models for concurrency. *Electr. Notes Theor. Comput. Sci.*, 192(1) :93–108, 2007. Proceedings of SOS 2007.

[PV03] Marco Pistore and Moshe Y. Vardi. The planning spectrum - one, two, three, infinity. In *Proceedings of 18th LICS*, pages 234–243. IEEE Computer Society, 2003.

[RE96] Grzegorz Rozenberg and Joost Engelfriet. Elementary net systems. In *Dagstuhl Lecturs on Petri Nets*, volume 1491 of *LNCS*, pages 12–121. Springer, 1996.

[Rei85] Wolfgang Reisig. *Petri Nets : An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of 29th POPL*, pages 154–165, 2002.

[RT86] Grzegorz Rozenberg and P.S. Thiagarajan. Petri nets : Basic notions, structure, behaviour. In *Current Trends in Concurrency*, volume 224 of *LNCS*, pages 585–668. Springer, 1986.

[San94] Davide Sangiorgi. Locality and true-concurrency in calculi for mobile processes. In *Proceedings of TACS*, volume 789 of *LNCS*, pages 405–424. Springer, 1994.

[San95] Davide Sangiorgi. Internal mobility and agent passing calculi. In *Proc. ICALP'95*, 1995.

[Sch07] Matthias Schmalz. Extensions of an algorithm for generalised fair model checking. Master's thesis, Univeristy of Lübeck, 2007.

[Schdf] Matthias Schmalz. Extensions of an algorithm for generalised fair model checking. Diploma Thesis, Univ. of Lübeck, Germany, 2007, www.infsec.ethz.ch/people/mschmalz/diplomathesis.pdf.

[Sco72] Dana S. Scott. Continuous lattices. In F. William Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, volume 274 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1972.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, M.I.T., 1995.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of 25th ICALP*, volume 1443 of *LNCS*, pages 695–706, 1998.

[SL95] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2) :250–273, 1995. An extended abstract appears in *Proceedings of 5th CONCUR*, Uppsala, Sweden, LNCS 836, pages 481–496, August 1994.

[Smy92] Mike B. Smyth. Topology. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1 : Background : Mathematical Structures, pages 641–761. Oxford University Press, 1992.

[Spi01] Frank Spitzer. *Principles of random walk*. Springer, 2001.

[Sto02] Mariëlle Stoelinga. An introduction to probabilistic automata. *Bulletin of the European Association for Theoretical Computer Science*, 78 :176–198, 2002.

[SVV07] Matthias Schmalz, Hagen Völzer, and Daniele Varacca. Model checking almost all paths can be less expensive than checking all paths. In *Proceedings of 27th FSTTCS*, volume 4855 of *LNCS*, pages 532–543. Springer, 2007.

[SVV09] Matthias Schmalz, Daniele Varacca, and Hagen Völzer. Counterexamples in probabilistic ltl model checking for markov chains. In *Proceedings of 20th CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 587–602. Springer, 2009.

[SW02] Davide Sangiorgi and David Walker. *The π-calculus*. Cambridge University Press, 2002.

[Tel87] Rastislav Telgársky. Topological games : On the 50th anniversary of the Banach-Mazur game. *Rocky Mountain Journal of Mathematics*, 17(2) :227–276, 1987.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, chapter 4, pages 133–192. Elsevier, 1990.

[Tur95] David N. Turner. *The Polymorphic pi-calculus*. PhD thesis, University of Edinburgh, 1995.

[Udd84] J.T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Department of Math. and C.S., Eindhoven University of Technology, Eindhoven, 1984.

[Vol01] Hagen Völzer. Randomized non-sequential processes. In *Proceedings of 12th CONCUR*, volume 2154 of *LNCS*, pages 184–201, 2001. Extended version as Technical Report 02-28 - SVRC - University of Queensland.

[Var03] Daniele Varacca. *Probability, Nondeterminism and Concurrency. Two Denotational Models for Probabilistic Computation*. PhD thesis, BRICS - Aarhus University, 2003. Available at http ://www.brics.dk/DS/03/14.

[Vas94] Vasco Vasconcelos. Typed concurrent objects. In *Proc. ECOOP'94*, 1994.

[VB93] K. Van Berkel. *Handshake Circuits : an Asynchronous Architecture for VLSI Design*, volume 5 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1993.

[vGP09] Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures, event structures and petri nets. *Theor. Comput. Sci.*, 410(41) :4111–4159, 2009.

[Vol02] Hagen Völzer. Refinement-robust fairness. In *Proceedings of 13th CONCUR 2002*, volume 2421 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2002.

[Vol05] Hagen Völzer. On conspiracies and hyperfairness in distributed computing. In *Proceedings of 19th DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2005.

[VV06] Daniele Varacca and Hagen Völzer. Temporal logics and model checking for fairly correct systems. In *Proceedings of 21st LICS*, pages 389–398. IEEE Computer Society, 2006.

[VV12] Hagen Völzer and Daniele Varacca. Defining fairness in reactive and concurrent systems. *JACM*, 59(3) :471–522, 2012.

[VVK05] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. In *Proceedings of 16th CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2005.

[VVW04] Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. In *Proceedings of 15th CONCUR*, volume 3170 of *LNCS*, pages 481–496. Springer, 2004.

[VVW06] Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. *Theor. Comput. Sci.*, 358(2-3) :173–199, 2006.

[VY06] Daniele Varacca and Nobuko Yoshida. Typed event structures and the π-calculus. In *Proceedings of XXII MFPS*, ENTCS, 2006. Full version available at `www.pps.jussieu.fr/~varacca`.

[VY07] Daniele Varacca and Nobuko Yoshida. The Probabilistic π-Calculus and Event Structures. In *Proceedings of QAPL*, ENTCS, 2007.

[VY10] Daniele Varacca and Nobuko Yoshida. Typed event structures and the linear pi-calculus. *Theor. Comput. Sci.*, 411(19) :1949–1973, 2010.

[Wil91] David Williams. *Probability with Martingales*. Cambridge University Press, 1991.

[Win80] Glynn Winskel. *Events in Computation*. Ph.D. thesis, Dept. of Computer Science, University of Edinburgh, 1980.

[Win82] Glynn Winskel. Event structure semantics for CCS and related languages. In *Proceedings of 9th ICALP*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.

[Win87] Glynn Winskel. Event structures. In *Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.

[Win05a] Glynn Winskel. Name generation and linearity. In *Proceedings of 20th LICS*, pages 301–310. IEEE Computer Society, 2005.

[Win05b] Glynn Winskel. Relations in concurrency. In *Proceedings of 20th LICS*, pages 2–11. IEEE Computer Society, 2005.

[WN95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of logic in Computer Science*, volume 4. Clarendon Press, 1995.

[YBH01] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong Normalisation in the π-Calculus. In *Proceedings of LICS'01*, pages 311–322. IEEE, 2001. The full version in *Journal of Inf. & Comp..*, 191 (2004) 145–202, Elsevier.

[YH99] Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order processes. In *Proc. of 10th CONCUR*, LNCS n. 1664, pages 557–572, 1999.

[YHB02] Nobuko Yoshida, Kohei Honda, and Martin Berger. Linearity and bisimulation. In *FoSSaCs02*, volume 2303 of *LNCS*, pages 417–433. Springer, 2002.

[Yos02] Nobuko Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. Technical Report 2002-20, MCS Technical Report, University of Leicester, 2002.

[ZPK02] Lenore D. Zuck, Amir Pnueli, and Yonit Kesten. Automatic verification of probabilistic free choice. In *VMCAI*, volume 2294 of *Lect. Notes in Comp. Sci.*, pages 208–224. Springer, 2002.

Annexe A

Event structure semantics of the linearly typed π-calculus

A.1 Introduction

Models for concurrency can be classified according to different criteria. One possible classification distinguishes between *interleaving* models and *causal* models (also known as *true concurrent* models). In interleaving models, concurrency is reduced to the nondeterministic choice between all possible sequential schedulings of the concurrent actions. Instances of such models are *traces* and *labelled transition systems* [WN95]. Interleaving models are very successful in defining observational equivalences, by means of bisimulation [Mil89]. In causal models, causality and concurrency are explicitly represented. Instances of such models are *Petri nets* [RT86], *Mazurkiewicz traces* [Maz86] and *event structures* [NPW81]. True concurrent models can easily represent interesting behavioural properties such as absence of conflict, independence of the choices and sequentiality [RT86].

In this chapter we address a particular true concurrent model : the model of *event structures* [NPW81, Win87]. Event structures have been used to give semantics to concurrent process languages. The earliest and possibly the most intuitive is Winskel's semantics of Milner's CCS [Win82].

The first contribution of this document is to present a compositional typing system for event structures that ensures two important behavioural properties : *conflict freeness* and *confusion freeness*.

Conflict freeness is the true concurrent version of confluence. In a conflict free system, the only nondeterminism allowed is due to the scheduling of independent components. To illustrate the less familiar notion of confusion freeness, let us suppose that a system is composed of two processes P and Q. Suppose the

system can reach a state where P has a choice between two different actions a_1, a_2, and where Q, independently, can perform action b. We say that such a state is *confused* if the occurrence of b changes the choices available to P (for instance by disabling a_2, or by enabling a third action a_3). Intuitively the choice of process P is not local to that process in that it can be influenced by an independent action. We say that the system is confusion free if none of its reachable states is confused. Intuitively, in a confusion free system, all choices are *localised*. The locations of the choices are called the *cells*.

Confusion freeness was first identified in the context the theory of Petri nets [RT86]. It has been studied in that context, in the form of free choice nets [DE95]. Confusion free event structures are also known as *concrete data structures* [BC82], and their domain-theoretic counterpart are the *concrete domains* [KP93].

Confusion freeness has been recognised as an important property also in the context of probabilistic models. Probabilistic models for concurrency have an extensive literature : most of the studies concern interleaving models [LS91, Seg95, DEP02], but recently, true concurrent ones have also been studied [Kat96, FMW02, AB06, VVW06, Vol01].

In a confusion free system, the intuition is that local choices can be resolved by a probability distribution within each cell. The results in [VVW06] show that by assigning probability distributions to cells one obtains a unique probability measure over the set of maximal configurations.

If confluence entails the property of having only one maximal computation, up to the order of concurrent events, it is then reasonable to define probabilistic confluence as the property of having only one maximal probabilistic computation, where a probabilistic computation is defined as a probability measure over the set of computations. The results in [VVW06] can then be interpreted as saying that probabilistic confusion free systems are *probabilistically confluent*.

The second contribution of this document is to give the first direct event structure semantics of a fragment of the π-calculus [Mil99]. Various causal semantics of the π-calculus existed before [JJ95, BG95, Eng96, BS98, DP99, CS00], but none was given in terms of event structures. The technical difficulty in extending CCS semantics to the π-calculus lies in the handling of α-conversion, which is the main ingredient to represent dynamic creation of names. We are able to solve this problem for a restricted version of the π-calculus, a linearly typed version of Sangiorgi's πI-calculus (more precisely, the extension of the calculus in [BHY01] to the nondeterministic one). This fragment is expressive enough to encode the typed λ-calculus (in fact, to encode it *fully abstractly* [BHY01, YBH01]). We argue that in this fragment, α-conversion need not be performed dynamically (at "run time"), but can be done during the typing (at "compile time"), by choosing in advance all the names that will be created during the computation. This is possible because the typing system guarantees that, in a sense, every process knows in advance which processes it will communicate with.

To substantiate this intuition, we provide a fully abstract encoding of the linearly typed fragment of the π-calculus into an intermediate process language,

which is syntactically similar to the π-calculus except that α-conversion is not allowed. We devise a typing system for this language that makes use of the event structure types. We then provide the language with a semantics in terms of typed event structures. Via this fully abstract intermediate translation, we thus obtain a sound event structure semantics of the π-calculus, which follows the same lines as Winskel's : syntactic nondeterministic choice is modelled by *conflict*, prefix is modelled using *causality*, and parallel composition generates *concurrent* events. Moreover, since our semantics is given in terms of typed event structures, we obtain that all processes of this fragment are confusion free. Our typing system generalises an early idea by Milner, who devised a syntactic restriction of CCS (a kind of a typing system) that guarantees confluence of the interleaving semantics [Mil89]. As a corollary of our work we show that a similar restriction applied to the π-calculus guarantees the property of conflict freeness.

We then extend the above setting to a probabilistic variant of the π-calculus We introduce a probabilistic variant of the π-calculus, similar to the ones presented in [HP00, CP05], which it is obtained by adding a probabilistic primitive to the linearly typed fragment.

We then provide an interleaving and a true concurrent semantics to this probabilistic extension of the lineraly typed π-calculus. The interleaving semantics is given, as in [HP00, CP05], as Segala automata [Seg95], which are an operational model that combine probability and nondeterminism. The nondeterminism is necessary to account for the different possible schedulings of the independent parts of a system. The true concurrent semantics is given as probabilistic event structures. In this model, we do not have to account for the different schedulings, and that leads to the probabilistic confluence result (Theorem A.10.2).

In order to relate the two semantics, we show how a probabilistic event structure generates a Segala automaton. This allows us to show an operational correspondence between the two semantics.

Structure of the chapter This chapter is a fusion of the two papers [VY10, VY07], with slight modifications. As in [VY10], we provide the full definition of the intermediate language which was omitted from the conference verision [VY06].

Section A.2 presents a linearly typed version of the πI-calculus. This section is inspired from [YBH01], but our fragment is extended to allow nondeterministic choice. Section A.3 introduces the basic definitions of event structures and defines formally the notion of confusion freeness. We briefly introduce the category of event structures and we explicitly describe the categorical product. The product of event structures is one of the basic ingredients in the definition of the parallel composition. The explicit definition we present allows us to carry out the proofs in the following sections. Section A.4 presents our new typing system and an event structure semantics of the types. We then define a notion of typing of event structures by means of the morphisms of the category of

event structures. Typed event structures are confusion free by definition. The main theorem of this section is that the parallel composition of typed event structures is again typed, and thus confusion free. In Section A.5, we present the intermediate process language which is used to bridge between the typed event structures and the linear π-calculus. We call this calculus *Name Sharing CCS* or NCCS. We define a notion of typing for NCCS processes and its typed operational semantics. In Section A.6, we give a semantics of typed NCCS processes in terms of event structures. The main result of this section is that the semantics of a typed process is a typed event structure. We also show that this semantics is sound with respect to bisimulation. In Section A.7, we provide a fully abstract translation of the the typed πI-calculus, into NCCS. Through the sound event structure semantics of NCCS, we obtain a sound semantics of the π-calculus in terms of event structures. The main result of the section is that the semantic of a πI-calculus term is a typed event structure, and thus it is confusion free. In Section A.8, we present the probabilistic extension of the π-calculus, obtained by adding a probability distribution on outputs. We give its operational semantics in temrs of Segala automata. In Section A.9, we recall the probabilistic event structures of [VVW06], with some minor modifications, and we use them to give a semantics to the calculus. In Section A.10, we show the correspondence between the event structures semantics and the Segala automata semantics, and we state the probabilistic confluence result. Section A.11 presents some related and future works. Finally, in Section A.12 we detail the proofs of the theorems and propositions.

A.2 A linear version of the π-calculus

This section briefly summarises an extension of linear version of the π-calculus in [BHY01] to non-determinism [Yos02]. The reader may refer to [BHY01, Yos02] for a more detailed description and more examples.

A.2.1 Syntax and reduction

As anticipated, we consider a restricted version of the π-calculus [Mil99], where only bound names are passed in interaction. The resulting calculus is called the πI-calculus in the literature [San95]. Syntactically we restrict all outputs to be of the form $(\boldsymbol{\nu}\, \tilde{y})\overline{x}\langle\tilde{y}\rangle.P$ (where $\tilde{y}$ represents a tuple of pairwise distinct names), which we henceforth write $\overline{x}(\tilde{y}).P$. We consider a version of the calculus more general than the one presented in [BHY01], in that both input and output are nondeterministic. Nondeterministic input is called *branching*, and it is already present in [BHY01], while nondeterministic output, called *selection*, is a novelty of this work. Branching is similar to the "case" construct and selection is "injection" in the typed λ-calculi; these constructs have been studied in other typed π-calculi [Vas94]. The formal grammar of the calculus is defined below.

$$\begin{array}{rcl} P & ::= & x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i \;\mid\; \overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y_i}).P_i \\ & \mid & P \,|\, Q \;\mid\; (\boldsymbol{\nu}\, x)P \;\mid\; \mathbf{0} \;\mid\; !x(\tilde{y}).P \end{array}$$

The process $x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i$ (resp. $\overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y_i}).P_i$) is a branching input (resp. selecting output), where I denotes a finite or countably infinite indexing set. The names in $\tilde{y_i}$ are bound in the continuation P_i. The process $!x(\tilde{y}).P$ is a replicated input, binding $\tilde{y}$. $P \,|\, Q$ is a parallel composition and $(\boldsymbol{\nu}\, x)P$ is a restriction that binds x. We omit the empty tuple : for example, $\overline{x}$ stands for $\overline{x}()$. When the index in the branching indexing set is a singleton we use the notation $x(\tilde{y}).P$ when it is binary, we write $x((\tilde{y_1}).P_1 \& (\tilde{y_2}).P_2)$ (and similarly for selection). Notions of bound/free names, $\alpha-$ and structural equivalences, and of evaluation contexts are defined as usual [Mil99, BHY01, YBH01, HY95].

Processes where all selection indexing sets are singletons are called *deterministic*. Deterministic processes where also branching indexing sets are singletons are called *simple*.

The reduction semantics is as follows :

$$x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i \mid \overline{x} \bigoplus_{j\in J} \mathtt{in}_j(\tilde{y_j}).Q_j \;\longrightarrow\; (\boldsymbol{\nu}\, \tilde{y_h})(P_h \,|\, Q_h) \quad (h \in I \cap J)$$

$$!x(\tilde{y}).P \mid \overline{x}(\tilde{y}).Q \;\longrightarrow !x(\tilde{y}).P \mid (\boldsymbol{\nu}\, \tilde{y})(P \,|\, Q)$$

closed under evaluation contexts and structural equivalence. A nondeterministic branching synchronises with a selection on one of the common branches, the communicated names are restricted, and the continuations triggered. An output can also synchronise with a replicated server which is still present after the reduction. Note that α-conversion may be necessary for two processes to synchronise. For instance, consider the process $x(y).P \mid \overline{x}(z).Q$. Assuming y is fresh for Q, it can be α-converted to $x(y).P \mid \overline{x}(y).Q[y/z]$, allowing synchronisation.

A.2.2 Types and typings

The linear type discipline restricts the behaviour of processes as follows.

(A) for each linear name there are a unique input and a unique output ; and

(B) for each replicated name there is a unique replicated input with zero or more dual outputs.

In the context of deterministic processes, the typing system guarantees confluence. We will see that in the presence of nondeterminism this typing system guarantees confusion freeness.

As an example for the first condition, let us consider :

$$Q_1 \stackrel{\text{def}}{=} \overline{x}.y \,|\, \overline{x}.z \,|\, x \qquad\qquad Q_2 \stackrel{\text{def}}{=} y.\overline{x} \,|\, z.\overline{y} \,|\, x.(\overline{z} \,|\, \overline{w})$$

Then Q_1 is not typable as x appears twice as output, while Q_2 is typable since each channel appears at most once as input and output. Typability of simple

processes such as Q_2 offers only deterministic behaviour. However branching and selection can provide non-deterministic behaviour, preserving linearity :

$$Q_3 \stackrel{\text{def}}{=} \overline{x}.(y \oplus z) \,|\, x.(\overline{w} \,\&\, \overline{v})$$

Q_3 is typable, and we have either $Q_3 \longrightarrow (y\,|\,\overline{w})$ or $Q_3 \longrightarrow (z\,|\,\overline{v})$. As an example of the second constraint, let us consider the following two processes :

$$Q_4 \stackrel{\text{def}}{=} !\,y.\overline{x} \,|\, !\,y.\overline{z} \qquad Q_5 \stackrel{\text{def}}{=} !\,y.\overline{x} \,|\, \overline{y} \,|\, !\,z.\overline{y}$$

Q_4 is untypable because y is associated with two replicators : but Q_5 is typable since, while output at y appears twice, a replicated input at y appears only once.

Channel types are inductively made up from type variables and action modes : the two *input modes* $\downarrow, \mathbf{!}$, and the two *output modes* $\uparrow, \mathbf{?}$. We let $p, p', \ldots$ denote modes. We define $\overline{p}$, the *dual* of p, by : $\overline{\downarrow} = \uparrow$, $\overline{\mathbf{!}} = \mathbf{?}$ and $\overline{\overline{p}} = p$. Then the syntax of types is given as follows :

$$\begin{array}{lcccccccc} \sigma & ::= & \&_{i\in I}\,(\tilde{\sigma}_i)^{\downarrow} & | & \bigoplus_{i\in I}\,(\tilde{\sigma}_i)^{\uparrow} & | & (\tilde{\sigma})^{\mathbf{!}} & | & (\tilde{\sigma})^{\mathbf{?}} \\ & & \text{(branching)} & & \text{(selection)} & & \text{(offer)} & & \text{(request)} \\ \tau & ::= & \sigma & | & \updownarrow & & \text{(closed type)} & & \end{array}$$

where $\tilde{\sigma}$ is a tuple of types. We write $MD(\tau)$ for the outermost mode of τ. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes, with $\updownarrow$ being self-dual. A type environment Γ is a finite mapping from channels to channel types. Sometimes we will write $x \in \Gamma$ to mean $x \in Dom(\Gamma)$.

Types restrict the composability of processes : if P is typed under environment Γ_1, Q is typed under Γ_2 and if Γ_1, Γ_2 are "compatible", then a new environment $\Gamma_1 \odot \Gamma_2$ is defined, such that $P \,|\, Q$ is typed under $\Gamma_1 \odot \Gamma_2$. If the environments are not compatible, $\Gamma_1 \odot \Gamma_2$ is not defined and the parallel composition cannot be typed. Formally, we introduce a partial commutative operation $\odot$ on types, defined as follows :

$$\begin{array}{lll} (1) & \tau \odot \overline{\tau} = \updownarrow & \text{with } MD(\tau) = \downarrow \\ (2) & \tau \odot \overline{\tau} = \overline{\tau}, \quad \tau \odot \tau = \tau & \text{with } MD(\tau) = ? \end{array}$$

Then, the environment $\Gamma_1 \odot \Gamma_2$ is defined homomorphically. Intuitively, the rules in (2) say that a server should be unique, but an arbitrary number of clients can request interactions. The rules in (1) say that once we compose input-output linear channels, the channel becomes uncomposable. Other compositions are undefined. The definitions (1) and (2) ensure the two constraints (A) and (B).

The rules defining typing judgments $P \triangleright \Gamma$ are defined in Figure A.1. They are identical to the affine π-calculus [BHY01] except a straightforward modification to deal with the non-deterministic output. We refer to [BHY01] for an informal discussion on the meaning of the rules. We just note here that, in the rule (Par) the use of $\Gamma_1 \odot \Gamma_2$ guarantees the consistent channel usage. For instance it guarantees that linear inputs are only composed with linear outputs.

$$\frac{P \triangleright \Gamma, x:\tau \quad x \notin \Gamma \quad MD(\tau) = \mathbf{!}, \updownarrow}{(\boldsymbol{\nu}\, x)P \triangleright \Gamma}\ \text{Res} \qquad \frac{}{\mathbf{0} \triangleright \varnothing}\ \text{Zero} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x:\updownarrow}\ \text{WeakCl}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma \quad I \subseteq J}{\overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \bigoplus_{i\in J}(\tilde{\tau}_i)^{\uparrow}}\ \text{LOut} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}}\ \text{WeakOut}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma}{x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I}(\tilde{\tau}_i)^{\downarrow}}\ \text{LIn} \qquad \frac{P_i \triangleright \Gamma_i \quad (i = 1,2)}{P_1 \mid P_2 \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad x \notin \Gamma \quad \forall (z:\tau) \in \Gamma.\ MD(\tau) = ?}{!x(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{!}}}\ \text{RIn} \qquad \frac{P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}, \tilde{y} : \tilde{\tau}}{\overline{x}(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}}\ \text{ROut}$$

FIGURE A.1 – Linear Typing Rules

A.2.3 A typed labelled transition relation

Typed transitions describe the observations a typed observer can make of a typed process. The typed transition relation is a proper subset of the untyped transition relation, while not restricting $\boldsymbol{\tau}$-actions : hence typed transitions restrict observability, not computation.

Labels are generated by the following grammar :

$$\begin{array}{rcccccccc} \alpha, \beta & ::= & x\mathtt{in}_i\langle\tilde{y}\rangle & | & \overline{x}\mathtt{in}_i\langle\tilde{y}\rangle & | & x\langle\tilde{y}\rangle & | & \overline{x}\langle\tilde{y}\rangle \\ & & \text{(branching)} & & \text{(selection)} & & \text{(offer)} & & \text{(request)} \\ \boldsymbol{\tau} & ::= & (x,\overline{x})\mathtt{in}_i\langle\tilde{y}\rangle & | & (x,\overline{x})\langle\tilde{y}\rangle & & \text{(synchronisation)} & & \end{array}$$

With the notation above, we say that x is the *subject* of the label β, denoted as $subj(\beta)$, while $\tilde{y} = y_1, \ldots, y_n$ are the *object* names, denoted as $obj(\beta)$. For branching/selection labels, the index i is the *branch* of the label. The notation "$\mathtt{in}_i$" comes from the injection of the typed λ-calculus. The partial operation $\alpha \bullet \beta$ is defined as follows : $x\mathtt{in}_i\langle\tilde{y}_i\rangle \bullet \overline{x}\mathtt{in}_i\langle\tilde{y}_i\rangle = (x,\overline{x})\mathtt{in}_i\langle\tilde{y}_i\rangle$, $x\langle\tilde{y}\rangle \bullet \overline{x}\langle\tilde{y}\rangle = (x,\overline{x})\langle\tilde{y}\rangle$, and undefined otherwise. It is convenient, for the proofs, to use synchronisation labels that keep track of which synchronisation took place. However, as it is customary, we consider synchronisation transitions not to be observable. Thus for the purpose of defining observational equivalences, all $\boldsymbol{\tau}$-labels will be identified.

The standard untyped transition relation is defined in Figure A.2. We define the predicate "Γ allows β" which represents how an environment restricts observability :

- for all Γ, Γ allows $\boldsymbol{\tau}$;
- if $MD(\Gamma(x)) = \downarrow$, then Γ allows $x\mathtt{in}_i\langle\tilde{y}\rangle$;
- if $MD(\Gamma(x)) = \uparrow$, then Γ allows $\overline{x}\mathtt{in}_i\langle\tilde{y}\rangle$;
- if $MD(\Gamma(x)) = \mathbf{!}$, then Γ allows $x\langle\tilde{y}\rangle$;
- if $MD(\Gamma(x)) = \mathbf{?}$, then Γ allows $\overline{x}\langle\tilde{y}\rangle$.

$$\overline{x}\bigoplus_{i\in I}(\tilde{y}_i).P_i \xrightarrow{\overline{x}\mathtt{in}_j\langle\tilde{y}_j\rangle} P_j \quad x\,\&_{i\in I}(\tilde{y}_i).P_i \xrightarrow{x\mathtt{in}_j\langle\tilde{y}_j\rangle} P_j$$

$$!x(\tilde{y}).P \xrightarrow{x\langle\tilde{y}\rangle} P \mid !x(\tilde{y}).P \quad \overline{x}(\tilde{y}).P \xrightarrow{\overline{x}\langle\tilde{y}\rangle} P$$

$$\frac{P \xrightarrow{\beta} P'}{P \mid Q \xrightarrow{\beta} P' \mid Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q' \quad obj(\alpha) = \tilde{y}}{P \mid Q \xrightarrow{\alpha\bullet\beta} (\boldsymbol{\nu}\,\tilde{y})(P' \mid Q')}$$

$$\frac{P \xrightarrow{\beta} P' \quad subj(\beta) \neq x}{(\boldsymbol{\nu}\,x)P \xrightarrow{\beta} (\boldsymbol{\nu}\,x)P'} \qquad \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\beta} Q}{P \xrightarrow{\beta} Q}$$

FIGURE A.2 – Labelled Transition System for the πI-Calculus

Whenever Γ allows β, we define a new environment $\Gamma \setminus \beta$ as follows :
- for all Γ, $\Gamma \setminus \boldsymbol{\tau} = \Gamma$;
- if $\Gamma = \Delta, x : \&_{i\in I}\,(\tilde{\tau}_i)^{\downarrow}$, then $\Gamma \setminus x\mathtt{in}_i\langle\tilde{y}\rangle = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : \bigoplus_{i\in I}(\tilde{\tau}_i)^{\uparrow}$, then $\Gamma \setminus \overline{x}\mathtt{in}_i\langle\tilde{y}\rangle = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\tilde{\tau})^{!}$, then $\Gamma \setminus x\langle\tilde{y}\rangle = \Gamma, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\tilde{\tau})^{?}$, then $\Gamma \setminus \overline{x}\langle\tilde{y}\rangle = \Gamma, \tilde{y} : \tilde{\tau}$.

The environment $\Gamma \setminus \beta$ represents what remains of Γ after the transition labelled by β has happened. Linear channels are consumed, while replicated channels are not consumed. The new previously bound channels are released.

The typed transition, written $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma'$ is defined by

$$\text{if } P \xrightarrow{\beta} Q \text{ and } \Gamma \text{ allows } \beta \quad \text{then} \quad P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta$$

The above rule does not allow a linear input action and an output action when there is a complementary channel in the process. For example, if a process has $x : (\tilde{\tau})^{!}$ in its action type, then output at x is excluded since such actions can never be observed in a typed context – cf. [BHY01]. For a concrete example, consider the process $\overline{x}.y \mid \overline{y}.x$ which is typed in the environment $x : \updownarrow, y : \updownarrow$. Although the process has some untyped transitions, none of them is allowed by the environment.

By induction on the rules in Figure A.2, we can obtain :

Proposition A.2.1.

- *If $P \triangleright \Gamma$, $P \xrightarrow{\beta} Q$ and Γ allows β, then $Q \triangleright \Gamma \setminus \beta$.*
- (Subject reduction) *If $P \triangleright \Gamma$ and $P \xrightarrow{\boldsymbol{\tau}} Q$, then $Q \triangleright \Gamma$.*

– (Church Rosser for deterministic processes) *Suppose* $P \triangleright \Gamma$ *and* P *is deterministic. Assume* $P \xrightarrow{\tau} Q_1$, *and* $P \xrightarrow{\tau} Q_2$. *Then* $Q_1 \equiv_\alpha Q_2$ *or there exists* R *such that* $Q_1 \xrightarrow{\tau} R$ *and* $Q_2 \xrightarrow{\tau} R$.

Finally we define the notion of typed bisimulation. Let $\mathcal{R}$ be a symmetric relation between judgments such that if $(P \triangleright \Gamma)\,\mathcal{R}\,(P' \triangleright \Gamma')$, then $\Gamma = \Gamma'$. We say that $\mathcal{R}$ is a bisimulation if the following is satisfied :

- whenever $(P \triangleright \Gamma)\,\mathcal{R}\,(P' \triangleright \Gamma)$, $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta$, then there exists Q' such that $P' \triangleright \Gamma \xrightarrow{\beta} Q' \triangleright \Gamma \setminus \beta$, and $(Q \triangleright \Gamma \setminus \beta)\,\mathcal{R}\,(Q' \triangleright \Gamma \setminus \beta)$.

As anticipated, in the above definition we allow a $\boldsymbol{\tau}$ label to be matched by a different $\boldsymbol{\tau}$ label. The identities of different $\boldsymbol{\tau}$ labels are considered only in some of the proofs.

If there exists a bisimulation between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma)$. It can be proved that $\approx$ is a congruent relation. The proof is analogous to the one in Appendix C.3 of [YBH01].

A.3 Event structures

Event structures were introduced by Nielsen, Plotkin and Winskel [NPW81, Win80], and have been subject of several studies since. They appear in different forms. The one we introduce in this work is sometimes referred to as *prime event structures* [Win87]. For the relations of event structures with other models for concurrency, the standard reference is [WN95].

A.3.1 Basic definitions

An *event structure* is a triple $\mathcal{E} = \langle E, \leq, \smile \rangle$ such that

- E is a countable set of *events* ;
- $\langle E, \leq \rangle$ is a partial order, called the *causal order* ;
- for every $e \in E$, the set $[e) := \{e' \mid e' < e\}$, called the *enabling set* of e, is finite ;
- $\smile$ is an irreflexive and symmetric relation, called the *conflict relation*, satisfying the following : for every $e_1, e_2, e_3 \in E$ if $e_1 \leq e_2$ and $e_1 \smile e_3$ then $e_2 \smile e_3$.

The reflexive closure of conflict is denoted by $\asymp$. We say that the conflict $e_2 \smile e_3$ is *inherited* from the conflict $e_1 \smile e_3$, when $e_1 < e_2$. If a conflict $e_1 \smile e_2$ is not inherited from any other conflict we say that it is *immediate*, denoted by $e_1 \smile_\mu e_2$. The reflexive closure of immediate conflict is denoted by $\asymp_\mu$. If two events are not causally related nor in conflict they are said to be *concurrent*. The set of maximal elements of $[e)$ is denoted by $parents(e)$. A *configuration* x of an event structure $\mathcal{E}$ is a conflict free downward closed subset of E, i.e. a subset x of E satisfying : (1) if $e \in x$ then $[e) \subseteq x$ and (2) for every $e, e' \in x$, it is not the case that $e \smile e'$. Therefore, two events of a configuration are either causally dependent or concurrent, i.e., a configuration represents a run of an event structure where events are partially ordered. The set of configurations of $\mathcal{E}$, partially ordered by inclusion, is denoted as $\mathcal{L}(\mathcal{E})$. It is a coherent ω-algebraic domain [NPW81], whose compact elements are the finite configurations.

A *labelled event structure* is an event structure $\mathcal{E}$ together with a labelling function $\lambda : E \to L$, where L is a set of labels. Events should be thought of

as occurrences of actions. Labels allow us to identify events which represent different occurrences of the same action. Labels are also essential in defining the parallel composition, and play a major role in the typed setting. A labelled event structure generates a labelled transition system as follows.

Definition A.3.1. Let $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$ be a labelled event structure and let e be one of its minimal events. The event structure $\mathcal{E}\lfloor e = \langle E', \leq', \smile', \lambda' \rangle$ is defined by : $E' = \{e' \in E \mid e' \not\asymp e\}$, $\leq' = \leq_{|E'}$, $\smile' = \smile_{|E'}$, and $\lambda' = \lambda_{E'}$.

Roughly speaking, $\mathcal{E}\lfloor e$ is $\mathcal{E}$ minus the event e, and minus all events that are in conflict with e. We can then generate a labelled transition system on event structures as follows : if $\lambda(e) = \beta$, then

$$\mathcal{E} \xrightarrow{\beta} \mathcal{E}\lfloor e \ .$$

The reachable transition system with initial state $\mathcal{E}$ is denoted as $TS(\mathcal{E})$.

A.3.2 Conflict free and confusion free event structures

An interesting subclass of event structures is the following.

Definition A.3.2. An event structure is *conflict free* if its conflict relation is empty.

Conflict freeness is the true concurrent version of confluence. Indeed it is easy to verify that if $\mathcal{E}$ is conflict free, then $TS(\mathcal{E})$ is confluent.

As informally explained, in a confusion free event structure every conflict is *localised*. To specify what "local" means in this context, we need the notion of *cell*, a set of pairwise conflicting events with the same causal predecessors.

Definition A.3.3. A *partial cell* is a set c of events such that $e, e' \in c$ implies $e \asymp_\mu e'$ and $[e) = [e')$. A maximal partial cell is called a *cell*.

In general, two events in immediate conflicts need not belong to the same cell. If a cell is thought of as a location, this means that not all conflicts are localised. This leads us to the following definition.

Definition A.3.4. An event structure is *confusion free* if its cells are closed under immediate conflict.

Equivalently, in a confusion free event structure reflexive immediate conflict is an equivalence relation with cells as its equivalence classes [VVW04].

A.3.3 A category of event structures

Event structures form the class of objects of a category [WN95]. The morphisms are defined as follows. Let $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2 \rangle$ be two event structures. A *morphism* $f : \mathcal{E}_1 \to \mathcal{E}_2$ is a partial function $f : E_1 \to E_2$ such that

- f preserves configurations : if x is a configuration of $\mathcal{E}_1$, then $f(x)$ is a configuration of $\mathcal{E}_2$;
- f is locally injective : let x be a configuration of $\mathcal{E}_1$, if $e, e' \in x$ and $f(e), f(e')$ are both defined with $f(e) = f(e')$, then $e = e'$.

It is straightforward to verify that the identity is a morphism and that morphisms compose, so that what we obtain is indeed a category.

Morphisms reflect conflict and causality and preserve concurrency. They can be equivalently characterised as follows.

Proposition A.3.5 ([WN95])**.** *A partial function $f : E_1 \to E_2$ is a morphism of event structures $f : \mathcal{E}_1 \to \mathcal{E}_2$ if and only if the following are satisfied :*

- *f reflects causality : if $f(e_1)$ is defined, then $[f(e_1)] \subseteq f([e_1])$;*
- *f reflect reflexive conflict : if $f(e_1), f(e_2)$ are defined, and if $f(e_1) \asymp f(e_2)$, then $e_1 \asymp e_2$.*

There are various ways of dealing with labels. For the general treatment we refer to [WN95]. Here we present the simplest notion : take two labelled event structures $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1, \lambda_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2, \lambda_2 \rangle$ on the same set of labels L. A morphism $f : \mathcal{E}_1 \to \mathcal{E}_2$ is said to be *label preserving* if, whenever $f(e_1)$ is defined, $\lambda_2(f(e_1)) = \lambda_1(e_1)$.

A.3.4 Operators on event structures

We can define several operations on labelled event structures.

- Prefixing $\alpha.\mathcal{E}$, where $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$. It is the event structure $\langle E', \leq' , \smile', \lambda' \rangle$, where $E' = E \uplus \{e'\}$ for some new event e', $\leq'$ coincides with $\leq$ on E and moreover, for every $e \in E$ we have $e' \leq e$, the conflict relation $\smile'$ coincides with $\smile$, that is e' is in conflict with no event. Finally λ' coincides with λ on E and $\lambda'(e') = \alpha$. Intuitively, we add a new initial event, labelled by α.
- Prefixed sum $\sum_{i \in I} \alpha_i.\mathcal{E}_i$. This is obtained by disjoint union of copies of the event structures $\alpha_i.\mathcal{E}_i$, where the order relation is the disjoint union of the orders, the labelling function is the disjoint union of the labelling functions, and the conflict is the disjoint union of the conflicts extended by putting in conflict every two events in two different copies. This is a generalisation of prefixing, where we add an initial *cell*, instead of an initial event.
- Restriction $\mathcal{E} \setminus X$ where $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$ and $X \subseteq L$ is a set of labels. This is obtained by removing from E all events with label in X and all events that are above one of those. On the remaining events, order, conflict and labelling are unchanged.
- Relabelling $\mathcal{E}[f]$. This is just composing the labelling function λ with a function $f : L \to L$. The new event structure has thus labelling function $f \circ \lambda$.

It is easy to verify that all these constructions preserve the class of confusion free event structures. Also, with the obvious exception of the prefixed sum, they preserve the class of conflict free event structures

A.3.5 The parallel composition

The parallel composition of event structures is defined in [WN95] as the categorical product followed by restriction and relabelling. The existence of the product is deduced via general categorical arguments, but not explicitly constructed.

In order to carry out our proofs, we needed a more concrete representation of the product. We have devised such a representation, which is inspired by the one given in [DDM88], but which is more suitable to an inductive reasoning.

Let $\mathcal{E}_1 := \langle E_1, \leq_1, \smile_1 \rangle$ and $\mathcal{E}_2 := \langle E_2, \leq_2, \smile_2 \rangle$ be two event structures. Let $E_i^* := E_i \uplus \{*\}$. Consider the set $\tilde{E}$ obtained as the initial solution of the equation $X = \mathcal{P}_{fin}(X) \times E_1^* \times E_2^*$. Its elements have the form (x, e_1, e_2) for x finite, $x \subseteq \tilde{E}$. Initiality allows us to define inductively a notion of *height* of an element of $\tilde{E}$ as

$$h(\varnothing, e_1, e_2) = 0 \quad \text{and} \quad h(x, e_1, e_2) = \max\{h(e) \mid e \in x\} + 1$$

Most of our reasoning will be by induction on the height of the elements. We now carve out of $\tilde{E}$ a set E which will be the support of our product event structure $\mathcal{E}$. At the same time we define the order relation and the conflict relation on $\mathcal{E}$.

Base : we have that $(\varnothing, e_1, e_2) \in E$ if

- $e_1 \in E_1, e_2 \in E_2$, and e_1 minimal in E_1, e_2 minimal in E_2 or
- $e_1 \in E_1, e_2 = *$ and e_1 minimal in E_1 or
- $e_1 = *, e_2 \in E_2$ and e_2 minimal in E_2.

The order on the elements of height 0 is trivial.

Finally we have $(\varnothing, e_1, e_2) \asymp (\varnothing, d_1, d_2)$ if $e_1 \asymp d_1$ or $e_2 \asymp d_2$.

Inductive Case : assume that all elements in E of height $\leq n$ have been defined. Assume that an order relation and a conflict relation has been defined on them. Let (x, e_1, e_2) be of height $n+1$. Let y be the set of maximal elements of x. Let $y_1 = \{d_1 \in E_1 \mid (z, d_1, d_2) \in y\}$ and $y_2 = \{d_2 \in E_2 \mid (z, d_1, d_2) \in y\}$, be the projections of y onto the two components. We have that $(x, e_1, e_2) \in E$ if x is downward closed and conflict free, and furthermore :

- Suppose $e_1 \in E_1, e_2 = *$. Then it must be the case that $y_1 = \mathit{parents}(e_1)$.
- Suppose $e_2 \in E_2, e_1 = *$. Then it must be the case that $y_2 = \mathit{parents}(e_2)$.
- Suppose $e_1 \in E_1, e_2 \in E_2$. Then
  - if $(z, d_1, d_2) \in y$, then either $d_1 \in \mathit{parents}(e_1)$ or $d_2 \in \mathit{parents}(e_2)$;
  - for all $d_1 \in \mathit{parents}(e_1)$, there exists $(z, d_1, d_2) \in x$;
  - for all $d_2 \in \mathit{parents}(e_2)$ there exists $(z, d_1, d_2) \in x$.
- Let $x_1 = \{d_1 \in E_1 \mid (z, d_1, d_2) \in x\}$ and $x_2 = \{d_2 \in E_2 \mid (z, d_1, d_2) \in x\}$. Then for no $d_1 \in x_1$, $d_1 \asymp e_1$ and for no $d_2 \in x_2$, $d_2 \asymp e_2$.

The partial order is extended by $e \leq (x, e_1, e_2)$ if $e \in x$, or $e = (x, e_1, e_2)$. Note that if $e < e'$ then $h(e) < h(e')$.

For the conflict, let $e = (x, e_1, e_2)$ and $d = (z, d_1, d_2)$, with either $h(e) = n+1$ or $h(d) = n+1$ or both. Then we define $e \smile d$ if one of the following holds :

- $e_1 \asymp d_1$ or $e_2 \asymp d_2$, and $e \neq d$;
- there exists $e' = (x', e_1', e_2') \in x$ such that $e_1' \asymp d_1$ or $e_2' \asymp d_2$, and $e' \neq d$;
- there exists $d' = (z', d_1', d_2') \in z$ such that $e_1 \asymp d_1'$ or $e_2 \asymp d_2'$, and $e \neq d'$;
- there exists $e \in x, d \in z$ such that $e \smile d$.

As the following lemma shows, some of the clauses above are redundant, but are kept for simplicity.

Lemma A.3.6 (Stability)**.** *If $(x, e_1, e_2), (x', e_1, e_2) \in E$ and $x \neq x'$, then there exist $d \in x, d' \in x'$ such that $d \smile d'$.*

Now we are ready to state the main new result of this section : take two event structures $\mathcal{E}_1, \mathcal{E}_2$, and let $\mathcal{E} = \langle E, \leq, \smile \rangle$ be defined as above. Then we have :

Theorem A.3.7. *The structure $\mathcal{E}$ is an event structure and it is the categorical product of $\mathcal{E}_1, \mathcal{E}_2$.*

We will not make explicit use of the properties of the categorical product, except that projections preserve configurations. However Theorem A.3.7 is necessary to fit in the general framework of models for concurrency, and to avoid building "ad hoc" models.

For event structures with labels in L, the labelling function of the product takes on the set $L_* \times L_*$, where $L_* := L \uplus \{*\}$. We define $\lambda(x, e_1, e_2) = (\lambda_1^*(e_1), \lambda_2^*(e_2))$, where $\lambda_i^*(e_i) = \lambda_i(e_i)$ if $e_i \neq *$, and $\lambda_i^*(*) = *$. A *synchronisation algebra* S is given by a partial binary operation $\bullet_S$ defined on L_* [WN95]. Given two labelled event structures $\mathcal{E}_1, \mathcal{E}_2$, the parallel composition $\mathcal{E}_1 \|_S \mathcal{E}_2$ is defined as the categorical product followed by restriction and relabelling : $(\mathcal{E}_1 \times \mathcal{E}_2 \setminus X)[f]$ where X is the set of pairs $(\alpha_1, \alpha_2) \in L_* \times L_*$ for which $\alpha_1 \bullet_S \alpha_2$ is undefined, while the function f : is defined as $f(\alpha_1, \alpha_2) = \alpha_1 \bullet_S \alpha_2$. The subscripts S are omitted when the synchronisation algebra is clear from the context.

The simplest possible synchronisation algebra is defined as $\alpha \bullet * = * \bullet \alpha = \alpha$, and undefined in all other cases. In this particular case, the induced parallel composition can be represented as the disjoint union of the sets of events, of the causal orders, and of the conflict. This can be also generalised to an arbitrary family of event structures $(\mathcal{E}_i)_{i \in I}$. In such a case we denote the parallel composition as $\prod_{i \in I} \mathcal{E}_i$.

Parallel composition does not preserve in general the classes of conflict free and confusion free event structures. New conflicts can be created through synchronisation. One of the main reasons to devise a typing system for event structures is to guarantee the preservation of these two important behavioural properties.

A.3.6 Examples of event structures

We collect in this section a series of examples, with graphical representation.

Example A.3.1. Consider the following event structures $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$, defined on the same set of events $E := \{a, b, c, d, e\}$. In $\mathcal{E}_1$, we have $a \leq b, c, d, e$ and $b \smile_\mu c$, $c \smile_\mu d$, $b \smile_\mu d$. In $\mathcal{E}_2$, we do not have $a \leq d$, while in $\mathcal{E}_3$, we do not have $b \smile_\mu d$. The three event structures are represented in Figure A.3,

where curly lines represent immediate conflict, while the causal order proceeds upwards along the straight lines.

The event structure $\mathcal{E}_1$ is confusion free, with three cells : $\{a\}, \{b, c, d\}, \{e\}$. In $\mathcal{E}_2$, there are four cells : $\{a\}, \{b, c\}, \{d\}, \{e\}$. $\mathcal{E}_2$ is not confusion free, because some cells are not closed under immediate conflict. This is an example of *asymmetric* confusion [RE96]. In $\mathcal{E}_3$ there are four cells : $\{a\}, \{b, c\}, \{c, d\}, \{e\}$. $\mathcal{E}_3$ is not confusion free, because immediate conflict is not transitive. This is an example of *symmetric* confusion.

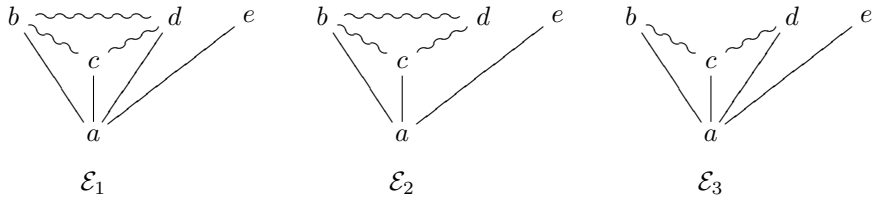


FIGURE A.3 – Event structures

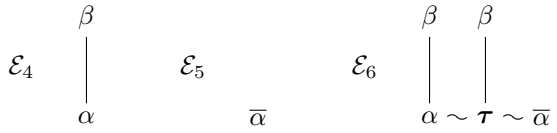


FIGURE A.4 – Parallel composition of event structures

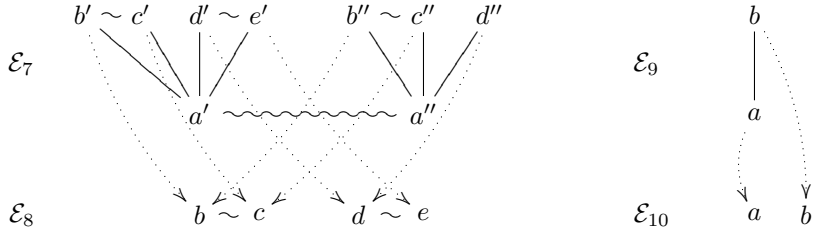


FIGURE A.5 – Morphisms of event structures

Example A.3.2. Next we show an example of parallel composition, see Figure A.4. Consider the two labelled event structures $\mathcal{E}_4, \mathcal{E}_5$, where $E_4 = \{a, b\}, E_5 = \{a'\}$, conflict and order being trivial, and $\lambda(a) = \alpha, \lambda(b) = \beta, \lambda(a') = \overline{\alpha}$. Consider the symmetric synchronisation algebra $\alpha \bullet \overline{\alpha} = \boldsymbol{\tau}$, $\alpha \bullet * = \alpha$, $\overline{\alpha} \bullet * = \overline{\alpha}$, $\beta \bullet * = \beta$ and undefined otherwise. Then $\mathcal{E}_6 := \mathcal{E}_4 \| \mathcal{E}_5$ is as follows : $E_6 = \{e := (\varnothing, a, *), e' := (\varnothing, *, a'), e'' := (\varnothing, a, a'), d := (\{e\}, a', *), d'' := (\{e''\}, a', *)\}$, with the ordering defined as $e \leq d, e'' \leq d''$, while the conflict is defined as $e \smile e''$, $e' \smile e''$, $e \smile d''$, $e' \smile d''$, $e'' \smile d$, $d \smile d''$. The labelling function is $\lambda(e) = \alpha, \lambda(e') = \overline{\alpha}, \lambda(e'') = \boldsymbol{\tau}, \lambda(d) = \lambda(d'') = \beta$. Note that, while $\mathcal{E}_4, \mathcal{E}_5$ are confusion free, $\mathcal{E}_6$ is not, since reflexive immediate conflict is not transitive.

Example A.3.3. Finally we show two examples of morphisms. First, consider the two event structures $\mathcal{E}_7, \mathcal{E}_8$ defined as follows :

- $E_7 = \{a', b', c', d', e', a'', b'', c'', d''\}$ with $a' \smile_\mu a''$ $b' \smile_\mu c', d' \smile_\mu e', b'' \smile_\mu c''$ and $a' \leq b', c', d', e'$ and $a'' \leq b'', c'', d''$.
- $E_8 = \{b, c, d, e\}$ with $b \smile_\mu c, d \smile_\mu e$, and trivial ordering.

Note that both $\mathcal{E}_7$ and $\mathcal{E}_8$ are confusion free.

We define a morphism $f : \mathcal{E}_7 \to \mathcal{E}_8$ by putting $f(x') = f(x'') = x$ for $x = b, c, d, e$ while f is undefined on a', a''. Note that b' and b'' are mapped to the same element b, and they are indeed in conflict, because they inherit the conflict $a' \smile a''$.

For another example consider the two event structures $\mathcal{E}_9, \mathcal{E}_{10}$, where $E_9 = E_{10} = \{a, b\}$, both have empty conflict, and in $\mathcal{E}_9$ we have $a \leq b$. The identity function on $\{a, b\}$ is a morphism $\mathcal{E}_9 \to \mathcal{E}_{10}$ but not vice versa. We can say that the causal order of $\mathcal{E}_9$ refines the causal order of $\mathcal{E}_{10}$.

A.4 Typed event structures

In this section we present a notion of types for an event structure, which are inspired from the types for the linear π-calculus. Every such type is represented by an event structure which interprets the causality between the names contained in the type. We then assign types to event structures by allowing a more general notion of causality.

A.4.1 Types and environments

Types and type environments are generated by the following grammar

$$\begin{array}{rcll} \Gamma, \Delta & ::= & y_1 : \sigma_1, \ldots, y_n : \sigma_n & \text{(type environment)} \\ \tau, \sigma & ::= & \&_{i \in I} \Gamma_i & \text{(branching)} \\ & | & \bigoplus_{i \in I} \Gamma_i & \text{(selection)} \\ & | & \bigotimes_{i \in I} \Gamma_i & \text{(offer)} \\ & | & \biguplus_{i \in I} \Gamma_i & \text{(request)} \\ & | & \updownarrow & \text{(closed type)} \end{array}$$

A type environment Γ is *well formed* if any name appears at most once. Only well formed environments are considered for typing event structures. An environment

can also be thought of as a partial function from names to types. In this view we can talk of *domain* and *range* of an environment.

We say a name is *confidential* for a type environment Γ if it appears inside a type in the range of Γ. A name is *public* if it is in the domain of Γ. Intuitively, confidential names are used to identify different occurrences of events that have the same public label. We will see this explicitly when we introduce the event structure semantics. Technically, in client and server types, we also require the enviroments Γ_i to be nonempty in order to distinguish different components. This is not restrictive, as we can always introduce "dummy" names.

The form of event structures types and environments is similar to those of the π-calculus. In the π-calculus we only keep track of the types of the object names, as their precise identity is irrelevant. In event structure types we recursively keep track not only of the types, but also of the identity of the confidential names. Moreover server and client types explicitly represent each copy of the resource.

Branching types represent the notion of "environmental choice" : several choices are available for the environment to choose. Selection types represent the notion of "process choice" : some choice is made by the process. In both cases the choice is alternative : one excludes all the others. Server types represent the notion of "available resource" : I offer to the environment something that is available regardless of whatever else happens. Client types represent the notion of "concurrent request" : I want to reserve a resource that I may use at any time.

It is straightforward to define duality between types by exchanging branching and offer, with selection and request, respectively. Therefore, for every type τ and environment Γ, we can define their dual $\overline{\tau}$, $\overline{\Gamma}$. However types and environments enjoy a more general notion of duality that is expressed by the following definition. We define a notion of matching for types. The matching of two types also produces a set of names that are to be considered as "closed", as they have met their dual. Finally, after two types have matched, they produce a "residual" type.

We define the relations $match[\tau, \sigma] \to S$, $match[\Gamma, \Delta] \to S$ symmetric in the first two arguments, and the partial function $res[\tau, \sigma]$ as follows :

- let $\Gamma = x_1 : \sigma_1 \dots x_n : \sigma_n$ and $\Delta = y_1 : \tau_1 \dots y_m : \tau_m$. Then $match[\Gamma, \Delta] \to S$ if $n = m$, for every $i \leq n$ $x_i = y_i$, $match[\sigma_i, \tau_i] \to S_i$ and $S = \bigcup_{i \leq n} S_i \cup \{x_i\}$;
- let $\tau = \&_{i \in I} \Gamma_i$ and $\sigma = \bigoplus_{j \in J} \Delta_j$. Then $match[\tau, \sigma] \to S$ if $I = J$, for all $i \in I$, $match[\Gamma_i, \Delta_i] \to S_i$ and $S = \bigcup_{i \in I} S_i$ In such a case $res[\tau, \sigma] = \updownarrow$;
- let $\tau = \bigotimes_{i \in I} \Gamma_i$ and $\sigma = \biguplus_{j \in J} \Gamma_j$ Then $match[\tau, \sigma] \to S$ if $J \subseteq I$, for all $j \in J$, $match[\Gamma_j, \Delta_j] \to S_j$, and $S = \bigcup_{j \in J} S_j$ In such a case $res[\tau, \sigma] = \bigotimes_{i \in I \setminus J} \Gamma_i$.
- $match[\updownarrow, \updownarrow] \to \varnothing$, $res[\updownarrow, \updownarrow] = \updownarrow$.

A branching type matches a corresponding selection types, all their names are closed and the residual type is the special type recording that the matching has taken place. A client type matches a server type if every request corresponds to an available resource. The residual type records which resources are still

available.

We now define the composition of two environments. Two environments can be composed if the types of the common names match. Such names are given the residual type by the resulting environment. All the closed names are recorded. Client types can be joined, so that the two environments are allowed to independently reserve some resources. Given two type environments Γ_1, Γ_2 we define the environment $\Gamma_1 \odot \Gamma_2 \stackrel{\text{def}}{=} \Gamma$ and the set of names $cl(\Gamma_1, \Gamma_2)$ as follows :

- if $x \notin Dom(\Gamma_1)$ and no name in $\Gamma_2(x)$ appears in Γ_1, then $\Gamma(x) = \Gamma_2(x)$, $S_x = \varnothing$ and symmetrically ;
- if $\Gamma_1(x) = \tau, \Gamma_2(x) = \sigma$ and $match[\tau, \sigma] \to S$, then $\Gamma(x) = res[\tau, \sigma]$ and $S_x = S$;
- if $\Gamma_1(x) = \biguplus_{i \in I} \Delta_i$ and $\Gamma_2(x) = \biguplus_{j \in J} \Delta_j$ and no name appears in both Δ_i and Δ_j for every $i, j \in I \cup J$ we have then $\Gamma(x) = \biguplus_{i \in I \cup J} \Delta_i$ and $S_x = \varnothing$;
- if any of the other cases arises, then Γ is not defined ;
- $cl(\Gamma_1, \Gamma_2) = \bigcup_{x \in Dom(\Gamma_1, \Gamma_2)} S_x$.

A.4.2 Semantic of types

Type environments are given a semantics in terms of labelled confusion free event structures.

The labels are the ones described in the Section A.2. Labels can be *allowed* or *disallowed* by a type environments, similarly to the π-calculus case, but recursively considering the confidential names. Consider a label α, an environment Γ, and suppose $\Gamma(x) = \sigma$, then :

- if $\alpha = x\mathtt{in}_j\langle\tilde{y}\rangle$, and if $\sigma = \&_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j, then α is allowed by Γ ;
- if $\alpha = \overline{x}\mathtt{in}_j\langle\tilde{y}\rangle$, and if $\sigma = \bigoplus_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed by Γ ;
- if $\alpha = x\langle\tilde{y}\rangle$, and if $\sigma = \bigotimes_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed by Γ ;
- if $\alpha = \overline{x}\langle\tilde{y}\rangle$, and if $\sigma = \biguplus_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed by Γ ;
- if $\alpha = \tau$, then α is allowed by Γ ;
- if α is allowed by any of the environments appearing in the types in the range of Γ, then α is allowed by Γ.

Note that if a label is allowed, the definition of well-formedness guarantees that it is allowed in a unique way. Note also that if a label α has subject x and x does not appear in Γ, then α is not allowed by Γ. Let $Dis(\Gamma)$ be the set of labels that are *not allowed* by the environment Γ.

The semantics of types is presented in Figure A.6, where we assume that $\tilde{y}_i$ represents the sequence of names in the domain of Γ_i. A name used for branching/selection identifies a cell. A name used for offer/request identifies a "cluster" of parallel events. The semantics of selection and branching is obtained using the sum of event structures. The semantics of client and server is given using the parallel composition. To define the parallel composition, we use a sym-

$$[\![y_1 : \sigma_1, \ldots, y_n : \sigma_n]\!] \;=\; [\![y_1 : \sigma_1]\!] \| \ldots \| [\![y_n : \sigma_n]\!]$$

$$[\![x : \&_{i\in I}\, \Gamma_i]\!] \;=\; \textstyle\sum_{i\in I} x\mathtt{in}_i\langle \tilde{y}_i\rangle.[\![\Gamma_i]\!] \qquad [\![x : \bigoplus_{i\in I}\, \Gamma_i]\!] \;=\; \textstyle\sum_{i\in I} \overline{x}\mathtt{in}_i\langle \tilde{y}_i\rangle.[\![\Gamma_i]\!]$$

$$[\![x : \bigotimes_{i\in I}\, \Gamma_i]\!] \;=\; \textstyle\prod_{i\in I} x\langle \tilde{y}_i\rangle.[\![\Gamma_i]\!] \qquad [\![x : \uplus_{i\in I}\, \Gamma_i]\!] \;=\; \textstyle\prod_{i\in I} \overline{x}\langle \tilde{y}_i\rangle.[\![\Gamma_i]\!]$$

$$[\![x : \updownarrow]\!] \;=\; \varnothing$$

FIGURE A.6 – Denotational semantics of types

metric synchronisation algebra which extends the one defined in Section A.2 : $\alpha \bullet * = \alpha$, $x\mathtt{in}_i\langle \tilde{y}_i\rangle \bullet \overline{x}\mathtt{in}_i\langle \tilde{y}_i\rangle = (x, \overline{x})\mathtt{in}_i\langle \tilde{y}_i\rangle$, $x\langle \tilde{y}\rangle \bullet \overline{x}\langle \tilde{y}\rangle = (x, \overline{x})\langle \tilde{y}\rangle$, and undefined otherwise. Also the semantics of an environment is obtained as the parallel composition of the semantics of the types, with initial events labelled using the corresponding names. Such parallel compositions do not involve synchronisation due to the condition on uniqueness of names and thus, as we already explained, they can be thought of as disjoint unions.

The following result is a sanity check for our definitions. It shows that matching of types corresponds to parallel composition with synchronisation.

Proposition A.4.1. *Take two environments* Γ_1, Γ_2*, and suppose* $\Gamma_1 \odot \Gamma_2$ *is defined. Then* $([\![\Gamma_1]\!] \| [\![\Gamma_2]\!]) \setminus (Dis(\Gamma_1 \odot \Gamma_2) \cup \boldsymbol{\tau}) = [\![\Gamma_1 \odot \Gamma_2]\!]$.

A.4.3 Typing event structures

Given a labelled confusion free event structure $\mathcal{E}$ on the same set of labels as above, we define when $\mathcal{E}$ is typed in the environment Γ, written as $\mathcal{E} \triangleright \Gamma$. A type environment Γ defines a general behavioural pattern via its semantics $[\![\Gamma]\!]$. The intuition is that for an event structure $\mathcal{E}$ to have type Γ, $\mathcal{E}$ should follow the pattern of $[\![\Gamma]\!]$, possibly "refining" the causal structure of $[\![\Gamma]\!]$ and possibly omitting some of its actions.

Definition A.4.2. We say that $\mathcal{E} \triangleright \Gamma$, if the following conditions are satisfied :
- each cell in $\mathcal{E}$ is labelled by x, $\overline{x}$ or $(x, \overline{x})$, and labels of the events correspond to the label of their cell in the obvious way ;
- there exists a label-preserving morphism of labelled event structures $f : \mathcal{E} \to [\![\Gamma]\!]$ such that $f(e)$ is undefined if and only if $\lambda(e) \in \boldsymbol{\tau}$.

Roughly speaking a confusion free event structure $\mathcal{E}$ has type Γ if cells are partitioned into branching, selection, request, offer and synchronisation cells, all the non-synchronisation events of $\mathcal{E}$ are represented in Γ and causality in $\mathcal{E}$ refines causality in $[\![\Gamma]\!]$.

As we said, the parallel composition of confusion free event structures is not confusion free in general. The main result of this section shows that the parallel

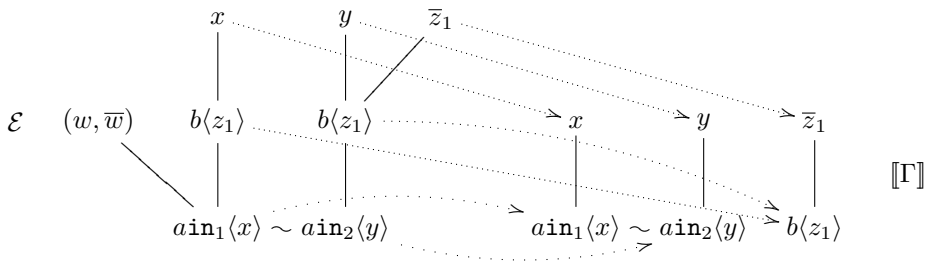


FIGURE A.7 – Typed event structure

composition of typed event structures is still confusion free, and moreover is typed.

Lemma A.4.3. *Suppose $\mathcal{E} \triangleright \Gamma$, and let $e, e' \in E$ be distinct events.*
- *If $\lambda(e) = \lambda(e') \neq \boldsymbol{\tau}$, then $e \smile e'$.*
- *If $\lambda(e), \lambda(e') \neq \boldsymbol{\tau}$ and $\lambda(e)$ and $\lambda(e')$ have the same subject and different branch,then $e \smile e'$.*
- *If $e \smile_\mu e'$, then $\lambda(e)$ and $\lambda(e')$ have the same subject and different branch.*

Theorem A.4.4. *Take two labelled confusion free event structures $\mathcal{E}_1, \mathcal{E}_2$. Suppose $\mathcal{E}_1 \triangleright \Gamma_1$ and $\mathcal{E}_2 \triangleright \Gamma_2$. Assume $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2))$ is confusion free and $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2)) \triangleright \Gamma_1 \odot \Gamma_2$.*

The proof relies on the fact that the typing system, in particular the uniqueness condition on well formed environments, guarantees that no new conflict is introduced through synchronisation.

Special cases are obtained when some or all cells are singletons. We call a typed event structure *deterministic* if its selection cells and its $\boldsymbol{\tau}$ cells are singletons. We call a typed event structure *simple* if all its cells are singletons. In particular, a simple event structure is conflict free.

Theorem A.4.5. *Take two labelled deterministic (resp. simple) event structures $\mathcal{E}_1 \triangleright \Gamma_1$ and $\mathcal{E}_2 \triangleright \Gamma_2$. Suppose $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus Dis(\Gamma_1 \odot \Gamma_2)$ is deterministic (resp. simple).*

A.4.4 Examples

In the following, when the indexing set of a branching type is a singleton, we use the abbreviation $(\Gamma)^\downarrow$. Similarly, for a singleton selection type we write $(\Gamma)^\uparrow$. When the indexing set of a type is $\{1, 2\}$, we write $(\Gamma_1 \& \Gamma_2)$ or $(\Gamma_1 \otimes \Gamma_2)$.

Example A.4.1. Consider the types $\tau_1 = (x : ()^\downarrow \,\&\, y : ()^\downarrow), \sigma_1 = \biguplus_{i \in \{2\}} (z_i :\updownarrow)$ $\tau_2 = (x : ()^\uparrow \oplus y : ()^\uparrow), \sigma_2 = \bigotimes_{i \in \{1,2,3\}} (z_i :\updownarrow)$. We have $match[\tau_1, \tau_2]$, with $res[\tau_1, \tau_2] = \updownarrow$; and $match[\sigma_1, \sigma_2]$, with $res[\sigma_1, \sigma_2] = \bigotimes_{i \in \{2\}} (z_i :\updownarrow)$. If we put

$\Gamma_1 = a : \tau_1, b : \sigma_1$, and $\Gamma_2 = a : \tau_2, b : \sigma_2$, we have that $\Gamma_1 \odot \Gamma_2 = a :\updownarrow, b : \bigotimes_{i\in\{1,3\}}(z_i :\updownarrow)$.

Example A.4.2. As an example of typed event structures, consider the environment $\Gamma = a : (x : ()^{\downarrow} \;\&\; y : ()^{\downarrow}), b : \biguplus_{i\in\{1\}}(z_i : ()^{\uparrow})$. Figure A.7 shows an event structure $\mathcal{E}$, such that $\mathcal{E} \triangleright \Gamma$, together with a morphism $\mathcal{E} \to [\![\Gamma]\!]$. Note that the two events in $\mathcal{E}$ labelled with $b\langle z_1\rangle$ are mapped to the same event and indeed they are in conflict.

A.5 Name sharing CCS

Our goal is to use typed event structures to interpret the linearly typed π-calculus. We would like this interpretation to be similar, in a sense to extend, Winskel's semantics of CCS [Win82]. However we face two main difficulties.

The first problem is that Winskel's semantics is strictly related to the labelled transition semantics of CCS. The labelled transition semantics of the π-calculus is more complex, and in particular the communication rule involves α-conversion. This rule seems difficult to represent using the available techniques. The second problem is that we want to use typed, and therefore confusion free, event structures. However, even if we applied Winskel's semantics to a fragment of the π-calculus without name passing, we would obtain confused event structures. This is due to the replicated server. If we interpret the replicated sever as an infinite parallel composition of copies of the resource, Winskel's semantics allows each of such components to compete for the same client. This competition creates some spurious conflicts that break confusion freeness. Alternatively, we can model the server as one single resource that, after providing its service, spawns another copy of itself. This would create another spurious conflict between two clients to decide who is going to be served first.

To see this with an example, imagine the server to be a post office. A post office allows a client to post a letter. How do we implement this service ? If the post office has only one employee, that accepts one letter at a time, then two clients could end up fighting for the right of going first. If the post office has infinitely many employees, still two clients may fight over the same one (for instance because she is more efficient), or two employees could fight over the same client (because their salary is proportional to their activity).

Our solution to this problem would be to assign *in advance* an employee for each client, so that when the client decides to post his letter, he knows which till to go to. This solution has also the advantage to solve the α-conversion problem. If we know in advance whom we are going to communicate with, we can also decide in advance which "private" channels we are going to share. In a sense we perform α-conversion *before* we start the computation, or, one could say, at *compile time*.

To formalise this intutition we first introduce a variant of CCS that will be interpreted using typed event structures. Our language differs from CCS in many technical details, but the only relevant difference is that synchronisation

between actions happens only if the actions share the same confidential names. In a second moment we will see the correspondence between this calculus, and the π-calculus.

A.5.1 Syntax

Syntactically the calculus we present is very similar to the π-calculus. Communication happens along channels, and information is "passed" along such channels. The difference between the two is in the semantics. In our variant of CCS names are not sent from a process to another : processes decide their confidential names before communicating, and there is not α-conversion. If the chosen names do not coincide, the processes do not synchronise.

Another important technical difference from standard π-calculus and CCS is that we allow infinite parallel composition and infinite restriction. The former is necessary in order to translate replicated processes of the π-calculus. The standard intuition in the π-calculus is that the process $!P$ represents the parallel composition of infinitely many copies of P. We need to represent this explicitly in order to be able to provide each copy with different confidential names. Infinite restriction is also necessary, because we need to restrict all confidential names that are shared between two processes in parallel, and these are in general infinitely many.

We call this language Name Sharing CCS, or NCCS. The syntax is as follows :

P	$::=$	$x \,\&_{i\in I}\, \mathtt{in}_i\langle\tilde{y_i}\rangle.P_i$	branching
	$\mid$	$\overline{x} \bigoplus_{i\in I} \mathtt{in}_i\langle\tilde{y_i}\rangle.P_i$	selection
	$\mid$	$x\langle\tilde{y}\rangle.P$	single offer
	$\mid$	$\overline{x}\langle\tilde{y}\rangle.P$	single request
	$\mid$	$\prod_{i\in I} P_i$	parallel composition
	$\mid$	$P \setminus S$	restriction
	$\mid$	$\mathbf{0}$	zero

For the notation, we use conventions analogous to the π-calculus. Processes are identified up to a straightforward structural congruence, which includes the rule $(P\setminus S)\setminus T \equiv P\setminus(S\cup T)$, but no notion of α-equivalence. Names of a process are partitioned into *public* and *confidential*, similarly to the free/bound partition in the π-calculus. The change of name undelines the fact that α-conversion is not allowed.

As for the π-calculus, the fragment of NCCS where the indexing sets of branching and selection are singleton is called *simple*. The fragment where the selection is always a singleton, but the branching is arbitrary is called *deterministic*. The general language is for clarity denoted as the *nondeterministic* fragment.

The operational semantics is completely analogous to the one of CCS, and it is shown in Figure A.8. Labels are the same as for the π-calculus, and synchronisation labels are globally denoted by $\boldsymbol{\tau}$. The main difference with CCS is the presence of the confidential names that are used only for synchronisation. Note also that only the subject of an action is taken into account for restriction.

$$\overline{x}\bigoplus_{i\in I}\mathtt{in}_i\langle\tilde{y}_i\rangle.P_i \xrightarrow{\overline{x}\mathtt{in}_j\langle\tilde{y}_j\rangle} P_j \qquad x\,\&_{i\in I}\,\mathtt{in}_i\langle\tilde{y}_i\rangle.P_i \xrightarrow{x\mathtt{in}_j\langle\tilde{y}_j\rangle} P_j$$

$$\overline{x}\langle\tilde{y}\rangle.P \xrightarrow{\overline{x}\langle\tilde{y}\rangle} P \qquad x\langle\tilde{y}\rangle.P \xrightarrow{x\langle\tilde{y}\rangle} P$$

$$\frac{P\xrightarrow{\beta}P' \quad subj(\beta)\notin S}{P\setminus S\xrightarrow{\beta}P'\setminus S} \qquad \frac{P\xrightarrow{\boldsymbol{\tau}}P'}{P\setminus S\xrightarrow{\boldsymbol{\tau}}P'\setminus S}$$

$$\frac{P_n\xrightarrow{\beta}P'}{\prod_{i\in\mathbb{N}}P_i\xrightarrow{\beta}(\prod_{i\in\mathbb{N}\setminus\{n\}}P_i)\mid P'} \qquad \frac{P_n\xrightarrow{\alpha}P' \quad P_m\xrightarrow{\beta}P''}{\prod_{i\in\mathbb{N}}P_i\xrightarrow{\alpha\bullet\beta}(\prod_{i\in\mathbb{N}\setminus\{n,m\}}P_i)\mid P'\mid P''}$$

FIGURE A.8 – Labelled Transition System for Name Sharing CCS

Example A.5.1. For instance the process

$$(x\langle y\rangle.P\mid\overline{x}\langle z\rangle.R)\setminus\{x\}$$

cannot perform any transition, because y and z do not match. The process

$$(x\langle y\rangle.P\mid\overline{x}\langle y\rangle.Q\mid\overline{x}\langle y\rangle.R)\setminus\{x\}$$

can perform two different initial $\boldsymbol{\tau}$ transitions. Since the name x is not bound, it does not become private to the subprocesses involved in the communication. The process

$$(x\,\&_{i\in\{1,2\}}\,\mathtt{in}_i.P_i\mid\overline{x}\bigoplus_{i\in\{1,2\}}\mathtt{in}_i.R_i)\setminus\{x\}$$

can perform, nondeterministically, two $\boldsymbol{\tau}$ transitions to $(P_1\mid R_1)\setminus\{x\}$ or to $(P_2\mid R_2)\setminus\{x\}$.

A.5.2 Typing rules

Using the notions of type and type enviroment presented in Section A.4, we are going to present a typing system for NCCS. This typing system is very similar to the one of the π-calculus.

Before introducing the typing rules, we have to define the operation of "parallel composition of environments". This operation intuitevly combine environments for which the only possible shared public names are client requests.

Let Γ_h $h\in H$ be a family of enviroments such that for every name x, either for every h, $\Gamma_h(x)=\biguplus_{k_h\in K_h}\Delta_{k_h}$, or $x\in Dom(\Gamma_h)$ for at most one h. We define $\Gamma=\prod_{h\in H}\Gamma_h$ as follows. If for every h, $\Gamma_h(x)=\biguplus_{k_h\in K_h}\Delta_{k_h}$, then $\Gamma(x)=\biguplus_{k_h\in K_h,h\in H}\Delta_{k_h}$, assuming all the names involved are distinct. If $x\in Dom(\Gamma_h)$ for at most one h, then $\Gamma(x)=\Gamma_h(x)$.

A special case, which will be of particular interest when encoding the π-calculus, is when all the Γ_h are different instances of the same environment, up to renaming of the confidential names. For any set K, let $F_K : Names \to \mathcal{P}(Names)$ be a function such that, for every name x, there is a bijection between K and $F_K(x)$. Concretely we can represent $F_K(x) = \{x^k \mid k \in K\}$. In the following we assume that each set K is associated to a unique F_K, and that for distinct x, y, $F_K(x) \cap F_K(y) = \varnothing$.

Given a type τ, and an index k, define τ^k as follows :

- $\bigotimes_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)^k = \bigotimes_{h\in H}(\tilde{y}_h^k : \tilde{\tau}_h^k)$, where $\tilde{y}_h = (y_{i,h})_{i\in I}$ and $\tilde{y}_h^k = (y_{i,h}^k)_{i\in I}$;
- and similarly for all other types.

Given an environment Γ, we define Γ^k where for every name $x \in Dom(\Gamma)$, $\Gamma^k(x) = \Gamma(x)^k$. The environment $\Gamma[K]$ is defined as $\prod_{k\in K}\Gamma^k$, and is thus defined only when for every $x \in Dom(\Gamma)$, $MD(\Gamma(x)) = \mathbf{?}$. We will also assume that all names in the range of the substitution are fresh, in the sense that no name in the range of F_K appears in the domain of Γ. Under this assumption we easily have that if Γ is well formed and if $\Gamma[K]$ is defined, then $\Gamma[K]$ is also well formed.

We are now ready to write the rules : see Figure A.9. The rule for weakening of the client type tells us that we can request a resource even if we are not actually using it. The rule for the selection tells us that we can choose less than what the types offers. The parallel composition is well typed only if the names used for communication have matching types, and if the matched names are restricted. This makes sure that communication can happen, and that the shared names are indeed private to the processes involved.

A.5.3 Typed semantics

The relation Γ allows β was defined in Section A.4. We also need a definition of the environment $\Gamma \setminus \beta$, similar to the one defined in Section A.2.

- $\Gamma \setminus \boldsymbol{\tau} = \Gamma$;
- if $\Gamma = \Delta, x : \binampersand_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)$, then $\Gamma \setminus x\mathtt{in}_i(\tilde{y}_i) = \Delta, \tilde{y}_i : \tilde{\tau}_i$;
- if $\Gamma = \Delta, x : \bigoplus_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)$, then $\Gamma \setminus \overline{x}\mathtt{in}_i(\tilde{y}_i) = \Delta, \tilde{y}_i : \tilde{\tau}_i$;
- if $\Gamma = \Delta, x : \bigotimes_{h\in H\uplus\{j\}}(\tilde{y}_h : \tilde{\tau}_h)$, then $\Gamma \setminus x(\tilde{y}_j) = \Delta, \tilde{y}_j : \tilde{\tau}_j, x : \bigotimes_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)$;
- if $\Gamma = \Delta, x : \biguplus_{h\in H\uplus\{j\}}(\tilde{y}_h : \tilde{\tau}_h)$, then $\Gamma \setminus \overline{x}(\tilde{y}_j) = \Delta, \tilde{y}_j : \tilde{\tau}_j, x : \biguplus_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)$.

Note that $\Gamma \setminus \beta$ is defined precisely when Γ allows β. We have the following

Proposition A.5.1. *If $P \triangleright \Gamma$, $P \xrightarrow{\beta} Q$ and Γ allows β, then $Q \triangleright \Gamma \setminus \beta$.*

Corollary A.5.2 (Subject Reduction)**.** *If $P \triangleright \Gamma$, $P \xrightarrow{\tau} Q$ then $Q \triangleright \Gamma$.*

$$\frac{}{0 \triangleright \varnothing}\ \text{Zero} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \biguplus_{h \in H} \Gamma_h}\ \text{WeakReq} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \updownarrow}\ \text{WeakCl}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma}{x \,\&_{i \in I}\, \mathtt{in}_i \langle \tilde{y}_i \rangle . P_i \triangleright \Gamma, x : \&_{i \in I} (\tilde{y}_i : \tilde{\tau}_i)}\ \text{Branch}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma \quad I \subseteq J}{\overline{x} \bigoplus_{i \in I} \mathtt{in}_i p_i \langle \tilde{y}_i \rangle . P_i \triangleright \Gamma, x : \bigoplus_{i \in J} (\tilde{y}_i : \tilde{\tau}_i)}\ \text{Sel}$$

$$\frac{P \triangleright \Gamma, \tilde{w}_j : \tilde{\tau}_j, x : \biguplus_{h \in H} (\tilde{w}_h : \tilde{\tau}_h) \quad \tilde{w}_j \text{ fresh}}{\overline{x} \langle \tilde{w}_j \rangle . P \triangleright \Gamma, x : \biguplus_{h \in H \uplus \{j\}} (\tilde{w}_h : \tilde{\tau}_h)}\ \text{Req}$$

$$\frac{P_h \triangleright \Gamma_h, \tilde{y}_h : \tilde{\tau}_h \quad a \notin \Gamma}{\prod_{h \in H} x \langle \tilde{y}_h \rangle . P_h \triangleright \prod_{h \in H} \Gamma_h, a : \bigotimes_{h \in H} (\tilde{y}_h : \tilde{\tau}_h)}\ \text{Offer}$$

$$\frac{P \triangleright \Gamma, x : \tau \quad MD(\tau) = \mathbf{!}, \updownarrow}{P \setminus x \triangleright \Gamma}\ \text{Res} \qquad \frac{P_i \triangleright \Gamma_i \quad (i = 1, 2) \quad S = cl(\Gamma_1, \Gamma_2)}{(P_1 \| P_2) \setminus S \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par}$$

FIGURE A.9 – Typing Rules for NCCS

Proposition A.5.1 allows us to define the notion of *typed transition*, written $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma'$ by adding the constraint :

$$\frac{P \xrightarrow{\beta} Q \quad \Gamma \text{ allows } \beta}{P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta}$$

We are going to define a notion of bisimulation which is slightly different from one might expect. The reason is that labels, as we have presented them, contain somehow too much information, more than a typed context should recognise. Normal CCS bisimulation would be too fine and our full abstraction result would fail. In principle a label should represent what a context can observe. But a typed context cannot really take apart two processes with different confidential names. Either the context does not synchronise on the subject of the label, and then the confidential names do not matter. Or, if it does synchronise, the typing rules ensure it must do it with the proper confidential names, whatever they are. We want thus to allow processes that use different confidential names to be identified.

In the following ρ will be a fresh injective renaming of the confidential names of an environment Δ. In such a case then $\Delta[\rho]$ is also a well formed environment.

Definition A.5.3. Let $\mathcal{R}$ be a symmetric relation between judgments such that if $(P \triangleright \Gamma)\, \mathcal{R}\, (P' \triangleright \Gamma')$, then $\Gamma' = \Gamma[\rho]$, for some injective renaming ρ. We say that $\mathcal{R}$ is a *bisimulation up to renaming* if the following is satisfied :

$$
\begin{array}{rcl}
[\![0 \triangleright \varnothing]\!] & = & \varnothing \\
[\![P \triangleright \Gamma, x : \biguplus_{h\in H} \Gamma_h]\!] & = & [\![P \triangleright \Gamma]\!] \\
[\![P \triangleright \Gamma, x : \updownarrow]\!] & = & [\![P \triangleright \Gamma]\!] \\
[\![P \setminus x \triangleright \Gamma]\!] & = & [\![P \triangleright \Gamma, x : \tau]\!] \setminus \{x\} \\
[\![\overline{x} \bigoplus_{i\in I} \mathtt{in}_i\langle\tilde{y}_i\rangle.P_i \triangleright \Gamma, x : \bigoplus_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)]\!] & = & \sum_{i\in I} \overline{x}\mathtt{in}_i\langle\tilde{y}_i\rangle.[\![P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i]\!] \\
[\![x \,\&_{i\in I}\, \mathtt{in}_i\langle\tilde{y}_i\rangle.P_i \triangleright \Gamma, x : \,\&_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)]\!] & = & \sum_{i\in I} x\mathtt{in}_i\langle\tilde{y}_i\rangle.[\![P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i]\!] \\
[\![\overline{x}\langle\tilde{y}\rangle.P \triangleright \Gamma, x : \biguplus_{k\in K\uplus\{j\}}(\tilde{y}_k : \tilde{\tau}_k)]\!] = \overline{x}\langle\tilde{y}\rangle.[\![P \triangleright \Gamma, x : \biguplus_{k\in K}(\tilde{y}_k : \tilde{\tau}_k), \tilde{y}_j : \tilde{\tau}_j]\!] & & \\
[\![\prod_{k\in K} x\langle\tilde{y}_k\rangle.P_k \triangleright \prod_{k\in K}\Gamma_k, x : \bigotimes_{k\in K}(\tilde{y}_k : \tilde{\tau}_k)]\!] & = & \prod_{k\in K} x\langle\tilde{y}_k\rangle.[\![P_k \triangleright \Gamma_k, \tilde{y}_k : \tilde{\tau}_k]\!] \\
[\![(P_1 \| P_2) \setminus S \triangleright \Gamma_1 \odot \Gamma_2]\!] = [\![P_1 \triangleright \Gamma_1]\!] \mid [\![P_2 \triangleright \Gamma_2]\!] \setminus (Dis(\Gamma_1 \odot \Gamma_2)) & &
\end{array}
$$

FIGURE A.10 – Denotational semantics of simple Name Sharing CCS

– whenever $(P \triangleright \Gamma)\ \mathcal{R}\ (P' \triangleright \Gamma')$, $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta$, then there exists a renaming ρ and a process Q' such that $P'[\rho] \triangleright \Gamma'[\rho] \xrightarrow{\beta} Q' \triangleright \Gamma[\rho' \circ \rho] \setminus \beta$, and $(Q \triangleright \Gamma \setminus \beta)\ \mathcal{R}\ (Q' \triangleright \Gamma'[\rho] \setminus \beta)$.

If there exists a bisimulation up to renaming between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma')$.

A.6 Event structure semantics of Name Sharing CCS

A.6.1 Semantics of nondeterministic NCCS

The event structure semantics of typed NCCS is presented in Figure A.10. It is given in terms of labelled event structures, using the operations, in particular the parallel composition, as defined in Section A.3.5. This construction is perfectly analogous to the one in [WN95], the only difference being the synchronisation algebra. However, since the synchronisation algebra is the same for both the operational and the denotational semantics, we obtain automatically the correspondence between the two, as in [WN95].

In the parallel composition, we have to restrict all the channels that are subject of communication. More generally, we need to restrict all the actions that are not allowed by the new type environment.

The main property of the semantics is that the denotation of a typed process is a typed event structure. In particular all denoted event structures are confusion free.

Theorem A.6.1. *Let P be a process and Γ an environment such that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is confusion free, and $[\![P \triangleright \Gamma]\!] \triangleright [\![\Gamma]\!]$.*

A.6.2 Semantics of deterministic NCCS

The syntax of NCCS introduces the conflict explicitly, therefore we cannot obtain conflict free event structures. The result above shows that no new conflict is introduced through synchronisation. Moreover, in the deterministic fragment, synchronisation does indeed resolve the conflicts.

First it is easy to show that the semantics of deterministic NCCS is in term of deterministic event structures :

Proposition A.6.2. *Suppose P is a deterministic process, and that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is deterministic.*

The main theorem is the following, which justifies the term "deterministic". It states that once all choices have been matched with selections, or cancelled out, what remains is a conflict free event structure.

Theorem A.6.3. *If let X be the set of names in P, then $[\![P \triangleright \Gamma]\!] \setminus X$ is a conflict free event structure.*

Corollary A.6.4. *If $[\![\Gamma]\!] = \varnothing$, then $[\![P \triangleright \Gamma]\!]$ is conflict free.*

A.6.3 Semantics of simple NCCS

Although the syntax of NCCS does not introduce directly any conflict, there is in principle the possibility that conflict is introduced by the parallel composition. The typing system is designed in such a way that this is not the case.

Theorem A.6.5. *Suppose P is a simple process such that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is conflict free.*

A.6.4 Correspondence between the semantics

In order to show the correspondence between the operational and the denotational semantics, we invoke Winskel and Nielsen's handbook chapter [WN95]. Note that our semantics are a straightforward modification of the standard CCS semantics. This is the main reason why we chose the formalism presented here : we wanted to depart as little as possible from the treatment of [WN95].

The main difference is that typed semantics modifies the behaviour, by forbidding some of the actions. However this modification acts precisely as a special form of name restriction : in the labelled transition system it blocks some action, while in event structures it cancel them out (together with all events enabled by them). With a straightforward generalisation of the notion of restriction, we then preserve the correspondence between the two semantics and the proof technique of [WN95] carries over. In particular we have

Theorem A.6.6. *Take two typed NCCS processes $P \triangleright \Gamma, Q \triangleright \Gamma$. Suppose that $[\![P \triangleright \Gamma]\!] = [\![Q \triangleright \Gamma]\!]$, then $P \triangleright \Gamma \approx Q \triangleright \Gamma$.*

This theorem is the best result we can get : indeed, as for standard CCS, we cannot expect the event structure semantics to be fully abstract. Bisimilarity is a "interleaving" semantics, equating the two processes $\boldsymbol{\tau}\|\boldsymbol{\tau}$ and $\boldsymbol{\tau}.\boldsymbol{\tau}$, which have different event structure semantics.

A more direct correspondence is described next. Recall the way we derive a transition system from an event structure, as presented in Section A.3 : if $\lambda(e) = \beta$, then $\mathcal{E} \xrightarrow{\beta} \mathcal{E}\lfloor e$. We can therefore state the following correspondence :

Theorem A.6.7. *Let $\cong$ denote isomorphism of labelled event structures ;*

- *if $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$, then $[\![P \triangleright \Gamma]\!] \xrightarrow{\beta} \cong [\![P' \triangleright \Gamma \setminus \beta]\!]$.*
- *if $[\![P \triangleright \Gamma]\!] \xrightarrow{\beta} \mathcal{E}'$ then $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ and $\mathcal{E}' \cong [\![P' \triangleright \Gamma \setminus \beta]\!]$.*

The proof is by induction on the operational rules. The only difficult case is the parallel composition.

A.7 Correspondence between the calculi

A.7.1 Translation

We are now going to present a fully abstract translation of the π-calculus into Name Sharing CCS. The translation is parametrised over a fixed choice for the confidential names. This parametrisation is necessary because π-calculus terms are identified up to α-conversion, and so the identity of bound names is irrelevant, while in Name Sharing CCS, the identity of confidential names is important. Also, since servers are interpreted as infinite parallel compositions, every bound name of a server must correspond to infinitely many names in the interpretation.

The translation is a family of functions $pc[\![-]\!]^{\rho}$, that take a judgment of the π-calculus and return a process of NCCS. The semantic functions are indexed by a "choice" function ρ that for every bound name assigns a set (possibly a singleton) of fresh distinct names. In order to make this work, we have to use the convention that all bound names in the π-calculus are distinct, and different from the free names. In this way ρ cannot identify two different bindings. Although the translation is defined for all ρ, the target process will not always be typable. In particular, for some choice of renaming, the parallel composition in NCCS will not be typed.

We define the translation by induction on the derivation of the typing judgment. Without loss of generality, we will assume that all the weakenings are applied to the empty process. The translation is defined in Figure A.11.

The notation has to be explained. The notation $\rho, y \to S$ denotes the function ρ extended on a name y not already in the domain of ρ, and such that all names in S are fresh and distinct from any other name in the range of ρ. In the translation of the server, we use Y to denote the set of confidential names of the translation of P. We also use the choice function $\rho[K]$ defined as follows : assume the range of ρ are only singletons, say for every name x in the domain,

$$
\begin{array}{lcl}
pc[\![0 \triangleright x_i : (\tau_i)^{\mathtt{?}}, y_j : \updownarrow]\!]^{\rho} & = & 0 \\
pc[\![(\boldsymbol{\nu}\, x)P \triangleright \Gamma]\!]^{\rho} & = & pc[\![P \triangleright \Gamma, x : \tau]\!]^{\rho} \setminus x \\
pc[\![P_1 \| P_2 \triangleright \Gamma_1 \odot \Gamma_2]\!]^{\rho_1 \cup \rho_2} & = & (pc[\![P_1 \triangleright \Gamma_1]\!]^{\rho_1} \| pc[\![P_2 \triangleright \Gamma_2]\!]^{\rho_2}) \setminus S
\end{array}
$$

$$
\begin{array}{l}
pc[\![\overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \bigoplus_{i\in I}(\tilde{\tau}_i)^{\uparrow}]\!]^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} = \\
\qquad \overline{x} \bigoplus_{i\in I} \mathtt{in}_i\langle \tilde{z}_i \rangle.pc[\![P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i]\!]^{\rho}
\end{array}
$$

$$
\begin{array}{l}
pc[\![x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I}(\tilde{\tau}_i)^{\downarrow}]\!]^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} = \\
\qquad x \,\&_{i\in I}\, \mathtt{in}_i\langle \tilde{z}_i \rangle.pc[\![P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i]\!]^{\rho}
\end{array}
$$

$$
\begin{array}{lcl}
pc[\![!x(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{!}]\!]^{\rho[K],\tilde{y}\to\{\tilde{y}_k\}_{k\in K}} & = & \prod_{k\in K} x\langle \tilde{y}^k \rangle.pc[\![P \triangleright \Gamma]\!]^{\rho}[\tilde{y}^k/\tilde{y}][Y^k/Y] \\
pc[\![\overline{x}(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{\mathtt{?}}]\!]^{\rho,\tilde{y}\to\tilde{w}} & = & \overline{x}\langle \tilde{w} \rangle.pc[\![P \triangleright \Gamma, x : (\tilde{\tau})^{\mathtt{?}}[\tilde{w}/\tilde{y}]]\!]^{\rho}
\end{array}
$$

FIGURE A.11 – Translation from π to NCCS

$\rho(x) = \{y\}$. Then $\rho[K](x) = \{y_k \mid k \in K\}$, where y_k are obtained by a function $F_K : Names \to \mathcal{P}(Names)$ as in Section A.5. In the translation of the parallel composition, S denotes the set of names that are in the range of both ρ_1 and ρ_2.

Once past the rather heavy notation, the translation is rather simple. Note the way bound variables become confidential names. Observe also that the server is translated into an infinite parallel composition.

We said that the translation is not always typable. In particular, for the wrong choice of ρ_1, ρ_2, the parallel composition may not be typed because the chosen confidential names may not match. However it is always possible to find suitable ρ_1, ρ_2. Intuitively we can say that in translating typed π into typed NCCS, we perform α-conversion "at compile time".

Lemma A.7.1. *For every judgment $P \triangleright \Gamma$ in the π-calculus, there exists a choice function ρ and a type environment Δ such that $pc[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$. Moreover, for every injective fresh renaming ρ', if $pc[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$ then $pc[\![P \triangleright \Gamma]\!]^{\rho' \circ \rho} \triangleright \Delta[\rho']$.*

Example A.7.1. We demonstrate how the process which generates an infinite behaviour with infinite new name creation is interpreted into NCCS. Consider the process $\mathtt{Fw}\langle ab \rangle = !a(x).\overline{b}(y).y.\overline{x}$. This agent links two locations a and b and it is called a *forwarder*. It can be derived that $\mathtt{Fw}\langle ab \rangle \triangleright a : \tau, b : \overline{\tau}$ with $\tau = (()^{\uparrow})^{!}$. Consider the process $P_\omega = \mathtt{Fw}\langle ab \rangle \,|\, \mathtt{Fw}\langle ba \rangle$ so that $P_\omega \triangleright (a : \tau, b : \overline{\tau}) \odot (b : \tau, a : \overline{\tau})$, that is $P_\omega \triangleright a, b : \tau$. One possible translation for $\mathtt{Fw}\langle ab \rangle \triangleright a : (()^{\uparrow})^{!}, b : (()^{\downarrow})^{\mathtt{?}}$ is

$$Q_1 = \prod_{k\in K} a\langle x^k \rangle.\overline{b}\langle y^k \rangle.y^k.\overline{x}^k \triangleright a : \bigotimes_{k\in K}(x^k : ()^{\uparrow}), b : \biguplus_{k\in K}(y^k : ()^{\downarrow})$$

while for $\mathtt{Fw}\langle ba\rangle \triangleright b : (()^{\uparrow})^{!}, a : (()^{\downarrow})^{?}$ is

$$Q_2 = \prod_{h\in H} b\langle z^h\rangle.\overline{a}\langle w^h\rangle.w^h.\overline{z}^h \triangleright b : \bigotimes_{h\in H}(z^h : ()^{\uparrow}), a : \biguplus_{h\in H}(()^{?}w^h : ()^{\downarrow})$$

Assuming there are two "synchronising" injective functions $f : K \to H, g : H \to K$, such that $y^k = z^{f(k)}, w^h = x^{g(h)}$ (if not, we can independently perform a fresh injective renaming on both environments), we obtain that the corresponding types for a, b match, so that we can compose the two environments. Therefore the translation of $P_\omega \triangleright a, b : \tau$ is $(Q_1 \mid Q_2) \setminus S \triangleright \Delta$ for

$$\Delta = a : \bigotimes_{k\in K\setminus g(H)}(x^k : ()^{\uparrow}), b : \bigotimes_{h\in H\setminus f(K)}(z^h : ()^{\uparrow}).$$

The reader can check that any transition of P_ω is matched by a corresponding transition of its translation. This is what we formally show next.

A.7.2 Full abstraction

To show the correctness of the translation, we first prove the correspondence between the labelled transition semantics. If ρ is a choice function and S is a set of names, by $\rho \setminus S$ we denote the function ρ restricted to the names not in S.

Theorem A.7.2. *Suppose $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ in the π-calculus, then there exists ρ and Δ such that $pc[\![P \triangleright \Gamma]\!]^\rho \triangleright \Delta$ and $pc[\![P \triangleright \Gamma]\!]^\rho \triangleright \Delta \xrightarrow{\beta} pc[\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho\setminus obj(\beta)} \triangleright \Delta \setminus \beta$.*

Conversely, suppose $pc[\![P \triangleright \Gamma]\!]^\rho \triangleright \Delta \xrightarrow{\beta} Q \triangleright \Delta \setminus \beta$. Then there exists P' such that $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ and $pc[\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho\setminus obj(\beta)} = Q$.

The full abstraction is then a corollary.

Theorem A.7.3 (Full abstraction)**.** *We have $P \triangleright \Gamma \approx P' \triangleright \Gamma$ if and only if for some $\rho, \rho', \Delta, \Delta'$ we have $pc[\![P \triangleright \Gamma]\!]^\rho \triangleright \Delta \approx pc[\![P' \triangleright \Gamma]\!]^{\rho'} \triangleright \Delta'$.*

Recall that in NCCS we use bisimilarity up to renaming.

A.7.3 Event structure semantics of the π-calculus

By composing the translation obtained in this section with the event structure semantics of Section A.6, we obtain an event structure semantics of the π-calculus.

Given a π-calculus judgment $P \triangleright \Gamma$, we define

$$[\![P \triangleright \Gamma]\!]^\rho_\Delta = [\![pc[\![P \triangleright \Gamma]\!]^\rho \triangleright \Delta]\!]$$

We thus have

Lemma A.7.4. *For every judgment $P \triangleright \Gamma$ in the π-calculus, there exist ρ and Δ such that $[\![P \triangleright \Gamma]\!]^\rho_\Delta$ is defined. When this is the case $[\![P \triangleright \Gamma]\!]^\rho_\Delta$ is a confusion free event structure, and $[\![P \triangleright \Gamma]\!]^\rho_\Delta \triangleright \Delta$.*

Proposition A.7.5 (Soundness)**.** *Suppose that for some* $\rho, \rho', \Delta, \Delta'$, $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta} = [\![P' \triangleright \Gamma]\!]^{\rho'}_{\Delta'}$. *Then* $P \triangleright \Gamma \approx P' \triangleright \Gamma$.

Note that the event structure semantics of CCS is already not fully abstract with respect to bisimulation [Win82], hence the other direction does not hold in our case either.

However, there is another kind of correspondence between the labelled transition systems and the event structures, analogous to the one discussed in Section A.6.4. Combining Theorem A.6.7 with Theorem A.7.2, we obtain :

Theorem A.7.6. *Suppose* $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ *in the* π*-calculus, and that* $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta}$ *is defined. Then* $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta} \xrightarrow{\beta} \cong [\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho \setminus obj(\beta)}_{\Delta \setminus \beta}$.

Conversely, suppose $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta} \xrightarrow{\beta} \mathcal{E}'$. *Then there exists* P' *such that* $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ *and* $[\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho \setminus obj(\beta)}_{\Delta \setminus \beta} \cong \mathcal{E}'$.

A.8 A probabilistic π-calculus

A.8.1 Syntax and Operational Semantics

In this section we modify the π-calculus as presented in A.2, by replacing the nondeterministic choice on output with a probabilistic choice. As in the non probabilistic case, input is similar to the "case" construct and selection is "injection" in the typed λ-calculi. The formal grammar of the calculus is defined below with $p_i \in [0, 1]$.

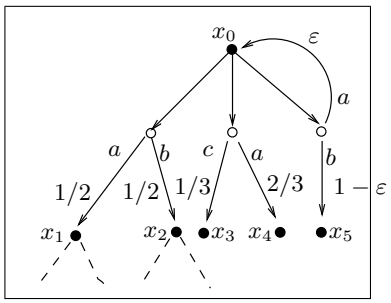
$$P ::= x \,\&_{i \in I}\, \mathtt{in}_i(\tilde{y_i}).P_i \mid \overline{x} \bigoplus_{i \in I} p_i \mathtt{in}_i(\tilde{y_i}).P_i \mid P|Q \mid (\boldsymbol{\nu}\, x)P \mid \mathbf{0} \mid\, !x(\tilde{y}).P$$

The construct $\overline{x} \bigoplus_{i \in I} p_i \mathtt{in}_i(\tilde{y_i}).P_i$ is a probabilistic selecting output, and the events are given probability denoted by the p_i, with the requirement that $\sum_{i \in I} p_i = 1$.

To give an operational semantics to the probabilistic π-calculus we use *Segala automata*, a model that combines probability and nondeterminism. Segala automata can be seen as an extension both of Markov chains and of labelled transition systems. They were introduced by Segala and Lynch [SL95, Seg95]. A recent presentation of Segala automata can be found in [Sto02]. The name "Segala automata" appears first in [BSV03]. It is non standard in the literature, but we prefer it to the more common, but ambiguous, "probabilistic automata".

A *probability distribution* over a finite or countable set X is a function $\xi : X \to [0, 1]$ such that $\sum_{x \in X} \xi(x) = 1$. The set of probability distributions over X is denoted by $V(X)$. By $\mathcal{P}(X)$, we denote the powerset of X.

A *Segala automaton* over a set of labels A is given by a finite or countable set of states X together with a transition function $t : X \to \mathcal{P}(V(A \times X))$. This model represents a process that, when it is in a state x, nondeterministically chooses a probability distribution ξ in $t(x)$ and then performs action a and enters in state y with probability $\xi(a, y)$.



In the initial state x_0 there are three possible transition groups, corresponding to its three hollow children. The left-most transition group is $x_0\{ \xrightarrow[p_i]{a_i} x_i\}_{i\in I}$ where $I = \{1, 2\}$, $a_1 = a, a_2 = b$ and $p_1 = p_2 = 1/2$. The right-most transition group is $x_0\{ \xrightarrow[p_j]{a_j} x_j\}_{j\in J}$ where $J = \{0, 5\}$, $a_0 = a, a_5 = b$ and $p_0 = \varepsilon, p_5 = 1 - \varepsilon$.

FIGURE A.12 – A Segala automaton

The notation we use comes from [HP00]. Consider a transition function t. Whenever a probability distribution ξ belongs to $t(x)$ for a state $x \in X$ we will write

$$x\{ \xrightarrow[p_i]{a_i} x_i\}_{i\in I} \tag{A.1}$$

where $x_i \in X$, $i \neq j \Rightarrow (a_i, x_i) \neq (a_j, x_j)$, and $\xi(a_i, x_i) = p_i$. Probability distributions in $t(x)$ are also called *transition groups* of x.

A good way of visualising probabilistic automata is by using alternating graphs [Han91]. In Figure A.12, black nodes represent states, hollow nodes represent transition groups.

The operational semantics of the probabilistic version of the π calculus is given in terms of Segala automata. The rules for deriving the transitions are presented in Figure A.13.

In particular, the selecting output synchronises with the branching input, and a synchronisation step takes place, with the probability chosen by the output process.

A.8.2 Linear types for the probabilistic π-calculus

The rules defining typing judgements $P \triangleright \Gamma$ are identical to the non-probabilistic case, except a straightforward modification to deal with the generative output, which is still basically the same rule as for confusion free processes, without any additional complexity due to the probability.

As in the nonprobabilistic case, we obtain a typed version of the operational semantics by restricting the actions that are not allowed by the type environment.

The typed automaton, $P \triangleright \Gamma\{ \xrightarrow[p_i]{\beta_i} P_i \triangleright \Gamma_i\}_{i\in I}$, is defined by adding the following constraint :

$$\text{if } P\{ \xrightarrow[p_i]{\beta_i} P_i\}_{i\in I} \text{ and } \Gamma \text{ allows } \beta_i \text{ for all } i \in I \quad \text{then} \quad P \triangleright \Gamma\{ \xrightarrow[p_i]{\beta_i} P_i \triangleright \Gamma_i\}_{i\in I}$$

$$\overline{x}\bigoplus_{i\in I} p_i \mathtt{in}_i(\tilde{y}_i).P_i\{ \xrightarrow[p_i]{\overline{x}\mathtt{in}_i\langle\tilde{y}_j\rangle} P_i\}_{i\in I} \qquad x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i\{ \xrightarrow[1]{x\mathtt{in}_j\langle\tilde{y}_j\rangle} P_j\}$$

$$!x(\tilde{y}).P\{ \xrightarrow[1]{x\langle\tilde{y}\rangle} P \mid !x(\tilde{y}).P\} \qquad \overline{x}(\tilde{y}).P\{ \xrightarrow[1]{\overline{x}\langle\tilde{y}\rangle} P\}$$

$$\frac{P\{ \xrightarrow[p_i]{\beta_i} P_i\}_{i\in I} \quad subj(\beta_i)\neq x}{(\boldsymbol{\nu}\, x)P\{ \xrightarrow[p_i]{\beta_i} (\boldsymbol{\nu}\, x)P_i\}_{i\in I}} \qquad \frac{P\{ \xrightarrow[p_i]{\beta_i} P_i\}_{i\in I}}{P \mid Q\{ \xrightarrow[p_i]{\beta_i} P_i \mid Q\}_{i\in I}}$$

$$\frac{P\{ \xrightarrow[p_i]{\alpha_i} P_i\}_{i\in I} \quad Q\{ \xrightarrow[1]{\beta_i} Q_i\} \quad obj(\alpha_i)=\tilde{y}}{P \mid Q\{ \xrightarrow[p_i]{\alpha_i\bullet\beta_i} (\boldsymbol{\nu}\,\tilde{y})(P_i \mid Q_i)\}_{i\in I}} \qquad \frac{P\equiv_\alpha P' \quad P\{ \xrightarrow[p_i]{\beta_i} Q_i\}_{i\in I}}{P'\{ \xrightarrow[p_i]{\beta_i} Q_i\}_{i\in I}}$$

FIGURE A.13 – Segala automaton for the probabilistic πI-Calculus

The nature of the typing system is such that for every transition group, either all actions are allowed, or all are not, and therefore the above semantics is well defined.

A.8.3 Example of a probabilistic process

We consider the model of traffic lights from [RP02]. Let a be a driver, and let $\mathtt{in}_{\mathtt{red}}, \mathtt{in}_{\mathtt{yell}}, \mathtt{in}_{\mathtt{green}}$ represent colours of the traffic light. The process $\overline{a}\mathtt{in}_{\mathtt{red}}(y)$ represents the traffic light signalling to the driver it is red, at the same time communicating the name y of the crossing. The behaviour of the driver at the crossing is either braking, staying still, or driving ($\mathtt{in}_{\mathtt{brake}}, \mathtt{in}_{\mathtt{still}}, \mathtt{in}_{\mathtt{drive}}$).

A cautious driver is represented by the process :

$$D_c^a = a \,\&_{i\in\{\mathtt{red},\mathtt{yell},\mathtt{green}\}}\, \mathtt{in}_i(y).P_i \quad \text{with}$$
$$P_{\mathtt{red}} = \overline{y}(0.2\mathtt{in}_{\mathtt{brake}} \oplus 0.8\mathtt{in}_{\mathtt{still}})$$
$$P_{\mathtt{yell}} = \overline{y}(0.9\mathtt{in}_{\mathtt{brake}} \oplus 0.1\mathtt{in}_{\mathtt{drive}})$$
$$P_{\mathtt{green}} = \overline{y}(\mathtt{in}_{\mathtt{drive}})$$

A cautious driver watches what colour the light is and behaves accordingly. If it is red, she stays still, or finishes braking. If it is yellow, most likely she brakes. If it is green, she drives on.

A driver in a hurry is represented by the process

$$D_h^a = a \,\&_{i\in\{\mathtt{red},\mathtt{yell},\mathtt{green}\}}\, \mathtt{in}_i(y).Q_i \quad \text{with}$$
$$Q_{\mathtt{red}} = \overline{y}(0.3\mathtt{in}_{\mathtt{brake}} \oplus 0.6\mathtt{in}_{\mathtt{still}} \oplus 0.1\mathtt{in}_{\mathtt{drive}})$$
$$Q_{\mathtt{yell}} = \overline{y}(0.1\mathtt{in}_{\mathtt{brake}} \oplus 0.9\mathtt{in}_{\mathtt{drive}})$$
$$Q_{\mathtt{green}} = \overline{y}(\mathtt{in}_{\mathtt{drive}})$$

This is similar to the cautious driver, but he is more likely to drive on at red and yellow. In fact, both have the same *type*, they check the light, and they choose a behaviour :

$$D_c^a, D_h^a \triangleright a : \&_{i\in\{\texttt{red},\texttt{yell},\texttt{green}\}}\ (\bigoplus_{j\in\{\texttt{brake},\texttt{still},\texttt{drive}\}}()^{\uparrow})^{\downarrow}$$

where $\&_{i\in I}\ (\tau_i)^{\downarrow}$ is a branching type which inputs a value of type τ_i and $\bigoplus_{i\in I}\ (\tau_i)^{\uparrow}$ is a selection type which selects a branch i with a value of type τ_i. Note that the type actually states that the driver chooses the behaviour *after* seeing the light. We can represent two independent drivers :

$$D2 = (\nu a, a')(\overline{a}\texttt{in}_{\texttt{red}}(y).R \mid D_c^a \mid \overline{a'}\texttt{in}_{\texttt{green}}(y).R \mid D_h^{a'})$$

where $R = y\ \&_{i\in\{\texttt{brake},\texttt{still},\texttt{drive}\}}\ \texttt{in}_i()$ represents the traffic light accepting the behaviour of the driver. We have that $D2$ has two transition groups, corresponding to the two drivers. Note that the typing system guarantees that each driver can perform only one of three actions, i.e. either `brake`, `still` or `drive` at any one time.

A.9 Probabilistic event structure semantics

A.9.1 Untyped probabilistic event structures

We now present the model of probabilistic event structures, that we use to give an alternative semantics to the probabilistic π-calculus. Probabilistic event structures were first introduced by Katoen [Kat96], as an extension of the so called *bundle* event structures. A probabilistic version of *prime* event structures was introduced in [VVW06]. In this work we use prime event structures as we think they are the simplest and easiest to understand of all variants of event structures. Moreover the confluence theorem uses results of [VVW06].

Given a confusion-free event structure, we can associate a probability distribution with some cells. Intuitively it is as if, for every such cell, we have a die local to it, determining the probability with which the events at that cell occur.

We can think of the cells with a probability distribution as *generative*, while the other cells will be called *reactive*. Reactive cells are awaiting a synchronisation with a generative cell in order to be assigned a probability.

Definition A.9.1. Let $\mathcal{E} = \langle E, \leq, \smile\rangle$ be a confusion free event structure, let G be a set of cells of $\mathcal{E}$ and let G' be the set of events of the cells in G. The cells in G are called *generative*. The cells not in G are called *reactive*. A *cell valuation* on $(\mathcal{E}, G)$ is a function $\mu : G' \to [0, 1]$ such that for every $c \in G$, we have $\sum_{e\in c} \mu(e) = 1$. A *partial probabilistic event structure* is a confusion free event structure together with a cell valuation. It is called simply *probabilistic event structure* if $G' = E$.

This definition generalises the definition given in [VVW06], where it is assumed that $G' = E$. Note also that a confusion free event structure can be seen as a probabilistic event structure where the set G is empty.

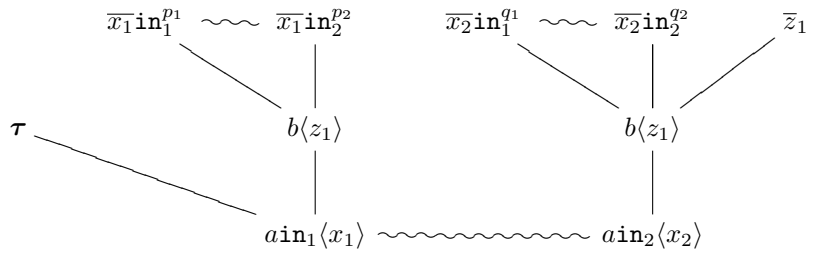


FIGURE A.14 – A typed event structure

Beside the standard operator on event structures, as defined in Section A.3, we also have the *probabilistic prefixed sum* $\sum_{i\in I} p_i a_i.\mathcal{E}_i$, where $\mathcal{E}_i$ are partial probabilistic event structures. This is obtained as adding a new initial generative cell with the condition that the probability of the new initial events are p_i.

All constructors, except the parallel composition, preserve the class of partial probabilistic event structures. The linear typing allows parallel composition to preserve that class.

A.9.2 Typed probabilistic event structures

To type a partial probabilistic event structure, we type it as a non probabilistic event structure. We also make sure that only the branching cells are reactive, as they are waiting to synchronise with a dual selection cell.

Definition A.9.2. Let $\mathcal{E} = \langle E, \leq, \smile, \lambda, G, \mu\rangle$ be a partial probabilistic event structure. We say that $\mathcal{E} \triangleright \Gamma$, if the following conditions are satisfied :
- $\mathcal{E} \triangleright \Gamma$ as for the non-probabilistic case ;
- G includes all cells, except the branching ones.

From the fact that the parallel composition of typed event structures is typed, one can easily derive that the parallel composition of typed probabilistic event structures is still a probabilistic event structure, and that it is typed.

The distinction between reactive branching input and generative selecting output is akin to the one in [ABG04].
Figure A.14 represents a typed (partial) probabilistic event structure $\mathcal{E} \triangleright \Gamma$, where

$$\Gamma \quad = \quad a : \&_{i\in\{1,2\}}(x_i : \oplus_{k\in\{1,2\}}()),\ b : \otimes_{i\in\{1\}}(z_i : \uplus_{k\in\{1\}}())$$

The selection cells $\overline{x_1}\mathtt{in}_1, \overline{x_1}\mathtt{in}_2$ and $\overline{x_2}\mathtt{in}_1, \overline{x_2}\mathtt{in}_2$ are generative. The branching cell $a\mathtt{in}_1\langle x_1\rangle, a\mathtt{in}_2\langle x_2\rangle$ is reactive. Every other cell is generative, and contains only one event, that has probability 1. We can see that the causality in $\mathcal{E}$ refines the name causality in Γ : for instance, Γ forces the labels with subject x_i

$$
\begin{aligned}
[\![0 \triangleright x_i : (\tau_i)^{\mathbf{?}}, y_j : \updownarrow]\!]^{\rho} &= \varnothing \\
[\![(\boldsymbol{\nu}\, x)P \triangleright \Gamma]\!]^{\rho} &= [\![P \triangleright \Gamma, x : \tau]\!]^{\rho} \setminus x \\
[\![P_1 \| P_2 \triangleright \Gamma_1 \odot \Gamma_2]\!]^{\rho_1 \cup \rho_2} &= \\
&\quad ([\![P_1 \triangleright \Gamma_1]\!]^{\rho_1} \| [\![P_2 \triangleright \Gamma_2]\!]^{\rho_2}) \setminus S \\
[\![\overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \bigoplus_{i\in I} (\tilde{\tau}_i)^{\uparrow}]\!]^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} &= \\
&\quad \textstyle\sum_{i\in I} \overline{x}\mathtt{in}_i\langle \tilde{z}_i\rangle.[\![P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i]\!]^{\rho} \\
[\![x \,\&_{i\in I}\, p_i \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I} (\tilde{\tau}_i)^{\downarrow}]\!]^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} &= \\
&\quad \textstyle\sum_{i\in I} p_i x\mathtt{in}_i\langle \tilde{z}_i\rangle.[\![P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i]\!]^{\rho} \\
[\![!x(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{!}]\!]^{\rho[K],\tilde{y}\to\{\tilde{y}_k\}_{k\in K}} &= \\
&\quad \textstyle\prod_{k\in K} x\langle \tilde{y}^k\rangle.[\![P \triangleright \Gamma]\!]^{\rho}[\tilde{y}^k/\tilde{y}][Y^k/Y] \\
[\![\overline{x}(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}]\!]^{\rho,\tilde{y}\to\tilde{w}} &= \\
&\quad \overline{x}\langle \tilde{w}\rangle.[\![P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}[\tilde{w}/\tilde{y}]]\!]^{\rho}
\end{aligned}
$$

FIGURE A.15 – Event Structure Semantics of the probabilistic π-Calculus

to be above the label $a\mathtt{in}_i\langle x_i\rangle$, but does not force the causal link between the events labelled by $a\mathtt{in}_i\langle x_i\rangle$ and $b\langle z_1\rangle$.

A.9.3 Semantics of the calculus

We now present the event structure semantics of the π-calculus and its properties. As before, the semantics is given by a family of partial functions $[\![-]\!]^{\rho}$, parametrised by a choice function ρ, that take a judgement of the π-calculus and return an event structure.

It is a slight modification of the nonprobabilistic case, to take into account the generative cells.

As in the non-probabilistic case, we have the following result.

Theorem A.9.3. *For every judgement $P \triangleright \Gamma$ in the π-calculus, there exist a choice function ρ and a type environment Δ such that $[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$. Moreover, for every injective fresh renaming ρ', if $[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$ then $[\![P \triangleright \Gamma]\!]^{\rho' \circ \rho} \triangleright \Delta[\rho']$.*

Theorem A.9.4. *Let P be a process and Γ an environment such that $P \triangleright \Gamma$. Suppose that $[\![P \triangleright \Gamma]\!]^{\rho}$ is defined. Then there is a environment Δ such that $[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$.*

This theorem means that all denoted event structures are indeed partial probabilistic event structures. Note that the set of generative cells includes all synchronisation cells. Therefore a closed process denotes a probabilistic event structure.

Corollary A.9.5. *The event structure* $[\![P \triangleright \varnothing]\!]^\rho$ *is a probabilistic event structure.*

This implies that there exists a unique probability measure over the set of maximal runs [VVW06]. In other words, for closed processes, the scheduler only influences the order of independent events, in accordance with the intuition that probabilistic choices are local and not influenced by the environment.

A.10 Event structures and Segala automata

In this section we show a formal correspondence between Segala automata and probabilistic event structures. We first introduce the notion of scheduler.

A.10.1 Runs and schedulers

An *initialised* Segala automaton, is a Segala automaton together with an initial state x_0. A *finite path* of an initialised Segala automaton is an element in $(X \times V(X \times A) \times A)^* X$, written as $x_0\xi_1 a_1 x_1 \dots \xi_n a_n x_n$, such that $\xi_{i+1} \in t(x_i)$. An *infinite* path is defined in a similar way as an element of $(X \times V(X \times A) \times A)^\omega$.

The probability of a finite path $\tau := x_0\xi_1 a_1 x_1 \dots \xi_n a_n x_n$ is defined as

$$\Pi(\tau) = \prod_{1 \le i \le n} \xi_i(a_i, x_i)\,.$$

The last state of a finite path τ is denoted by $l(\tau)$. A path τ is *maximal* if it is infinite or if $t(l(\tau)) = \varnothing$.

A *scheduler* for a Segala automaton with transition function t is a partial function $\mathcal{S} : (X \times V(X \times A) \times A)^* X \to V(X \times A)$ such that, if $t(l(\tau)) \neq \varnothing$ then $\mathcal{S}(\tau)$ is defined and $\mathcal{S}(\tau) \in t(l(\tau))$. A scheduler chooses the next probability distribution, knowing the history of the process. Using the representation with alternating graphs, we can say that, for every path ending in a black node, a scheduler chooses one of his hollow sons.

Given an (initial) state $x_0 \in X$ and a scheduler $\mathcal{S}$ for t, we consider the set $\mathcal{B}(t, x_0, \mathcal{S})$ of maximal paths, obtained from t by the action of $\mathcal{S}$. Those are the paths $x_0\xi_1 a_1 x_1 \dots \xi_n a_n x_n$ such that $\xi_{i+1} = \mathcal{S}(x_0\xi_1 a_1 x_1 \dots \xi_i a_i x_i)$. The set of maximal paths is endowed with the σ-algebra generated by the finite paths. A scheduler induces a probability measure on such σ-algebra as follows : for every finite path τ, let $K(\tau)$ be the set of maximal paths extending τ. Define $\zeta_{\mathcal{S}}(K(\tau)) := \Pi(\tau)$, if $\tau \in \mathcal{B}(t, x_0, \mathcal{S})$, and 0 otherwise. It can be proved [Seg95] that $\zeta_{\mathcal{S}}$ extends to a unique probability measure on the σ-algebra generated by the finite paths.

Given a set of labels $B \subseteq A$ we define $\zeta_{\mathcal{S}}(B)$ to be $\zeta_{\mathcal{S}}(Z)$, where Z is the set of all maximal paths containing some label from B.

A.10.2 From event structures to Segala automata

Definition A.10.1. Let $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$ be a labelled event structure and let e be one of its minimal events. The event structure $\mathcal{E}\lfloor e = \langle E', \leq', \smile', \lambda' \rangle$ is defined by : $E' = \{e' \in E \mid e' \not\geq e\}$, $\leq' = \leq_{|E'}$, $\smile' = \smile_{|E'}$, and $\lambda' = \lambda_{|E'}$.

Roughly speaking, $\mathcal{E}\lfloor e$ is $\mathcal{E}$ minus the event e, and minus all events that are in conflict with e. We can then generate a Segala automaton on event structures as follows :

$$\mathcal{E}\{ \xrightarrow[p_i]{a_i} \mathcal{E}\lfloor e_i\}_{i \in I}$$

if there exists a minimal generative cell $c = \{e_i \mid i \in I\}$, such that $\mu(e_i) = p_i$ and $\lambda(e_i) = a_i$. We also put

$$\mathcal{E}\{ \xrightarrow[1]{a} \mathcal{E}\lfloor e\}$$

if there exists an event e belonging to a minimal reactive cell, such that $\lambda(e) = a$. The initialised Segala automaton generated by an event structure $\mathcal{E}$ is the above automation initialised at $\mathcal{E}$.

A probabilistic event structure (where every cell is generative) generates a somewhat "deterministic" Segala automaton. The general formalisation of this property requires several technicalities (see [VVW06], for instance). Here we state a simplified result.

Let $\mathcal{E}$ be a probabilistic event structure, and consider the Segala automaton (t, x_0), generated as above. Consider a scheduler $\mathcal{S}$ for such a Segala automaton. We say that $\mathcal{S}$ is *fair* if for every path $\tau \in \mathcal{B}(t, x_0, \mathcal{S})$, there does not exist a generative cell c of the event structure, and an index j, such that for all $i > j$, the transition group corresponding to c is enabled but it is not chosen by $\mathcal{S}$.

Theorem A.10.2. *Let $\mathcal{E}$ be a probabilistic event structure, and consider the corresponding Segala automaton. For all sets of labels B, and for all fair schedulers $\mathcal{S}, \mathcal{T}$, we have $\zeta_{\mathcal{S}}(B) = \zeta_{\mathcal{T}}(B)$.*

In a non-probabilistic confluent system, all (fair) resolutions of the nondeterministic choices give rise to the same set of events, possibly in different order. In this sense we can see Theorem A.10.2 as expressing probabilistic confluence.

Figure A.16 shows an example of a (partial) probabilistic event structure. The generative cells are $\{\alpha', \alpha''\}, \{\beta'', \gamma''\}$ and the probability is indicated as superscript of the label. Figure A.17 shows the Segala automaton corresponding to the event structure of Figure A.16.

A.10.3 The adequacy theorem

The correspondence between the two semantics of the π-calculus is formalised by the following theorem.

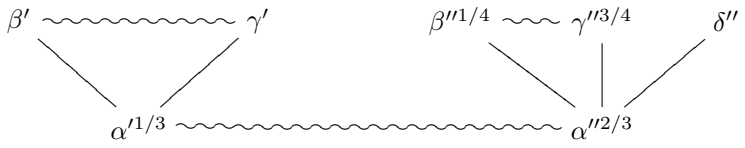


FIGURE A.16 – A probabilistic event structure

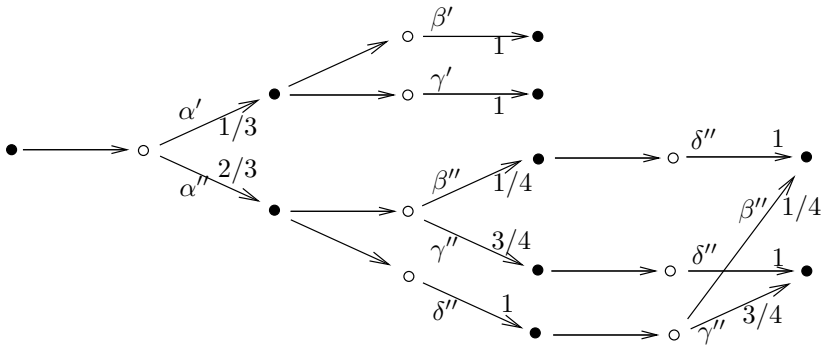


FIGURE A.17 – The corresponding Segala automaton

Theorem A.10.3. *Let* $\cong$ *denote isomorphism of probabilistic event structures.*

Suppose $P \triangleright \Gamma\{ \xrightarrow[p_i]{\beta_i} P_i \triangleright \Gamma_i\}_{i\in I}$ *in the* π*-calculus. Then there exist* ρ, ρ_i *such that* $[\![P \triangleright \Gamma]\!]^\rho$ *is defined and* $[\![P \triangleright \Gamma]\!]^\rho\{ \xrightarrow[p_i]{\beta_i} \cong [\![P_i \triangleright \Gamma_i]\!]^{\rho_i}\}_{i\in I}$.

Conversely, suppose $[\![P \triangleright \Gamma]\!]^\rho\{ \xrightarrow[p_i]{\beta_i} \mathcal{E}_i\}_{i\in I}$ *, for some* ρ*. Then there exist* P_i, ρ_i *such that* $P \triangleright \Gamma\{ \xrightarrow[p_i]{\beta_i} P_i \triangleright \Gamma \setminus \beta_i\}_{i\in I}$ *and* $[\![P_i \triangleright \Gamma \setminus \beta_i]\!]^{\rho_i} \cong \mathcal{E}_i$*, for all* $i \in I$*.*

The proof is completely analogous to the one for the non-probabilistic case by induction on the operational rules, the difficult case being the parallel composition.

A.10.4 Example of probabilistic confluence

Theorem A.10.3 and Theorem A.10.2 together show that the linearly typed probabilistic π-calculus is "probabilistically confluent". Note that Theorem A.10.2 applies only to *fully* probabilistic event structures, that is event structures which do not contain reactive cells. In particular, in light of Corollary A.9.5, it applies

to closed processes. More generally it applies to processes whose free names do not include linear inputs.

To exemplify the confluence theorem, consider a process P such that $P \triangleright a : \bigoplus_{i \in I} \varnothing$. This is a process that emits only one visible action, whose subject is a. For every $j \in I$ we can define the probability P emits $a\mathtt{in}_j$ as $p_{\mathcal{S}}(a\mathtt{in}_j)$ for some fair scheduler $\mathcal{S}$. By Theorem A.10.2, we have that this probability is independent from the scheduler, so we can define it as $p(a\mathtt{in}_j)$. This independence from the scheduling policy is what we call probabilistic confluence.

Note also that it can be shown that $\sum_{i \in I} p(a\mathtt{in}_i) \leq 1$. When the inequation is strict, the missing probability is the probability that the process does not terminate. This reasoning relies on the typing in that there exist untyped processes that are not probabilistically confluent. For instance consider

$$(\nu b)(\overline{b} \mid b.\overline{a} \bigoplus_{i \in \{1,2\}} p_i \mathtt{in}_i \mid b.\overline{a} \bigoplus_{i \in \{1,2\}} q_i \mathtt{in}_i)$$

The above process also emits only one visible action, whose subject is a. The probability of $\overline{a}\mathtt{in}_1$ is p_1, or q_1, depending on which synchronisation takes place, i.e. depending on the scheduler. Note, however, that this process is not typable.

A.11 Conclusions and related work

In this chapter we have provided a typing system for event structures and exploited it to give an event structure semantics of the π-calculus. As far as we know, this work offers the first formalisation of a notion of types in event structures, and the first direct event structure semantics of the π-calculus.

The work is quite technical and it requires a little effort to be read. The readers may ask themselves what they gain from this effort. We think the main contributions are as follows.

- It is a standard intuition that confluence means absence of conflict, determinism. In this work we have *formalised* this intuition. In the process of this formalisation some conflict situations that are *hidden* by the interleaving semantics were discovered. This fact can be underlined by noting that the standard event structure semantics of the so called *confluent* CCS [Mil89] is *not* conflict free.
- Similarly, we have formalised the notion of *probabilistic confluence*, and we have shown that the same typing system that guarantees confluence in the non probabilistic case can be used to enforce probabilistic confluence.
- It is well known how to compose event structures in order to obtain event structures. However it was not known how to compose confusion free event structures in order to obtain confusion free event structures. Our work offers a solution to this problem. Concrete data structures, a fundamental concept in various fields of semantics, can be seen as confusion free event structures. Therefore our work also shows how to compose concrete data structures.
- Although several causal semantics of the π-calculus exist (see related work below), no one ever gave a *direct* event structure semantics, that could be

seen as an extension of Winskel's semantics of CCS. We believe the main difficulty of an event structures semantics of the π-calculus lies in the handling of name generation. Name generation is an inherently *dynamic* operation, while event structures have a more *static*, denotational flavour. We have shown that, by restricting the amount of concurrency to that permitted by the linear type discipline, we can deal with name generation statically, and thus we can extend Winskel's semantics. This restricted π-calculus is still very expressive, in that it can encode fully abstractly functional programming languages.

A.11.1 Possible lines of research

The typed λ-calculus can be encoded into the typed π-calculus. This provides an event structure semantics of the λ-calculus, that we want to study in detail. Also the types of the λ-calculus are given an event structure semantics. We aim at comparing this "true concurrent" semantics of the λ-types with concurrent games [Mel05], and with ludics nets [FM05].

An event structure *terminates* if all its maximal configurations are finite. It would be interesting to study a typing system of event structures that guarantees termination applying the idea of the strongly normalising typing system of the π-calculus [YBH01].

In the probabilistic case we have shown a correspondence between event structures and Segala automata. We would have liked to extend this correspondence to a categorical adjunction between two suitable categories, ideally extending the setting presented in [WN95]. It is possible to do so, by a simple definition of morphisms for Segala automata, and by extending the notion of probabilistic event structures. Unfortunately neither category has products, which are used in [WN95] to define parallel composition. The reason for this is nontrivial and it is has to do with the notion of stochastic correlation, a phenomenon already discussed in [VVW06] in the context of true concurrent models. This issue needs to be investigated further.

The linearly typed π-calculus is the target of a sound and complete encodings of functional language [BHY01, YBH01]. Our traffic light example in Section 5 suggests that our calculus captures the core part of the expressiveness represented by the Stochastic Lambda Calculus [RP02]. We plan to perform similar encodings in the probabilistic version, notably the probabilistic functional language [RP02], probabilistic λ-calculus [DHW05] and Probabilistic PCF [DH02]. Since the linear type structures are originated from game semantics [HO00], this line of study would lead to a precise expressive analysis between the probabilistic event structures, Segala automata, probabilistic programming languages and probabilistic game semantics [DH02], bridged by their representations of or encodings into probabilistic π-calculi. Finally, there are connections between event structures, concurrent games [Mel04], and ludics [FM05, FP07a] that should be investigated also in the presence of probabilities.

A.11.2 Related work

There are several causal models for the π-calculus, that use different techniques. In [BS98, DP99], the causal relations between transitions are represented by "proofs" of the transitions which identify different occurrences of the same transition. In our case a similar role is played by names in types. In [CS00], a more abstract approach is followed, which involves indexed transition systems. In [JJ95], a semantics of the π-calculus in terms of pomsets is given, following ideas from dataflow theory. The two papers [BG95, Eng96] present Petri nets semantics of the π-calculus. Since we can unfold Petri nets into event structures, these could indirectly provide event structure semantics of the π-calculus. In [BCM99], an event structure unfolding of double push-out rewriting systems is studied, and this could also indirectly provide an event structure semantics of the π-calculus, via the double push-out semantics of the π-calculus presented in [MP95]. In [BS01], Petri Nets are used to provide a type theory for the Join-calculus, a language with several features in common with the π-calculus. None of the above semantics directly uses event structures and no notion of compositional typing systems in true concurrent models is presented. In addition, none of them is used to study a correspondence between semantics and behavioural properties of the π-calculus in our sense.

A recent work [BMM06] claims to provide an event structure semantics of the full π-calculus. However they cater only for the *reduction* semantics. Consequently their semantics is not compositional, nor it is an extension of Winskel's semantics of CCS.

In [Win05b], event structures are used in a different way to give semantics to a process language, a kind of value passing CCS. That technique does not apply yet to the π-calculus where we need to model creation of new names, although recent work [Win05a] is moving in that direction.

Some interesting results connecting game semantics and event structures can be found in [FP07b]. A fundamental work on the connections between the linear π-calculus and polarised linear logic is [HL10]. See also [FP07c].

Infinite behaviour is introduced in our version of CCS by means of the infinite parallel composition. Infinite parallel composition is similar to replication in that it provides infinite behaviour "in width" rather that "in depth". Recent studies on recursion versus replication are [BGZ03, GSV04].

For the probabilistic case, the natural comparison is with the probabilistic π-calculus by Herescu and Palamidessi [HP00]. Their and our calculi both have a semantics in terms of Segala automata, while we also provide an event structure semantics. The key of our construction is the typing system, which allows us to stay within the class of probabilistic event structures.

Our typing system is designed to provide a "probabilistically confluent" calculus, and therefore their calculus is more expressive, as it allows non-confluent computations. At the core of their calculus, there is a renormalisation of probabilities, which is absent in our setting, i.e. in our calculus, all probabilistic choices are local, and are not influenced by the environment.

A simpler calculus, without renormalisation, is presented in [CP05]. This

version is very similar to ours, in that all choices are local; in fact, the protocol example presented in [CP05] (via an encoding into our calculus) is linearly typable. We believe we could apply a typing system similar to ours to the calculus in [CP05], prove the same results in this paper and identify a good class of probabilistic name-passing behaviours.

A.12 Proofs

A.12.1 Proof of Lemma A.3.6

We prove it by induction on the joint size of x, x'. The base case is vacuously true. Now take $(x, e_1, e_2), (x', e_1, e_2) \in E$ with $x \neq x'$. Since x, x' are downward closed sets, if their maximal elements coincide, they coincide. Therefore, w.l.o.g. there must be a maximal element $(y, d_1, d_2) \in x$ such that $(y, d_1, d_2) \notin x'$. By definition of E, and without loss of generality, we can assume that $d_1 \in parents(e_1)$. Therefore, by definition of E, there must be a $(y', d_1, d_2') \in x'$. Suppose $d_2 \neq d_2'$. Then by definition of conflict $(y, d_1, d_2) \smile (y', d_1, d_2')$. If $d_2 = d_2'$ then it must be $y \neq y'$. Then by induction hypothesis there exist $f \in y, f' \in y'$ such that $f \smile f'$. And since x, x' are downward closed, we have $f \in x, f' \in x'$.

A.12.2 Proof of Theorem A.3.7

Recall the the definition of $(E, \leq, \smile)$. In order to show that it is an event structure, we first o have to show that the relation $\leq$ is a partial order. We have that

- it is reflexive by construction;
- it is antisymmetric : suppose $e' \leq e = (x, e_1, e_2)$. If $e' \neq e$, then, by construction $h(e') < h(e)$, so that it cannot be $e \leq e'$.
- it is transitive : suppose $e' \leq e \leq d = (y, d_1, d_2)$. This means that $e \in y$. Since, by construction, y is downward closed, this means that $e' \in y$, so that $e' \leq d$.

Next, for every event $e = (x, e_1, e_2)$, we have that $[e)$ is finite, as it coincides with x.

Then we need to show that the conflict is irreflexive and hereditary. It is hereditary essentially by definition : suppose $e := (x, e_1, e_2) \smile d := (y, d_1, d_2)$, and let $d \leq d' := (y', d_1', d_2')$. By considering all the cases of the definition of $e \smile d$, we derive $e \smile d'$. For instance, suppose there exists $e' := (x', e_1', e_2') \leq e$ such that $e_1' \asymp d_1$, and $e' \neq d$. This means that $e' \smile d$. Notice that $e' \leq e$, and $d \leq d'$. By the fourth clause of the definition, $e \smile d'$. The other cases are analogous.

To prove that the conflict relation is irreflexive, suppose $(x, e_1, e_2) \smile (x, e_1, e_2)$. There are two possible ways of deriving this. First, if there are $e, d \in x$ such that $e \smile d$, but this contradicts the fact that x is a configuration. The other possibility is that, there exist $(x', e_1', e_2') \in x$ such that $(x', e_1', e_2') \smile (x, e_1, e_2)$.

Take a minimal such. Then it must be $e'_1 \asymp e_1$ or $e'_2 \asymp e_2$. But this contradicts the definition of E.

Now we have to show that such event structure is the categorical product of $\mathcal{E}_1, \mathcal{E}_2$. First thing to show is that projections are morphisms. Using Proposition A.3.5, it is enough to show that they reflect reflexive conflict and preserves downward closure.

- Take $e, e' \in E$ and suppose by that $\pi_1(e) \asymp \pi_1(e')$. Then, by definition we have $e \asymp e'$.
- To show that π_1 preserves downward closure let $e = (x, e_1, e_2)$ suppose $e'_1 \leq e_1 = \pi_1(e)$. Then we show that there is a $e' \leq e$ such that $\pi_1(e') = e'_1$. By induction on the height of e : the basis is vacuously true, since e_1 is minimal. For the step, consider first the case where $e'_1 \in \mathit{parents}(e_1)$. Then, by definition of E, we have that there exists $e' = (x', e'_1, e'_2) \in x$. Therefore $e' \leq e$ and $\pi_1(e') = e'_1$. If $e'_1 \notin \mathit{parents}(e_1)$, then there is a $e''_1 \in \mathit{parents}(e_1)$ such that $e'_1 \leq e''_1 \leq e_1$ so that there is $e'' = (x'', e''_1, e''_2) \in x$. By induction hypothesis there is $e' \in x''$ such that $\pi_1(e') = e'_1$. And by transitivity, $e' \leq e$.

Now we want to show that $\mathcal{E}$ enjoys the universal property that makes it a categorical product. That is for every event structure $\mathcal{D}$, such that there are morphisms $f_1 : \mathcal{D} \to \mathcal{E}_1, f_2 : \mathcal{D} \to \mathcal{E}_2$, there exists a unique $f : \mathcal{D} \to \mathcal{E}$ such that $\pi_1 \circ f = f_1$ and $\pi_2 \circ f = f_2$.

Clearly, if such f exists, it must be defined as $f(d) = (x, f_1(d), f_2(d))$, for some x. By this we mean $f(d) = (x, f_1(d), *)$, if $f_2(d)$ is undefined, $f(d) = (x, *, f_2(d))$, if $f_1(d)$ is undefined, and undefined if both are undefined. We now define x, by induction on the size of $\lceil d \rceil$. Suppose d is minimal. Then, since f_1, f_2 are morphisms and thus preserve downward closure, we have that $f_1(d), f_2(d)$ are both minimal. Since every maximal element of x must contain the parent of at least one of them, the only possibility is that x be empty.

Putting $f(d) = (\varnothing, f_1(d), f_2(d))$, we obtain, that, on element of height 0,

- $f(d)$ is uniquely defined : we have seen that all choices are forced
- f reflects reflexive conflict : suppose $(\varnothing, f_1(d), f_2(d)) \asymp (\varnothing, f_1(d'), f_2(d'))$, then either $f_1(d) \asymp f_1(d')$ or $f_2(d) \asymp f_2(d')$. In the first case, since f_1 is a morphism, and thus reflects reflexive conflict, we have $d \asymp d'$. Symmetrically for the other case.
- f preserves downward closure vacuously

Now suppose f is uniquely defined for all elements of height less or equal than n, it reflects reflexive conflict and preserves downward closure. Consider d of height $n+1$. We want to define $f(d) = (x, f_1(d), f_2(d))$. Define x as follows. For a set A, let $\downarrow A$ be the downward closure of A. Let $X = \{f(d') \mid d' < d \,\&\, [\ f_1(d') \in \mathit{parents}(f_1(d)) \text{ or } f_2(d') \in \mathit{parents}(f_2(d))\ ]\}$ and define x as $\downarrow X$. First of all we should check that this is indeed an element of E. x is downward closed by definition. It is finite because X is and each element of X has finitely many predecessors. Suppose there are $d', d'' < d$ such that $f(d') \smile f(d'')$. We know by induction that f reflects reflexive conflict on elements of height smaller than d, which means that $d' \smile d''$, contradiction.

Now the maximal elements of x contain either a parent of $f_1(d)$ or a parent of $f_2(d)$ by construction. Take a parent e_1 of $f_1(d)$. I claim that e_1 is of the form $f_1(d')$ for some $d' < d$. Since $e_1 \in \mathit{parents}(f_1(d))$, in particular $e_1 \leq f_1(d)$. since f_1 preserves downward closure, there must exists d' as above. Thus all parents are represented in X. Finally, suppose there is $(z, e_1, e_2) \in x$ such that $e_1 \asymp f_1(d)$ or $e_2 \asymp f_2(d)$. If $(z, e_1, e_2) \in X$, then $(z, e_1, e_2) = f(d')$ for some $d' < d$. So that $e_1 = f_1(d')$, and $e_2 = f_2(d')$. Since f_1, f_2 reflect reflexive conflict, we would have $d' \smile d$, contradiction. Otherwise there must be $f(d') \in X$ such that $(z, e_1, e_2) < f(d')$. Since f preserves downward closure on elements of height less or equal than n, there must be $d'' < d'$ such that $f(d'') = (z, e_1, e_2)$. As above we conclude $d'' \smile d$, contradiction.

Thus putting $f(d) = (x, f_1(d), f_2(d))$, we have that f is well defined on d. Moreover

- $f(d)$ is uniquely defined : suppose we have another possible x. Since f must preserve downward closure, for all $e \in x$, we have that $e = f(d')$ for some $d' < d$. Now, suppose there is an element $f(d') \in X$ which is not in x. W.l.o.g assume that $f_1(d') \in \mathit{parents}(f_1(d))$. Then, there must be an element $e' = (y, f_1(d'), d'_2)$ maximal in x. By the observation above it must be $e' = f(d')$, contradiction.
- f preserves downward closure : take d, and consider $e \leq f(d)$. By construction, either $e \in X$, in which case we have $e = f(d')$ for some $d' < d$, of $e \leq e' \in X$, in which case we have $e' = f(d')$ for $d' < d$. Since, by induction f preserves downward closure, we have $e = f(d'')$ for $d'' < d' < d$.
- f reflects reflexive conflict : suppose $(x, f_1(d), f_2(d)) \asymp (x', f_1(d'), f_2(d'))$, then
  - either $f_1(d) \asymp f_1(d')$ or $f_2(d) \asymp f_2(d')$. In either case, since f_1, f_2 reflects reflexive conflict, we have $d \asymp d'$.
  - there exists $(x'', e_1, e_2) \leq (x', f_1(d'), f_2(d')$, such that $f_1(d) \asymp e_1$ or $f_2(d) \asymp e_2$. Since f preserves downward closure, we have $(x'', e_1, e_2) = f(d'')$ for some $d'' < d'$ and we reason as above.
  - the symmetric case is similar
  - there exists $(y, e_1, e_2) \leq (x, f_1(d), f_2(d))$ and there exists $(y', e'_1, e'_2) \leq (x', f_1(d'), f_2(d'))$, and the reasoning is as above, using that f preserves downward closure.

Thus f is a morphism, is uniquely defined for every $d \in D$, and commutes with the projections. This concludes the proof.

A.12.3 Proof of Proposition A.4.1

Consider a minimal element of $[\![\Gamma_1]\!]$.

- If it synchronises, by the condition on the definition of $\Gamma_1 \odot \Gamma_2$, it must synchronise with a dual minimal element in $[\![\Gamma_2]\!]$. Every event above these two events is either a $\boldsymbol{\tau}$, or it is not allowed, therefore it is deleted by the restriction.
- If it does not synchronise it is left alone, with all above it not synchronising either, and not being restricted.

We can think of $[\![\Gamma_1 \odot \Gamma_2]\!]$, as a disjoint union of $[\![\Gamma_1]\!]$, $[\![\Gamma_2]\!]$, plus some hiding.

A.12.4 Proof of Lemma A.4.3

Suppose $\mathcal{E} \triangleright \Gamma$, witnessed by a morphism $f : \mathcal{E} \to [\![\Gamma]\!]$.

- Let $e, e' \in E$ be such that $\lambda(e) = \lambda(e') \neq \boldsymbol{\tau}$. Therefore, by uniqueness of the labels in $[\![\Gamma]\!]$, $f(e) = f(e')$, and since f reflects reflexive conflict, we have $e \smile e'$.
- A similar reasoning applies for the case when $\lambda(e) = a\mathtt{in}_i\langle\tilde{x}\rangle$ and $\lambda(e') = a\mathtt{in}_j\langle\tilde{y}\rangle$. Then $f(e), f(e')$ belong to the same cell, and thus they are in conflict. Since f reflects conflict, we have $e \smile e'$.
- Suppose $E \triangleright \Gamma$, and let $e, e' \in E$ be such that $e \smile_\mu e'$. Then they belong to the same cell, and by definition they must have same subject but different branch.

A.12.5 Proof of Theorem A.4.4

Define $\Gamma = \Gamma_1 \odot \Gamma_2$ Suppose $\mathcal{E}_1 \triangleright \Gamma_1$, and $\mathcal{E}_2 \triangleright \Gamma_2$. Let $\mathcal{E} = (\mathcal{E}_1 \| \mathcal{E}_2) \setminus Dis(\Gamma)$. We invite the reader to review the definition of the product of event structures, and the consequent definition of parallel composition.

Lemma A.12.1. *Let $(x, e_1, e_2), (y, d_1, d_2)$ be two events in $\mathcal{E}$. Suppose $(x, e_1, e_2) \smile (y, d_1, d_2)$. Then there exists $(x', e_1', e_2') \leq x, (y', d_1', d_2') \leq y$ such that either $e_1' \smile_\mu d_1'$ or $d_1' \smile_\mu d_2'$.*

We check this by cases, on the definition of conflict.

- $e_1 \smile d_1$. In this case there must exists $e_1' \leq e_1$ and $e_2' \leq e_2$ such that $e_1' \smile_\mu e_2'$. Since projection are morphisms of event structures, and since in particular preserve configurations, for every event f below e_1 there must be an event in E below (x, e_1, e_2) that is projected onto f. And similar for d_1. Therefore there are $(x', e_1', e_2') \in x, (y', d_1', d_2') \in y$ for some x', y', e_2', d_2'. Note also that $(x', e_1', e_2') \smile (y', d_1', d_2')$.
- $e_2 \smile d_2$ is symmetric.
- $e_1 = d_1$ and $e_2 \neq d_2$. This is the crucial case, where we use the typing. In this case it is not possible that $e_2 = *$ and $d_2 \neq *$ (nor symmetrically). This is because of the typing. If the label dual of e_1 is not in Γ_2 then both $e_2, d_2 = *$. If the label dual of e_1 is in Γ_2, then the label of e_1 is matched and thus it becomes disallowed, so that the event $(x, e_1, *)$ is removed. So both e_2 and d_2 have the same label (the dual of the label of e_1). Thus they are mapped on the same event in $[\![\Gamma_2]\!]$, and thus they must be in conflict. Then we reason as above.
- $e_2 = d_2$ and $e_1 \neq d_1$ is symmetric.
- $e_1 = d_1$ and $e_2 = d_2$. Then the conclusion follow from stability (Lemma A.3.6).
- suppose there exists $(\bar{x}, \bar{e_1}, \bar{e_2} \in x$ such that $\bar{e_1} \asymp d_1$ or $\bar{e_2} \asymp d_2$. Then we reason as above to find $(x', e_1', e_2') \in \bar{x}, (y', d_1', d_2') \in y$ such that either $e_1' \smile_\mu d_1'$ or $d_1' \smile_\mu d_2'$. Note that, by transitivity, $(x', e_1', e_2') \in x$.

– the symmetric case is analogous.
– Suppose there is $e \in x$, and $d \in y$ such that $e \smile d$. By wellfoundedness this case can be reduce to one of the previous ones.

Lemma A.12.2. *If $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, then their labels have the same subject, but different branch and different confidential names.*

By Lemma A.12.1, either $e_1 \asymp_\mu d_1$ or $e_2 \asymp_\mu d_2$ (or both). In the first case, the labels of e_1, d_1 have the same subject. Thus the labels of $(x, e_1, e_2), (y, d_1, d_2)$ also have the same subject (whether they are synchronisation labels or not). The second case is symmetric.

Lemma A.12.3. *If $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, then $x = y$*

First suppose $e_2 = d_2 = *$. Then $e_1 \asymp_\mu d_1$. Dually when $e_1 = d_1 = *$. Finally, suppose $e_1, d_1, e_2, d_2 \neq *$. Without loss of generality we have $e_1 \asymp_\mu d_1$. But then $e_2 \asymp d_2$, because they have dual labels. Then it must be $e_2 \asymp_\mu d_2$ because otherwise we would not have $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$.

In all cases we have that $(x, d_1, d_2) \in E$. Indeed it satisfies the condition for being in the product (because $parents(e_1) = parents(d_1)$ and $parents(e_2) = parents(d_2)$), and it is allowed if and only if (x, e_1, e_2) is allowed. Suppose $x \neq y$. By stability we have that there are $e' \in x, d' \in y$ such that $e' \smile d'$. Which contradicts $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$.

Lemma A.12.4. *The relation $\asymp_\mu$ is transitive in $\mathcal{E}$.*

Suppose $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, and $(y, d_1, d_2) \asymp_\mu (z, g_1, g_2)$, Then reasoning as above we have that $e_1 \asymp_\mu d_1 \asymp_\mu g_1$ and $e_2 \asymp_\mu d_2 \asymp_\mu g_2$. Which implies $e_1 \asymp_\mu g_1$ and $e_2 \asymp_\mu g_2$, from which we derive $(x, e_1, e_2) \asymp_\mu (z, g_1, g_2)$.

Lemmas A.12.3, and A.12.4 together prove that $\mathcal{E}$ is confusion free.

To prove that $\mathcal{E} \triangleright \Gamma$, suppose $f_1 : E_1 \to [\![\Gamma_1]\!]$ and $f_2 : E_2 \to [\![\Gamma_2]\!]$. Recall that $[\![\Gamma]\!] = ([\![\Gamma_1]\!] \| [\![\Gamma_2]\!]) \setminus (Dis(\Gamma) \cup \boldsymbol{\tau})$. As we observed we can think of $[\![\Gamma]\!]$ as the disjoint union of $[\![\Gamma_1]\!]$ and $[\![\Gamma_2]\!]$, plus some hiding.

We define the following partial function $f : \mathcal{E} \to [\![\Gamma]\!]$. $f(x, e_1, *) = f_1(e_1)$, $f(x, *, e_2) = f_2(e_2)$ (where by equality we mean *weak equality*), and undefined otherwise. We have to check that f satisfies the conditions required. The first two conditions are a consequence of (the proof) of the first part of the theorem. It remains to show that f is a morphism of event structures. This follows from general principles, but we repeat the proof here.

We have to check that if $d \leq f(x, e_1, e_2)$ in $[\![\Gamma]\!]$, then there exists (y, d_1, d_2) in $\mathcal{E}$ such that $f(y, d_1, d_2) = d$. Without loss of generality, we assume $e_2 = *$, so that $f(x, e_1, e_2) = f_1(e_1)$. Let $d \leq f_1(e_1)$. Since f_1 is a morphism, then there is $d_1 \leq e_1$ such that $f_1(d_1) = d$. Since projections are morphisms, there must be a $(y, d_1, d_2) \leq (x, e_1, e_2)$. I claim that d_2 must be equal to $*$, so that $f(y, d_1, d_2) = f_1(d_1) = d$. If d_2 were not $*$, then its label would be dual to label of d_1. This means that both labels are in $Dis(\Gamma)$, and that no event in $[\![\Gamma]\!]$, and in particular the d, can be labelled by either of them. This contradicts $f_1(d_1) = d$.

Then we have to check that f reflects $\asymp$. So, suppose $f(x, e_1, e_2) \asymp f(x', e_1', e_2')$. By the structure of $[\![\Gamma]\!]$ it cannot be that $f(x, e_1, *) \asymp f(x', *, e_2')$, because they are mapped to disjoint concurrent components. Therefore, w.l.o.g, the only case to consider is $f(x, e_1, *) \asymp f(x', e_1', *)$. This means $f_1(e_1) \asymp f_1(e_1')$. Since f_1 is a morphism, then $e_1 \asymp e_1'$, which implies $(x, e_1, *) \asymp (x', e_1', *)$.

A.12.6 Proof of Proposition A.5.1

By a straightforward case analysis.

A.12.7 Proof of Theorem A.6.1

The proof is by induction on the semantics. All the cases are easily done directly, with the exception of the parallel composition. The case of the parallel composition is a direct consequence of Theorem A.4.4.

A.12.8 Proof of Theorem A.6.7

The proof is by induction on the rules of the operational semantics. All cases are rather straightforward, except the parallel composition. For this we need the following lemma. To avoid distinguishing different cases, lets say that, for every event structure $\mathcal{E}$, we have $\mathcal{E} \xrightarrow{*} \mathcal{E}\lfloor * = \mathcal{E}$.

Lemma A.12.5. *Let $\cong$ denote isomorphism of event structures. We have that $\mathcal{E}_1 \xrightarrow{\alpha} \mathcal{E}_1\lfloor e_1$, and $\mathcal{E}_2 \xrightarrow{\beta} \mathcal{E}_2\lfloor e_2$ if and only if $\mathcal{E}_1 \| \mathcal{E}_2 \xrightarrow{\alpha \bullet \beta} \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\varnothing, e_1, e_2)$. Moreover, in such a case, we have $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\varnothing, e_1, e_2) \cong (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$.*

The first part of theorem is straightforward : if e_1, e_2 are minimal in $\mathcal{E}_1, \mathcal{E}_2$, then $(\varnothing, e_1, e_2)$ is a minimal event in $\mathcal{E}_1 \| \mathcal{E}_2$, and vice versa. Assuming this is the case, we are now going to prove that $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\varnothing, e_1, e_2) \cong (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$. We will define a bijective function $f : \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\varnothing, e_1, e_2) \to (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$, such that both f and f^{-1} are morphism of event structure. We define f by induction on the height of the events. Also by induction we show the properties required. That is we prove that

- for every n, f is bijective on elements of height n ;
- f preserves and reflects the conflict relation ;
- f preserves and reflects the order relation ;
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, where Π_1, Π_2 denote the projections in the parallel composition.

In particular. the above properties imply that both f, and f^{-1} are morphisms of event structure. The preservation of the labels follows from the last point, noting that the labels of an event in the product depend only on the labels of the projected events.

Base : height $= 0$

Events of height 0 in $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\varnothing, e_1, e_2)$ are of two forms :

- the form $(\varnothing, d_1, d_2)$, with d_1 minimal in $\mathcal{E}_1$ and d_2 minimal in $\mathcal{E}_2$ (when different from $*$) [1]. In such a case we define $f(\varnothing, d_1, d_2) = (\varnothing, d_1, d_2)$.
- the form $((\varnothing, e_1, e_2), d_1, d_2)$, with $e_1 \leq d_1$ and d_2 minimal in $\mathcal{E}_2$, or $e_2 \leq d_2$ and d_1 minimal in $\mathcal{E}_1$, or both $e_1 \leq d_1, e_2 \leq d_2$. In such a case we define $f(\varnothing, d_1, d_2) = (\varnothing, d_1, d_2)$.

Note that from the discussion above, it follows that the events $(\varnothing, d_1, d_2)$ and $((\varnothing, e_1, e_2), d_1, d_2)$ cannot be both in $\mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$. We prove that f is well defined on events of height 0. Consider $d = (\varnothing, d_1, d_2)$. Then both d_1, d_2 are minimal in $\mathcal{E}_1, \mathcal{E}_2$ respectively. Also it is not the case that $d_1 \asymp e_1$, nor $d_2 \asymp e_2$, as otherwise we would have $(\varnothing, d_1, d_2) \asymp (\varnothing, e_1, e_2)$. This means that d_1, d_2 belong to $\mathcal{E}_1 \lfloor e_1, \mathcal{E}_2 \lfloor e_2$ and are minimal there. So that $f(d) = (\varnothing, d_1, d_2) \in (\mathcal{E}_1 \lfloor e_1) \| (\mathcal{E}_2 \lfloor e_2)$. A similar reasoning applies when $d = ((\varnothing, e_1, e_2), d_1, d_2)$. Now we prove

- f is bijective on events of height 0 ; it is surjective : take an event $(\varnothing, d_1, d_2)$ in $(\mathcal{E}_1 \lfloor e_1) \| (\mathcal{E}_2 \lfloor e_2)$. There are several cases. If both d_1 is minimal in $\mathcal{E}_1$ and d_2 is minimal in $\mathcal{E}_2$, and it is not the case that $e_1 \asymp d_1$ nor $e_2 \asymp d_2$, then $(\varnothing, d_1, d_2) \in \mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$. Similarly, in the other cases, it is easy to see that $((\varnothing, e_1, e_2), d_1, d_2) \in \mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$. Also f is injective. The only thing to check is that $(\varnothing, d_1, d_2)$ and $((\varnothing, e_1, e_2), d_1, d_2)$ cannot be both events in $\mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$, which, as we have observed, is the case.
- f preserves and reflects conflict on events of height 0. This is easily verified by checking all the cases of definition of conflict. Note that it cannot be the case that $(\varnothing, d_1, d_2) \asymp (\varnothing, e_1, e_2)$, as such events do not belong to $\mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$.
- f preserves and reflects order on events of height 0, trivially.
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, by definition.

Step : height $= n + 1$

We assume that f is defined for all events of height $\leq n$, and that it satisfies the required properties there. On events of height $n+1$, we define f as follows. $f(x, d_1, d_2) = (f(x), d_1, d_2)$. We prove that f is well defined. Note that in order to show that $(f(x), d_1, d_2)$ is an event, we only use properties of $\Pi_1(f(x))$ and $\Pi_2(f(x))$, by induction hypothesis they coincide with $\Pi_1(x), \Pi_2(x)$ respectively. We consider one case, the others being similar. Suppose $d_1 \in E_1, d_2 \in E_2$. Then let y be the set of maximal elements of x. Since f preserves and reflects order, we have that $f(y)$ is the set of maximal elements of $f(x)$. Let $y_1 = \Pi_1(y), y_2 = \Pi_2(y)$. Note that we also have $y_1 = \Pi_1(f(y)), y_2 = \Pi_2(f(y))$. Since (x, d_1, d_2) is an event, we have

- if $(z, d_1, d_2) \in y$, then either $d_1 \in \mathit{parents}(e_1)$ or$d_2 \in \mathit{parents}(e_2)$;
- for all $d_1 \in \mathit{parents}(e_1)$, there exists $(z, d_1, d_2) \in x$;
- for all $d_2 \in \mathit{parents}(e_2)$ there exists $(z, d_1, d_2) \in x$.
- for no $d_1 \in \Pi_1(x)$, $d_1 \asymp e_1$ and for no $d_2 \in \Pi_2(x)$, $d_2 \asymp e_2$.

These conditions, show that $(f(x), d_1, d_2)$ is also an event.

We now prove that

- f is bijective on event of height $n + 1$. First, if (x, d_1, d_2) is of height

1. We omit this remark in the following : it will be considered implicit throughout.

$n+1$, so is $(f(x), d_1, d_2)$, because by induction hypothesis, f is bijective on events of height n, so that x contains one such event if and only if $f(x)$ does. To prove that f is surjective, consider now an event $(y, d_1, d_2) \in (\mathcal{E}_1 \lfloor e_1) \| (\mathcal{E}_2 \lfloor e_2)$. Since f is biejctive on events of height $\leq n$, we have that there exists x such that $y = f(x)$, and moreover since f preserves and reflects order and conflict, x is a configuration if and only if $f(x)$ is. We have to argue that if $(f(x), d_1, d_2)$ is an event of $(\mathcal{E}_1 \lfloor e_1 \| \mathcal{E}_2 \lfloor e_2)$ then (x, d_1, d_2) is an event of $\mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\varnothing, e_1, e_2)$. This is done in a similar way than the base case. To prove that f is injective, consider $(x, d_1, d_2), (x', d_1, d_2)$, such that $f(x) = f(x')$. By induction hypothesis f is injective, so that $x = x'$ and we are done.

- f preserves and reflects conflict. This is done as in the base case.
- f preserves and reflects order. In fact by definition $d \in x$ if and only if $f(d) \in f(x)$, which is precisely what we need.
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, by definition.

This concludes the proof.

A.12.9 Proof of Lemma A.7.1

Given a NCCS type σ, we define its *erasure* $er(\sigma)$ to be the π type obtained from σ by removing all confidential names. It is a partial function defined as follows

- $er(y_1 : \sigma_1, \ldots, y_n : \sigma_n) = er(\sigma_1), \ldots, er(\sigma_n)$
- $er(\binampersand_{i \in I} \Gamma_i) = (\binampersand_{i \in I} er(\Gamma_i))^{\downarrow}$
- $er(\bigoplus_{i \in I} \Gamma_i) = (\bigoplus_{i \in I} er(\Gamma_i))^{\uparrow}$
- $er(\bigotimes_{i \in I} \Gamma_i) = (er(\Gamma))^{!}$ if for all $i \in I$, $er(\Gamma_i) = er(\Gamma)$.
- $er(\biguplus_{i \in I} \Gamma_i) = (er(\Gamma))^{?}$ if for all $i \in I$, $er(\Gamma_i) = er(\Gamma)$.
- $er(\updownarrow) = \updownarrow$

Lemma A.12.6. *Suppose $er(\sigma) = \overline{er(\tau)}$, and suppose σ, τ have disjoint sets of names. Suppose for every type of the form $\bigotimes_{k \in K} \Gamma_k$, the set K is* infinite. *Then there is a renaming ρ, such that $match[\tau, \sigma[\rho]] \to S$ and if $res[\tau, \sigma[\rho]] = \bigotimes_{k \in K} \Gamma_k$, then K is infinite.*

By induction on the structure of the types.

We want to prove that for every judgement $P \triangleright \Gamma$, there exists a choice function ρ and an environment Δ, such that $pc[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$. We will prove it by induction on the typing rules. However we need a stronger statement for the induction to go through. We prove that a Δ exists such that it has the following properties

- if $\Delta(x) = \tau$, then $\Gamma(x) = er(\tau)$
- if $\Gamma(x) = \tau$, then there exists τ' such that $\Delta(x) = \tau'$ and $er(\tau') = \tau$.
- for every type of the form $\bigotimes_{k \in K} \Gamma_k$, the set K is *infinite*.

Finally we prove that if $pc[\![P \triangleright \Gamma]\!]^{\rho} \triangleright \Delta$, then for every fresh renaming ρ', $pc[\![P \triangleright \Gamma]\!]^{\rho' \circ \rho} \triangleright \Delta[\rho']$.

The proof is trivial for Zero, WeakCl, WeakOut, Res, LIn, LOut, Rout. For Rin, one has just to take care to choose K to be infinite. For the parallel

composition, assume $pc[\![P_1 \triangleright \Gamma_1]\!]^{\rho_1} \triangleright \Delta_1$ and $pc[\![P_2 \triangleright \Gamma_2]\!]^{\rho_2} \triangleright \Delta_2$. First rename all the variables in Δ_1, Δ_2, so that they are disjoint. In this way we can substitute a name of Δ_1 for a name in Δ_2, and Δ_2 would still be well formed.

Then consider a judgement $a : \tau$ in Γ_1 such that there is a matching judgement $\overline{a : \sigma}$ in Γ_2. Consider the type τ' such that $a : \tau'$ is in Δ_1. Since $er(\tau) = \overline{er(\tau')}$, by Lemma A.12.6 we find a ρ_a such that $match[\tau, \sigma[\rho_a]] \to S$. For every matching name, we obtain such a renaming. All renamings can be joined to obtain a fresh injective renaming ρ, because no name is involved in two different renamings. Therefore $\Delta_1 \odot \Delta_2[\rho]$ is defined.

A.12.10 Proof of Theorem A.7.2

The proof is by structural induction on $P \triangleright \Gamma$. All the cases are rather easy, taking into account that π-calculus terms can perform any fresh α-variant of an action. For the parallel composition, one has to notice that names that are closed *after* the transition in the π-calculus are closed *before* the transition in NCCS.

A.12.11 Proof of Theorem A.7.3

One direction of the proof (soundness) is easy and it is left to the reader.

To prove full abstraction we define a relation as follows (we omit the environments for simplicity) : $(pc[\![P]\!]^{\rho}, pc[\![Q]\!]^{\rho'}) \in \mathcal{R}$ if and only if $P \approx Q$. We want to prove it is a bisimulation. Suppose $pc[\![P]\!]^{\rho} \xrightarrow{\beta} R$. Then $P \xrightarrow{\beta} P'$ and $R = pc[\![P']\!]^{\rho \setminus obj(\beta)}$. Since $P \approx Q$, then $Q \xrightarrow{\beta} Q'$ with $P' \approx Q'$. Then there exists ρ'' such that $pc[\![Q]\!]^{\rho''} \xrightarrow{\beta} pc[\![Q']\!]^{\rho'' \setminus obj(\beta)}$. The choice function ρ'' can be obtained from ρ' via a bijection of names ρ''' (note that the cardinality of the Ks is always the same). Then we can write $pc[\![Q]\!]^{\rho'}[\rho'''] \xrightarrow{\beta} pc[\![Q']\!]^{\rho'' \setminus obj(\beta)}$. We conlcude by noting that $(pc[\![P']\!]^{\rho \setminus obj(\beta)}, pc[\![Q']\!]^{\rho'' \setminus obj(\beta)}) \in \mathcal{R}$.

A.12.12 Proof of Theorem A.10.2

Before proving Theorem A.10.2, we need a lemma. Consider the Segala automaton (t, x_0) generated by a probabilistic event structure $\mathcal{E}$. Transition groups correspond to certain sets of events. A path of the Segala automaton can be seen as performing a sequence of events.

Recall that a *configuration* of an event structure is a set of events that is conflict free and downward closed.

Lemma A.12.7. *The set of events along a path of (t, x_0) form a configuration of $\mathcal{E}$. The probability of a path is the same as the probability of the corresponding configuration.*

A scheduler creates paths, and therefore configurations. A infinite path cannot be extended further, but the corresponding configuration is not maximal in general. However this is the case for *fair* schedulers.

Lemma A.12.8. *The set of events along a path in $\mathcal{B}(t, x_0, \mathcal{S})$, for $\mathcal{S}$ fair, is a maximal configuration. Conversely, given a fair scheduler $\mathcal{S}$ and a maximal configuration, there exists a corresponding path in $\mathcal{B}(t, x_0, \mathcal{S})$.*

Proof of Theorem A.10.2. Consider a fully probabilistic event structure $\mathcal{E}$. According to Theorem 4.2 in [VVW06], there exists a unique probability distribution μ over the set of maximal configurations. Consider a label a. We define $p(a) := \mu(X_a)$, where X_a is the set of maximal configurations that contain the label a (it is easy to show this set is indeed measurable).

We want to prove that, given a fair scheduler $\mathcal{S}$ for the Segala automaton generated by $\mathcal{E}$, we have $p_{\mathcal{S}}(a) = p(a)$, and therefore it is independent from $\mathcal{S}$. Consider the set of paths in $\mathcal{B}(t, x_0, \mathcal{S})$ that contain a label a. This is the disjoint union of the paths that contain the first a at the i-th position, for $i > 0$. Let B_i be such sets. The measure of B_i is the sum of the probabilities of the finite paths of length i that contain the label a in the last position. Let such paths form the set F_i. The configurations corresponding to the paths in F_i have the same probability. Using Lemma A.12.8, we show that every maximal configuration containing an event labelled by a is above a unique configuration in some F_i. This shows that the measure of the set of maximal configurations containing an a coincides with the set of infinite paths in $\mathcal{B}(t, x_0, \mathcal{S})$.

Annexe B

Event structure semantics of the untyped π-calculus

B.1 Introduction

In the previous chapter we have used event structures to give a causal and compositional semantics to a very restricted fragment of the π-calculus. In this chapter we present an event structure semantics of the full, unrestricted version of the calculus. It is the first compositional event structure semantics of the π-calculus. The semantics we propose generalises Winskel's semantics of CCS [Win82], it is operationally adequate with respect to the standard labelled transition semantics, and consequently it is sound with respect to bisimilarity.

B.1.1 The internal π calculus

As a first step we give a semantics of a very expressive subcalculus, the πI-calculus [San95], that we presented in its typed version in the previous chapter. Recall that the distinctive feature of theπI-calculus is that the output of free names is disallowed. The symmetry of input and output prefixes, that are both binders, simplifies considerably the theory, while preserving most of the expressiveness of the calculi with free name passing [Bor98, MS04, Pal03]. The πI-calculus comes in two variants : synchronous and asynchronous. Contrary to the full calculus, the asynchronous variant is not a subcalculus of the synchronous one and therefore they have to be treated independently.

In order to provide an event structure semantics of any version of the π-calculus, one has in particular to be able to represent dynamic creations of new synchronisation channels, a feature that is not present in traditional process algebras. In Winskel's event structure semantics of CCS [Win82], the parallel composition is defined as product in a suitable category followed by relabelling and hiding. The product represents all conceivable synchronisations, the hiding removes synchronisations that are not allowed, while the relabelling chooses

suitable names for synchronisation events. In CCS one can decide statically whether two events are allowed to synchronise, whereas in the π-calculus, a synchronisation between two events may depend on which synchronisations took place before.

Consider for instance the π-process $a(x).\overline{x}(u).\mathbf{0} \mid \overline{a}(z).z(v).\mathbf{0}$ where $a(x).P$ is an input at a, $\overline{a}(z).Q$ is an output of a new name z to a and $\mathbf{0}$ denotes the inaction. This process contains two synchronisations, first along the channel a and then along a private, newly created, channel z. The second synchronisation is possible only since the names x and z are made equal by the previous synchronisation along a. To account for this phenomenon, we define the semantics of the parallel composition by performing hiding and relabelling not uniformly on the whole event structure, but relative to the causal history of events.

The full symmetry underlying the πI-calculus theory has a further advantage : it allows a uniform treatment of causal dependencies. Causal dependencies in the π-processes arise in two ways [BS98, DP99] : by nesting prefixes (called *structural* or *prefixing* or *subject* causality) and by using a name that has been bound by a previous action (called *link* or *name* or *object* causality). While subject causality is already present in CCS, object causality is distinctive of the π-calculus. In the synchronous πI-calculus, object causality always appears under subject causality, as in $a(x).x(y).\mathbf{0}$ or in $(\boldsymbol{\nu} c)a(x).(c(z).\mathbf{0} \mid \overline{c}(w).x(y).\mathbf{0})$, where the input on x causally depends in both senses from the input on a. As a result, the causality of synchronous πI-calculus can be naturally captured by the standard prefixing operator of the event structures, as in CCS.

On the other hand, in the asynchronous πI-calculus, the bound output process is no longer a prefix : in $\overline{a}(x)P$, the continuation process P can perform any action α before the output of x on a, provided that α does not contain x. Thus the asynchronous output has a looser causal dependency. For example, in $(\boldsymbol{\nu} c)\overline{a}(x)(c(z).\mathbf{0} \mid \overline{c}(w)x(y).\mathbf{0})$, $\overline{a}(x)$ only binds the input at x, and the interaction between $c(z)$ and $\overline{c}(w)$ can perform before $\overline{a}(x)$, thus there exists no subject causality. Representing this output object causality requires a novel operator on event structures that we call *rooting*, whose construction is inspired from a recent study on Ludics [CF05b].

In this chapter, we present these new constructions, and use them to obtain compositional, sound and adequate semantics for both synchronous and asynchronous πI-calculus.

B.1.2 Free Name Passing and Scope Extrusion

After having dealt with the simpler case of the πI-calculus we then provide the semantics for the full calculus.

The main issues when dealing with the full π-calculus are name passing and the extrusion of bound names. These two ingredients are the source of the expressiveness of the calculus, but they are problematic in that they allow complex forms of causal dependencies, as detailed below.

Free name passing Compared to pure CCS, (either free or bound) name passing adds the ability to dynamically acquire new synchronization capabilities. For instance consider the π-calculus process $P = n(z).(\overline{z}\langle a\rangle \mid m(x))$, that reads from the channel n and uses the received name to output the name a in parallel with a read action on m. Hence a synchronization along the channel m is possible if a previous communication along the channel n substitutes the variable z exactly with the name m. Then, in order to be compositional, the semantics of P must also account for "potential" synchronizations that might be activated by parallel compositions, like the one on channel m.

To account for this phenomenon, we define the parallel composition of event structures so that synchronization events that involve input and output on different channels, at least one of which is a variable, are not deleted straight away. Moreover, the events produced by the parallel composition are relabelled by taking into account their causal history. For instance, the event corresponding to the synchronization pair $(\overline{z}\langle a\rangle, m(x))$ is relabelled into a τ action if, as in the process P above, its causal history contains a synchronization that substitutes the variable z with the name m.

Causal dependencies in π-calculus processes arise in two ways [BS98, DP99] : by nesting prefixes (called *structural* or *prefixing* or *subject* causality) and by using a name that has been bound by a previous action (called *link* or *name* or *object* causality). While subject causality is already present in CCS, object causality is distinctive of the π-calculus. The interactions between the two forms of causal dependencies are quite complex. We illustrate them by means of examples.

Parallel Scope extrusion. Consider the two processes $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle.n(x))$ and $Q = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid n(x))$. The causal dependence of the action $n(x)$ on the output $\overline{a}\langle n\rangle$ is clear in the process P (i.e. there is a structural causal link), however, a similar dependence appears also in Q since a process cannot synchronize on the fresh name n before receiving it along the channel a (i.e. there is an objective causal link). Now consider the process $P_1 = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid \overline{b}\langle n\rangle)$: in the standard interleaving semantics of π-calculus only one output extrudes, either $\overline{a}\langle n\rangle$ or $\overline{b}\langle n\rangle$, and the other one does not. As a consequence, the second (free) output *depends* on the previous extruding output. However, in a true concurrent model we can hardly say that there is a dependence between the two parallel outputs, which in principle could be concurrently executed resulting in the parallel/simultaneous extrusion of the same name n to two different threads reading respectively on channel a and on channel b.

Dynamic Addition of New Extruders. We have seen that a bound name may have multiple extruders. In addition, the coexistence of free and bound outputs allows the set of extruders to dynamically change during the computation. Consider the process $P_2 = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \,|\, n(z)) \mid a(x).(\overline{x}\langle b\rangle \,|\, \overline{c}\langle x\rangle)$. It can either open the scope of n by extruding it along the channel a, or it can evolve to the process $(\boldsymbol{\nu} n)(n(z) \mid \overline{n}\langle b\rangle \mid \overline{c}\langle n\rangle)$ where the output of the variable x has

become a new extruder for both the actions with subject n. Hence after the first synchronization there is still the possibility of opening the scope of n by extruding it along the channel c.

The lesson we learned. The examples above show that the causal dependencies introduced by the scope extrusion mechanisms distinctive of the π-calculus can be understood in terms of the two ingredients of extrusion : name restriction and communication.

1. The restriction $(\boldsymbol{\nu} n)P$ adds to the semantics of P a causal dependence between *every* action with subject n and *one of* the outputs with object n.
2. The communication of a restricted name adds *new* causal dependencies since both new extruders and new actions that need an extrusion may be generated by variable substitution.

A causal semantics for the π-calculus should account for such a dynamic additional objective causality introduced by scope extrusion. In particular, the first item above hints at the fact that we have to deal with a form of disjunctive (objective) causality. Prime event structures are stable models that represent disjunctive causality by duplicating events and so that different copies causally depend on different (alternative) events. In our case this amounts to represent different copies of any action with a bound subject, each one causally depending on different (alternative) extrusions. However, the fact that the set of extruders dynamically changes complicates the picture since new copies of any action with a bound subject should be dynamically spawned for each new extruder. In this way the technical details quickly become intractable, as discussed in Section C.7.

In this work we eventually chose to follow a completely different approach that leads to an extremely simple technical development. The idea is to represent the disjunctive objective causality in a so-called inclusive way : in order to trace the causality introduced by scope extrusion it is sufficient to ensure that whenever an action with a bound subject is executed, at least one extrusion of that bound name must have been already executed, but it is not necessary to record which output was the real extruder. Clearly, such an inclusive-disjunctive causality is no longer representable with stable structures like prime event structures. However, we show that an operational adequate true concurrent semantics of the π-calculus can be given by encoding a π-process simply as a pair $(\mathcal{E}, X)$ where $\mathcal{E}$ is a prime event structure and X is a set of (bound) names. Intuitively, the causal relation of $\mathcal{E}$ encodes the structural causality of a process. Instead, the set X affects the computation on $\mathcal{E}$: we define a notion of *permitted configurations*, ensuring that any computation that contains an action whose subject is a bound name in X, also contains a previous extrusion of that name. Hence a further benefit of this semantics is that it clearly accounts for both forms of causality : subjective causality is captured by the causal relation of event structures, while objective causality is implicitly captured by permitted configurations.

Structure of the chapter. This chapter is the fusion of the two papers [CVY07, CVY12] with some modifications. in Section B.2, we recall the definitions of the π-calculus, the internal variant, and the asynchronous internal variant. In Section B.3, we briefly recall the definition of event structure as already presented in Section A.3. In Section B.4, we present the semantics of the internal π-calculus (synchronous and asyncrhonous). In Section B.5, we see how to handle the substitution of free names. In Section B.6 ; we see how to handle the parallel scope extrusion, and we provide the semantics of the full calculus. Section B.7 contains some discussions about our modeling choices. Section B.8 discusess related and future work.

B.2 The π-Calculus

B.2.1 The full calculus

In this section we illustrate the synchronous, monadic π-calculus that we consider. We presuppose a countably-infinite set of names and a countably-infinite set of variables ranged over by $m, .., q$ and by $x, .., z$, respectively. We use a, b, c to range over both names and variables.

Prefixes	π	::=	$a(x) \mid \overline{a}\langle b\rangle$
Processes	P, Q	::=	$\sum_{i\in I} \pi_i.P_i \mid P \mid Q \mid (\boldsymbol{\nu} n)P \mid A\langle \tilde{x}, \tilde{p} \mid \mathbf{z}, \mathbf{n}\rangle$
Definitions	$A(\tilde{x}, \tilde{p} \mid \mathbf{z}, \mathbf{n}) = P_A$		

The syntax consists of the parallel composition, name restriction, finite summation of guarded processes and recursive definition. In $\sum_{i\in I} \pi_i.P_i$, I is a finite indexing set ; when I is empty we simply write $\mathbf{0}$ and denote with $+$ the binary sum. A process $a(x).P$ can perform an input at a and the variable x is the placeholder for the name so received. The output case is symmetric : a process $\overline{a}\langle b\rangle.P$ can perform an output along the channel a. Notice that an output can send a name (either free or restricted) or else a variable.

We assume that every constant A has a unique defining equation $A(\tilde{x}, \tilde{p} \mid \mathbf{z}, \mathbf{n}) = P_A$. The symbol $\tilde{p}$, resp. $\tilde{x}$, denotes a tuple of distinct names, resp. variables, that correspond to the free names, resp. variables, of P_A. $\mathbf{n}$, resp. $\mathbf{z}$, represents an infinite sequence of distinct names $\mathbb{N} \to Names$, resp. distinct variables $\mathbb{N} \to Variables$, that is intended to enumerate the (possibly infinite) bound names, resp. bound variables, of P_A. The parameters $\mathbf{n}$ and $\mathbf{z}$ do not usually appear in recursive definitions in the literature. The reason we add them is that we want to maintain the following Basic Assumption :

Every bound name/variable is different from any other name/variable, either bound or free.

In the π-calculus, this policy is usually implicit and maintained along the computation by dynamic α-conversion : every time the definition A is unfolded, a copy of the process P_A is created whose bound names and variables must be

$$\text{(In Late)}\quad \frac{}{a(x).P \xrightarrow{a(x)} P} \qquad \text{(Out)}\quad \frac{}{\overline{a}\langle b\rangle.P \xrightarrow{\overline{a}\langle b\rangle} P}$$

$$\text{(Comm)}\quad \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\overline{a}\langle b\rangle} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'} \qquad \text{(Par)}\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{(Open)}\quad \frac{P \xrightarrow{\overline{a}\langle n\rangle} P' \quad n \neq a}{(\boldsymbol{\nu} n)P \xrightarrow{\overline{a}(n)} P'} \qquad \text{(Close)}\quad \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\overline{a}(n)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} n)(P'\{^{n}/_{x}\} \mid Q')}$$

$$\text{(Res)}\quad \frac{P \xrightarrow{\alpha} P' \quad n \notin \mathsf{fn}(\alpha)}{(\boldsymbol{\nu} n)P \xrightarrow{\alpha} (\boldsymbol{\nu} n)P'} \qquad \text{(Sum)}\quad \frac{P_i \xrightarrow{\alpha} P_i' \quad i \in I}{\sum_{i\in I} P_i \xrightarrow{\alpha} P_i'}$$

$$\text{(Rec)}\quad \frac{P_A\{^{\tilde{y},\tilde{q}}/_{\tilde{x},\tilde{p}}\}\{^{\mathbf{w},\mathbf{m}}/_{\mathbf{z},\mathbf{n}}\} \xrightarrow{\alpha} P' \quad A(\tilde{x},\tilde{p} \mid \mathbf{z},\mathbf{n}) = P_A}{A\langle \tilde{y},\tilde{q} \mid \mathbf{w},\mathbf{m}\rangle \xrightarrow{\alpha} P'}$$

FIGURE B.1 – Labelled Transition System of the π-calculus

fresh. This dynamic choice is difficult to interpret in the event structures. Hence, in order to obtain a precise semantic correspondence, our recursive definitions prescribe all the names and variables that will be possibly used in the recursive process. Notice that this assumption has no impact on the process behaviour since every π-process can be α-renamed so that it satisfies that assumption.

The sets of free and bound names and free and bound variables of P, denoted by $\mathsf{fn}(P), \mathsf{bn}(P), \mathsf{fv}(P), \mathsf{bv}(P)$, are defined as usual but for constant processes, whose definitions are as follows : $\mathsf{fn}(A\langle \tilde{x},\tilde{p} \mid \mathbf{z},\mathbf{n}\rangle) = \{\tilde{p}\}$, $\mathsf{bn}(A\langle \tilde{x},\tilde{p} \mid \mathbf{z},\mathbf{n}\rangle) = \mathbf{n}(\mathbb{N})$, $\mathsf{fv}(A\langle \tilde{x},\tilde{p} \mid \mathbf{z},\mathbf{n}\rangle) = \{\tilde{x}\}$ and $\mathsf{bv}(A\langle \tilde{x},\tilde{p} \mid \mathbf{z},\mathbf{n}\rangle) = \mathbf{z}(\mathbb{N})$.

Example B.2.1. Consider $A(x \mid \mathbf{z}) = x(z_0).A\langle z_0 \mid \mathbf{z}'\rangle \mid x(z_1).A\langle z_1 \mid \mathbf{z}''\rangle$, where $\mathbf{z}'(n) = \mathbf{z}(2n+2)$ and $\mathbf{z}''(n) = \mathbf{z}(2n+3)$ (we omit to mention $\mathbf{n}$). In this case the sequence of variables $\mathbf{z}$ is partitioned into two infinite subsequences $\mathbf{z}'$ and $\mathbf{z}''$ (corresponding to even and odd variable occurrences), so that the bound variables used in the left branch of A are different from those used in the right branch. Intuitively $A\langle a \mid \mathbf{z}\rangle$ partially "unfolds" to $a(z_0).(z_0(z_2).A\langle z_2 \mid \mathbf{z}_1'\rangle \mid z_0(z_4).A\langle z_4 \mid \mathbf{z}_2'\rangle) \mid a(z_1).(z_1(z_3).A\langle z_3 \mid \mathbf{z}_1''\rangle \mid z_1(z_5).A\langle z_3 \mid \mathbf{z}_2''\rangle)$ with suitable $\mathbf{z}_1', \mathbf{z}_2', \mathbf{z}_1'', \mathbf{z}_2''$.

The operational semantics is given in Figure B.1 in terms of an LTS (in late style) where we let α, β range over the set of labels $\{\tau, a(x), \overline{a}\langle b\rangle, \overline{a}(n)\}$. The syntax of labels shows that the object of an input is always a variable, whereas

the object of a free output is either a variable (e.g. $b(x)$ or $\overline{a}\langle x\rangle$) or a name. On the other hand, the object of a bound output is always a name, since it must occur under a restriction. The definition of the substitution $\{\mathbf{w}/\mathbf{z}\}$ is as follows. Let $A(\tilde{x} \mid \mathbf{z}) = P_A$ be a recursive definition. The sequence $\mathbf{z}$ contains all bound names of P_A, and in particular the names of all sequences $\mathbf{z}'$ that appear in P_A. For each such sequence, there exists an injective function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathbf{z}'(n) = \mathbf{z}(f(n))$. To obtain the process $P_A\{\mathbf{w}/\mathbf{z}\}$, for each bound name of the form $\mathbf{z}(n)$ we substitute $\mathbf{w}(n)$, and for each sequence $\mathbf{z}'$ we substitute the sequence $\mathbf{w}'$ defined as $\mathbf{w}'(n) = \mathbf{w}(f(n))$.

The symmetric rules to (Comm)(Par)(Close) are omitted. Note also that the use of Assumption B.2.1, makes it unnecessary to have the side condition that usually accompany (Par).

B.2.2 The synchronous internal calculus

This section gives basic definitions of the πI-calculus [San95]. This subcalculus captures the essence of name passing with a simple labelled transition relation.

Syntax. The syntax of the monadic, synchronous πI-calculus [San95] is the following. In this calculus, we do not need to distinguish between names and variables — as is the case already for the original presentation of the full calculus.

$$\begin{array}{lrcl} \textit{Definitions} & A(\tilde{x} \mid \mathbf{z}) = P_A & & \\ \textit{Prefixes} & \pi & ::= & a(x) \mid \overline{a}(x) \\ \textit{Processes} & P, Q & ::= & \sum_{i\in I} \pi_i.P_i \mid P \mid Q \mid (\boldsymbol{\nu} a)P \mid A\langle \tilde{x} \mid \mathbf{z}\rangle \end{array}$$

It is very similar to the full π-calculus. The only difference are the output prefix. The process $\overline{a}(x).P$ corresponds to $(\boldsymbol{\nu} x)\overline{a}\langle x\rangle.P$. Therefore, in the πI-calculus only bound names can be communicated, modelling the so called *internal mobility*.

Operational Semantics. The operational semantics is given in the following in terms of an LTS (in late style) where we let α, β range over the set of labels $\{\tau, a(x), \overline{a}(n)\}$.

The rules of Fig B.2 illustrate the internal mobility characterising the πI-calculus communication. In particular, according to (Comm), we have that $a(x).P \mid \overline{a}(y).Q \xrightarrow{\tau} (\boldsymbol{\nu} y)(P\{y/x\} \mid Q)$ where the fresh name y appearing in the output is chosen as the "canonical representative" of the private value that has been communicated.

Proposition B.2.1. *Let P be a process that satisfies the Basic Assumption. Suppose $P \xrightarrow{\alpha} P'$. Then P' satisfies the Basic Assumption.*

We end this section with the definition of strong bisimilarity in the πI-calculus.

$$\text{(In)}\quad \frac{}{a(x).P \xrightarrow{a(x)} P} \qquad \text{(Out)}\quad \frac{}{\overline{a}(x).P \xrightarrow{\overline{a}(x)} P}$$

$$\text{(Comm)}\quad \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\overline{a}(y)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} y)(P'\{y/x\} \mid Q')} \qquad \text{(Par)}\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{(Sum)}\quad \frac{P_i \xrightarrow{\alpha} P_i'}{\sum_{i\in I} P_i \xrightarrow{\alpha} P_i'}\ i \in I \qquad \text{(Res)}\quad \frac{P \xrightarrow{\alpha} P'}{(\boldsymbol{\nu} a)P \xrightarrow{\alpha} (\boldsymbol{\nu} a)P'}\ a \notin \mathsf{fn}(\alpha)$$

$$\text{(Rec)}\quad \frac{P_A\{\tilde{y}/\tilde{x}\}\{\mathbf{w}/\mathbf{z}\} \xrightarrow{\alpha} P' \quad A(\tilde{x} \mid \mathbf{z}) = P_A}{A\langle \tilde{y} \mid \mathbf{w}\rangle \xrightarrow{\alpha} P'}$$

FIGURE B.2 – Labelled Transition System of the π-calculus

Definition B.2.2 (πI strong bisimilarity)**.** A symmetric relation $\mathcal{R}$ on πI processes is a strong bisimulation if $P\ \mathcal{R}\ Q$ implies :

- whenever $P \xrightarrow{\tau} P'$, there is Q' s.t. $Q \xrightarrow{\tau} Q'$ and $P'\mathcal{R}Q'$.
- whenever $P \xrightarrow{a(x)} P'$, there is Q' s.t. $Q \xrightarrow{a(y)} Q'$ and $P'\{z/x\}\mathcal{R}Q'\{z/y\}$.
- whenever $P \xrightarrow{\overline{a}(x)} P'$, there is Q' s.t. $Q \xrightarrow{\overline{a}(y)} Q'$ and $P'\{z/x\}\mathcal{R}Q'\{z/y\}$.

with z being any fresh variable/name. Two processes P, Q are *bisimilar*, written $P \sim Q$, if they are related by some strong bisimulation.

This definition differs from the corresponding definition in [San95] because we do not have the α-conversion rule, and thus we must allow Q to mimic P using a different bound name. The relation $\sim$ is a congruence.

B.2.3 The asynchronous internal calculus

We now present the *asynchronous* πI-calculus [Bou92, HT91, MS04].

$$\begin{array}{lllll} \textit{Processes} & P, Q & ::= & \sum_{i\in I} a_i(x_i).P_i \mid \overline{a}(x)P & \\ & & \mid & P \mid Q \mid (\boldsymbol{\nu} a)P \mid A\langle \tilde{x} \mid \mathbf{z}\rangle & \\ \textit{Definition} & & & A(\tilde{x} \mid \mathbf{z}) = P_A & \end{array}$$

The new syntax of the bound output reflects the fact that there is a looser causal connection between the output and its continuation. A process $\overline{a}(x)P$ is different from $\overline{a}(x).P$ in that it can activate the process P even if the name x

has not been emitted yet along the channel a. The operational semantics can be obtained from that of the synchronous calculus by removing the rule (Out) and adding the following three rules :

$$\text{(Out)}\quad \frac{}{\overline{a}(x)P \xrightarrow{\overline{a}(x)} P} \qquad \text{(Async)}\quad \frac{P \xrightarrow{\alpha} P'}{\overline{a}(x)P \xrightarrow{\alpha} \overline{a}(x)P'}\ n \notin \mathsf{fn}(\alpha)$$

$$\text{(Asynch Comm)}\quad \frac{P \xrightarrow{a(x)} P'}{\overline{a}(y)P \xrightarrow{\tau} (\boldsymbol{\nu} y)P'\{y/x\}}$$

Relying on this LTS, the definition of strong bisimilarity for the asynchronous πI-calculus is identical to that in Section B.2.2.

B.3 Event structures

This section reviews basic definitions of event structures, as presented more extensively in Section A.3.

Definition B.3.1 (Event Structure)**.** An event structure is a triple $\mathcal{E} = \langle E, \leq, \smile \rangle$ s.t.

- E is a countable set of *events* ;
- $\langle E, \leq \rangle$ is a partial order, called the *causal order* ;
- for every $e \in E$, the set $\lceil e) := \{e' \mid e' < e\}$, called the *enabling set* of e, is finite ;
- $\smile$ is an irreflexive and symmetric relation, called the *conflict relation*, satisfying the following : for every $e_1, e_2, e_3 \in E$ if $e_1 \leq e_2$ and $e_1 \smile e_3$ then $e_2 \smile e_3$.

The reflexive closure of conflict is denoted by $\asymp$. We say that the conflict $e_2 \smile e_3$ is *inherited* from the conflict $e_1 \smile e_3$, when $e_1 < e_2$. If a conflict $e_1 \smile e_2$ is not inherited from any other conflict we say that it is *immediate*. If two events are not causally related nor in conflict they are said to be *concurrent*.

Definition B.3.2 (Labelled event structure)**.** Let L be a set of labels. A labelled event structure $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$ is an event structure together with a labelling function $\lambda : E \to L$ that associates a label to each event in E.

Definition B.3.3. Let $\mathcal{E} = \langle E, \leq, \smile, \lambda \rangle$ be a labelled event structure and let e be one of its minimal events. The event structure $\mathcal{E} \lfloor e = \langle E', \leq', \smile', \lambda' \rangle$ is defined by : $E' = \{e' \in E \mid e' \not\asymp e\}$, $\leq' = \leq_{|E'}$, $\smile' = \smile_{|E'}$, and $\lambda' = \lambda_{E'}$. If $\lambda(e) = \beta$, we write $\mathcal{E} \xrightarrow{\beta} \mathcal{E} \lfloor e$.

The reachable LTS with initial state $\mathcal{E}$ corresponds to the computations over $\mathcal{E}$. It is usually defined using the notion of *configuration* [WN95]. However,

by relying on the LTS as defined above, the adequacy theorem has a simpler formulation. A precise correspondence between the two notions of LTS can be easily defined.

Event structures have been shown to be the class of objects of a category [WN95], whose morphisms are defined as follows. Let $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2 \rangle$ be event structures. A *morphism* $f : \mathcal{E}_1 \to \mathcal{E}_2$ is a partial function $f : E_1 \to E_2$ such that (i) f reflects causality : if $f(e_1)$ is defined, then $\big[f(e_1)\big) \subseteq f\big([e_1)\big)$; (ii) f reflects reflexive conflict : if $f(e_1), f(e_2)$ are defined, and if $f(e_1) \asymp f(e_2)$, then $e_1 \asymp e_2$.

It is easily shown that an isomorphism in this category is a bijective function that preserves and reflects causality and conflict. In the presence of labelled event structures $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1, \lambda_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2, \lambda_2 \rangle$ on the same set of labels L, we will consider only *label preserving* isomorphisms, i.e. isomorphisms $f : \mathcal{E}_1 \to \mathcal{E}_2$ such that $\lambda_2(f(e_1)) = \lambda_1(e_1)$. If there is an isomorphism $f : \mathcal{E}_1 \to \mathcal{E}_2$, we say that $\mathcal{E}_1, \mathcal{E}_2$ are isomorphic, written $\mathcal{E}_1 \cong \mathcal{E}_2$.

We recall briefly the main operations that can be defined on labelled event structures :

- *Prefixing* $a.\mathcal{E}$;
- *Prefixed sum* $\sum_{i \in I} a_i.\mathcal{E}_i$.
- *Restriction* (or *Hiding*) $\mathcal{E} \setminus X$ where $X \subseteq L$ is a set of labels.
- *Relabelling* $\mathcal{E}[f]$ where L and L' are two sets of labels and $f : L \to L'$.

B.3.1 The parallel composition

The parallel composition of two event structures $\mathcal{E}_1$ and $\mathcal{E}_2$ gives a new event structure $\mathcal{E}'$ whose events model the parallel occurrence of events $e_1 \in E_1$ and $e_2 \in E_2$. In particular, when the labels of e_1 and e_2 match according to an underlying synchronisation model, $\mathcal{E}'$ records (with an event $e' \in E'$) that a synchronisation between e_1 and e_2 is possible, and deals with the causal effects of such a synchronisation.

The parallel composition is defined as the categorical product followed by restriction and relabelling [WN95]. The categorical product is unique up to isomorphism, but it can be explicitly constructed in different ways. One that is useful for our purpose was presented in Section A.3.

The synchronisation model underlying the relabelling operation needed for parallel composition is formalised by the notion of *synchronisation algebra* [WN95]. A synchronisation algebra S is a partial binary operation $\bullet_S$ defined on L_*. If α_i are the labels of events $e_i \in E_i$, then $\alpha_1 \bullet_S \alpha_2$ is the label of the event $e' \in E'$ representing the synchronisation of e_1 and e_2. If $\alpha_1 \bullet_S \alpha_2$ is undefined, the synchronisation event is given a distinguished label bad indicating that this event is not allowed and should be deleted.

Definition B.3.4 (Parallel Composition of Event Structures)**.** Let $\mathcal{E}_1, \mathcal{E}_2$ two event structures labelled over L, let S be a synchronisation algebra, and let $f_S : L_* \to L' = L_* \cup \{\mathsf{bad}\}$ be a function defined as $f_S(\alpha_1, \alpha_2) = \alpha_1 \bullet_S \alpha_2$, if S is defined on (α_1, α_2), and $f_S(\alpha_1, \alpha_2) = \mathsf{bad}$ otherwise. The parallel

composition $\mathcal{E}_1 \|_S \mathcal{E}_2$ is defined as the categorical product followed by relabelling and restriction [1] : $\mathcal{E}_1 \|_S \mathcal{E}_2 = (\mathcal{E}_1 \times \mathcal{E}_2)[f_S] \setminus \{\mathsf{bad}\}$. The subscripts S are omitted when the synchronisation algebra is clear from the context.

A large CPO of event structures. We say that an event structure $\mathcal{E}$ is a *prefix* of an event structure $\mathcal{E}'$, denoted $\mathcal{E} \leq \mathcal{E}'$ if there exists $\mathcal{E}'' \cong \mathcal{E}'$ such that $E \subseteq E''$ and no event in $E'' \setminus E$ is below any event of E.

Winskel [Win82] has shown that the class of event structures with the prefix order is a large CPO, and thus the limits of countable increasing chains exist. Moreover all operators on event structures are continuous. We will use this fact to define the semantics of the recursive definitions.

B.4 Semantics of πI-Calculus

This section defines the denotational semantics of πI-processes in terms of labelled event structures. Given a process P, we associate to P an event structure $\mathcal{E}_P$ whose events e represent the occurrence of an action $\lambda(e)$ in the LTS of P. Our main issue is compositionality : the semantics of the process $P \mid Q$ should be defined as $\mathcal{E}_P \parallel \mathcal{E}_Q$ so that the operator $\parallel$ satisfactorily models the parallel composition of P and Q.

B.4.1 Generalised relabelling

It is clear from Definition B.3.4 that the core of the parallel composition of event structures is the definition of a relabelling function encoding the intended synchronisation model. As discussed in the Introduction, name dependences appearing in πI-processes let a synchronisation between two events possibly depend on the previous synchronisations. We then define a generalised relabelling operation where the relabelling of an event depends on (the labels of) its causal history. Such a new operator is well-suited to encode the πI-communication model and allows the semantics of the πI-calculus to be defined as an extension of CCS event structure semantics.

Definition B.4.1 (Generalised Relabelling)**.** Let L and L' be two sets of labels, and let $Pom(L')$ be a pomset (i.e., partially ordered multiset) of labels in L'. Given an event structure $\mathcal{E} = \langle E, \leq, conf(,)\lambda \rangle$ over the set of labels L, and a function $f : Pom(L') \times L \longrightarrow L'$, we define the relabelling operation $\mathcal{E}[f]$ as the event structure $\mathcal{E}' = \langle E, \leq, conf(,)\lambda' \rangle$ with labels in L', where $\lambda' : E \longrightarrow L'$ is defined as follows by induction on the height of an element of E :

$$\text{if } h(e) = 0 \text{ then } \lambda'(e) = f(\,\varnothing,\, \lambda(e)\,)$$

$$\text{if } h(e) = n+1 \text{ then } \lambda'(e) = f(\,\lambda'([e)),\, \lambda(e)\,)$$

1. In [WN95], the definition of parallel composition is $(\mathcal{E}_1 \times \mathcal{E}_2 \setminus X)[f]$, where X is the set of labels (pairs) for which f is undefined. We can prove that such a definition is equivalent to ours, which is more suitable to be generalised to the π-calculus.

In words, an event e is relabelled with a label $\lambda'(e)$ that depends on the (pomset of) labels of the events belonging to its causal history $[e)$.

The set of labels we consider is $L = \{a(x), \overline{a}(x), \tau \mid a, x \in Names\}$. In order to define the parallel composition we also need an auxiliary set of labels $L' = \{a(x), \overline{a}(x), \tau_{x=y} \mid a, x, y \in Names\} \cup \{\mathsf{bad}, \mathsf{hide}\}$, where bad and hide are distinguished labels.

In L', the silent action τ is tagged with the couple of bound names that get identified through the synchronisation. This extra piece of information carried by τ-actions is essential in the definition of the generalised relabelling function. Let for instance e encode the parallel occurrence of two events e_1, e_2 labelled, resp., $x(x')$ and $\overline{y}(y')$, then e_1 and e_2 do synchronise only if x and y are equal, that is only if in the causal history of e there is an event labelled with $\tau_{x=y}$; in such a case e can then be labelled with $\tau_{x'=y'}$.

The distinguished label bad denotes, as before, synchronisations that are not allowed, while the new label hide denotes the hiding of newly generated names. Both labels are finally deleted.

Let $f_\pi : Pom(L') \times (L \uplus \{*\} \times L \uplus \{*\}) \longrightarrow L'$ be the relabelling function defined as :

$$
\begin{array}{rcccl}
f_\pi(X, \langle a(y), \overline{a}(z)\rangle) & = & f_\pi(X, \langle \overline{a}(z), a(y)\rangle) & = & \tau_{y=z} \\
f_\pi(X, \langle a(y), \overline{b}(z)\rangle) & = & f_\pi(X, \langle \overline{b}(z), a(y)\rangle) & = & \left\{ \begin{array}{ll} \tau_{y=z} & \text{if } \tau_{a=b} \in X \\ \mathsf{bad} & \text{otherwise} \end{array} \right. \\
f_\pi(X, \langle \alpha, * \rangle) & = & f_\pi(X, \langle *, \alpha \rangle) & = & \left\{ \begin{array}{ll} \mathsf{hide} & \text{if } \tau_{a=b} \in X \\ & \& \ \alpha = a(y), \overline{a}(y) \\ \alpha & \text{otherwise} \end{array} \right. \\
 & & f_\pi(X, \langle \alpha, \beta \rangle) & = & \mathsf{bad} \quad \text{otherwise}
\end{array}
$$

The function f_π encodes the πI-synchronisation model in that it only allows synchronisations between input and output over the same channel, or over two channels whose names have been identified by a previous communication. The actions over a channel a that has been the object of a previous synchronisation are relabelled as hide since, according to internal mobility, a is a bound name.

The extra information carried by the τ-actions is only necessary in order to *define* the relabelling, but it should later on be forgotten, as we do not distinguish τ-actions in the LTS. Hence we apply a second relabelling er that simply erases the tags :

$$
er(\alpha) = \left\{ \begin{array}{ll} \tau & \text{if } \alpha = \tau_{x=y} \\ \alpha & \text{otherwise} \end{array} \right.
$$

B.4.2 Definition of the semantics

The semantics of the πI-calculus is then defined as follows by induction on processes, where the parallel composition of event structure is defined by

$$
\mathcal{E}_1 \|_\pi \mathcal{E}_2 = ((\mathcal{E}_1 \times \mathcal{E}_2)\ [f_\pi][er])\ \backslash \{\mathsf{bad}, \mathsf{hide}\}
$$

To deal with recursive definitions, we use an index k to denote the level of unfolding.

$$\begin{aligned}
\{\!|\, \mathbf{0} \,|\!\}_k &= \varnothing \\
\{\!|\, (\boldsymbol{\nu} a)P \,|\!\}_k &= \{\!|\, P \,|\!\}_k \backslash \{l \in L \mid a \text{ is the subject of } l\} \\
\{\!|\, P \mid Q \,|\!\}_k &= \{\!|\, P \,|\!\}_k \parallel_\pi \{\!|\, Q \,|\!\}_k \\
\{\!|\, \textstyle\sum_{i\in I} \pi_i.P_i \,|\!\}_k &= \textstyle\sum_{i\in I} \pi_i.\{\!|\, P_i \,|\!\}_k \\
\{\!|\, A\langle \tilde{x} \mid \mathbf{z}\rangle \,|\!\}_0 &= \varnothing \\
\{\!|\, A\langle \tilde{x} \mid \mathbf{z}, \rangle \,|\!\}_{k+1} &= \{\!|\, P_A\{\tilde{y}/\tilde{x}\}\{\mathbf{w}/\mathbf{z}\} \,|\!\}_k
\end{aligned}$$

Recall that all operators on event structures are continuous with respect to the prefix order. It is thus easy to show that, for any k, $\{\!|\, P \,|\!\}_k \leq \{\!|\, P \,|\!\}_{k+1}$. We define $\{\!|\, P \,|\!\}$ to be the limit of the increasing chain $...\{\!|\, P \,|\!\}_k \leq \{\!|\, P \,|\!\}_{k+1} \leq \{\!|\, P \,|\!\}_{k+2}...$:

$$\{\!|\, P \,|\!\} = \sup_{k\in\mathbb{N}} \{\!|\, P \,|\!\}_k$$

Since all operators are continuous w.r.t. the prefix order we have the following result :

Theorem B.4.2 (Compositionality)**.** *The semantics $\{\!|\, P \,|\!\}$ is compositional, i.e.*
- $\{\!|\, P \mid Q \,|\!\} = \{\!|\, P \,|\!\} \parallel_\pi \{\!|\, Q \,|\!\}$,
- $\{\!|\, \sum_{i\in I} \pi_i.P_i \,|\!\} = \sum_{i\in I} \pi_i.\{\!|\, P_i \,|\!\}$, *and*
- $\{\!|\, (\boldsymbol{\nu} a)P \,|\!\}k = \{\!|\, P \,|\!\}\backslash\{l \in L \mid a \text{ is the subject of } l\}$.

B.4.3 Examples

Example B.4.1. As a first example, consider the process $P = a(x).\overline{x}(u) \mid \overline{a}(z).z(v)$ discussed in the Introduction. We show in the following the two event structures $\mathcal{E}_1, \mathcal{E}_2$ associated to the basic threads, as well as the event structure corresponding to $\{\!|\, P \,|\!\} = \mathcal{E}_1 \parallel_\pi \mathcal{E}_2$. Figure B.3 shows two intermediate steps involved in the construction of $\{\!|\, P \,|\!\}$, according to the definition of the parallel composition operator.

Example B.4.2. As a second example, consider $Q = \overline{a}(w) \mid P$, where P is the process above. In Q two different communications may take place along the channel a : either the fresh name w is sent, and the resulting process is stuck, or the two threads in P can synchronise as before establishing a private channel for a subsequent communication. The behaviour of Q is illustrated by the following event structure which corresponds to $\{\!|\, Q \,|\!\} = \mathcal{E}_3 \parallel_\pi \{\!|\, P \,|\!\}$, where $\mathcal{E}_3 = \{\!|\, \overline{a}(w) \,|\!\}$ is a simple event structure consisting of a single event labeled by $\overline{a}(w)$.

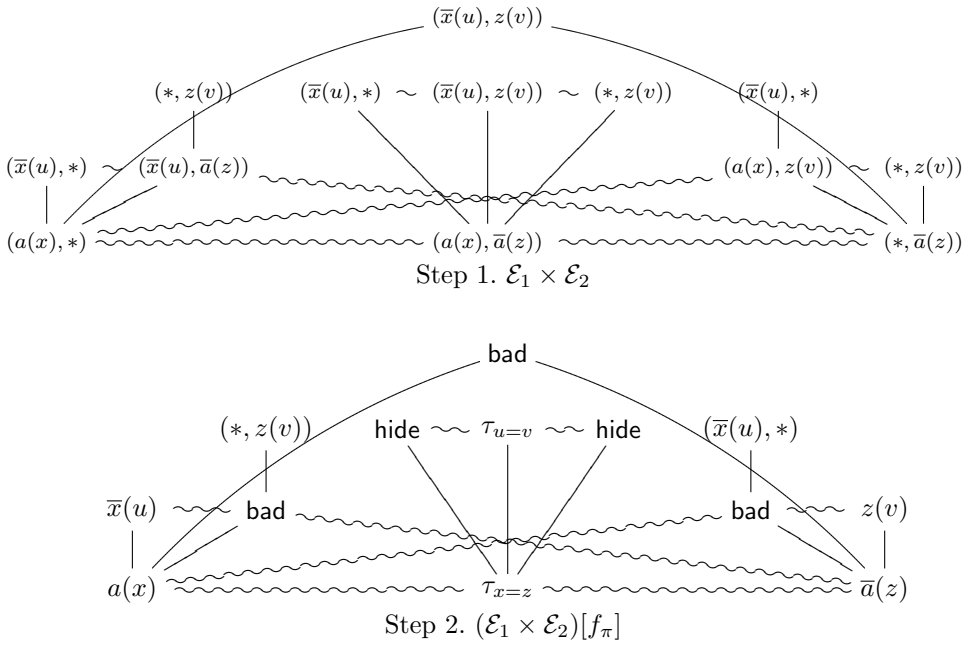
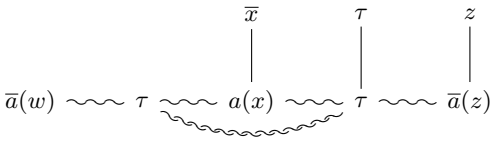
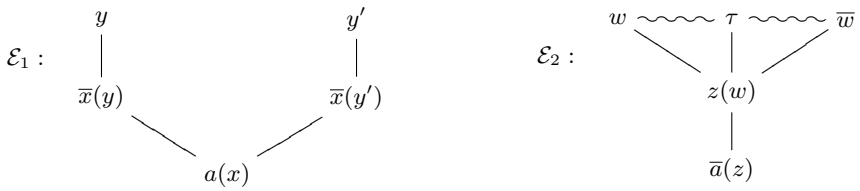


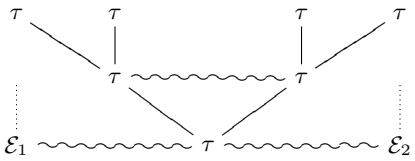
FIGURE B.3 – Event structure corresponding to $a(x).\overline{x}(u) \mid \overline{a}(z).z(v)$



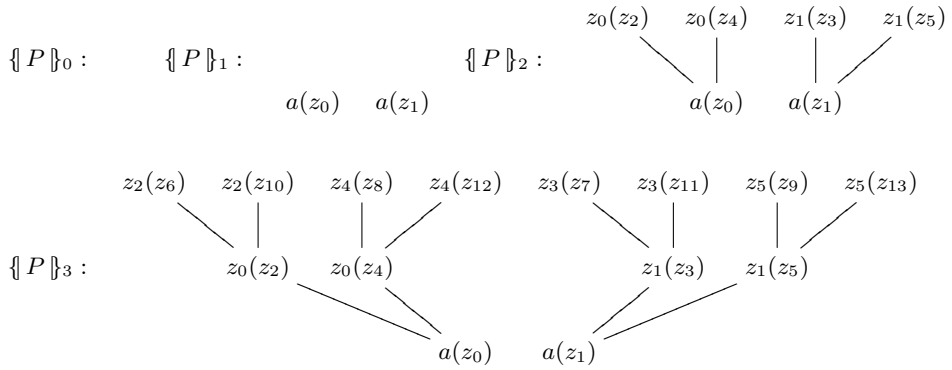
Example B.4.3. As a further example, consider the process $R = a(x).\big(\overline{x}(y).y \mid \overline{x}(y').y'\big) \mid \overline{a}(z).\big(z(w).(w \mid \overline{w})\big)$ whose two threads correspond to the following two event structures :



R allows a first communication on a that identifies x and z and triggers a further synchronisation with one of the outputs over x belonging to $\mathcal{E}_1$. This second communication identifies w with either y or y', which can now compete with w for the third synchronisation. The event structure corresponding to $\{\!|\, R \,|\!\} = \mathcal{E}_1 \parallel_\pi \mathcal{E}_2$ is the following.



Example B.4.4. Consider the recursive process, seen in Example B.2.1, $A(x \mid \mathbf{z}) = x(z_0).A\langle z_0 \mid \mathbf{z}'\rangle \mid x(z_1).A\langle z_1 \mid \mathbf{z}''\rangle$, where $\mathbf{z}'(n) = \mathbf{z}(2n{+}2)$ and $\mathbf{z}''(n) = \mathbf{z}(2n{+}3)$. In the following, we draw the first approximations of the semantics of $P = A\langle a \mid \mathbf{z}\rangle$:



B.4.4 Properties of the semantics

The operational correspondence is stated in terms of the labelled transition system defined in Section B.3.

Theorem B.4.3 (Operational Adequacy)**.** *Suppose* $P \xrightarrow{\beta} P'$ *in the* πI*-calculus. Then* $\{\!|\, P \,|\!\} \xrightarrow{\beta} \cong \{\!|\, P' \,|\!\}$*. Conversely, suppose that* $\{\!|\, P \,|\!\} \xrightarrow{\beta} \mathcal{E}'$*. Then there exists* P' *such that* $P \xrightarrow{\beta} P'$ *and* $\{\!|\, P' \,|\!\} \cong \mathcal{E}'$*.*

The proof technique is similar to the one used in [VY06], but it takes into account the generalised relabelling. As an easy corollary, we get that if two πI processes have isomorphic event structure semantics, their LTSs are isomorphic too. This clearly implies soundness w.r.t. bisimilarity.

Theorem B.4.4 (Soundness)**.** *If* $\{\!|\, P \,|\!\} \cong \{\!|\, Q \,|\!\}$*, then* $P \sim Q$*.*

The converse of the soundness theorem (i.e. completeness) does not hold. In fact this is always the case for event structure semantics (for instance the one in [Win82]), because bisimilarity abstracts away from causal relations, which are instead apparent in the event structures. As a counterexample, we have $a.b + b.a \;\sim\; a \mid b$ but $\{\!|\, a.b + b.a \,|\!\} \ncong \{\!|\, a \mid b \,|\!\}$.

Isomorphism of event structures is indeed a very fine equivalence, however it is, in a sense behavioural, as it is *strictly* coarser than structural congruence.

Proposition B.4.5. *If $P \equiv Q$ then $\{\!|\, P \,|\!\} \cong \{\!|\, Q \,|\!\}$*

The converse of the previous proposition does not hold : $\{\!|\, (\boldsymbol{\nu} a)a.P \,|\!\} \cong \{\!|\, \mathbf{0} \,|\!\} = \varnothing$ but $(\boldsymbol{\nu} a)a.P \not\equiv \mathbf{0}$. Also, the two processes $(\boldsymbol{\nu} a)(a(x).\overline{x}(u) \mid \overline{a}(y).y(v))$ and $(\boldsymbol{\nu} a, b)(a(x).\overline{b}(u) \mid \overline{a}(y).b(v))$ are not structurally congruent, but they both correspond to the same event structure containing only two events e_1, e_2 with $e_1 \leq e_2$ and $\lambda(e_1) = \lambda(e_2) = \tau$.

B.4.5 Semantics of the asynchronous calculus

The event structure semantics of the asynchronous πI-calculus requires to encode the output process $\overline{a}(x)P$, introducing the following novel operator, called *rooting*.

Definition B.4.6 (Rooting $a[X].\mathcal{E}$)**.** Let $\mathcal{E}$ be an event structure labelled over L, let a be a label and $X \subseteq L$ be a set of labels. We define the rooting operation $a[X].\mathcal{E}$ as the event structure $\mathcal{E}' = \langle E', \leq', conf('), \lambda' \rangle$, where $E' = E \uplus \{e'\}$ for some new event e', $\leq'$ coincides with $\leq$ on E and for every $e \in E$ such that $\lambda(e) \in X$ we have $e' \leq' e$, the conflict relation $conf(')$ coincides with $conf()$, that is e' is in conflict with no event. Finally, λ' coincides with λ on E and $\lambda'(e') = a$.

The rooting operation adds to the event structure a new event, labeled by a, which is put below the events with labels in X (and any event above them). This operation is used to give the semantics of asynchronous bound output : given a process $\overline{a}(x)P$, every action performed by P that depends on x should be rooted with $\overline{a}(x)$. In addition to that, in order to model asynchrony, we need to also consider the possible synchronisations between $\overline{a}(x)$ and P (for example, consider $\overline{a}(x)a(z).b.x$, whose operational semantics allows an initial synchronisation between $\overline{a}(x)$ and $a(z).b.x$).

The formal construction is then obtained as follows. Given a process $\overline{a}(x)P$, every action performed by P that has x as subject is rooted with a distinctive label $\perp$. The resulting structure is composed in parallel with $\overline{a}(x)$, so that (i) every "non-blocked" action in P, (i.e. every action that does not depend on x) can synchronise with $\overline{a}(x)$, and (ii) the actions rooted by $\perp$ (i.e. those depending on x) become causally dependent on the action $\overline{a}(x)$.

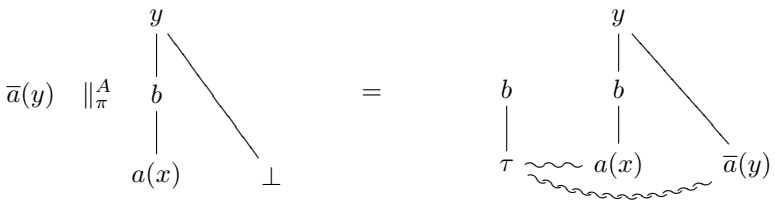
Such a composition is formalised the parallel composition operator $\|^A_\pi$ built around the generalised relabelling function $f^A_\pi : Pom(L') \times (L \uplus \{*, \perp\} \times L \uplus \{*, \perp\}) \longrightarrow L'$ that extends f_π with the following two clauses dealing with the new labels :

$$\begin{array}{rcl} f^A_\pi(X, \langle \perp, \overline{a}(x) \rangle) & = & f^A_\pi(X, \langle \overline{a}(x), \perp \rangle) \;=\; \overline{a}(x) \\ f^A_\pi(X, \langle \perp, * \rangle) & = & f^A_\pi(X, \langle *, \perp \rangle) \;=\; \mathsf{bad} \end{array}$$

The denotational semantics of asynchronous πI-processes is then identical to that in Section B.4, with a new construction for the output :

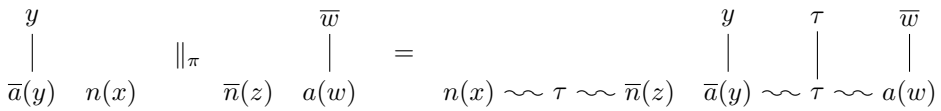
$$\{\!|\, \overline{a}(x)P \,|\!\}_k \;=\; \overline{a}(x) \;\|^A_\pi\; \perp[X].\{\!|\, P \,|\!\}_k \qquad X = \{\alpha \in L \mid x \text{ is the subject of } \alpha\}$$

Example B.4.5. Let R be the process $\overline{a}(y)(a(x).b.y)$; its semantics is defined by the following event structure :



First a new event labelled by $\perp$ is added below any event whose label has y as subject. In this case there is only one such event, labelled by y. Then the resulting event structure is put in parallel with the single event labelled by $\overline{a}(y)$. This event can synchronise with the $\perp$ event or with the $a(x)$ event. The first synchronisation simply substitutes the label $\overline{a}(y)$ for $\perp$. The second one behaves as a standard synchronisation.

Example B.4.6. Consider the process $P = \overline{a}(y)(n(x) \mid y) \mid \overline{n}(z)(a(w).\overline{w})$. The semantics of P is the following event structure :



Note that the causality between the $a(w)$ event and the $\overline{w}$ event is both object *and* subject, and it is due to the prefix constructor. The causality between the $\overline{a}(y)$ event and the y event is only object, and it is due to the rooting.

As for the synchronous case, the semantics is adequate with respect to the labelled transition system.

Theorem B.4.7 (Operational Adequacy)**.** *Suppose* $P \xrightarrow{\beta} P'$ *in the aynchronous* πI*-calculus. Then* $\{\!|\, P \,|\!\} \xrightarrow{\beta} \cong \{\!|\, P' \,|\!\}$*. Conversely, suppose that* $\{\!|\, P \,|\!\} \xrightarrow{\beta} \mathcal{E}'$*. Then there exists* P' *such that* $P \xrightarrow{\beta} P'$ *and* $\{\!|\, P' \,|\!\} \cong \mathcal{E}'$*.*

The proof is analogous to the synchronous case, with a case analysis for the rooting.

Theorem B.4.8 (Soundness)**.** *If* $\{\!|\, P \,|\!\} \cong \{\!|\, Q \,|\!\}$*, then* $P \sim Q$*.*

B.5 Free names

We present the event structure semantics of the full π-calculus in two phases, dealing separately with the two main issues of the calculus. We start in this section discussing free name passing, and we postpone to the next section the treatment of scope extrusion.

The core of a compositional semantics of a process calculus is parallel composition. When a process P is put in parallel with another process, new synchronizations can be triggered. Hence the semantics of P must also account for "potential" synchronizations that might be activated by parallel compositions. In Winskel's event structure semantics of CCS [Win82], the parallel composition is defined as a product in a suitable category followed by relabelling and hiding, as we have presented in Section B.3. For the semantics of the π-calculus, when the parallel composition of two event structures is computed, synchronisation events that involve input and output on different channels cannot be hidden straight away. If at least one of the two channels is a variable, then it is possible that, after prefixing and parallel composition, the two channels will be made equal.

We then resort to a technique similar to the one used in [CVY07] : we consider a generalized notion of relabelling that takes into account the history of a (synchronization) event. Such a relabelling is defined according to the following ideas :

- each pair $(a(x), \overline{a}\langle b\rangle)$ made of two equal names or two equal variables is relabelled $\tau_{x\to b}$, indicating that it represents a legal synchronization where b is substituted for x. Moreover, such a substitution must be propagated in all the events that causally depend on this synchronization. Anyway, after all substitution have taken place, there is no need to remember the extra information carried by the τ action, than the subscripts of the τ events are erased.
- Synchronisations pairs, like $(a(x), \overline{b}\langle c\rangle)$, that involve different channels (at least) one of which is a variable, is relabelled $(a(x), \overline{b}\langle c\rangle)_{x\to c}$, postponing the decision whether they represent a correct synchronization or not.
- Each pair $(n(x), \overline{m}\langle b\rangle)$ made of two different names is relabelled bad to denote a synchronization that is not allowed.

Definition B.5.1 (Generalised Relabelling)**.** Let L and L' be two sets of labels, and let $Pom(L')$ be a pomset (i.e., partially ordered multiset) of labels in L'. Given an event structure $\mathcal{E} = \langle E, \leq, \smile, \lambda\rangle$ over the set of labels L, and a function $f : Pom(L') \times L \longrightarrow L'$, we define the relabelling operation $\mathcal{E}[f]$ as the event structure $\mathcal{E}' = \langle E, \leq, \smile, \lambda'\rangle$ with labels in L', where $\lambda' : E \longrightarrow L'$ is defined as follows by induction on the height of an element of E : if $h(e) = 0$ then $\lambda'(e) = f(\,\varnothing,\, \lambda(e)\,)$, if $h(e) = n + 1$ then $\lambda'(e) = f(\,\lambda'([e)),\, \lambda(e)\,)$.

In words, an event e is relabelled with a label $\lambda'(e)$ that depends on the (pomset of) labels of the events belonging to its causal history $[e)$.

In the case of π-calculus with free names, let $L = \{a(x), \overline{a}\langle b\rangle, \tau \;\mid\; a, b \in Names \cup Variables,\; x \in Variables\}$ be the set of labels used in the LTS of π-calculus without restriction. We define the relabelling function needed by the parallel composition operation around the extended set of labels $L' = L \cup \{(\alpha, \beta)_{x\to b} \mid \alpha, \beta \in L\} \cup \{\tau_{x\to b},\, \mathsf{bad}\}$, where bad is a distinguished label. The relabelling function $f_\pi : Pom(L') \times (L' \uplus \{*\} \times L' \uplus \{*\}) \longrightarrow L'$ is defined as

follows (we omit the symmetric clauses) :

$$
\begin{array}{rcl}
f_\pi(X, \langle a(y), \overline{a}\langle b\rangle\rangle) & = & \tau_{y\to b} \\
f_\pi(X, \langle a(x), \overline{y}\langle c\rangle\rangle) & = & \left\{ \begin{array}{ll} \tau_{x\to c} & \text{if } \alpha_{y\to a} \in X \\ (a(x), \overline{y}\langle c\rangle)_{x\to c} & \text{otherwise} \end{array} \right. \\
f_\pi(X, \langle n(y), \overline{m}\langle b\rangle\rangle) & = & \mathsf{bad} \\
f_\pi(X, \langle y(x), \overline{a}\langle n\rangle\rangle) & = & \left\{ \begin{array}{ll} \tau_{x\to n} & \text{if } \alpha_{y\to a} \in X \\ (y(x), \overline{a}\langle n\rangle)_{x\to n} & \text{otherwise} \end{array} \right. \\
f_\pi(X, \langle y(x), *\rangle) & = & \left\{ \begin{array}{ll} a(x) & \text{if } \alpha_{y\to a} \in X \\ y(x) & \text{otherwise} \end{array} \right. \\
f_\pi(X, \langle \overline{y}\langle b\rangle, *\rangle) & = & \left\{ \begin{array}{ll} \overline{a}\langle b\rangle & \text{if } \alpha_{y\to a} \in X \\ \overline{y}\langle b\rangle & \text{otherwise} \end{array} \right. \\
f_\pi(X, \langle \alpha, *\rangle) & = & \alpha \\
f_\pi(X, \langle \alpha, \beta\rangle) & = & \mathsf{bad} \quad \text{otherwise}
\end{array}
$$

The extra information carried by the τ-actions, differently from that of "incomplete synchronization" events, is only necessary in order to *define* the relabelling, but there is no need to keep it after the synchronization has been completed. Hence we apply a second relabelling er that simply erases the subscript of τ actions.
The semantics of the π-calculus is then defined as follows by induction on processes, where the parallel composition of event structure is defined by

$$\mathcal{E}_1 \|_\pi \mathcal{E}_2 = ((\mathcal{E}_1 \times \mathcal{E}_2)\ [f_\pi][er])\ \backslash\{\mathsf{bad}\}$$

To deal with recursive definitions, we use an index k to denote the level of unfolding.

$$
\begin{array}{rcl}
\{\!|\, \mathbf{0} \,|\!\}_k & = & \varnothing \\
\{\!|\, \textstyle\sum_{i\in I} \pi_i.P_i \,|\!\}_k & = & \textstyle\sum_{i\in I} \pi_i.\{\!|\, P_i \,|\!\}_k \\
\{\!|\, P \mid Q \,|\!\}_k & = & \{\!|\, P \,|\!\}_k \parallel_\pi \{\!|\, Q \,|\!\}_k \\
\{\!|\, A\langle \tilde{y}, \tilde{q} \mid \mathbf{w}, \mathbf{m}\rangle \,|\!\}_0 & = & \varnothing \\
\{\!|\, A\langle \tilde{y}, \tilde{q} \mid \mathbf{w}, \mathbf{m}\rangle \,|\!\}_{k+1} & = & \{\!|\, P_A\{^{\tilde{y},\tilde{q}}/_{\tilde{x},\tilde{p}}\}\{^{\mathbf{w},\mathbf{m}}/_{\mathbf{z},\mathbf{n}}\} \,|\!\}_k
\end{array}
$$

Recall that all operators on event structures are continuous with respect to the prefix order. It is thus easy to show that, for any k, $\{\!|\, P \,|\!\}_k \leq \{\!|\, P \,|\!\}_{k+1}$. We define $\{\!|\, P \,|\!\}$ to be the limit of the increasing chain $...\{\!|\, P \,|\!\}_k \leq \{\!|\, P \,|\!\}_{k+1} \leq \{\!|\, P \,|\!\}_{k+2}...$, that is $\{\!|\, P \,|\!\} = \sup_{k\in\mathbb{N}} \{\!|\, P \,|\!\}_k$
Since all operators are continuous w.r.t. the prefix order we have :

Theorem B.5.2 (Compositionality)**.** *The semantics* $\{\!|\, P \,|\!\}$ *is compositional, i.e.* $\{\!|\, P \mid Q \,|\!\} = \{\!|\, P \,|\!\} \parallel_\pi \{\!|\, Q \,|\!\}$, $\{\!|\, \sum_{i\in I} \pi_i.P_i \,|\!\} = \sum_{i\in I} \pi_i.\{\!|\, P_i \,|\!\}$,

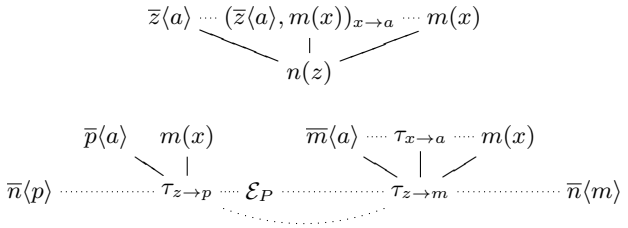


FIGURE B.4 –

Example B.5.1. As an example, consider the process $P = n(z).(\overline{z}\langle a\rangle \mid m(x))$. The synchronization along the channel m can be only performed if the previous synchronization along n substitutes the variable z with the name m. Accordingly, the semantics of the process P is the first event structure in Figure B.4, denoted by $\mathcal{E}_P$. Moreover, the second structure in Figure B.4 corresponds to the semantics of the process $P \mid \overline{n}\langle m\rangle \mid \overline{n}\langle p\rangle$.

The following theorem shows that the event structure semantics is operationally correct. Indeed, given a process P, the computational steps of P in the LTS of Section B.2 are reflected by the semantics $\{\!|\, P \,|\!\}$.

Theorem B.5.3 (Operational Adequacy)**.** *Let $\beta \in L = \{a(x), \overline{a}\langle b\rangle, \tau\}$. Suppose $P \xrightarrow{\beta} P'$ in the π-calculus. Then $\{\!|\, P \,|\!\} \xrightarrow{\beta} \cong \{\!|\, P' \,|\!\}$. Conversely, suppose $\{\!|\, P \,|\!\} \xrightarrow{\beta} \mathcal{E}'$. Then there exists P' such that $P \xrightarrow{\beta} P'$ and $\{\!|\, P' \,|\!\} \cong \mathcal{E}'$.*

Note that the correspondence holds for those labels that appear in the LTS of the calculus. Labels that identify "incomplete synchronizations" have been introduced in the event structure semantics for the sake of compositionality, but they are not considered in the theorem above since they do not correspond to any operational step.

B.6 Scope extrusion

In this section we show how the causal dependencies introduced by scope extrusion can be captured by event structure-based models. As we discussed in Section B.1, the communication of bound names implies that any action with a bound subject causally depends on a dynamic set of possible extruders of that bound subject. Hence dealing with scope extrusion requires modelling some form of disjunctive causality. Prime event structures are stable models that represent an action α that can be caused either by the action β_1 or the action β_2 as two different events e, e' that are both labelled α but e causally depends on the event labeled β_1 while e' is caused by the event labelled β_2. In order to avoid the proliferation of events representing the same action with different extruders, we follow here a different approach, postponing to the next section a more detailed discussion on the use of prime event structures.

B.6.1 Event structure with bound names

We define the semantics of the full π-calculus in terms of pairs $(\mathcal{E}, X)$, where $\mathcal{E}$ is a prime event structure, and is a X a set of names. We call such a pair an *event structure with bound names*. Intuitively, the causal relation of $\mathcal{E}$ encodes the structural causality of a process, while the set X records bound names. The names in the set X affect the notion of computation on $\mathcal{E}$: for instance, if a minimal event in $\mathcal{E}$ has a label whose subject is in X, then such a minimal event cannot be executed since it requires a previous extrusion. In other terms, given a pair $(\mathcal{E}, X)$ we define a notion of *permitted configurations*, ensuring that any computation that contains an action whose subject is a bound name, also contains a previous extrusion of that name. Objective causality is then implicitly captured by permitted configurations.

Definition B.6.1 (Semantics)**.** The semantics of the full π-calculus is inductively defined as follows, where k denote the level of unfolding of recursive definitions, and we write $\mathcal{E}^k_P$, resp. X^k_P, for the first, resp. the second, projection of the pair $\{\!|\, P \,|\!\}_k$

$$\begin{aligned}
\{\!|\, \mathbf{0} \,|\!\}_k &= (\varnothing, \varnothing) \\
\{\!|\, \textstyle\sum_{i\in I} \pi_i.P_i \,|\!\}_k &= (\, \textstyle\sum_{i\in I} \pi_i.\mathcal{E}^k_{P_i} \;,\; \biguplus_{i\in I} X^k_{P_i} \,) \\
\{\!|\, P \mid Q \,|\!\}_k &= (\, \mathcal{E}^k_P \parallel_\pi \mathcal{E}^k_Q \;,\; X^k_P \uplus X^k_Q \,) \\
\{\!|\, (\boldsymbol{\nu} n)P \,|\!\}_k &= (\, \mathcal{E}^k_P \;,\; X^k_P \uplus \{n\} \,) \\
\{\!|\, A\langle \tilde{y}, \tilde{q} \mid \mathbf{w}, \mathbf{m}\rangle \,|\!\}_0 &= (\varnothing, \{\mathbf{w}(\mathbb{N})\}) \\
\{\!|\, A\langle \tilde{y}, \tilde{q} \mid \mathbf{w}, \mathbf{m}\rangle \,|\!\}_{k+1} &= (\, \mathcal{E}^k_{P_A\{^{\tilde{y},\tilde{q}}/_{\tilde{x},\tilde{p}}\}\{^{\mathbf{w},\mathbf{m}}/_{\mathbf{z},\mathbf{n}}\}}, \;\; \{\mathbf{w}(\mathbb{N})\} \,)
\end{aligned}$$

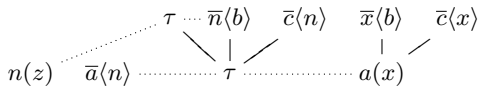
It is easy to show that, for any k, $\mathcal{E}^k_P \leq \mathcal{E}^{k+1}_P$ and $X^k_P = X^{k+1}_P = X_P$. Then the semantics of a process P is defined as the following limit :

$$\{\!|\, P \,|\!\} \;=\; (\, \sup_{k\in\mathbb{N}} \mathcal{E}^k_P \;,\; X_P \,) \;.$$

This semantics is surprisingly simple : a restricted process like $(\boldsymbol{\nu} n)P$ is represented by a prime event structure that encodes the process P where the scope of n has been opened, and we collect the name n in the set of bound names. As for parallel composition, the semantics $\{\!|\, P \mid Q \,|\!\}$ is a pair $(\mathcal{E}, X)$ where X collects the bound names of both $\{\!|\, P \,|\!\}$ and $\{\!|\, Q \,|\!\}$ (remind that we assumed that bound names are pairwise different), while the event structure $\mathcal{E}$ is obtained exactly as in the previous sections. This is since the event structures that get composed correspond to the processes P and Q where the scope of any bound name has been opened. It is immediate to prove the following property.

Proposition B.6.2. *Let P, Q be any two processes of the π-calculus, then $\{\!|\, (\boldsymbol{\nu} n)P \mid Q \,|\!\} = \{\!|\, (\boldsymbol{\nu} n)(P \mid Q) \,|\!\}$.*

Example B.6.1. Consider the process $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid n(z)) \mid a(x).(\overline{x}\langle b\rangle \mid \overline{c}\langle x\rangle)$, whose first synchronization produces a new extruder $\overline{c}\langle n\rangle$ for the bound name n. The semantics of P is the pair $(\mathcal{E}_P, \{n\})$, where $\mathcal{E}_P$ is the following e.s. :



In order to study the operational correspondence between the LTS semantics of the π-calculus and the event structure semantics above, we first need to adapt the notion of computational steps of the pairs $(\mathcal{E}, X)$. The definition of labelled transitions between prime event structures, i.e., Definition B.3.3, is generalized as follows.

Definition B.6.3 (Permitted Transitions)**.** Let $(\mathcal{E}, X)$ be a labelled event structure with bound names. Let e be a minimal event of $\mathcal{E}$ with $\lambda(e) = \beta$. We define the following permitted labelled transitions :

- $(\mathcal{E}, X) \xrightarrow{\beta} (\mathcal{E}\lfloor e,\ X)$, if $\beta \in \{\tau, a(x), \overline{a}\langle b\rangle\}$ with $a, b \notin X$.
- $(\mathcal{E}, X) \xrightarrow{\overline{a}(n)} (\mathcal{E}\lfloor e,\ X\backslash\{n\})$, if $\beta = \overline{a}\langle n\rangle$ with $a \notin X$ and $n \in X$.

According to this definition, the set of bound names constrains the set of transitions that can be performed. In particular, no transition whose label has a bound subject is allowed. On the other hand, when a minimal event labeled $\overline{a}\langle n\rangle$ is consumed, if the name n is bound, the transition's labels records that this event is indeed a bound output. Moreover, in this case we record that the scope of n is opened by removing n from the set of bound names of the target pair. Finally, observe that the previous definition only allows transitions whose labels are in the set $L = \{\tau, a(x), \overline{a}\langle b\rangle, \overline{a}(n)\}$, which is exactly the sets of labels in the LTS of Section B.2.

Theorem B.6.4 (Operational Adequacy)**.** *Let $\beta \in L = \{a(x), \overline{a}\langle b\rangle, \overline{a}(n), \tau\}$. Suppose $P \xrightarrow{\beta} P'$ in the π-calculus. Then $\{\!|\, P \,|\!\} \xrightarrow{\beta} \cong \{\!|\, P' \,|\!\}$. Conversely, suppose $\{\!|\, P \,|\!\} \xrightarrow{\beta} \mathcal{E}'$. Then there exists P' such that $P \xrightarrow{\beta} P'$ and $\{\!|\, P' \,|\!\} \cong \mathcal{E}'$.*

B.6.2 Subjective and objective causality

Given an event structure with bound names $(\mathcal{E}, X)$, Definition B.6.3 shows that some configuration of $\mathcal{E}$ is no longer allowed. For instance, if e is minimal but its label has a subject that is a name in X, e.g, $\lambda(e) = n(x)$ with $n \in X$, then the configuration $\{e\}$ is no longer allowed since the event e requires a previous extrusion of the name n.

Definition B.6.5 (Permitted Configuration)**.** Let $(\mathcal{E}, X)$ be an event structure with bound names. Given a configuration C of $\mathcal{E}$, we say that C is *permitted* in $(\mathcal{E}, X)$ whenever, for any $e \in C$ whose label has subject n with $n \in X$,

- $C \setminus \{e\}$ is permitted, and
- $C \setminus \{e\}$ contains an event e' whose label is an output action with object n.

The first item of the definition above is used to avoid circular definitions that would allow wrong configurations like $\{\overline{n}\langle m\rangle, \overline{m}\langle n\rangle\}$ with $X = \{n, m\}$. Now, the two forms of causality of the π-calculus can be defined using event structures with bound names and permitted configurations.

Definition B.6.6 (Subjective and Objective Causality)**.** Let P be a process of the π-calculus, and $\{\!|\, P \,|\!\} = (\mathcal{E}_p, X_P)$ be its semantics. Let be $e_1, e_2 \in E_P$, then
- e_2 has a *subjective dependence* on e_1 if $e_1 \leq_{\mathcal{E}_P} e_2$;
- e_2 has a *objective dependence* on e_1 if (i) the label of e_1 is the output of a name in X which is also the subject of the label of e_2, and if (ii) there exists a configuration C that is permitted in $(\mathcal{E}, X)$ and that contains both e_1 and e_2.

Example B.6.2. Let consider again the process P in Example B.6.1. The configurations $C_1 = \{\overline{a}\langle n\rangle, n(z)\}$ and $C_2 = \{\tau, \overline{c}\langle n\rangle, n(z)\}$ are both permitted by $\{\!|\, P \,|\!\}$, and they witness the fact that the action $n(z)$ has an objective dependence on $\overline{a}\langle n\rangle$ and [2] on $\overline{c}\langle n\rangle$.

Example B.6.3. Let be $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid \overline{b}\langle n\rangle \mid n(x))$, then $\{\!|\, P \,|\!\} = (\mathcal{E}_P, \{n\})$ where $\mathcal{E}_P$ has three concurrent events. In this process there is no subjective causality, however the action $n(x)$ has an objective dependence on $\overline{a}\langle n\rangle$ and on $\overline{b}\langle n\rangle$ since both $C_1 = \{\overline{a}\langle n\rangle, n(x)\}$ and $C_2 = \{\overline{b}\langle n\rangle, n(x)\}$ are permitted configurations.

Example B.6.4. Let be $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle.\overline{b}\langle n\rangle.n(x))$, then $\{\!|\, P \,|\!\} = (\mathcal{E}_P, \{n\})$ where $\mathcal{E}_P$ is a chain of three events. According to the causal relation of $\mathcal{E}_P$, the action $n(x)$ has a structural dependence on both the outputs. Moreover, the permitted configuration $C = \{\overline{a}\langle n\rangle, \overline{b}\langle n\rangle, n(x)\}$ shows that $n(x)$ has an objective dependence on $\overline{a}\langle n\rangle$ and on $\overline{b}\langle n\rangle$ [3].

B.6.3 The meaning of labelled causality

In this work we focus on compositional semantics, studying a true concurrent semantics that operationally matches the LTS semantics of the π-calculus. Alternatively, one could take as primitive the reduction semantics of the π-calculus, taking the perspective that only τ-events are "real" computational steps of a concurrent system. Therefore one could argue that the concept of causal dependency makes only sense between τ events. In this perspective, we propose to interpret the causal relation between non-τ events as an anticipation of the causal relations involving the synchronizations they will take part in. In other terms, non-τ events (from now on simply called labelled events) represent "incomplete" events, that are waiting for a synchronization or a substitution to be completed. Hence we can prove that in our semantics two labelled events e_1 and e_2 are causally dependent *if and only if* the τ-events they can take part in

2. We could also say that $n(z)$ objectively depends *either* on $\overline{a}\langle n\rangle$ *or* on $\overline{c}\langle n\rangle$.
3. In this case we do not know which of the two outputs really extruded the bound name, accordingly to the inclusive disjunctive causality approach we are following.

are causally dependent. This property is expressed by the following theorem in terms of permitted configurations. Recall that the parallel composition of two event structures is obtained by first constructing the cartesian product. Therefore there are projection morphisms π_1, π_2 on the two composing structures. Let call τ-configuration a configuration whose events are all τ-events. Note that every τ-configuration is permitted.

Theorem B.6.7. *Let P be a process. A configuration C is permitted in $\{\!|\, P \,|\!\}$ if and only if there exists a process Q and a τ-configuration C' in $\{\!|\, P \mid Q \,|\!\}$ such that $\pi_1(C') = C$.*

Let e_1, e_2 be two labelled events of $\{\!|\, P \,|\!\} = (\mathcal{E}_p, X_p)$. If e_1, e_2 are structurally dependent, i.e., $e_1 \leq_{\mathcal{E}_P} e_2$, then such a structural dependence is preserved and reflected in the τ-actions they are involved in because of the way the parallel composition of event structures is defined. On the other hand, let be e_1, e_2 objectively dependent. Consider the parallel composition $(\mathcal{E}_P || _\pi \mathcal{E}_Q, X_P \cup X_Q)$ for some Q such that there is a τ events e_2' in $\mathcal{E}_P \;||\; _\pi \mathcal{E}_Q$ with $\pi_1(e_2') = e_2$ and $[e_2')$ is a τ-configuration. Then there must be an event $e_1' \in [e_2')$ such that $\pi_1(e_1') = e_1$.

B.7 Discussion

As we discussed in Section B.1, objective causality introduced by scope extrusion requires for the π-calculus a semantic model that is able to express some form of disjunctive causality. In the previous section we followed an approach that just ensures that some extruder (causally) precedes any action with a bound subject. However, we could alternatively take the effort of tracing the identity of the actual extruders. We could do it by duplicating the events corresponding to actions with bound subject and letting different copies depend on different, alternative, extruders. Such a duplication allows to use prime event structures as semantics models. In this section we discuss this alternative approach showing to what extent it can be pursued.

As a first example, the semantics of the process $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid \overline{b}\langle n\rangle \mid n(x))$, containing two possible extruders for the action $n(x)$, can be represented by left-most prime event structure in Figure B.5. When more than a single action use as subject the same bound name, each one of these actions must depend on one of the possible extruders. Then the causality of the process $(\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid \overline{b}\langle n\rangle \mid n(x).n(y))$, represented by the rightmost event structure in Figure B.5, shows that the two read actions might depend either on the same extruder or on two different extrusions.

Things get more complicate when dealing with the dynamic addition of new extruders by means of communications. In order to guarantee that there are distinct copies of any event with a bound subject that causally depend on different extruders, we have to consider the objective causalities generated by a communication. More precisely, when the variable x is substituted with a bound name n by effect of a synchronization, (i) any action with subject

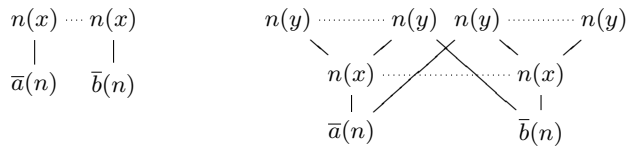
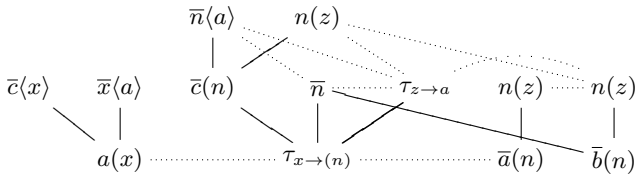


FIGURE B.5 –

x appearing in the reading thread becomes an action that requires a previous scope extrusion, and (ii) the outputs with object x become new extruders for any action with subject n or x. To exemplify, consider the process $P' = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle \mid \overline{b}\langle n\rangle \mid n(z)) \mid a(x).(\overline{x}\langle a\rangle \mid \overline{c}\langle x\rangle)$ that initially contains two extruders for n, and with the synchronization along the channel a evolves to $(\boldsymbol{\nu} n)(\overline{b}\langle n\rangle \mid n(z) \mid \overline{n}\langle a\rangle \mid \overline{c}\langle n\rangle)$. Its causal semantics can be represented with the following prime event structure :



The read action $n(z)$ may depend on one of the two initial extruders $\overline{a}\langle n\rangle$ and $\overline{b}\langle n\rangle$, or on the new extruder $\overline{c}\langle n\rangle$ that is generated by the first communication. Accordingly, three different copies of the event $n(z)$ appear over each of the three extruders. On the other hand, the output action on the bound name n is generated by the substitution entailed by the communication along the channel a, hence any copy of that action keeps a (structural) dependence on the corresponding τ event. Moreover, since it is an action with bound subject, there must be a copy of it for each of the remaining extruders of n, that is $\overline{b}\langle n\rangle$ and $\overline{c}\langle n\rangle$. To enhance readability, the event structure resulting from the execution of the communication along the channel a is the leftmost e.s. in Figure B.6.

So far so good, in particular it seems possible to let the causal relation of prime event structures encode both structural and objective causality of π-processes. However, this is not the case. To see this, consider the process $P = (\boldsymbol{\nu} n)(\overline{a}\langle n\rangle.\overline{b}\langle n\rangle.n(z))$ of Example B.6.4. If we just duplicate the event $n(z)$ to distinguish the fact that it might depend on an extrusion along a or along b, we obtain the rightmost structure in Figure B.6, that we denote $\mathcal{E}_p$. In particular, even if the two copies intends to represent two different objective causalities, nothing distinguishes them since since they both structurally depend on both outputs. This is a problem when we compose the process P in parallel with, e.g., $Q = a(x).\overline{c}\langle x\rangle \mid b(y).\overline{d}\langle y\rangle$. After the two synchronizations we would like to obtain two copies of the read actions on n that depend on the two different remaining extruders $\overline{c}\langle n\rangle$ and $\overline{d}\langle n\rangle$. However, in order to obtain such an event structure as the parallel composition of the semantics of P and the semantics of Q we must be able to record somehow the different objective causality that distinguishes the two copies of $n(z)$ in the semantics of P.

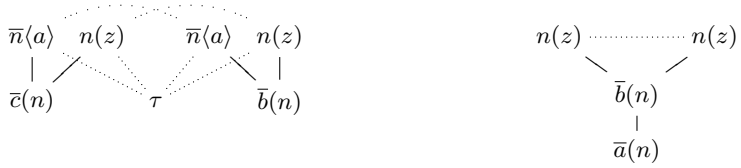


FIGURE B.6 –

The technical solution would be to enrich the event labels so that the label of an event e also records the identity of the extruder events that e (objectively) causally depends on. A precise account of this approach is technically wired and intractable, so that the intuition on the semantics gets lost. Moreover, we think that this final example sheds light on the fact that structural and objective causality of π-processes cannot be expressed by the sole causal relation of event structures. To conclude, at the price of losing the information about which extruder an event depends on, the approach we developed in the previous section brings a number of benefits : it is technically much simpler, it is operationally adequate, it gives a clearer account of the two forms of causality distinctive of π-processes.

B.8 Related and future work

There are several causal models for the π-calculus, that use different techniques. There exist noninterleaving semantics in terms of labelled transition systems, where the causal relations between transitions are represented by "proofs" which allow to distinguish different occurrences of the same transition [San94, BS98, DP99]. In [CS00], a more abstract approach is followed, which involves indexed transition systems. In [JJ95], a semantics of the π-calculus in terms of pomsets is given, following ideas from dataflow theory. The two papers [BG95, Eng96] present Petri nets semantics of the π-calculus. However, none of these aproaches accounts for parallel extrusion. We finally recall [MP95] that introduces a graph rewriting-based semantics of the π-calculus that allows parallel extrusions.

[BMM06] gives an unlabelled event structure semantics of the full calculus which only corresponds to the *reduction* semantics, hence which is not compositional.

We plan for future work the application of the present semantics to the study of a labelled reversible semantics of the π-calculus that would extend the work of Danos and Krivine [DK04]. Phillips and Ulidowski [PU07] noted the strict correspondence between reversible transition systems and event structures. A first step in this direction is [LMS10], which proposes a reversible semantics of the π-calculus that only considers reductions. It would also be interesting to study which kind of configuration structures [vGP09] can naturally include our definition of permitted configuration.

Annexe C

Semantic subtyping for the π-calculus

C.1 Introduction and motivations

C.1.1 Semantic subtyping

The language $\mathbb{C}$Duce [FCB08, Fri04] is a functional programming language for XML manipulation, with a very rich type system. Types and subtyping play a central role in $\mathbb{C}$Duce : for its design (patterns and pattern matching are built around types), for its execution (functions can be overloaded with run-time code selection), and for its implementation (pattern matching compilation and query computation use static type information to optimise execution). All these usages of types rely on a common foundational core : the *semantic subtyping* framework. An introduction to semantic subtyping can be found in [CF05a], while [Cas05] discusses several aspects and perspectives ; technical details are given in [FCB08, Fri04]. In a nutshell, given a typed language with some (possibly recursive) type constructors (e.g., $\rightarrow$, $\times$, `list`(), ...), semantic subtyping is a technique to enrich the language with *type combinators*, i.e. set-theoretic union, intersection, and negation types. The behaviour of combinators is specified via the subtyping relation (rather than via the typing of the terms). The subtyping relation is "semantic" since instead of axiomatising it by a set of inference rules, one describes a set-theoretic interpretation of the types $[\![\ ]\!] : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ (where $\mathcal{P}$ denotes the powerset operator and $\mathcal{D}$ some domain) and then defines the subtyping relation as $s \leq t \stackrel{\text{def}}{\Leftrightarrow} [\![s]\!] \subseteq [\![t]\!]$. Such a set-theoretic interpretation must satisfy at least three design goals.

1. It must ensure that type *combinators* have a set-theoretic interpretation. This is done by imposing that union, intersection, and negation types are respectively interpreted as the set-theoretic union, intersection, and complement operations of $\mathcal{P}(\mathcal{D})$.
2. It must ensure that type *constructors* have a "natural" interpretation

(at least, for what concerns subtyping), e.g., that product types are interpreted as set-theoretic products, function types as sets of maps from domain to co-domain, and so on.

3. It must allow for an interpretation of types as sets of values. This means that if we take as $\mathcal{D}$ the set of values of the language and as interpretation the function that maps a type to the set of all values of that type, then this new interpretation must induce the very same subtyping relation as the one used to type values.

Finding a domain $\mathcal{D}$ and an interpretation function $[\![-]\!]$ that satisfy the last two points is far from being trivial : a set-theoretic interpretation of functional and recursive types or the circularity between the typing of values and definition of subtyping are difficult constraints. As described in [CF05b] and outlined later on, semantic subtyping provides a technique to do so.

C.1.2 Subtyping for processes

In this work we apply the semantic subtyping framework to define a type system for a concurrent process language in which values are exchanged between agents via communication channels that can be dynamically generated. The language we consider is a variant of the asynchronous π-calculus [Bou92, HT91], in which communication is subjected to pattern matching.

There exists a well established literature on typing and subtyping for the π-calculus (e.g. [PS96, Sew98, YH99, SW02]). However, all the approaches we are aware of rely on subtyping relations or on type equivalences that are defined syntactically, by means of structural rules. In our view, such syntactic formalisations of typing relations miss a clean semantic intuition of types. Consider, for example, the type system defined by Hennessy and Riely [HR02], which is one of the most advanced type systems for variants of the π-calculus. It includes read-only and write-only channels, as well as union and intersection types. In that system the following equality is used to *define* the union type :

$$ch^+(\mathtt{t}_1) \vee ch^+(\mathtt{t}_2) = ch^+(\mathtt{t}_1 \vee \mathtt{t}_2) \tag{C.1}$$

where $ch^+(t)$ is the type of channels from which we can only read values of type t, and $\vee$ denotes union. We would like to understand the precise semantic intuition that underlies an equation such as (C.1).

Channels as boxes. In order to understand how channels and channel types relate, we have to provide a semantic account of channels. Our intuition is that a channel is a box in which we can put things (write) and from which we can take things (read). The type of a channel, then, is characterised by the set of the things the box can contain. That is, a channel of type $ch^+(t)$ is a box in which we must expect to find objects of type t and, similarly, a channel of type $ch^-(t)$ is a box in which we are allowed to put objects of type t. This is a particular interpretation (see Section C.5.5 for alternative intuitions), but if one takes this stand, then equality (C.1) does not seem to be justified. Consider the types $ch^+(\mathtt{candy}) \vee ch^+(\mathtt{coal})$ and $ch^+(\mathtt{candy} \vee \mathtt{coal})$. Both represent boxes. If we have a box of the first type, then we expect to find in it either a candy or a

piece of charcoal, but we know it is always one of the two. For instance, if we use the box twice, the second time we will know what present it contains. A box of the second type, instead, is a "surprise box" as it can always give us both candies and charcoal. Our intuition suggests that the two types above are different because they characterise two different kinds of objects.

The role of the language. So why did Hennessy and Riely require (C.1) ? The point is that, if in the language under consideration there is no syntactic construction that can tell apart a `candy` from a `coal` *and then branch*, that is, if it is not possible to branch to different pieces of code for messages of different types (e.g. a typecase, an exception trapping, an overloaded function, ...), then it is not possible to operationally observe any difference between the types in (C.1). Hennessy and Riely do not have such a construction, therefore (C.1) is sound.

On the contrary, suppose we are sent a channel c of type $ch^+(\texttt{candy})\ \texttt{V}\ ch^+(\texttt{coal})$ If it is possible to test whether c is of type $ch^+(\texttt{candy})$ or of type $ch^+(\texttt{coal})$, then we can continue assuming that on c we will receive messages of only one of the two types. In this case a rule such as (C.1) would be unsound, because it would make it possible to receive on c both `candy` and `coal` and this could make the code crash.

We define a variant of the π-calculus that exploits the full power of our new type system, and in particular that permits dynamically testing the type of values received on a channel. We implement the dynamic test by endowing input actions with patterns, and allowing synchronisation when pattern matching succeeds. The result is a simple and elegant formalism that can be easily extended with product types, to obtain a polyadic π-calculus, and with a restricted form of recursive types. To allow for full recursive types, the calculus must be restricted to the *local* variant, where channels received in input can only be used in output position. Finally we show that in that setting the typing and subtyping relations are decidable

C.1.3 Functions as processes

It would be possible to extend the $\mathbb{C}\pi$-calculus with functional types and values, the semantic machinery should be able the scale. However the question arises whether this extension is necessary or whether it is possible to encode functions as processes. It is well known that several such encodings are possible from the λ-calculus into the π-calculus [Mil92, SW02, YBH01]. In the Join-calculus language [F$^+$96], the functional part is simply syntactic sugar for its coding in the concurrent part.

In this work, we describe an encoding of $\mathbb{C}$Duce into the $\mathbb{C}\pi$-calculus. The encoding turned out not to be so straightforward as one may expect. The difficulty arises in finding an encoding of the types that respects the subtyping relation. The Milner-Turner translation of arrow types [SW02] respects the subtyping relation in the context of the simply typed λ-calculus, but it breaks down in the presence of intersection types.

As we detail in Sections C.7 and C.8, the translation we propose sheds new light on the Milner-Turner encoding as it shows the respective roles of argument and return channel that are used to simulate functions in a concurrent world. In particular, it shows that in the presence of type-case, the latter must be scrambled by introducing some noise at the type level so that the receiver cannot gain information by testing the type of the return channel. The translation is a further confirmation of the validity of the equational laws for union and intersection types in the π-calculus, since a different axiomatisation proposed in the literature is incompatible with the Milner-Turner technique. This is not the only contribution to the type theory of the π-calculus, since the encoding also outlines the different roles played by the two contra-variant constructors of $\mathbb{C}\pi$, namely input channel and negation, and shows how they interplay when considering them from a logical point of view. Finally, at term level the translation formalises the nice correspondence between functional pattern matching and π-calculus guarded sums on a same input channel.

C.1.4 Related work

The first work on subtyping for π was done by Pierce and Sangiorgi [PS96] and successively extended in several other works [Sew98, D$^+$00, YH99].

The work closest to ours, at least for the expressiveness of the types, is the already cited work of Hennessy and Riely [HR02]. As far as π-types are concerned, our work subsumes their system in the sense that it defines a richer subtyping relation; this can be checked by observing that their type $\mathsf{rw}\langle s, t\rangle$ corresponds to the intersection $ch^+(s) \wedge ch^-(t)$ of our formalism.

The works of Acciai and Boreale [AB05] and of Brown *et al.* [BLM05], define languages similar to ours, with XDuce-like pattern matching. However their type systems are less rich than ours and, most importantly, their subtyping relations are defined syntactically.

As for the technical issues of semantic subtyping, our starting point is the work developed by Frisch *et al.* for functional programming languages [FCB08, Fri04], that led to the design of $\mathbb{C}$Duce.

C.1.5 Structure of the chapter

This chapter is a fusion of the papers [CDV08, CDV06].

In Section C.2 we describe the types, their semantics, and subtyping relation whose decidability is shown in Section C.3. In Section C.4 we introduce $\mathbb{C}\pi$, a variant of π-calculus tailored on the previous types, and show examples of its usage. In Section C.5 we discuss possible extensions of $\mathbb{C}\pi$ and in particular we we present the local variant of the $\mathbb{C}\pi$-calculus. In Section C.6 we present the functional core of $\mathbb{C}$Duce. Section C.7 is devoted to explaining the main difficulties we encountered when encoding $\mathbb{C}$Duce types into $\mathbb{C}\pi$ types. Section C.8 contains the formalisation of the encoding of the language, while Section C.9 presents the correctness results. In Section C.10 we conclude by giving some insight on more general aspects of this work and trying to convey the intuition

of why we believe that the main contribution lie well beyond the technical result we present. In order to lighten the presentation, we postpone the proofs to Section C.11.

C.2 Types and subtyping

We shall present in detail a relatively simple system with just base types, channels, and boolean combinators. In Section C.5, we will then sketch how to add the product type constructor, recursive types, and functional types.

C.2.1 Types

In the simplest of our type systems, a type is inductively built by applying *type constructors*, namely base type constructors (e.g. integers, strings, etc...), the input or the output channel type constructor, or by applying a *boolean combinator*, i.e., union, intersection, and negation :

Types	t	::=	$b \mid ch^+(t) \mid ch^-(t)$	constructors
		$\mid$	$\mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t$	combinators

Combinators are self-explaining, with $\mathbf{0}$ being the empty type and $\mathbf{1}$ the type of all values. The "set difference" combinator $s\backslash t$ will be used as a shorthand for $s \wedge \neg t$. For what concerns type constructors, $ch^+(t)$ denotes the type of those channels that can be used to *input* only values of type t. Symmetrically $ch^-(t)$ denotes the type of those channels that can be used to *output* only values of type t. The read and write channel type $ch(t)$ is absent from our definition. We shall use it only as syntactic sugar for $ch^-(t) \wedge ch^+(t)$, that is the type of channels that can be used to read only *and* to write only values of type t. The set of all types (sometimes referred to as "type algebra") will be denoted by $\mathcal{T}$.

In our approach channels are physical boxes where one can insert and withdraw objects of a given type. Our intuition is that there is not such a thing as a read-only or write-only box : each box is associated with a type t and one can always write and read objects of that type into and from such a box. Thus the type of $ch^+(t)$ can be considered just a constraint telling that a variable of that type will be bound only to boxes from which one can read objects of type t. If we know that a message has type $ch^+(t)$, it *does not* mean that we cannot write into it, we simply do not have any information about what can be written in it : for instance this message could be a box that cannot contain any object. What the type tells us is simply that we had better avoid writing into it since, in the absence of further information, no writing will be safe. Similarly, if a message is of type $ch^-(t)$, then we know that it can only be a box in which writing an object of type t is safe, but we have no information about what could be read from that channel, since the message might be a box that can contain any object. Therefore we had better avoid reading from it, unless we are ready to accept anything. However, if we *are* ready to accept anything, then our type system guarantees that we can read on a channel with type $ch^-(t)$ because, as we will see later, we have $ch^-(t) \leq ch^+(\mathbf{1})$.

C.2.2 Semantics of types

Our leading intuition is that a type should denote the set of values of that type. That is :

$$[\![t]\!] = \{v \mid \vdash v : t\} \ .$$

The basic types (integers, strings) should denote subsets of a set of basic values $\mathbb{B}$. The boolean operators over types should be interpreted by using the boolean operators over sets. By following our intuition we shall have that the interpretation of the type $ch(t)$ has to denote the set of all boxes (i.e. channels) that can contain objects of type t :

$$[\![ch(t)]\!] = \left\{c \mid c \text{ is a box for objects in } [\![t]\!]\right\} \ . \tag{C.2}$$

Since every box is uniquely associated to a type, then the interpretations of channel types are pairwise disjoint. This already gives invariance of channel types : $[\![ch(t)]\!] \subseteq [\![ch(s)]\!]$ if and only if $[\![t]\!] = [\![s]\!]$.

Starting from the above interpretation of $ch(t)$, we can now provide a semantics for $ch^+(t)$ and $ch^-(t)$. As said, the former should denote the set of all boxes from which one can safely expect to get only objects of type t. Thus we require that $ch^+(t)$ denotes all boxes for objects of type t, but also all boxes for objects of type s, for any $s \leq t$. Indeed, by subsumption, objects of types s are also of type t. Dually, $ch^-(t)$ should denote the set of all boxes in which one can safely put objects of type t. Therefore it will denote all boxes that can contain objects of type s, for any $s \geq t$. Let us write c^t to denote a box for objects of type t. We have

$$[\![ch^+(t)]\!] = \left\{c^s \mid s \leq t\right\} \ , \qquad [\![ch^-(t)]\!] = \left\{c^s \mid s \geq t\right\} \ .$$

Given the above semantic interpretation, from the viewpoint of types all the boxes of one given type t are indistinguishable, because either they all belong to the interpretation of one type or they all do not. This implies that the subtyping relation is insensitive to the actual number of boxes of a given type. We can thus assume that for every equivalence class of types, there is only one such box, which may as well be identified with $[\![t]\!]$, so that the intended semantics of channel types would be

$$[\![ch^+(t)]\!] = \left\{[\![s]\!] \mid s \leq t\right\} \ , \ [\![ch^-(t)]\!] = \left\{[\![s]\!] \mid s \geq t\right\} \ . \tag{C.3}$$

We have that this semantics induces covariance of input types and contravariance of output types. Moreover, as anticipated, we have that $ch(t) = ch^-(t) \wedge ch^+(t)$ since the types on both sides of the equality have the same semantics—namely, the singleton $\{[\![t]\!]\}$—and therefore it is justified to consider $ch(t)$ as syntactic sugar for $ch^-(t) \wedge ch^+(t)$, rather than a type constructor.

According to the discussion above, in order to define the semantics of a channel type, we need to know the subtyping relation. And here we are again in the presence of a circle. We use the subtyping relation in order to build the interpretation that we need in order to define the subtyping relation. We devote the next section to solve this problem.

C.2.3 Building a model

The minimal requirement for an interpretation function is that boolean combinators should be interpreted in the corresponding set-theoretical operators, and that basic values and channels should have disjoint interpretations.

Definition C.2.1 (Pre-model)**.** Let $\mathcal{D}$,and $\mathbb{B}$ be sets such that $\mathbb{B} \subseteq \mathcal{D}$, and let $[\![\ ]\!]$ be a function from $\mathcal{T}$ to $\mathcal{P}(\mathcal{D})$. The pair $(\mathcal{D}, [\![\ ]\!])$ is said to be a *pre-model* if

- $[\![b]\!] \subseteq \mathbb{B}$, $[\![ch^+(t)]\!] \cap \mathbb{B} = \varnothing$, $[\![ch^-(t)]\!] \cap \mathbb{B} = \varnothing$;
- $[\![\mathbf{1}]\!] = \mathcal{D}$, $[\![\mathbf{0}]\!] = \varnothing$;
- $[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$;
- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$, $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$.

We use this interpretation to build another interpretation, according to the intended meaning of equations (C.3). The symbol + will denote disjoint union of sets.

Definition C.2.2 (Extensional interpretation)**.** Let $(\mathcal{D}, [\![\ ]\!])$ be a pre-model. Let $[\![\mathcal{T}]\!]$ denote the image of the function $[\![\ ]\!]$. The *extensional* interpretation of the types is the function $\mathbb{E}(\,) : \mathcal{T} \to \mathcal{P}(\mathbb{B} + [\![\mathcal{T}]\!])$, defined as follows :

- $\mathbb{E}(b) = [\![b]\!]$;
- $\mathbb{E}(\mathbf{1}) = \mathbb{B} + [\![\mathcal{T}]\!]$, $\mathbb{E}(\mathbf{0}) = \varnothing$;
- $\mathbb{E}(\neg t) = \mathbb{E}(\mathbf{1}) \setminus \mathbb{E}(t)$;
- $\mathbb{E}(t_1 \vee t_2) = \mathbb{E}(t_1) \cup \mathbb{E}(t_2)$, $\mathbb{E}(t_1 \wedge t_2) = \mathbb{E}(t_1) \cap \mathbb{E}(t_2)$;
- $\mathbb{E}(ch^+(t)) = \{[\![s]\!] \mid [\![s]\!] \subseteq [\![t]\!]\}$;
- $\mathbb{E}(ch^-(t)) = \{[\![s]\!] \mid [\![s]\!] \supseteq [\![t]\!]\}$.

A pre-model and its extensional interpretation induce, in principle, different preorders on types. We could use the extensional interpretation to build yet another interpretation, and so on. In order to close the circle, we shall consider a pre-model "acceptable" if it is a fixed point of this process, that is, if it induces the same containment relation as its extensional interpretation. This amounts to the following definition :

Definition C.2.3 (Model)**.** A pre-model $(\mathcal{D}, [\![\ ]\!])$ is a *model* if for every t_1, t_2, we have $[\![t_1]\!] \subseteq [\![t_2]\!]$ if and only if $\mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$.

The last (and quite hard) point is to show that there actually exists a model, that is, that the condition imposed by Definition C.2.3 can indeed be satisfied. Paradoxically the model itself is not important. The subtyping relation is essentially characterised by the definition of extensional interpretation $\mathcal{E}[\![\ ]\!]$. So what really matters is the proof that there exists at least one model. As the case of recursive types proves (see § C.5.2), the existence of such a model is far from being trivial, and naive syntactic solutions —such as a term model— cannot be used.

Theorem C.2.4. *There exists a model* $(\mathcal{D}, [\![\ ]\!])$.

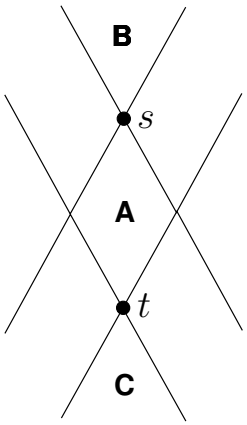


FIGURE C.1 – Channel types

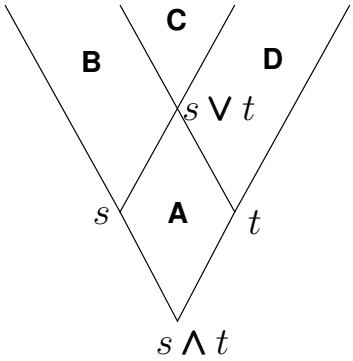


FIGURE C.2 – Some equations

Types are stratified according to the nesting of the channel constructor. The model $(\mathcal{D}, [\![\,]\!])$ is obtained as the limit of a chain of models $(\mathcal{D}_n, [\![\,]\!]_n)$, built exploiting this stratification. The long and technical proof can be found in Appendix C.11.2.

Finally, given a model for the types, we define

$$s \leq t \stackrel{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!] \,, \qquad\qquad s = t \stackrel{\text{def}}{\iff} [\![s]\!] = [\![t]\!] \,.$$

C.2.4 Examples of type (in)equalities and graphical representation

We list here some interesting equations and inequations between types that can be easily derived from the set-theoretic interpretation of types. A first simple example of equality and inequality is

$$ch(t) \leq ch\overline{(\mathbf{0})} = ch^+(\mathbf{1}) \tag{C.4}$$

which states that every channel c of whatever type $ch(t)$ can always be safely used in a process that does not write on c (since it has also type $ch\overline{(\mathbf{0})}$) and that does not care about what c returns (since it has type $ch^+(\mathbf{1})$).

Besides these fiddling relations, far more interesting relations can be deduced and, quite remarkably, in many cases this can be done graphically. Consider the definitions in (C.3) : they tell us that the interpretation of $ch^+(t)$ is the set of the interpretations of all types smaller than or equal to t. As such, it can be represented by the downward cone starting from t. Similarly, the upward cone starting from t represents $ch\overline{(t)}$. This illustrated in Figure C.1 where the upward cone B represents $ch\overline{(s)}$ and the downward cone C represents $ch^+(t)$. As the reader can easily verify, this representation immediatly gives covariance of input types and contravariance of output types.

If we now pass to Figure C.2 we see that $ch\overline{(s)}$ is the upward cone $\mathsf{B}{+}\mathsf{C}$ and $ch\overline{(t)}$ is the upward cone $\mathsf{C}{+}\mathsf{D}$. Their intersection is the cone C, that is the

upward cone starting from the least upper bound of s and t which yields the following equation

$$ch\overline{h}(s) \bigwedge ch\overline{h}(t) = ch\overline{h}(s \bigvee t) \ . \tag{C.5}$$

This states that if on a channel we can write values of type s and values of type t, this means that we can write on it values of type $s \bigvee t$. Dually, by turning Figure C.2 upside down it is easy to check the following equation :

$$ch^+(s) \bigwedge ch^+(t) = ch^+(s \bigwedge t) \tag{C.6}$$

which states that if a channel is such that we always read from it values of type s but also such that we always read from it values of type t, then what we read from it are actually values of type $s \bigwedge t$.

Similarly, note that the union of $ch\overline{h}(s)$ and $ch\overline{h}(t)$ is given by B+C+D and that this is strictly contained in the upward cone starting from $s \bigwedge t$, since the latter also contains the region A, whence the strictness of the following containment :

$$ch\overline{h}(s) \bigvee ch\overline{h}(t) \lneq ch\overline{h}(s \bigwedge t) \ . \tag{C.7}$$

Actually, the difference of the two types in the above inequality is the region A which represents $ch^+(s \bigvee t) \bigwedge ch\overline{h}(s \bigwedge t)$, from which we deduce

$$ch\overline{h}(s \bigwedge t) = ch\overline{h}(s) \bigvee ch\overline{h}(t) \bigvee (ch^+(s \bigvee t) \bigwedge ch\overline{h}(s \bigwedge t)) \ .$$

By turning Figure C.2 upside down again we can check the dual of equation (C.7) :

$$ch^+(s) \bigvee ch^+(t) \lneq ch^+(s \bigvee t) \tag{C.8}$$

As a final example consider the type $ch^+(s) \bigwedge ch\overline{h}(t)$, that is the type of a channel on which we can write values of type t and from which we expect to read values of type t. We have

$$ch^+(s) \bigwedge ch\overline{h}(t) = \mathbf{0} \tag{C.9}$$

if and only if $t \not\leq s$, i.e. we should expect to read at least what we can write. Once more this can be checked graphically on Figure C.1, but in order to show the role of our definitions, let us formally deduce this last equation. By definition, (C.9) holds if and only if $[\![ch^+(s) \bigwedge ch\overline{h}(t)]\!] = [\![\mathbf{0}]\!]$. By definition of model and the antisymmetry of $\subseteq$ this holds if and only if $\mathbb{E}(ch^+(s) \bigwedge ch\overline{h}(t)) = \mathbb{E}(\mathbf{0})$. By definition of $\mathbb{E}()$ this holds if and only if $\{[\![s]\!]' | [\![s]\!]' \subseteq [\![s]\!]\} \cap \{[\![t]\!]' | [\![t]\!] \subseteq [\![t']\!]\} = \varnothing$. By the reflexivity and transitivity of $\subseteq$ this holds if and only if $[\![t]\!] \not\subseteq [\![s]\!]$, that is, by definition of subtyping if and only if $t \not\leq s$.

C.3 Decidability of subtyping

For practical applications, it is essential that subtyping relations are decidable. The subtyping relation defined in Section C.2 is indeed decidable. The decision procedure is however a bit involved. As we show in details later in this section, we can always reduce the problem of deciding the subtyping between two types to deciding an inclusion of the following form :

$$ch^+(t_1) \bigwedge ch\overline{h}(t_2) \quad \leq \quad \bigvee_{h \in H} ch^+(t_3^h) \bigvee \bigvee_{k \in K} ch\overline{h}(t_4^k) \ . \tag{C.10}$$

While in some cases it is easy to decide the inclusion above (for instance, when $t_2 \not\leq t_1$ since then the left-hand side is empty), in general, this requires checking

whether a type is *atomic*, that is whether its only proper subtype is the empty type (for sake of simplicity the reader can think of the atomic types as the singletons of the type system [1]). To have an idea of why we have to push the check at the level of atomic types let us once more resort to the graphical representation. Consider the equation (C.10) above with only two types s and t with $t \lneq s$ (note the strictness of inclusion, which implies that $s\setminus t$ is not empty), and try to check whether :

$$ch^+(s) \wedge \overline{ch}(t) \leq \overline{ch}(s) \vee ch^+(t) \ .$$

The situation is represented in Figure C.1 where the region A represents the left-hand side of the inequality, while the region $\mathsf{B}+\mathsf{C}$ is the right hand side. So to check the subtyping above we have to check whether A is contained in $\mathsf{B}+\mathsf{C}$. At first sight these two regions look completely disjoint, but observe that they have at least two points in common, marked in bold in the figure (they are respectively the types $ch(s)$ and $ch(t)$). Now, the containment holds if the region A does not contain any other type besides these two. This holds true if and only if there is no other type between s and t, that is if and only if $s\setminus t$ is an atomic type.

Let us now present the technical details of the decision procedure (proofs can be found in Section C.11). First of all we need to define the notions of finite and atomic types.

Definition C.3.1 (Atomic and finite types)**.** An *atom* is a minimal non-empty type. A type is *finite* if it is equivalent to a finite union of atoms.

We start the description of the decision procedure by noting that deciding subtyping is equivalent to deciding the emptiness of a type.

$$s \leq t \Leftrightarrow s \wedge \neg t = \mathbf{0} \tag{C.11}$$

which can be derived as follows :

$$s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \ \Leftrightarrow \ [\![s]\!] \cap \complement[\![t]\!] = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = [\![\mathbf{0}]\!] \ \Leftrightarrow \ s \wedge \neg t = \mathbf{0} \ .$$

Thanks to the semantic interpretation we can directly apply set-theoretic equivalences to types (in the rest of the chapter we will do it without explicitly passing via the interpretation function). We then deduce that every type can be (effectively) represented in disjunctive normal form, i.e. as the union of intersections of literals, where a literal is a base type or a channel type, possibly negated. Since a union is empty only if all its addenda are empty, then in order to decide emptiness of a type —and thus in virtue of (C.11) to decide subtyping— it suffices to be able to decide whether an intersection of literals is empty. Since base types and channel types are interpreted in disjoint sets, intersections that involve literals of both kinds are either trivial, or can be simplified to intersections involving literals of only one kind. The problem is therefore reduced to decide whether

1. Nevertheless, notice that according to their definition, atomic types may be neither singletons nor finite. For instance, $ch(\mathbf{0})$ is atomic, but in the model defined by equation (C.2)—more precisely, in the model of values of Theorem C.4.5—it is the set of all the synchronisation channels ; these are just token identifiers on a countable alphabet, thus the type is countable as well.

$$(\bigwedge_{i\in P} b_i) \wedge (\bigwedge_{j\in N} \neg b_j) \quad \text{and} \quad (\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \wedge (\bigwedge_{j\in N} \neg ch^{\nu_j}(t_j))$$

are equivalent to $\mathbf{0}$ (where ν stands for either "+" or "−" and we grouped literals according to whether they are negated or not). The decision of emptiness of the left-hand side depends on the basic types that are used. For what concerns the right-hand side, we decompose this problem into simpler subproblems. More precisely, we reduce this problem to the problem of deciding subtyping between boolean combinations of the t_i's and t_j's. This problem is simpler, in the sense that it involves a strictly smaller nesting of channel types.

Using set-theoretic manipulations—in the case in point De Morgan's laws—the problem of deciding

$$(\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \wedge (\bigwedge_{j\in N} \neg ch^{\nu_j}(t_j)) = \mathbf{0}$$

can be shown to be equivalent to

$$(\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \leq (\bigvee_{j\in N} ch^{\nu_j}(t_j)) \ . \tag{C.12}$$

Because of equations (C.5) and (C.6), we can push the intersection on the left-hand side inside the constructors and reduce (C.12) to the equation (C.10) we met in the previous section, and that we recall below :

$$ch^+(t_1) \wedge ch\overline{(t_2)} \leq \bigvee_{h\in H} ch^+(t_3^h) \vee \bigvee_{k\in K} ch\overline{(t_4^k)} \tag{C.10}$$

where we grouped covariant and contravariant types together. In this way we simplified the left-hand side. Similarly we can get rid of redundant addenda on the right-hand side of (C.10) by eliminating :

1. all the covariant channel types on a t_3^h for which there exists a covariant addendum on a smaller or equal $t_3^{h'}$ (since the former channel type is contained in the latter) ;
2. all contravariant channel types on a t_4^k for which there exists a contravariant addendum on a larger or equal $t_4^{k'}$ (for the same reason as the above) ;
3. all the covariant channels on a t_3^h that is not larger than or equal to t_2 (since then $ch\overline{(t_2)} \cap ch^+(t_3^h) = \mathbf{0}$, so it does not change the inequation) ;
4. all contravariant channels on a t_4^k that is not smaller than or equal to t_1 (since then $ch^+(t_1) \cap ch\overline{(t_4^k)} = \mathbf{0}$).

Then the key property for decomposing the problem (C.10) into simpler subproblems is given by the following theorem :

Theorem C.3.2. *Suppose $t_1, t_2, t_3^h, t_4^k \in \mathcal{T}$, $k \in K$, $h \in H$. Suppose moreover that the following conditions hold :*

c1. for all distinct $h, h' \in H$, $t_3^h \not\leq t_3^{h'}$;

c2. for all distinct $k, k' \in K$, $t_4^k \not\leq t_4^{k'}$;

c3. for all $h \in H$, $t_2 \leq t_3^h$;

c4. for all $k \in K$, $t_4^k \leq t_1$.

Then

$$ch^+(t_1) \wedge ch\overline{(t_2)} \leq \bigvee_{h\in H} ch^+(t_3^h) \vee \bigvee_{k\in K} ch\overline{(t_4^k)} \tag{C.10}$$

if and only if one of the following conditions holds

LE. $t_2 \not\leq t_1$ *or*
R1. $\exists h \in H$ *such that* $t_1 \leq t_3^h$ *or*
R2. $\exists k \in K$ *such that* $t_4^k \leq t_2$ *or*
CA. for every choice of atoms $a_h \leq t_1 \backslash t_3^h$*, with* $h \in H$*, there exists* $k \in K$ *such that* $t_4^k \leq t_2 \vee \bigvee_{h \in H} a_h$.

The four hypotheses c1–c4 simply state that the right-hand side of the inequation was simplified according to the rules (1–4) described right before the statement of the theorem. The first condition (LE) says that $ch^+(t_1) \wedge c\bar{h}(t_2)$ is empty. The second condition (R1) and the third condition (R2) respectively make sure that one of the $ch^+(t_3^h)$ and, respectively, one of the $c\bar{h}(t_4^h)$ contains $ch^+(t_1) \wedge c\bar{h}(t_2)$. Finally the fourth and more involved [2] condition (CA) says that, every time we add to t_2 atoms of t_1 so that we are no longer below any t_3^h then we must end up above some of the t_4^k.

We have already shown at the beginning of this Section an example of the sensitivity of the subtyping relation to atoms. To obtain another, more concrete example of this fact, suppose there are three atoms $\mathtt{err}_1, \mathtt{err}_2, \mathtt{exc}$ and consider the case where $t_2 = \mathtt{int}$, $t_1 = t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2 \vee \mathtt{exc}$, $t_3 = t_2 \vee \mathtt{exc}$, $t_4 = t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2$. It is easy to see that $ch^+(t_1) \wedge c\bar{h}(t_2) \not\leq ch^+(t_3) \vee c\bar{h}(t_4)$ since, for example, the type $ch(t_2 \vee \mathtt{err}_1)$ is a subtype of the left-hand side, but not of the right-hand side. However if $\mathtt{err}_1 = \mathtt{err}_2$, the subtyping relation holds, because of condition (CA). Indeed in that case the indexing set H of Theorem C.3.2 is a singleton. The only atom in $t_1 \backslash t_3$ is $\mathtt{err}_1$, and it is true that $t_4 \leq t_2 \vee \mathtt{err}_1$.

As announced, Theorem C.3.2 decomposes the subtyping problem of (C.10) into a finite set of subtyping problems on simpler types (we must simplify the right hand side of inequation (C.10) by verifying the inequalities of conditions c1–c4, and possibly perform the $|H| + |K| + 1$ checks for LE, R1 and R1) *and* into the verification of condition (CA).

The condition (CA) involves a universal quantification on possibly infinite sets $t_1 \backslash t_3^h$, and therefore it is not possible to use it for a decision algorithm as it is. This problem can be avoided thanks to the following proposition

Proposition C.3.3. *If we replace condition (CA) with*
CA. Let* $H_f \subseteq H$ *be the set of those indices* h *for which* $t_1 \backslash t_3^h$ *is finite. For every choice of atoms* $a^h \leq t_1 \backslash t_3^h$*, with* $h \in H_f$*, there exists* $k \in K$ *such that* $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$.
then Theorem C.3.2 still holds.

Therefore it suffices to check the condition just for the $t_1 \backslash t_3^h$ that are finite. This can be done effectively provided that we are able to :

1. decide whether a type is finite and
2. if it is the case, list all its atoms.

We will assume that this is possible for base types and prove that this implies that it is possible for all types.

2. The original condition (CA) as it can be found in [CDV05] was even more involved. We renew our gratitute to the anonymous referee who suggested a major simplification.

Lemma C.3.4. *There is an algorithm that decides whether a type t is finite and if it is the case, outputs all its atoms.*

Theorem C.3.5. *The subtyping relation is decidable.*

We do not discuss here the complexity of the decision algorithm, nor the possibility of finding more efficient ways of doing it. We leave it for future work.

C.4 The $\mathbb{C}\pi$ calculus

We shall present a variant of the π-calculus, that exploits the type system of Section E.3.2. We will present its syntax, semantics, and typing rules, and prove the decidability of the typing relation.

C.4.1 Patterns

As we explained in the introduction, if we want to fully exploit the expressiveness of the type system, we must be able to check the type of the messages read on a channel. The simplest solution would be to add an explicit type-case process (e.g. $[M : t]P$ which reduces to P or $\mathbf{0}$ according whether M is of type t or not). Here, instead, we choose a more general approach, by endowing input actions with $\mathbb{C}$Duce patterns. Pattern matching includes dynamic type checks as a special case, and fits nicely in the semantic subtyping framework.

Definition C.4.1 (Patterns)**.** Given a type algebra $\mathcal{T}$, and a set of variables $\mathbb{V}$, a *pattern* p on $(\mathbb{V}, \mathcal{T})$ is a term generated by the following grammar

Patterns	p	$::=$	x	*capture,* $x \in \mathbb{V}$
		$\mid$	t	*type constraint,* $t \in \mathcal{T}$
		$\mid$	$p \wedge p$	*conjunction*
		$\mid$	$p \mid p$	*alternative*

such that for every subterm $p_1 \wedge p_2$ of p we have $(\langle p \rangle_1) \cap (\langle p \rangle_2) = \varnothing$, and for every subterm $p_1 | p_2$ of p we have $(\langle p \rangle_1) = (\langle p \rangle_2)$ (where $(\langle p \rangle)$ denotes the set of variables of $\mathbb{V}$ occurring in p).

Patterns are rather basic : they can test if a value is of a given type, capture it, and combine these tests via conjunctions and disjunctions. So for instance $x \wedge t$ is the pattern that captures a value in x if it is of type t. As a matter of fact, the patterns above lack the main capability peculiar of general patterns that is to deconstruct values. The reason is that here we consider a minimal type system in which the only type constructors are for channel types, and their values are not "constructed" from simpler values (e.g. pairs of values for product constructor) but are constants. So here patterns act more as a placeholder and they are interesting in view of the extension of our language with recursive types (Section C.5.2) product types (Section C.5.1) or other type constructors.

Following [FCB08] we define the semantics of patterns directly on models. A pattern is matched against an element of the domain $\mathcal{D}$ of a model of the types and the matching returns either a substitution for the free variables of the pattern, or a failure, denoted by Ω :

Definition C.4.2. Given a model $[\![\,]\!] : \mathcal{T} \to \mathcal{D}$, an element $d \in \mathcal{D}$, and a pattern p, the matching of d with p, noted by d/p, is the element of $\mathcal{D}^{(\langle p \rangle)} \cup \{\Omega\}$ defined as follows :

$$\begin{array}{lcll} d/t & = & \{\} & \text{if } d \in [\![t]\!] \\ d/t & = & \Omega & \text{if } d \in [\![\neg t]\!] \\ d/x & = & \{x \mapsto d\} & \end{array} \qquad \begin{array}{lcll} d/p_1 \wedge p_2 & = & d/p_1 \otimes d/p_2 & \\ d/p_1|p_2 & = & d/p_1 & \text{if } d/p_1 \neq \Omega \\ d/p_1|p_2 & = & d/p_2 & \text{if } d/p_1 = \Omega \end{array}$$

where $\gamma_1 \otimes \gamma_2$ is Ω when $\gamma_1 = \Omega$ or $\gamma_2 = \Omega$ and the union of the two otherwise.

A quite useful property of the pattern matching above is that the set of all elements for which a pattern p does not fail is the denotation of a type. Since this type is unique, we denote it by $⌊p⌋$. In other terms, for every (well-formed) pattern p, there exists a unique type $⌊p⌋$ such that $[\![⌊p⌋]\!] = \{d \in Dom \mid d/p \neq \Omega\}$. Not only, but this type can be calculated. Similarly, consider a pattern p and a type $t \leq ⌊p⌋$, then there is also an algorithm that calculates the type environment t/p that associates to each variable x of p the *exact* set of values that x can capture when p is matched against values of type t. Formally

Theorem C.4.3. *There is an algorithm mapping every pattern p to a type $⌊p⌋$ such that $[\![⌊p⌋]\!] = \{d \in \mathcal{D} \mid d/p \neq \Omega\}$.*

Theorem C.4.4. *There is an algorithm mapping every pair (t, p), where p is a pattern and t a type such that $t \leq ⌊p⌋$, to a type environment $(t/p) \in \mathcal{T}^{(\langle p \rangle)}$ such that $[\![(t/p)(x)]\!] = \{(d/p)(x) \mid d \in [\![t]\!]\}$.*

For such basic patterns the proofs of the properties above are really straightforward. What is remarkable is that these properties hold for polyadic $\mathbb{C}\pi$ with recursive types, as well (Section C.5.2).

C.4.2 The language

The syntax of our calculus is very similar to that of the asynchronous π-calculus a variant of the π-calculus for which message emission is non-blocking. The latter is generally considered as the calculus representing the essence of name passing with no redundant operation. The variant we consider is very similar to the original calculus, but we permit patterned input prefix and guarded choice between different patterns on the same input channel.

Channels	α	::=	x	variables
		|	c^t	constant
Messages	M	::=	n	constant
		|	α	channel
Processes	P	::=	$\overline{\alpha}M$	output
		|	$\sum_{i \in I} \alpha(p_i).P_i$	patterned input
		|	$P_1 \| P_2$	parallel
		|	$(\nu c^t)P$	restriction
		|	$!P$	replication

where I is a possibly empty finite set of indexes, t ranges over the types defined

in Section C.2.1 and p_i are patterns as given in Definition C.4.1. As customary we use the convention that the empty sum corresponds to the inert process, denoted by $\mathsf{0}$.

We want to comment on the presence of the simplified form of summation we have adopted : guarded sum of inputs on a single channel with possibly different patterns. Choice operators are very useful for specifying nondeterministic behaviours, but give rise to problems when considering implementation issues. Two main kinds of choice have to be considered : *external choice* that leaves the decision about the continuation to the external environment (usually having it dependent on the channel used by the environment to communicate) and *internal choice* that is performed by the process regardless of external interactions. Thanks to patterns we can offer an externally controllable choice, where the type of the received message, not the used channel, determines the continuation. Internal choice can also be modelled by specifying processes that perform input on the same channel according to the same pattern.

The other important difference with standard asynchronous π-calculus is that we distinguish between channel variables and channel constants and that the latter are decorated by the type of messages they communicate. This corresponds to our intuition that every box is intimately associated to the type of the objects it can contain. In what follows we will call channel constants also "typed channels", "boxes", or "channel values" to distinguish them from channel variables.

The *values* of the language are the closed messages, that is to say the typed channels and the constants : $v ::= n \mid c^t$.

We use $\mathcal{V}$ to denote the set of all values. Every value is associated to a type : every constant n is associated to an atomic basic type b_n (we also assume that every atomic basic type b_n has its corresponding basic value n), while every channel value is associated with the channel type that transport messages of the type indicated in the index. So all the values can be typed by the rules (const), (chan), and (subs) of Figure C.3 (actually with an empty Γ) where in the (subs) subsumption rule the $\leq$ is the subtyping relation induced by the model built to prove Theorem C.2.4 (see Section C.11.2).

C.4.3 Semantics

Let $\mathtt{M}(=)(\mathcal{D}_{\mathtt{M}}(,)[\![\ ]\!]_{\mathtt{M}}())$ be any model (that is, it satisfies Definition C.2.3). $\mathtt{M}()$ induces a subtyping relation $\leq_{\mathtt{M}}()$ defined as $s \leq_{\mathtt{M}} (t) \stackrel{\text{def}}{\Leftrightarrow} [\![s]\!]_{\mathtt{M}}(\subseteq)[\![t]\!]_{\mathtt{M}}()$. Consider the typing rules for Message in Figure C.3, use for the subsumption rule (subs) the $\leq_{\mathtt{M}}()$ relation, and denote by $\Gamma \vdash_{\mathtt{M}} (M) : t$ the corresponding typing relation.

Now consider this new interpretation function $[\![\ ]\!]_{\mathcal{V}} : \mathcal{T} \to \mathcal{P}(\mathcal{V})$ defined as $[\![t]\!]_{\mathcal{V}} = \{v \mid \Gamma \vdash_{\mathtt{M}} (v) : t\}$. It turns out that this interpretation, whatever the model is, satisfies the model conditions of Section C.2.3 and furthermore it generates the same subtyping relation as $\leq_{\mathtt{M}}()$. The circle we mentioned in the Introduction is now closed.

Messages

$$\frac{}{\Gamma \vdash n : b_n}\ (\text{const}) \qquad \frac{}{\Gamma \vdash c^t : ch(t)}\ (\text{chan})$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var}) \qquad \frac{\Gamma \vdash M : s \leq t}{\Gamma \vdash M : t}\ (\text{subs})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : c\bar{h}(t)}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

$$\frac{\Gamma \vdash \alpha : ch^+(t) \quad \Gamma, t/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I} \alpha(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

FIGURE C.3 – Typing rules

Theorem C.4.5 (Model of values)**.** *Let* $(\mathcal{D}, [\![\ ]\!])$ *be a model and* $\leq$ *and* $\Gamma \vdash M : t$ *be, respectively, the subtyping and typing relations it induces. Let* $[\![t]\!]_{\mathcal{V}} = \{v \mid \Gamma \vdash v : t\}$*. Then* $(\mathcal{V}, [\![\ ]\!]_{\mathcal{V}})$ *is a model and* $s \leq t \iff [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}$*.*

Since values are elements of a model of the types, Definition C.4.2 applies for d being a value. We can thus use it to define the reduction semantics of our calculus :

$$\overline{c^t} v \ \| \ \sum_{i\in I} c^t(p_i).P_i \quad \longrightarrow \quad P_j[v/p_j]$$

where $P[\sigma]$ denotes the application of substitution σ to process P. The asynchronous output of a *value* on the box c^t synchronises with an input on the same box only if at least one of the patterns guarding the sum matches the communicated value. If more than one pattern matches, then one of them is non-deterministically chosen and the corresponding process executed, but before its execution the pattern variables are replaced by the captured values. More refined matching policies (best match, first match, ...) can be easily encoded by a proper use of type combinators in patterns. As usual the notion of reduction must be completed with reductions in evaluation contexts and up to structural congruence, whose definitions are summarised in Figure C.4.

This operational semantics is the same as that of π-calculus but the actual process behavior has been refined in two points : (i) communication is subjected to pattern matching and (ii) communication can happen only along values (boxes).

The use of pattern matching is what makes it necessary to distinguish between typed channels and variables : matching is defined only for the formers as they are values, while a matching on variables must be delayed until they will be bound to a value.

Since we distinguish between variables and typed channels, it is reasonable to require that communication takes place only if we have a physical channel that can be used as a support for it ; thus, we forbid synchronisation if the channel

$$\mathrm{R}[\,] ::= [\,] \quad | \quad \mathrm{R}[\,]\|P \quad | \quad P\|\mathrm{R}[\,] \quad | \quad (\nu c^t)\mathrm{R}[\,]$$

$$P \longrightarrow Q \;\Rightarrow\; \mathrm{R}[P] \longrightarrow \mathrm{R}[Q] \qquad P' \equiv P \longrightarrow Q \;\Rightarrow\; P' \longrightarrow Q$$

$$P\|\mathbf{0} \equiv P \qquad P\|Q \equiv Q\|P \qquad P\|(Q\|R) \equiv (P\|Q)\|R$$

$$(\nu c^t)\mathbf{0} \equiv \mathbf{0} \qquad (\nu c^t)P \equiv (\nu d^t)P\{c^t \rightsquigarrow d^t\} \qquad !P \equiv !P\|P$$

$$(\nu c_1^{t_1})(\nu c_2^{t_2})P \equiv (\nu c_2^{t_2})(\nu c_1^{t_1})P \qquad \text{for } c_1 \neq c_2$$

$$(\nu c^t)(P\|Q) \equiv P\|(\nu c^t)Q \qquad \text{for } c^t \notin \mathsf{fn}(P)$$

where $P\{c^t \rightsquigarrow d^t\}$ is obtained from P by renaming all free occurrences of the box c^t into d^t, and assumes d^t is fresh.

FIGURE C.4 – Context and congruence closure

is still a variable. However there is a more technical reason to require this. Consider an environment $\Gamma = x : \mathbf{0}$. By subsumption we have $\Gamma \vdash x : ch(\texttt{int})$ and $\Gamma \vdash x : ch\overline{(\texttt{string})}$. Then, according to the typing rules of our system (see later on) the process $\overline{x}\,\texttt{ciao} \parallel x(y).\overline{x}(y \div y)$ is well typed, in the environment Γ, but it would give rise to a run time error by attempting to divide the string `ciao` by itself :

$$\overline{x}\,\texttt{ciao} \parallel x(y).\overline{x}(y \div y) \quad \longrightarrow \quad \overline{x}(\texttt{ciao} \div \texttt{ciao})$$

This reduction cannot happen in our calculus, because we can never instantiate a variable of type $\mathbf{0}$ (from a logical viewpoint, this corresponds to the classical *ex falso quodlibet* deduction rule).

C.4.4 Typing

In Figure C.3, we summarise typing rules that guarantee that, in well typed processes, channels communicate only values that correspond to their type.

The rules for messages do not deserve any particular comment. As customary, the system deduces only good-formation of processes without assigning them any type. The rules for replication and parallel composition are standard. The rule for restriction is slightly different since we do not need to store in the type environment the type of the channel [3]. In the rule for output we check that the message is compatible with the type of the channel.

The rule for input is the most involved one. The premises of the rule first infer the type t of the message that can be transmitted over the channel α, then for each summand i they use this type to calculate the type environment of the pattern variables (the environment (t/p_i) of Theorem C.4.4) and check whether under this environment the summand process P_i is typeable. This is all that is needed to have a sound type system. However the input construct is like a typecase/matching expression, so it seems reasonable to perform a check that

3. Strictly speaking, we do not restrict variables but values, so it would be formally wrong to store it in Γ. For the same reason, α-conversion is handled as a structural equivalence rule.

(*i*) patterns are exhaustive and (*ii*) there is no useless case[4]. The first check is performed by the side condition of the (input) rule : $t \leq \bigvee_{i\in I} \lbag p_i \rbag$ checks whether pattern matching is exhaustive, that is if for whatever value (of type t) sent on α there exists at least one pattern p_i that will accept it (the cases cover all possibilities). For the second condition one could naively think to add a second side condition such as $\lbag p_i \rbag \wedge t \neq \mathbf{0}$ for all $i \in I$ (we did this naivety in [CDV05]), which should check that the pattern matching is not redundant, by verifying that there does not exists a pattern p_i that will fail with every value of type t (no case is useless). However such a check is meaningful only if t is the *best* possible type we can deduce for the messages arriving on α. In a system with subsumption this condition can be always satisfied by considering a larger t (e.g., $t = \bigvee_{i\in I} \lbag p_i \rbag$), thus, without ensuring that all cases of the pattern matching are useful. Therefore we postpone the verification of this property till the definition of the typing algorithm (Section C.4.5) when this "best" type will be available.

As usual the basic result is the subject reduction, preceded by a substitution lemma. The proof of the theorem relies on the semantics of channel types as set of boxes, and can be found in Section C.11.5

Lemma C.4.6 (Substitution)**.**
- *If* $\Gamma, t/p \vdash M' : t'$ *and* $\Gamma \vdash v : t$*, then* $\Gamma \vdash M'[v/p] : t'$*.*
- *If* $\Gamma, t/p \vdash P$ *and* $\Gamma \vdash v : t$*, then* $\Gamma \vdash P[v/p]$*.*

Lemma C.4.7 (Congruence)**.** *If* $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$*.*

Theorem C.4.8 (Subject reduction)**.** *If* $\Gamma \vdash P$ *and* $P \to P'$*, then* $\Gamma \vdash P'$*.*

C.4.5 Typing algorithm

The decidability of the subtyping relation does not directly imply decidability of the typing relation (only semi-decidability is straightforward). The type algorithm is obtained from the typing rules in a standard way, namely by deleting the subsumption rule and embedding the checking of the subtyping relation in the elimination rules, in our case the (output) rule. As it is often the case, the typing algorithm also requires to compute a least upper bound of some given form. In particular, the algorithmic version of the (input) rules requires us to compute the least type of the form $ch^+(s)$ which is above a given type t, and it is not so evident that such a type exists (observe that our type algebra is *not* a complete lattice). Nevertheless, it turns out that such a type does exist (which gives us the minimum typing property) and furthermore it can be effectively computed.

Lemma C.4.9 (Upper bound channel)**.** *For every type* $s \leq ch^+(\mathbf{1})$ *there exists a least type* t *such that* $ch^+(t)$ *is an upper bound of* s*. We denote such type by* $\mathcal{C}(s)$*.*

4. In functional programming these checks are necessary for soundness since an expression non-complying to them may yield a type-error. In process algebræ non-compliance would just block synchronisation.

Messages

$$\frac{}{\Gamma \vdash n : b_n}\ (\text{const}) \qquad \frac{}{\Gamma \vdash c^t : ch(t)}\ (\text{chan}) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : s \quad s \leq ch\overline{(t)}}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

$$\mathcal{C}(s) \leq \bigvee_{i \in I} \wr p_i \int \frac{\Gamma \vdash \alpha : s \quad \Gamma, \mathcal{C}(s)/p_i \vdash P_i}{\Gamma \vdash \sum_{i \in I} \alpha(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

FIGURE C.5 – Algorithmic rules

The algorithmic rules are then defined as in Figure C.5. Soundness and completeness of these rules with respect to those in Figure C.3 are completely straightforward : soundness is obtained by a trivial application of the subsumption rule, while completeness can be easily deduced thanks to the fact that no type is inferred for processes (only good formation is checked), by using the fact that the type $\mathcal{C}(s)$ in the algorithmic (input) rule is always smaller than or equal to the type used by the corresponding rule in Figure C.3. Lemma C.4.9 and the decidability of $(\mathcal{C}(s)/p)$ (given by Theorem C.4.4) immediatly yield the following result.

Theorem C.4.10. *The typing relation is decidable.*

Finally, recall that in Section C.4.4 we hinted that we cannot statically check that all the branches of a pattern match are useful until we do not deduce the minimum type of the message that a channel can transport. Note that the algorithmic rules deduce for a channel its minimum type, and if this minimum type is, say, s, then by definition $\mathcal{C}(s)$ is the minimum type of the messages that the channel trasports. Therefore in order to check the usefulness of every branch it suffices to add to both the (input) rules in Figure C.3 and C.5 the side condition $\forall i \in I, \wr p_i \int \wedge \mathcal{C}(s) \neq \mathbf{0}$, and all the previous results carry along.

C.4.6 An example

We present here an example of a $\mathbb{C}\pi$ process. Consider the following situation. A web server is waiting on a channel α. The client wants the server to perform some computation on values it will send to the server. The server is able to perform two different kinds of computation, on values of type t_1 (say arithmetic operations), or on values of type t_2 (say list sorting). At the beginning of each session, the client can decide which operations it wants the server to perform, by sending a channel to the server, along which the communication can happen. The server checks the type of the channel, and provides the corresponding service.

$$P = \alpha(x : ch^+(t_1)).!x(y).P_1 + \alpha(x : ch^+(t_2)).!x(y).P_2$$

where we used the ℂDuce convention for patterns according to which $x : t$ is syntactic sugar for $x\mathsf{\wedge}t$ In the above process the channel α has type $ch^+(ch^+(t_1)\mathsf{\vee}\ ch^+(t_2))$. Note that, as explained in Section C.2.4 (equation (C.8)), $ch^+(t_1)\mathsf{\vee}\ ch^+(t_2) \neq ch^+(t_1 \mathsf{\vee} t_2)$. This means that the channel the server received on α will communicate *either* always values of type t_1 *or* always values of type t_2, and not interleaved sequences of the two, as $ch^+(t_1 \mathsf{\vee} t_2)$ would do.

As we discussed in the Introduction, this distinction is not present in analogous versions of process calculi where the axiom $ch^+(t_1)\mathsf{\vee}\, ch^+(t_2) = ch^+(t_1 \mathsf{\vee} t_2)$ is present. If such an axion were added to our theory, then we would program P defensively, as if α had the (morally larger) type $ch^+(ch^+(t_1 \mathsf{\vee}\, t_2))$

$$P' = \alpha(x).!(x(y : t_1).P_1 + x(y : t_2).P_2)$$

which is a less efficient server, since it performs pattern matching every time it receives a value.

C.5 Extensions and variations

C.5.1 Polyadic version

The first extension we propose consists in adding product to our type constructors. This is pretty straightforward. It requires adding $t ::= t \mathsf{\times} t$ to the productions of types, $M ::= (M, M)$ to the productions of messages, and $p ::= \mathbf{(}p_1\mathbf{,}p_2\mathbf{)}$ to the productions of patterns with the condition that for every subterm $\mathbf{(}p_1\mathbf{,}p_2\mathbf{)}$ of a pattern we have $(\langle p\rangle_1) \cap (\langle p\rangle_2) = \varnothing$.

The extensional interpretation becomes $\mathbb{E}(\,) : \mathcal{T} \to \mathcal{P}(\mathbb{B} + \mathcal{D}^2 + [\![\mathcal{T}]\!])$ and requires $\mathbb{E}(t_1 \mathsf{\times}\, t_2) = [\![t_1]\!] \times [\![t_2]\!]$. This completely characterises the subtyping relation. A semantic model can be built, in analogy with Section C.2.2. The subtyping relation is still decidable, as well as the typing relation.

The extensions described above suffice to obtain the polyadic calculus. In particular projections can be encoded by pattern matching. By using product types, together with the partially recursive types we show next, we can also encode more structured data, like lists or XML documents.

C.5.2 Partially recursive types

The types introduced so far can be represented as finite labelled trees. Recursive types are obtained without changing the syntax, by allowing trees to be infinite. As in the type system of ℂDuce we require such trees to be regular (so as they are finitely representable) and with the property that every infinite branch contains infinitely many nodes labelled by the product constructor (so as to avoid meaningless recursive definitions such as $t = t \mathsf{\wedge}\, t$).

Moreover we require that every branch can contain only finitely many nodes labelled with channel constructor. This amounts to require that the number of nested channel constructors is always bound. Or equivalently, if we were to define

recursive types with equations, this amounts to forbid the recursive variable being defined to be used inside a channel constructor (such as $x = ch(x) \vee \mathtt{int}$).

The reason for this is that, without this restriction, it is not possible to find a model. To see why, observe that we could have a recursive type t such that

$$t = b \vee (ch(t) \wedge ch(b))$$

for some nonempty base type b. If we have a model, either $t = b$ or $t \neq b$. Suppose $t = b$, then $ch(t) \wedge ch(b) = ch(b)$ and $b = t = b \vee ch(b)$. The latter implies $ch(b) \leq b$ which is not true when b is a base type. Therefore it must be $t \neq b$. According to our semantics this implies $ch(t) \wedge ch(b) = \mathbf{0}$, because they are two distinct atoms. Thus $t = b \vee \mathbf{0} = b$, contradiction.

Types are therefore stratified according to how many levels of nesting of the channel constructor there are and this stratification allows us to construct the model using the same ideas as presented in Section C.2. There are two main usages for arbitrary nested recursion of channels : one is to type "self application", that is a channel that can carry itself ; the other is for the definition of typed encodings. In our type system, we can already type self application by using, for instance, the type $ch(\mathbf{1})$: a channel that can carry everything, can clearly carry itself. Alternatively we can recover fully recursive types if we restrict to a *local* version of $\mathbb{C}\pi$ (see Section C.5.4 below) which is also enough for encoding functional languages.

Furthermore, note that recursion is still allowed with other type constructors, and a recursive type can appear inside a channel constructor provided that the number of occurrences of channel constructors is finite. For instance we are allowed to define the type $ch(IBlist)$, where *IBlist* is the type of heterogeneous lists of booleans and integers, defined as

$$IBlist = ((\mathtt{int} \vee \mathtt{bool}) \times IBlist) \vee ch(\mathbf{0})$$

(we use $ch(\mathbf{0})$ as the type of the empty list). Formally we have :

Definition C.5.1 (Types)**.** A type t is a possibly infinite regular tree generated by the following productions

$$\textit{Types} \qquad t \ ::= \ b \mid ch^{+}(t) \mid ch^{-}(t) \mid t \times t \mid \mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t$$

and such that on every infinite branch it has infinitely many occurrences of the product constructor and finitely many occurrences of the channel constructors.

With such recursive types it becomes interesting to use recursive patterns. If we relax the condition defined in Section C.5.1 for pair patterns and introduce a "constant pattern" as a case base for recursive pattern, then we can express the powerful patterns of CDuce.

Definition C.5.2 (Patterns)**.** A pattern p is a possibly infinite regular tree generated by the following productions

$$\textit{Patterns} \qquad p \ ::= \ x \mid t \mid (p,p) \mid (x := n) \mid p \wedge p \mid p|p$$

where x denotes a variable, t a type, and n a basic value. Additionally we require that on every infinite branch of p there are infinitely many occurrences of the pair pattern, that for every subterm $p_1 \wedge p_2$ of p $(\langle p \rangle_1) \cap (\langle p \rangle_2) = \varnothing$, and that for

every subterm $p_1|p_2$ of p $(\langle p\rangle_1) = (\langle p\rangle_2)$. Their semantics is defined as follows

$$\begin{array}{rcll} d/t & = & \{\} & \text{if } d \in [\![t]\!] \\ d/t & = & \Omega & \text{if } d \in [\![\neg t]\!] \\ d/p_1|p_2 & = & d/p_1 & \text{if } d/p_1 \neq \Omega \\ d/p_1|p_2 & = & d/p_2 & \text{if } d/p_1 = \Omega \end{array} \qquad \begin{array}{rcl} d/x & = & \{x \mapsto d\} \\ d/(x := d) & = & \{x \mapsto d\} \\ d/p_1 \wedge p_2 & = & d/p_1 \otimes d/p_2 \\ d/(p_1,p_2) & = & d/p_1 \otimes d/p_2 \end{array}$$

where $\gamma_1 \otimes \gamma_2$ is Ω when $\gamma_1 = \Omega$ or $\gamma_2 = \Omega$ and otherwise is the element $\gamma \in \mathcal{D}^{Dom(\gamma_1) \cup Dom(\gamma_2)}$ such that :

$$\gamma(x) = \begin{cases} \gamma_1(x) & \text{if } x \in Dom(\gamma_1) \backslash Dom(\gamma_2), \\ \gamma_2(x) & \text{if } x \in Dom(\gamma_2) \backslash Dom(\gamma_1), \\ (\gamma_1(x), \gamma_2(x)) & \text{if } x \in Dom(\gamma_2) \cap Dom(\gamma_1). \end{cases}$$

Let us give an example of recursive pattern that uses a constant pattern $(x := n)$. If we match a value of the type *IBlist* defined above, against the recursively defined pattern $p = (x : \texttt{int}, p)|(\_, p)|(x := nil^{\mathbf{0}})$, then we capture in x the list of all integers occurring in the matched value. More in details, the pattern is composed of three alternative subpatterns, each subpattern being applied only if the preceding ones fail. The first subpattern matches if the head of the list is of type `int`. In that case it captures the head in x and recursively applies the pattern to the tail. If the head is not of type `int`, then the second patterns skips it, and recursively applies the pattern to the tail. The constant pattern is applied only if the previous two patterns failed, that is if the matched value is not a pair (head,tail). This means that the value is the empty list, and therefore we associate $nil^{\mathbf{0}}$ to x. The third case of the the definition of γ states that for the whole pattern, x is associated to the list—actually the pair (head,tail)—of the values captured by x in each pair subpattern. Both Theorems C.4.3 and C.4.4 hold also for this extension (the proofs are similar to those found in [FCB08]) and the algorithm of the latter deduces for x the type $t = (\texttt{int} \times t) \vee ch(\mathbf{0})$, that is the type of the lists of integers.

This kind of recursive types and patterns are enough to encode XML data types and manipulate them *à la* ℂDuce. The reader can refer to [BCF03] for more details.

C.5.3 Arrow types

We can extend the type system further by adding function types, so that processes could send ℂDuce expressions as messages. To construct the model, we need to combine the techniques used for ℂDuce with the ones presented in this work.

However, we still cannot get full recursive types, due to the limitation described above. Moreover, we do not know whether the subtyping relation for this system is decidable. The techniques used for the simple system cannot be extended here, because we do not know how to decide whether an arrow type denotes a finite set.

C.5.4 The local calculus

We do not investigate in detail the lextensions proposed above, because, although theoretically challenging, they do not have much practical interest. In the applications, we may not want to have the full power of the π-calculus. In particular it has been observed [Mer04] that the *input capability*, the ability to use in input a received channel, is difficult to implement. In practice it is convenient to restrict to the so-called *local* variant of the π-calculus [Mer04], where the input capability is not allowed.

In our case this restriction has other important consequences :

- the covariant channel type $ch^+(t)$ is no longer necessary. The example of Section C.5.2 cannot be constructed, and indeed it is possible construct a model of the types with full recursion. The absence of input channel types makes also the decision algorithm considerably simpler, as condition CA is invoked only when channel types of different polarity are present. In particular the subtyping of channel types can be reduced to the following condition : $ch^-(t) \leq \bigvee_{i\in I} ch^-(t_i)$ if and only if there exists $i \in I$ such that $t_i \leq t$.
- it is possible to define a type-respecting encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$, similar to the Milner-Turner encoding of the simply typed λ-calculus in π (see for instance [SW02]). This makes explicit arrow types not necessary. However the standard translation of arrow types into channel types does not respect equality, therefore to devise a type-respecting encoding a more subtle approach was needed.

The syntax of types is restricted to the following one

Local $\mathbb{C}\pi$ *Types*	t	::=	$\mathbf{b} \mid ch^-(t) \mid t \times t$		constructors
		$\mid$	$\mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t$		combinators

Note the lack of input type. For these restricted set of types, the decision algorithm is much simpler than the one for the full $\mathbb{C}\pi$-calculus, as for intance, the condition CA is never checked.

The syntax of the local $\mathbb{C}\pi$ is a restriction of the full variant in that input can only happen on constant channels, and not on variables.This ensures that channels sent by other processes cannot be used in input.

Processes	P	::=	$\overline{\alpha}M$	output
		$\mid$	$\sum_{i\in I} c^t(p_i).P_i$	patterned input
		$\mid$	$P \parallel P$	parallel
		$\mid$	$(\nu c^t)P$	restriction
		$\mid$	$!P$	replication

C.5.5 Alternative models

Hitherto, the whole discussion is based on the intuition that channels always have both input and output capabilities, intuition that we materialised with the definition of the model given in Section C.11.2. However, this is just a *partic-*

Messages

$$\frac{}{\Gamma \vdash n : \mathbf{b}_n}\ (\text{const}) \qquad \frac{s_i \not\leq t}{\Gamma \vdash c^t : c\overline{h}(t) \wedge \neg c\overline{h}(s_1) \wedge \ldots \wedge \neg c\overline{h}(s_n)}\ (\text{chan})$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var}) \qquad \frac{\Gamma \vdash M : s \leq t}{\Gamma \vdash M : t}\ (\text{subs}) \qquad \frac{\Gamma \vdash M_1 : t_1,\ \Gamma \vdash M_2 : t_2}{\Gamma \vdash (M_1, M_2) : t_1 \times t_2}\ (\text{pair})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

$$\frac{t \leq \bigvee_{i \in I} \wr p_i \int \quad \Gamma, (t \wedge \wr p_i \int)/p_i \vdash P_i}{\Gamma \vdash \sum_{i \in I} c^t(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : c\overline{h}(t)}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

FIGURE C.6 – Local $\mathbb{C}\pi$ typing rules

ular model based on a *particular* intuition. As a matter of fact, the semantic subtyping approach provides two degrees of freedom in the definition of a model and, thus, of a subtyping relation :

1. We can give different definitions of the extensional interpretation (i.e., Definition C.2.2).
2. Once the extensional definition is set, there may exist different models, that is, different premodels that satisfy Definition C.2.3 for the given $\mathcal{E}$.

Both knobs can be turned to tune the subtyping relation, but between them the one that really matters is the first one.

The extensional interpretation is the one that devises the characteristics of the subtyping relation : from our experience, different models induce slight variations to the subtyping relation, if any at all. For instance, in the definition of CDuce the chosen extensional interpretation admits models that induce different subtyping relations [FCB08]. These models, however are rather difficult to find and differ only in the degree of sharing in recursive types [Fri04]. For this reason, we believe that, once the extensional interpretation is defined, the existence of a model matters much more than its definition. Moreover in our case we conjecture that all models for the extensional interpretation of Definition C.2.2 induce the same subtyping relation. This explains why we focused on the extensional interpretation and relegated the definition of the model to Section C.11.2.

On the contrary it can be very interesting to study alternative definitions of the extensional interpretation, since they correspond to different intuitive semantics and induce substantially different subtyping relations. The reason why we chose our current definition for the extensional interpretation is that it allows us to mix and compare channels of different polarities. This interpretation pushed the approach to its limits, as the issues with recursion and atomic types clearly show. But it is possible to consider different interpretations, in order to either recover existing subtyping relations, or make the subtyping relation more robust with respect to some features. As an example, let us briefly hint at four alternative definitions of the extensional interpretation.

1. We can define the extensional interpretation so that it reflects an intuitive

model in which not only read-and-write channels but also read-only channels and write-only channels are present. Here we would interpret $ch^+(t)$ as the set of all read-only and read-and-write channels for a type s smaller than or equal to t (and similarly for $c\bar{h}(t)$). Although $ch(t)$ would still be the intersection of $ch^+(t)$ and $c\bar{h}(t)$, this would substantially change the subtyping relation (there no longer is a type of all channels, channels of different polarities are less comparable, etc.) yielding a subtyping relation closer to the one defined by Pierce and Sangiorgi [PS96].

2. We can define an extensional interpretation sensitive to the identity of individual channels, that is, an interpretation in which the read-and-write channel type no longer is atomic. We would then obtain a subtyping relation which would be compatible with a language in which pattern matching can also test the name of a channel.
3. We can draw inspiration from the models of ℂDuce and interpret $ch^+(t)$ as the set of (the interpretations of) functions of type $\texttt{unit} \to t$, $c\bar{h}(t)$ as the set of (the interpretations of) functions of type $t \to \texttt{unit}$, and $ch(t)$ as their intersection. Once more this would induce a substantially different subtyping relation. In particular, this interpretation is compatible with an unconstrained definition of recursive types : since in ℂDuce the intersection of two function spaces is never empty, then the counterexample given in Section C.5.2 no longer works ($b \lesssim t$ holds in all models).
4. We can define a variant of the previous interpretation which instead of single functions uses records of functions to interpret channels. In particular we would interpret $ch^+(t)$ as the record type $\texttt{\{read: unit} \to t\texttt{\}}$, $c\bar{h}(t)$ as the record type $\texttt{\{write:}\ t \to \texttt{unit\}}$, and finally $ch(t)$ as the record type $\texttt{\{read: unit} \to t\texttt{, write:}\ t \to \texttt{unit\}}$. This interpretation, too, is compatible with full recursion (as an aside, this is the way in which references types are encoded and implemented in the language ℂDuce, which explains why pointers are possible even if ℂDuce features fully recursive types) but keeps the interpretation of read-only, write-only, and read-and-write channel types, distinct. This interpretation should also induce a conservative extension of the Pierce and Sangiorgi's subtyping relation.

The four above are just some of the possible different interpretations for channel types. Although in this work we considered one particular interpretation, we did not do so with the purpose to fix it as the best possible interpretation, but rather with the purpose to use it to illustrate how to apply the technique of semantic subtyping to mobile processes.

C.6 The functional language ℂDuce

ℂDuce is a very efficient functional language for rapid design and development of applications that manipulate XML data [BCF03]. In this work we concentrate on the foundational aspects of ℂDuce [FCB08] a detailed survey of which can be found in [CF05b]. In that respect, ℂDuce features the same syntactic types as $\mathbb{C}\pi$, with just a single exception, namely, the channel type

constructor is replaced by the function type constructor :

$$\begin{array}{lrcll} \mathbb{C}\textit{Duce Types} & \tau & ::= & \mathbf{b} \mid \tau \to \tau \mid \tau \times \tau & \text{constructors} \\ & & \mid & \mathbf{0} \mid \mathbf{1} \mid \neg\tau \mid \tau \vee \tau \mid \tau \wedge \tau & \text{combinators} \end{array}$$

where the same regularity and contractivity restrictions as in Section C.2.1 apply. We use σ, τ to range over $\mathbb{C}$Duce types and to typographically distinguish them from $\mathbb{C}\pi$ ones, these latter still ranged over by s and t.

Subtyping is characterised in the same way as for $\mathbb{C}\pi$, by defining an interpretation from the above types into a domain $\mathcal{D}$ (that we leave unspecified, see [FCB08]) which satisfies Definition (C.2.3). Definition C.2.2 is modified to account for the new type constructor for functions. We have $\mathbb{E}(-) : \mathcal{T} \to \mathcal{P}(\mathcal{D} + \mathcal{D}\times\mathcal{D} + \mathcal{P}(\mathcal{D}\times\mathcal{D}_\Omega))$ (where $\mathcal{D}_\Omega = \mathcal{D} + \{\Omega\}$, the disjoint union of the domain and of a distinguished error element Ω) while point (d.) of Definition C.2.2 becomes :

$$\text{d.} \quad \mathbb{E}(\sigma \to \tau) = \mathcal{P}\left(\overline{[\![\sigma]\!] \times \overline{[\![\tau]\!]}^{\mathcal{D}_\Omega}}^{\mathcal{D}\times\mathcal{D}_\Omega}\right)$$

where $\overline{X}^Y$ denotes the complement of X with respect to Y (i.e., $Y \setminus X$). In words, the extensional interpretation of $\sigma \to \tau$ is the set of graphs such that if the first element is in $[\![\sigma]\!]$, then the second element is in $[\![\tau]\!]$ (otherwise the second element can be anything, in particular the error Ω). Therefore, *for what concerns subtyping*, we can consider that arrow types are interpreted as follows :

$$[\![\sigma \to \tau]\!] = \{f \subseteq \mathcal{D} \times \mathcal{D}_\Omega \mid \forall (d_{\text{in}}, d_{\text{out}}) \in f.\ d_{\text{in}} \in [\![\sigma]\!] \Rightarrow d_{\text{out}} \in [\![\tau]\!]\}$$

As we did for $\mathbb{C}\pi$, we can use this characterisation to deduce several interesting type equality and containment relations.[5] For instance, we have $(\sigma_1 \vee \sigma_2 \to \tau_1 \wedge \tau_2) \lneq (\sigma_1 \to \tau_1) \wedge (\sigma_2 \to \tau_2)$, where the strictness of containment indicates that we have functions that can have different behaviours according to whether the argument is of type σ_1 or σ_2 (e.g., overloaded functions). For the goals of this work an utmostly interesting equation is

$$(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau' \tag{C.13}$$

whose validity can be easily checked by the reader, by applying the definition of $\mathbb{E}(-)$.

$\mathbb{C}$Duce is a λ-calculus with pairs, overloaded recursive functions, and pattern matching. This is reflected by the following syntax :

$$e ::= x \mid n \mid ee \mid (e,e) \mid \mu f^{\bigwedge_{i\in I}(\sigma_i \to \tau_i)}(x).e \mid \texttt{match}\ e\ \texttt{with}\ p \Rightarrow e \,|\, p \Rightarrow e$$

where patterns p are (but use $\mathbb{C}$Duce types). The type-case expression $(x = e \in \tau)?e_1 : e_2$ can be added as syntactic sugar for the matching expression $\texttt{match}\ e\ \texttt{with}\ x \wedge \tau \Rightarrow e_1 \,|\, x \wedge \neg\tau \Rightarrow e_2$.

Function abstractions use a μ-abstracted name for recursion and specify at their index several arrow types, indicating that the function has all these types

5. The error Ω is included in the codomain of the functions since without it every function would have type $\mathbf{1} \to \mathbf{1}$, therefore every application would be well-typed (with type $\mathbf{1}$). The error element Ω stands for the result of ill-typed applications. Thanks to it $\sigma \to \tau \leq \mathbf{1} \to \mathbf{1}$ does not hold in general, hence, it explicitly avoids the problem above.

$$\frac{}{\Delta;\Gamma \vdash n : \mathbf{b}_n}(const) \qquad \frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}(var)$$

$$\frac{}{\Delta;\Gamma \vdash f : \Delta(f)}(fvar) \qquad \frac{\Delta;\Gamma \vdash e : \sigma \leq \tau}{\Delta;\Gamma \vdash e : \tau}(subs)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash (e_1,e_2) : \tau_1 \times \tau_2}\ (pair)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \sigma \to \tau \quad \Delta;\Gamma \vdash e_2 : \sigma}{\Delta;\Gamma \vdash e_1 e_2 : \tau}\ (appl)$$

$$\frac{\begin{array}{c}(\text{for } \sigma_1 \equiv \sigma \wedge \lfloor p_1 \rfloor,\ \sigma_2 \equiv \sigma \wedge \neg \lfloor p_1 \rfloor) \\ \Delta;\Gamma \vdash e : \sigma \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor \quad \Delta;\Gamma,(\sigma_i/p_i) \vdash e_i : \tau_i\end{array}}{\Delta;\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 : \bigvee_{\{i|\sigma_i \not\simeq \mathbf{0}\}} \tau_i}\ (match)$$

$$\frac{\begin{array}{c}(\text{for } \tau \equiv \bigwedge_{i\in I}(\sigma_i \to \tau_i)) \quad (\forall i \in I, h \in I, j \in J) \\ \sigma_h \wedge \sigma_i = \mathbf{0} \qquad \tau \not\leq \sigma'_j \to \tau'_j \qquad \Delta, f : \tau; \Gamma, x : \sigma_i \vdash e : \tau_i\end{array}}{\Delta;\Gamma \vdash \mu f^{\tau}(x).e : \tau \wedge \bigwedge_{j\in J} \neg(\sigma'_j \to \tau'_j)}\ (abstr)$$

FIGURE C.7 – ℂDuce typing rules

(i.e., their intersection). This is formally stated by the rule (*abstr*) in Figure C.7 which for each $i \in I$ checks that the body e has type τ_i under the hypothesis that x has type σ_i. Note that the types of μ-abstracted variables are recorded in a distinct environment Δ. The distinction here is totally useless (we could have used a unique Γ) but it will be handy when we define the encoding (since μ-abstracted variables are translated into channel constants, then the encoding will be parametric only in Γ).

The only difficult rule is (*match*). It first deduces the type σ of the matched expression and checks whether patterns cover all its possible results (i.e., $\sigma \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor$) ; then it separately checks the first branch under the hypothesis that p_1 is selected (i.e. e is in $\sigma \wedge \lfloor p_1 \rfloor$) and the second branch under the hypothesis that p_2 is selected (i.e., e in $\sigma \wedge \neg \lfloor p_1 \rfloor$) ; finally it discards the return types of the branches *that cannot be selected*, which is safely approximated by the fact that the corresponding σ_i is empty. [6]

The rules in Figure C.7 are the same as those defined in [FCB08] (to which the reader can refer for more details) with just a single exception : in rule (*abstr*) we require that the arrows specified at the index of the function have disjoint domains : $\forall i, h < i . \sigma_h \wedge \sigma_i = \mathbf{0}$. This restriction is necessary (but not

6. The reader may wonder why the system does not return a type error when one of the two branches cannot be selected. As a matter of fact this is a key feature for typing overloaded functions, where the body is repeatedly checked under different hypothesis for some of which the σ_i of some typecase may be empty. This simple function should clarify the point : $\boldsymbol{\mu}\mathsf{f}^{(\mathsf{Int}\to\mathsf{Int};\mathsf{Bool}\to\mathsf{Bool})}(x).(y = x \in \mathsf{Int})?(y+1):\mathsf{not}(y)$
when we type the body under the hypothesis x : Int, then the second branch cannot be selected, while under x : Bool is the first one that cannot be selected. Without the selective union in the typing rule the best type we could have given to this function would have been $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$.

sufficient) in order to avoid the problem of output-driven overloading explained in Section C.7.2. However, it causes no loss of generality, since every ℂDuce function $\mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e$ can be put into this form by iterating on its index the rewriting that replaces $(\sigma_h \wedge \sigma_k \to \tau_h \wedge \tau_k) \wedge (\sigma_k \wedge \neg\sigma_h \to \tau_k) \wedge (\sigma_h \wedge \neg\sigma_k \to \tau_h)$ for every pair of arrows $\sigma_h \to \tau_h$, $\sigma_k \to \tau_k$ such that $\sigma_h \wedge \sigma_k \neq \mathbf{0}$. This rewriting is sound and it is easy to show that the two functions are operationally indistinguishable (e.g., by applicative bisimilarity).

As the intersection of negated channels in the rule (chan) ensures that values of ℂπ yield a model that induces the same subtyping relation as the initial one, so does for ℂDuce the intersection of negated arrows in the rule (*abstr*) : the interpretation defined by sets of values, where values are closed terms generated by $v ::= n \mid \mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e \mid (v, v)$ and types are ℂDuce types, enjoys the same properties.Therefore, we can again use the pattern semantics of Section C.4.1 to define the call-by-value operational semantics of ℂDuce (we omit the straightforward context rules that can be found in [FCB08]).

$$
\begin{array}{rcll}
v_1 v_2 & \longrightarrow & e[v_1/f; v_2/x] & \text{if} \quad v_1 = \mu f^{\tau}(x).e \\
\texttt{match } v \texttt{ with } p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \longrightarrow & e_1[v/p_1] & \text{if} \quad v/p_1 \neq \Omega \\
\texttt{match } v \texttt{ with } p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \longrightarrow & e_2[v/p_2] & \text{if} \quad v/p_1 = \Omega, v/p_2 \neq \Omega
\end{array}
$$

The calculus satisfies the subject reduction property [BCF03].

C.7 Roadmap to the encoding

In this section we discuss the main difficulties encountered in the definition of an encoding of ℂDuce into the local ℂπ-calculus. It lists some failed attempts which will clarify the reasons behind the successful attempt.

C.7.1 The Milner-Turner encoding

Since our encoding involves languages with subtyping, the first approach we tried was to adapt the Milner-Turner (MT) encoding of the call-by-value typed λ-calculus with subtyping into the typed π-calculus with subtyping, as presented in [SW02]. The translation of arrow types presented there is :

$$(\!|\sigma \to \tau|\!) \;=\; c\overline{h}((\!|\sigma|\!) \times c\overline{h}((\!|\tau|\!))) \;.$$

The encoding of λ-terms, decorated by their minimum types, is :

$$
\begin{array}{lcl}
(\!|x^{\tau}|\!)^{\Gamma,x:\tau}_{c} & = & \overline{c}(x) \\
(\!|\lambda x^{\sigma}.e^{\tau}|\!)^{\Gamma}_{c} & = & \boldsymbol{\nu}\, \tilde{n}^{(\!|\sigma\times c\overline{h}(|\tau|)\!|)}(\overline{c}(\tilde{n}) \parallel\, !(\tilde{n}(x,b).(\!|e^{\tau}|\!)^{\Gamma,x:\sigma}_{b})) \\
(\!|e_1^{\sigma\to\tau} e_2^{\rho}|\!)^{\Gamma}_{c} & = & \boldsymbol{\nu}\, \tilde{n}^{(\!|\sigma\times c\overline{h}(|\tau|)\!|)} \boldsymbol{\nu}\, b^{(\!|\rho|\!)}((\!|e_1^{\sigma\to\tau}|\!)^{\Gamma}_{\tilde{n}} \parallel\, \tilde{n}(w).((\!|e_2^{\rho}|\!)^{\Gamma}_{b} \parallel\, b(h).\overline{w}(h,c)))
\end{array}
$$

The encoding of an expression e is parametrised by a type environment Γ such that $\Gamma \vdash e : \tau$ and by a channel $c^{(\!|\tau|\!)}$ on which the value of the expression is returned to the environment. A function is represented by a channel (the "name" of the function) which can be called by sending the input value and a channel on which the output value should be returned. These two parameters are used by a replicated process (the "body" of the function) which returns the output value

upon termination. In the encoding of the application, the encoding of the function is called on the encoding of the argument, and the returned value is returned as the value of the whole expression. This encoding bears a strong resemblance with the continuation passing style transform. In this sense, the return channel of an expression could be seen as the address of the continuation.

Since we translate only well-typed terms, in the case of the application we must have $\rho \leq \sigma$. The encoding of the application (in particular, the $\overline{w}(h, c)$ subterm) is well-typed only if this implies $(\!|\rho|\!) \leq (\!|\sigma|\!)$. This holds true in the simply typed λ-calculus with subtyping, but fails as soon as we add intersection types. In that case, the translation of the types does not preserve the identity of types : in $\mathbb{C}$Duce, we have seen that the identity (C.13) holds (i.e., $(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau'$), while the same does not hold on the encodings of the types at issue since, in general, it is not true that

$$c\bar{h}(s \times c\bar{h}(t)) \wedge c\bar{h}(s \times c\bar{h}(t')) \leq c\bar{h}(s \times c\bar{h}(t \wedge t')) \ .$$

Using this observation we can indeed show that the MT encoding maps a well-typed $\mathbb{C}$Duce expression into an ill-typed $\mathbb{C}\pi$ process. Take $\rho = (\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{int} \to \mathbb{B})$, $\sigma = \mathtt{int} \to \mathbf{0}$, $\tau = (\sigma \to \mathtt{int}) \to \rho \to \mathtt{int}$, and $e = \lambda^{\tau} x.\lambda^{\rho \to \mathtt{int}} y.xy$. The expression e is well-typed with type τ since $\rho \leq \sigma$. The translation of $x : \sigma \to \mathtt{int}, y : \rho \vdash xy : \mathtt{int}$ on channel c' is :

$$P' = \boldsymbol{\nu}\, \tilde{n}'^{(\!|\sigma|\!) \times c\bar{h}(\mathtt{int})} \boldsymbol{\nu}\, b'^{(\!|\rho|\!)} (\overline{\tilde{n}}'(x) \parallel \tilde{n}'(w).(\overline{\tilde{b}}'(y) \parallel b'(h').\overline{w}(h', c')))$$

but the subterm $\overline{w}(h', c')$ is not well-typed. This is because the variable h' must have type $(\!|\rho|\!)$ being received on b'. However it cannot be sent on d' as $(\!|\rho|\!) \not\leq (\!|\sigma|\!)$. The translation of $\vdash e : \tau$ contains a ill-typed term and, therefore, is ill-typed .

C.7.2 Output-driven overloading

In order to give an operational intuition of why the MT encoding does not work, recall that intersections of arrow types are commonly assimilated to the types of overloaded functions. In $\mathbb{C}$Duce, the identity $(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau'$ is justified because overloaded functions can perform a type-case only on the type of the input. Therefore, if on the same input a function returns values of type τ *and* values of type τ' it must return only values that have both types.

In $\mathbb{C}\pi$, however, a process that encodes a function receives in input also the return channel. In principle such process could perform a type-case on this extra piece of information and then execute different computations according to whether the expected result is of type τ or τ'. Such "output-driven" overloaded function can, on the same input, return a value of type τ and a *different* value of type τ' (and not in τ). This is a function that is in $(\!|(\sigma \to \tau) \wedge (\sigma \to \tau')|\!)$ and not in $(\!|\sigma \to \tau \wedge \tau'|\!)$, therefore we expect that $(\!|\sigma \to \tau \wedge \tau'|\!) \lneq (\!|(\sigma \to \tau) \wedge (\sigma \to \tau')|\!)$ which is indeed the case.

C.7.3 The distributive law

At a first analysis, it may seem that the problem is the subtyping relation of $\mathbb{C}\pi$. We may be tempted to change it by adding the following inequation :

$$c\bar{h}(t_1 \wedge t_2) \;\leq\; c\bar{h}(t_1) \vee c\bar{h}(t_2)\ .$$

Since the converse inequality already holds (as seen in Section E.3.2), we would obtain a "contravariant" distributive law of the channel constructor over the intersection. A similar distributive law is used by Hennessy and Riely in [HR02] to *define* the intersection type. As explained in [CDV05], the above inequation is not justified in a calculus endowed with dynamic type-case. It is also not clear at first sight whether introducing the inequation is at all possible using a semantic approach. In any case, this new subtyping relation would not make the translation work either as it would introduce *too many* equations in the translation. For example, being $\mathtt{int} \wedge \mathbb{B} = \mathbf{0}$, we would get

$$c\bar{h}(\mathbf{0} \times c\bar{h}(\mathtt{int} \vee \mathbb{B})) \;\leq\; c\bar{h}(\mathtt{int} \times c\bar{h}(\mathbb{B})) \vee c\bar{h}(\mathbb{B} \times c\bar{h}(\mathtt{int}))\text{.}$$

The type on the left is the encoding of $\mathbf{0} \to \mathtt{int} \vee \mathbb{B}$ and the other type is the encoding of $(\mathtt{int} \to \mathbb{B}) \vee (\mathbb{B} \to \mathtt{int})$. This subtyping gives a problem already for the identity function, which has type $\mathbf{0} \to \mathtt{int} \vee \mathbb{B}$ but not $(\mathtt{int} \to \mathbb{B}) \vee (\mathbb{B} \to \mathtt{int})$.

C.7.4 The negation translation

Intuitively, to find an encoding that respects type equality, we need that, when encoding the arrow type, the operator that encodes the output type distributes over the intersection, while the operator that encodes the input type should not distribute over the intersection. One possible encoding that satisfies this requirement is the following :

$$(\!|\sigma \to \tau|\!) \;=\; c\bar{h}((\!|\sigma|\!) \times \neg(\!|\tau|\!))\ .$$

Indeed the negation is a contravariant constructor that distributes over the intersection. However it was not clear to us what operational interpretation we could attach to this translation. Under this translation of the types, the MT translation of the λ-terms would not be well-typed.

This however was the sparkle that brought us to our solution : (*i*) We want to preserve the naturalness of the MT encoding, that is, to encode functions calls by RPCs that send along with the argument a channel on which the call must return the result ; thus the type of the second argument of the call (i.e., the one that encodes the output type τ) must allow for messages of type $c\bar{h}((\!|\tau|\!))$. (*ii*) We also want the type of this argument to distribute over intersections, in order to respect the subtyping relation ; the use of negation, $\neg(\!|\tau|\!)$, seems to help in this direction. Finally, (*iii*) we want this second argument to be contravariant (since it is under a $c\bar{h}(\,)$, it will then respect the covariance of the output type it is meant to encode) ; but the joint use of two contravariant constructors, $c\bar{h}(\,)$ and $\neg$, would make it covariant, thus we may need to add a further negation to make it contravariant. All this yields, for the encoding of $\sigma \to \tau$, a second argument of type $\neg(c\bar{h}(\neg(\!|\tau|\!)))$, which is *almost* what we are looking for. We say "almost" since it still does not satisfy (*i*) insofar as it is not a supertype of $c\bar{h}((\!|\tau|\!))$; as we will explain in Section C.8.2 one point is still missing from it : $c\bar{h}(\mathbf{1})$ — to verify it, simply compute the difference $c\bar{h}((\!|\tau|\!)) \setminus \neg c\bar{h}(\neg(\!|\tau|\!))$. So we add it, obtaining for the second argument the following encoding $\neg c\bar{h}(\neg(\!|\tau|\!)) \vee c\bar{h}(\mathbf{1})$. This idea is

carried out in details and generalised in the following section.

C.8 The encoding

We propose a modification of the Milner-Turner encoding that respects type equality, and it is very close to the original translation.

C.8.1 The λ-channel constructor

The encoding of the types we propose is parametric with respect to a constructor of $\mathbb{C}\pi$ types that we call "λ-channel" type. This notion is designed to make the translation of types to respect the type equality (unlike the Milner-Turner and distributive approach), and to make the translation of terms to make sense (unlike the negation approach).

Definition C.8.1. A λ-*channel* (noted, $ch^{\lambda}(-)$) is a unary constructor of $\mathbb{C}\pi$ types s.t. :

1. $\overline{ch}(t) \leq ch^{\lambda}(t)$;
2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$;
3. $s \leq t \Leftrightarrow ch^{\lambda}(t) \leq ch^{\lambda}(s)$.

Observe that the three conditions of the definition correspond to the requirements (*i-iii*) we outlined at the end of the previous section. Therefore, Condition (1) is necessary for a meaningful translation of terms, while Conditions (2) and (3) are necessary for respecting the identity of types. Using λ-channel types we can now define a mapping of $\mathbb{C}$Duce types to $\mathbb{C}\pi$-calculus types that respects type equality.

Definition C.8.2. The interpretation function $\{\!|-|\!\} : \mathcal{T}_{\mathbb{C}\text{Duce}} \to \mathcal{T}_{\mathbb{C}\pi}$ is defined as follows

$$\begin{array}{llllllll}
\{\!|\mathbf{b}|\!\} & = & \mathbf{b} & \{\!|\mathbf{0}|\!\} & = & \mathbf{0} & \{\!|\mathbf{1}|\!\} & = & \mathbf{1} & \{\!|\neg\tau|\!\} & = & \neg\{\!|\tau|\!\} \\
\{\!|\sigma \vee \tau|\!\} & = & \{\!|\sigma|\!\} \vee \{\!|\tau|\!\} & & & & \{\!|\sigma \wedge \tau|\!\} & = & \{\!|\sigma|\!\} \wedge \{\!|\tau|\!\} & & & \\
\{\!|\sigma \times \tau|\!\} & = & \{\!|\sigma|\!\} \times \{\!|\tau|\!\} & & & & \{\!|\sigma \to \tau|\!\} & = & \overline{ch}(\{\!|\sigma|\!\} \times ch^{\lambda}(\{\!|\tau|\!\})). & & &
\end{array}$$

Theorem C.8.3. *Let σ and τ be $\mathbb{C}$Duce types. Then $\sigma \leq \tau \iff \{\!|\sigma|\!\} \leq \{\!|\tau|\!\}$.*

C.8.2 Incarnations of λ-channels and their intuition

Possible choices for $ch^{\lambda}(t)$ are of the form $ch^{\lambda_0}(t) \wedge \varphi$ where
– $ch^{\lambda_0}(t) = \neg\overline{ch}(\neg t) \vee \overline{ch}(\mathbf{1})$;
– φ is a constant type such that $\overline{ch}(\mathbf{0}) \leq \varphi$.

Proposition C.8.4. *The constructor $ch^{\lambda_0}(t) \wedge \varphi$ is a λ-channel.*

As the Condition (1) in Definition C.8.1 clearly states, the λ-channel $ch^{\lambda}(t)$ essentially is $\overline{ch}(t)$ plus some extra stuff, some "garbage", that makes the other two conditions —hence type identity preservation— hold. The extra stuff that is added to $\overline{ch}(t)$ is basically given by $ch^{\lambda_0}(t)$. To understand the precise role played by this garbage, it is interesting to consider the following properties :

a. $ch^{\lambda_0}(\mathbf{0}) = \mathbf{1}$
b. $ch^{\lambda_0}(\mathbf{1}) = \neg ch\overline{(}\mathbf{0}) \vee ch\overline{(}\mathbf{1})$
c. $[\![(ch^{\lambda_0}(t) \wedge \neg ch\overline{(}t)) \wedge ch\overline{(}\mathbf{0})]\!] = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbf{1}}\}$.

The first two properties say that $ch^{\lambda_0}(-)$ adds as garbage *at most* (point (a.)) everything and *at least* (point (b.)) all non-channel types plus the channel which outputs everything. In order to exactly determine which channels $ch^{\lambda_0}(t)$ adds to to $ch\overline{(}t)$ let us take out all $ch\overline{(}t)$ and consider just the channels that remained : this is exactly what $(ch^{\lambda_0}(t) \wedge \neg ch\overline{(}t)) \wedge ch\overline{(}\mathbf{0})$ does. Point (c.) states that these are all channels that can send values both inside and outside t. That is, these are all the channels for which it is not possible to predict the result of a test that checks whether the messages they transport are of type t.

This last observation is the key to understand why the complicated definition of $ch^{\lambda_0}(-)$ is necessary. We have observed that the MT translation does not work because it allows a "output-driven" overloading whereby a function can have different behaviours for different expected types of the result. The more general channel type $ch^{\lambda_0}(-)$ allows (potentially, in the types) the caller to "confuse" such output-driven functions, by sending "garbage" reply channels. Although in practice, encodings don't do that, the possibility of a output-driven function is ruled out also at the level of the types. It is like the presence of the Police in Utopia : everybody behaves well in Utopia, and the Police never works. But the presence of the Police is the visible representation of the fact the everybody behaves well.

To put it otherwise, if we take a channel that has type $ch^{\lambda}(s) \vee ch^{\lambda}(t)$, it is impossible to deduce whether it is only of type $ch^{\lambda}(s)$ or only of type $ch^{\lambda}(t)$. Even if it can transport all messages of type, say, t, it could be because the channel was in the garbage generated by $ch^{\lambda}(s)$. So λ-channels introduce some latent noise that makes it impossible to determine which output type they encode.

Although the constructor is parametric on a type φ, non-channel types play no active role in the encoding. Therefore it is reasonable (and it makes the encoding more understandable) to minimise φ (that is, $\varphi = ch\overline{(}\mathbf{0})$) so that $[\![ch^{\lambda}(t)]\!]$ only contains channels. In particular, this choice implies that $ch^{\lambda}(\mathbf{0}) = ch\overline{(}\mathbf{0})$ (all channels), $ch^{\lambda}(\mathbf{1}) = ch\overline{(}\mathbf{1})$ (just the channel which outputs everything). All the development, however, is independent from this choice.

C.8.3 Encoding of the terms

We describe here the mapping of $\mathbb{C}$Duce terms to $\mathbb{C}\pi$-calculus terms. What we translate are in fact typing derivations. To simplify the notation, we write e^{τ} assuming that τ is the type of e in the last step of the derivation. We use a similar convention for the immediate sub-expressions of e which are in the premises of the last applied rule. The translation is parametrised by a "continuation channel" α of type $ch\overline{(}\{\!\{\tau\}\!\})$. For readability we decorate the channels with their types only when we restrict them and in rule (*fvar*). We also adopt the $\mathbb{C}$Duce's convention to write $x{:}\tau$ for the pattern $x \wedge \tau$. The translation also requires a

straightforward translation of the patterns (it just encodes the types occurring in them) whose details are omitted.

Definition C.8.5. The translation of the expression e^τ on a channel α is defined by cases on the last applied typing rule :

(const) $\{\!\{n^{\mathbf{b}_n}\}\!\}^\Gamma_\alpha = \overline{\alpha}(n)$

(var) $\{\!\{x^\tau\}\!\}^{\Gamma,x:\tau}_\alpha = \overline{\alpha}(x)$

(fvar) $\{\!\{f^\tau\}\!\}^\Gamma_\alpha = \overline{\alpha}(f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\}\times ch^\lambda(\{\!\{\tau_i\}\!\}))})$ *(where* $\tau = \bigwedge_{i\in I}(\sigma_i \to \tau_i)$*)*

(pair) $\{\!\{(e_1^{\sigma_1},e_2^{\sigma_2})^\tau\}\!\}^\Gamma_\alpha = \boldsymbol{\nu}\, \tilde{n}^{\{\!\{\sigma_1\}\!\}}\boldsymbol{\nu}\, b^{\{\!\{\sigma_2\}\!\}}(\{\!\{e_1^{\sigma_1}\}\!\}^\Gamma_{\tilde{n}} \parallel \tilde{n}(w{:}\{\!\{\sigma_1\}\!\}).(\{\!\{e_2^{\sigma_2}\}\!\}^\Gamma_b \parallel b(h{:}\{\!\{\sigma_2\}\!\}).\overline{\alpha}(w,h)))$ *(where* $\tau = \sigma_1 \times \sigma_2$*)*

(appl) $\{\!\{(e_1^{\sigma\to\tau}e_2^\sigma)^\tau\}\!\}^\Gamma_\alpha = \boldsymbol{\nu}\, \tilde{n}^{\{\!\{\sigma\to\tau\}\!\}}\boldsymbol{\nu}\, b^{\{\!\{\sigma\}\!\}}(\{\!\{e_1^{\sigma\to\tau}\}\!\}^\Gamma_{\tilde{n}} \parallel \tilde{n}(w{:}\{\!\{\sigma\to\tau\}\!\}).(\{\!\{e_2^{\sigma}\}\!\}^\Gamma_b \parallel b(h{:}\{\!\{\sigma\}\!\}).\overline{w}(h,\alpha)))$

(subs) $\{\!\{(e^\sigma)^\tau\}\!\}^\Gamma_\alpha = \boldsymbol{\nu}\, \tilde{n}^{\{\!\{\sigma\}\!\}}(\{\!\{e^\sigma\}\!\}^\Gamma_{\tilde{n}} \parallel \tilde{n}(w{:}\{\!\{\sigma\}\!\}).\overline{\alpha}(w))$ *(where* $\sigma \le \tau$*)*

(match) $\{\!\{(\mathtt{match}\ e^\sigma\ \mathtt{with}\ p_1{\Rightarrow}e_1^{\tau_1}\ /p_2{\Rightarrow}e_2^{\tau_2})^\tau\}\!\}^\Gamma_\alpha = \boldsymbol{\nu}\, \tilde{n}^{\{\!\{\sigma\}\!\}}\boldsymbol{\nu}\, b^{(\{\!\{\sigma_1\}\!\}\times ch(\{\!\{\tau_1\}\!\}))\vee(\{\!\{\sigma_2\}\!\}\times ch(\{\!\{\tau_2\}\!\}))}((P_1 + P_2) \parallel Q)$

$$\begin{array}{rrcl} \textit{where} & P_1 & = & b(\{\!\{p_1\}\!\}, d{:}ch(\{\!\{\tau_1\}\!\})).\{\!\{e_1^{\tau_1}\}\!\}^{\Gamma,\sigma_1/p_1}_d, \\ & P_2 & = & b(\{\!\{p_2 \wedge \neg\lbag p_1\rbag\}\!\}, d{:}ch(\{\!\{\tau_2\}\!\})).\{\!\{e_2^{\tau_2}\}\!\}^{\Gamma,\sigma_2/p_2}_d, \\ & Q & = & \{\!\{e^\sigma\}\!\}^\Gamma_{\tilde{n}} \parallel \tilde{n}(h{:}\{\!\{\sigma\}\!\}).\bar{b}(h,\alpha) \\ & \sigma_1 & = & \sigma \wedge \lbag p_1\rbag, \quad \sigma_2 \;=\; \sigma \wedge \neg\lbag p_1\rbag, \quad \tau \;=\; \bigvee_{\{i\mid \sigma_i \not\simeq \mathbf{0}\}} \tau_i \end{array}$$

(abstr) $\{\!\{(\mu f^{\bigwedge_{i\in I}(\sigma_i\to\tau_i)}(x).e)^\tau\}\!\}^\Gamma_\alpha = \boldsymbol{\nu}\, f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\}\times ch^\lambda(\{\!\{\tau_i\}\!\}))}(\overline{\alpha}(f) \parallel \mathsf{body}(f))$

$$\begin{array}{rrcl} \textit{where} & \mathsf{body}(f) & = & !(\sum_{i\in I} f(x{:}\{\!\{\sigma_i\}\!\}, b{:}ch(\{\!\{\tau_i\}\!\})).\{\!\{e^{\tau_i}\}\!\}^{\Gamma,x:\sigma_i}_b \\ & & & \quad + f(x{:}\bigvee_{i\in I}\{\!\{\sigma_i\}\!\}, b{:}\bigvee_{i\in I}(ch^\lambda(\{\!\{\tau_i\}\!\}) \wedge \neg ch(\{\!\{\tau_i\}\!\})).\mathbf{0}) \\ & \tau & = & \bigwedge_{i\in I}(\sigma_i \to \tau_i) \wedge \bigwedge_{j\in J} \neg(\sigma'_j \to \tau'_j). \end{array}$$

In rule (*fvar*), we assume that every μ-abstracted variable f has a corresponding channel constant f^t for every suitable $\mathbb{C}\pi$ type t. This allows the encoding to be parametric only in the Γ environment, and not in the Δ one. In a match the expressions e_1 and e_2 play the role of two functions to be chosen according to the type of the argument e. Therefore we encode the match with a patterned sum of the encodings of e_1 and e_2 in parallel with the encoding of e.

The translation of a functional term is very similar to the original MT translation. To deal with overloading, the body of the function features a patterned choice. This choice includes all behaviours that the function can produce on different inputs, and the special sub-term $f(x{:}\bigvee_{i\in I}\{\!\{\sigma_i\}\!\}, b{:}\bigvee_{i\in I}(ch^\lambda(\{\!\{\tau_i\}\!\}) \wedge \neg ch(\{\!\{\tau_i\}\!\})).\mathbf{0}$, which we call the *functional garbage*. The role of this sub-term is to obtain well-typed terms. However we will see that, within the context of translation of $\mathbb{C}$Duce terms, the functional garbage choice is never taken. Indeed, carrying on with our analogy, this functional garbage corresponds to the prison of Utopia : it is there to capture misbehaving terms, even if we all know that there isn't any.

C.9 Correctness of the encoding

We start by stating that the translation produces well-typed terms.

Theorem C.9.1. *If* $\Delta;\Gamma \vdash e : \tau$, *then* $\{\!\{\Gamma\}\!\} \vdash \{\!\{e^\tau\}\!\}^\Gamma_{c\{\!\{\tau\}\!\}}$ *and* $\{\!\{\Gamma\}\!\}, x : ch\overline{(}\{\!\{\tau\}\!\}) \vdash \{\!\{e^\tau\}\!\}^\Gamma_x$, *where* $\{\!\{\Gamma\}\!\} = \{y : \{\!\{\sigma\}\!\} \mid y : \sigma \in \Gamma\}$.

In the following we convene that when we write $\{\!\{e\}\!\}^\Gamma_c$, then there are τ and Δ such that $\Delta;\Gamma \vdash e : \tau$ and $ch\overline{(}\{\!\{\tau\}\!\})$ is the type of c.

A first observation is that all reductions out of the encoding of a ℂDuce expression are deterministic (since patterns in sums are mutually exclusive) and never use the functional garbage in the body of functions. A *functional redex* is a redex of the shape $\mathsf{body}(f) \parallel \overline{f}(v, c)$. A reduction is *safe* if it is deterministic and each functional redex is reduced by choosing an alternative in $\mathsf{body}(f)$ different from the functional garbage. We denote safe reductions by $\longrightarrow_s$: as usual $\longrightarrow_s^*$ is the reflexive and transitive closure of $\longrightarrow_s$.

Lemma C.9.2. *All reductions starting from* $\{\!\{e\}\!\}^\varnothing_c$ *where* e *is an arbitrary* ℂ*Duce expression are safe.*

In order to state the correctness of the encoding, it is crucial to understand how ℂDuce values are mapped to ℂπ processes. As it is clear from the encoding, a functional value is mapped into the output of a private channel name in parallel with the encoding of the function body. We can then say that the ℂπ value corresponding to a functional value is a channel name. The encoding of a pair of ℂDuce values reduces to a process which outputs the pair of the corresponding ℂπ values in parallel with the function bodies of all functions which occur in the two values.

To formalise the above we will assume that *all function names* in the current value are *distinct* and *fixed*, so that we cannot rename them. We define two mappings, one from ℂDuce values to ℂπ values and one from ℂDuce values to sets of channel names.

Definition C.9.3.

1. The mapping $\mathsf{cpv}(-)$ is defined by induction on ℂDuce values as follows :
   - $\mathsf{cpv}(n) = n$;
   - $\mathsf{cpv}(\mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e) = f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\} \times ch^\lambda(\{\!\{\tau_i\}\!\}))}$;
   - $\mathsf{cpv}((v_1, v_2)) = (\mathsf{cpv}(v_1), \mathsf{cpv}(v_2))$.
2. The mapping $\mathsf{func}(-)$ is defined by induction on ℂDuce values as follows :
   - $\mathsf{func}(n) = \varnothing$;
   - $\mathsf{func}(\mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e) = \{f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\} \times ch^\lambda(\{\!\{\tau_i\}\!\}))}\}$;
   - $\mathsf{func}((v_1, v_2)) = \mathsf{func}(v_1) \cup \mathsf{func}(v_2)$.

The above mappings can express the normal forms of processes encoding values :

Lemma C.9.4. $\{\!\{v\}\!\}^\varnothing_c \longrightarrow_s^* \boldsymbol{\nu}\, \mathsf{func}(v)(\overline{c}(\mathsf{cpv}(v)) \| _{f\in\mathsf{func}(v)} \mathsf{body}(f))$.

More generally, one would like to have that if e is a well-typed ℂDuce expression and $e \longrightarrow^* v$, then $\{\!\{e\}\!\}^\varnothing_c \longrightarrow_s^* \boldsymbol{\nu}\, \mathsf{func}(v)(\overline{c}(\mathsf{cpv}(v)) \| _{f\in\mathsf{func}(v)} \mathsf{body}(f))$. Unfortunately, the corresponding result does not even hold for the MT encoding of λ-calculus into π-calculus [Mil92], *a fortiori* nor does for our encoding.

A reason for this failure is that when the whole λ-term is a β-redex its encoding reduces to a π-term which differs from the encoding of the corresponding β-contractum in the positions of the restriction and of the replicated input representing the reduced λ-abstraction. Moreover when a β-redex in argument position is contracted (following the call-by-value reduction strategy) the encoding of the reduced λ-term needs in its turn to be evaluated in order to be related with the encoding of the original λ-term.

Our encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$ being essentially an extension of the MT encoding has luckily no more problems than the original one, so we can show similar soundness results. To formulate these results we need to define for $\mathbb{C}\pi$ processes a standard notion of typed barbed congruence with respect to an environment Γ ($\Gamma \rhd P \cong Q$), see [SW02].

The main theorem of this section states that if a $\mathbb{C}$Duce expression reduces to a value, then its encoding reduces to the process which is barbed congruent to the encoding of that value, and vice versa if the evaluation of a $\mathbb{C}$Duce expression does not terminate, then the evaluation of its encoding does not terminate either.

Theorem C.9.5 (Correctness)**.**

1. *If* $e \longrightarrow^* v$*, then* $\{\!\{e\}\!\}_c^{\varnothing} \longrightarrow_s^* P$ *for some* P *such that* $\varnothing \rhd P \cong \boldsymbol{\nu}\, \mathsf{func}(v)(\overline{c}(\mathsf{cpv}(v))\|_{f \in \mathsf{func}(v)} \mathsf{body}(f))$.
2. *If* e *diverges, then* $\{\!\{e\}\!\}_c^{\varnothing}$ *diverges too.*

From this, and from compositionality, it is easy to obtain soundness. Given two $\mathbb{C}$Duce terms $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$ we denote by $\cdot; - e e' : \tau$ the standard Morris-style observational congruence (as defined, for instance, in [SW02] pag. 478).

Corollary C.9.6 (Soundness)**.** *If* $\Delta; \Gamma \vdash e : \tau$ *and* $\Delta; \Gamma \vdash e' : \tau$ *and* $\{\!\{\Gamma\}\!\} \rhd \{\!\{e\}\!\}_c^{\Gamma} \cong \{\!\{e'\}\!\}_c^{\Gamma}$*, then* $\cdot; - e e' : \tau$.

Notice that completeness fails for our encoding, for the same reason as it fails for the original MT encoding.

C.10 Conclusion

Pierce and Sangiorgi's subtyping for the π-calculus, though very elegant, is structurally very poor : it essentially amounts to compare the levels of nesting of channel constructors with the same polarity. In order to obtain a much richer and expressive subtyping relation, we combine here their types with union, intersection, and negation types. This is not a new idea—at least for what concerns unions and intersections—, but the originality of our approach is that the theory is semantically justified via a set theoretic interpretation of types as sets of values, which looks as quite a reasonable interpretation. The naturalness of the interpretation is justified and supported by several technical aspects, and reinforced by the definition of the type-preserving translation of $\mathbb{C}$Duce into the a local variant of the process calculus.

While the interpretation is very simple, its consequences are not. We have seen that deciding subtyping requires to enumerate and check one by one the atoms that compose the types involved in the verification. Such a degree of complexity is present only in the general framework. This is acceptable since our work aims at establishing the foundational basis of subtyping for π-calculus. Of course, such a degree of complexity makes the calculus unfit for practical applications. However in a practical scenario one would rather resort to the local variant and, in that case, the extra complexity of subtyping disappears, the subtyping algorithm being reduced to perform classic structural checks on syntactic types.

The fact that here we have to descend to the very structure that composes types (the world "atoms" is quite suggestive in this case) is not overly surprising. The point is that we are touching deep into the semantics of computations. This is witnessed by the fact that some characteristics (in some case, some "oddities") of $\mathbb{C}\pi$ are shared by completely different paradigms for which a semantic subtyping technique was used. For instance, $\mathbb{C}$Duce function values require some special non-structural typing rule which uses negated literals. This kind of rule becomes necessary also for $\mathbb{C}\pi$ as soon as one consider its local variant. A much more striking correspondence happens with atoms : we have shown that in order to decide the subtyping relation in $\mathbb{C}\pi$ one must be able to decide the atomicy of the types. Quite surprisingly the same problem appears in λ-calculus (actually, in any semantic subtyping based system) as soon as we try to extend it with polymorphic types. Imagine that we embed our types (whatever they are) with type variables $X, Y, \ldots$. Then the "natural" (semantic) extension of the subtyping relation is to quantify the interpretations over all substitutions for the type variables :

$$t_1 \leq t_2 \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \forall s. [\![t_1[s/X]]\!] \subseteq [\![t_2[s/X]]\!] \ . \tag{C.14}$$

Consider now the following inequality (taken from [HFC05]) where t is a closed type

$$(t, X) \leq (t \times \neg t) \vee (X \times t). \tag{C.15}$$

It is easy to see that this inequality holds if and only if t is atomic. If t is not atomic, then it has at least one non-empty proper subtype, and (C.14) does not hold when we substitute this subtype for X. If instead t is atomic, then for all X either $t \leq X$ or $t \leq \neg X$, whence (C.15). Note that this example does not use any fancy or powerful type constructor, such as arrows or channels : it only uses products and type variables. So it applies to all polymorphic extensions of semantic subtyping where, once more, deciding subtyping reduces to deciding whether some type is atomic or not.

These and other similarities are discussed in [Cas05] to which the reader can refer for deeper analysis and a discussion on perspectives.

C.11 Proofs

C.11.1 Characterising inclusion (Theorem C.3.2 and Proposition C.3.3)

In this section we first prove Theorem C.3.2 and then strengthen the result as in Proposition C.3.3.

We recall that in a boolean algebra, an *atom* is a minimal nonzero element. A boolean algebra is *atomic* if every nonzero element is greater than or equal to an atom. It is easy to prove that an atomic boolean algebra is equivalent to a subset of the powerset of its atoms.

Let $(D, \mathsf{\wedge}, \mathsf{\vee}, \mathbf{0}, \mathbf{1})$ be an atomic boolean algebra where, as customary, $d' \leq d$ if and only if $d' \mathsf{\vee} d = d$. For every $d \in D$ we denote $\downarrow d$ (that is, the set of all elements smaller than or equal to d) as $ch^+(d)$ and $\uparrow d$ (that is, the set of all elements larger than or equal to d) as $ch^-(d)$. We want to give an equivalent characterisation of the equation

$$\bigcap_{i\in I} ch^+(d_1^i) \cap \bigcap_{j\in J} ch^-(d_2^j) \subseteq \bigcup_{h\in H} ch^+(d_3^h) \cup \bigcup_{k\in K} ch^-(d_4^k)$$

that does not use the "operators" $ch^+(), ch^-()$. Notice that

$$\bigcap_{i\in I} ch^+(d_1^i) = ch^+(\bigwedge_{i\in I} d_1^i) \qquad \text{and} \qquad \bigcap_{j\in J} ch^-(d_2^j) = ch^-(\bigvee_{j\in J} d_2^j)\ .$$

Also, if there exist h, h' such that $d_3^{h'} \leq d_3^h$, then we can ignore $d_3^{h'}$ as $ch^+(d_3^{h'}) \subseteq ch^+(d_3^h)$. Dually for the d_4^k. Therefore we can concentrate on the case

$$ch^+(d_1) \cap ch^-(d_2) \subseteq \bigcup_{h\in H} ch^+(d_3^h) \cup \bigcup_{k\in K} ch^-(d_4^k)$$

where no two d_3^h are comparable, and no d_4^k are comparable.

The first case in which the inclusion holds is when $ch^+(d_1) \cap ch^-(d_2) = \varnothing$, which happens exactly when $d_2 \not\leq d_1$. If $d_2 \leq d_1$, without loss of generality we can also assume that $d_3^h \geq d_2$ for all $h \in H$ and that $d_4^k \leq d_1$ for all $k \in K$. This is because if $d_3^{\bar{h}} \not\geq d_2$ for some $\bar{h}$ then no element of $ch^-(d_2)$ can be in $ch^+(d_3^{\bar{h}})$. We can thus ignore such sets to test for the inclusion, and similarly for the d_4^k's.

The inclusion surely holds if for some $\bar{h}$ we have $d_1 \leq d_3^{\bar{h}}$, or if for some $\bar{k}$ we have $d_2 \geq d_4^{\bar{k}}$, since then, for instance in the former case, $ch^+(d_1)$ is contained in $ch^+(d_3^{\bar{h}})$ and so is its intersection with $ch^-(d_2)$. The most difficult case occurs when

- $d_2 \leq d_1$;
- for all $h \in H$, $d_3^h \geq d_2$;
- for all $k \in K$, $d_4^k \leq d_1$;
- for all $h \in H$, $d_3^h \not\geq d_1$;
- for all $k \in K$, $d_4^k \not\leq d_2$.

The way of thinking the inclusion is the following. (From now on it will be easier to think of D as a subset of the powerset of its atoms ; therefore we will sometimes say "contained" rather than "smaller", and so on.) Consider a d in $ch^+(d_1) \cap ch^-(d_2)$. If d is not below any of the d_3^h then it must be above one of the d_4^k. Suppose there is an element x of d_1 which is in no d_3^h (more precisely,

suppose that there is an atom $\overline{d}$ such that $\overline{d} \leq d_1$ and for all h, $\overline{d} \not\leq d_3^h$; to stress that it is an atom denote $\overline{d}$ by $\{x\}$). Then $d_2 \vee \{x\}$ is not contained in any of the d_3^h, and it must contain one of the d_4^k. This implies that for such d_4^k, $d_4^k \setminus d_2 \leq \{x\}$ [7]. Consider now two elements x_1, x_2 in d_1 such that if x_1 belongs to d_3^h then x_2 does not belong to d_3^h. Then $d_2 \vee \{x_1, x_2\}$ is not contained in any of the d_3^h, and it must contain one of the d_4^k. This implies that for such d_4^k, $d_4^k \setminus d_2 \leq \{x_1, x_2\}$.

More generally : for every $h \in H$ choose an element $x_h \in d_1 \setminus d_3^h$. Clearly we have that $d_2 \vee \{x_h \mid h \in H\}$ is not contained in any of the d_3^h. Reasoning as above we then have that there is a d_4^k such that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H\}$.

This proves the necessity of condition (CA) : for every choice of $x_h \in d_1 \setminus d_3^h$ there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H\}$.

We argued that the condition (CA) is necessary. It is also sufficient : if the condition holds, every set d included in d_1, containing d_2, and which is not contained in any of the d_3^h, must contain a set of the form $d_2 \vee \{x_h \mid h \in H\}$: just pick one witness of noncontainment for every d_3^h. Thus d contains one of the d_4^k.

We can strengthen the result as stated in Proposition C.3.3. Consider the case where for some h the sets $d_1 \setminus d_3^h$ are infinite. Let $H_i \subseteq H$ be the set of such h. Pick $\bar{h} \in H_i$, and let $\bar{H} = H \setminus \{\bar{h}\}$. Since there are only finitely many d_4^k, the condition is satisfied if and only if for at least two (in fact infinitely many) different choices $x'_{\bar{h}}$ and $x''_{\bar{h}}$ we have that the same d_4^k satisfies $d_4^k \setminus d_2 \leq \{x_h \mid h \in \bar{H}\} \bigvee \{x'_{\bar{h}}\}$, and $d_4^k \setminus d_2 \leq \{x_h \mid h \in \bar{H}\} \bigvee \{x''_{\bar{h}}\}$. Therefore we must have $d_4^k \setminus d_2 \subseteq \{x_h \mid h \in \bar{H}\}$. Repeating this for every index in H_i, we conclude that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H \setminus H_i\}$. Noting that $H \setminus H_i = H_f$, we conclude the proof that the condition (CA) is equivalent to condition (CA*) : for every choice of $x_h \in d_1 \setminus d_3^h$, $h \in H_f$, there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H_f\}$. (We could improve further by considering only those $d_1 \setminus d_3^h$ whose cardinality is not greater than the number of d_4^k - we do not need this for our purposes.)

C.11.2 The existence of a model

We shall construct here a model for the simplest of our type systems. This amounts to build a pre-model and then show that it satisfies Definition C.2.3. To understand the definitions and the proofs in this section, it is advisable to read first Sections C.3 and C.11.1.

Types are stratified according to the height of the nesting of the channel constructor. We define the height function ht as follows :

- $hb = h\mathbf{0} = h\mathbf{1} = 0$;
- $hch(t) = hch^+(t) = hch\overline{(t)} = ht + 1$;
- $ht_1 \bigvee t_2 = ht_1 \bigwedge t_2 = \max(ht_1, ht_2)$;
- $h\neg t = ht$.

Then we set $\mathcal{T}_n \stackrel{\text{def}}{=} \{t \mid ht \leq n\}$.

7. It is in fact $d_4^k \setminus d_2 = \{x\}$, since $d_4^k \not\leq d_2$.

Our pre-model for the types is built in steps. We start by providing a model for types of height 0, that is types in $\mathcal{T}_0$. Note that we must define the semantics only for type constructors, because the interpretation of the combinators is determined by the definition of pre-model. The only constructors of height 0 are the basic types, for these we assume existence of a universe of interpretation $\mathbb{B}$. We also assume that every basic type b has an interpretation $\mathcal{B}[\![b]\!] \subseteq \mathbb{B}$. Finally, we need a small technicality : we add to our types of height 0 the types $\overbrace{ch(\ldots(ch}^{k}(\mathbf{0})))$, that we denote here as $\mathbb{k}$. Although at higher levels these types are just syntactic sugar, we need them at level 0 to witness the existence of infinitely many channel types. The pre-model at level 0 is exactly formed by the basic types plus the positive natural numbers to model the $\mathbb{k}$. Therefore $\mathcal{D}_0 = \mathbb{B} + \mathbb{N}^+$ with $[\![b]\!]_0 = \mathcal{B}[\![b]\!]$ and $[\![\mathbb{k}]\!]_0 = \{k\}$. The boolean combinators are interpreted by using the corresponding set-theoretic combinators, according to Definition C.2.1.

Using this pre-model we define a subtyping relation over $\mathcal{T}_0$ as $t \leq_0 t'$ if and only if $[\![t]\!]_0 \subseteq [\![t']\!]_0$. We shall denote by $=_0$ the corresponding equivalence.

Now suppose we have a pre-model $\mathcal{D}_n$ for $\mathcal{T}_n$, with corresponding preorder $\leq_n$ and equivalence $=_n$. We call $\widetilde{\mathcal{T}}_n$ the set of equivalence classes $\mathcal{T}_n/_{=_n}$. Then $\mathcal{D}_{n+1}$ is defined as follows :

$$\mathcal{D}_{n+1} \stackrel{\text{def}}{=} \mathbb{B} + \widetilde{\mathcal{T}}_n \ .$$

with the following interpretation of channel types :

- $[\![ch^+(t)]\!]_{n+1} = \{[t']_{=_n} \mid t' \leq_n t\}$;
- $[\![ch^-(t)]\!]_{n+1} = \{[t']_{=_n} \mid t \leq_n t'\}$.

In principle each of these pre-models defines a different preorder between types. However, all such preorders coincide in the following sense :

Proposition C.11.1. *Let $t, t' \in \mathcal{T}_n$ and $k, h \geq n$, then $t \leq_k t'$ if and only if $t \leq_h t'$.*

Proof: To carry out the proof we use an interesting fact : every singleton of our pre-models is denoted by some type. For elements of $\mathbb{B}$ this was an assumption. For elements of $\widetilde{\mathcal{T}}_n$, observe that the singleton $\{[t]_{=_n}\}$ is denoted by the type $ch(t)$.

Suppose we have a model $\mathcal{D}_n$ for $\mathcal{T}_n$, with corresponding preorder $\leq_n$ and equivalence $=_n$. We call $\widetilde{\mathcal{T}}_n$ the set of equivalence classes $\mathcal{T}_n/=_n$. Then we set $\mathcal{D}_{n+1} \stackrel{\text{def}}{=} \mathbb{B} + \widetilde{\mathcal{T}}_n$, with the semantics of the channel types being

$$\begin{aligned} [\![ch^+(t)]\!]_{n+1} &= \{[t']_{=_n} \mid t' \leq_n t\}\,; \\ [\![ch^-(t)]\!]_{n+1} &= \{[t']_{=_n} \mid t \leq_n t'\}\,; \\ [\![\mathbb{k} + \mathbb{1}]\!]_{n+1} &= \{[\mathbb{k}]_{=_n}\}\,. \end{aligned}$$

Note that now the semantics of $\mathbb{1} = ch(\mathbf{0})$ is the expected one, and in general the semantics of $\mathbb{k} + \mathbb{1}$ coincides with the semantics of $ch(\mathbb{k})$. Therefore in the semantics at levels greater than 0 we can appropriately desugar the $\mathbb{k}$s, and ignore their existence.

When is a type t empty ? Given a type t we put it in disjunctive normal form. Clearly t is empty if and only if all summands are empty. If a summand contains literals of both basic types and channel types it is easy to decide emptiness : if it contains two positive literals of different kinds, then it is empty. If the positive literals are all of one kind, it is empty if and only if it is empty when removing the negative literals of the other kind. Finally the intersection of only negative literals is empty if the two kinds separately cover their own universe of interpretation. (That is if the union of all negated basic types is $\mathbb{B}$ and similarly for the channel types.)

Therefore it is enough to check emptiness for intersections of literals of one kind only. For base types :

$$\bigwedge_{b\in P} b \wedge \bigwedge_{b\in N} \neg b \ .$$

For channel types :

$$\bigwedge_{i\in I} ch^+(t_1^i) \wedge \bigwedge_{j\in J} ch\overline{(}t_2^j) \wedge \bigwedge_{h\in H} \neg ch^+(t_3^h) \wedge \bigwedge_{k\in K} \neg ch\overline{(}t_4^k) \ .$$

Using equations (C.5) and (C.6) of Section E.3.2 we can simplify the last expression to

$$ch^+(t_1) \wedge ch\overline{(}t_2) \wedge \bigwedge_{h\in H} \neg ch^+(t_3^h) \wedge \bigwedge_{k\in K} \neg ch\overline{(}t_4^k) \ .$$

To prove Proposition C.11.1, we now prove by induction the following statement : let $t \in \mathcal{T}_n$, then

- $t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;
- $|t|_n = l$ if and only if $|t|_{n+1} = l$;

where $|t|$ denotes the cardinality of t.

We start by the case $n = 0$. The "algorithm" for checking emptiness works in the same way for basic types. The only difference occurs for the types $\mathbb{k}$. The condition to check at level 0 is the following

$$\mathbb{N} \cap \bigcap_{\mathbb{k}\in P} [\![\mathbb{k}]\!]_0 \subseteq \bigcup_{\mathbb{k}\in N} [\![\mathbb{k}]\!]_0$$

which can be true only if there are two different $\mathbb{k} \in P$ or if the only $\mathbb{k}$ in P is also in N. It is important here that $\mathbb{N}$ is infinite, so no finite union of singletons can cover it. Therefore the condition above is equivalent to

$$\widetilde{\mathcal{T}_0} \cap \bigcap_{\mathbb{k}\in P} [\![\mathbb{k}]\!]_1 \subseteq \bigcup_{\mathbb{k}\in N} [\![\mathbb{k}]\!]_1$$

and therefore $t =_0 \mathbf{0}$ if and only if $t =_1 \mathbf{0}$. As for the cardinality : the proof is more general and it is the same as the inductive step case that we show next.

For the inductive step suppose that we know that for every type $t \in \mathcal{T}_n$ we have

- $t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;
- $|t|_n = l$ if and only if $|t|_{n+1} = l$.

Now take a type $t \in \mathcal{T}_{n+1}$, we want to prove that

- $t =_{n+1} \mathbf{0}$ if and only if $t =_{n+2} \mathbf{0}$;
- $|t|_{n+1} = l$ if and only if $|t|_{n+2} = l$.

Again the "algorithm" for checking the emptiness of basic types does not change. In the case of channel types we have to check that

$$[\![ch^+(t_1)]\!]_{n+1} \cap [\![ch^-(t_2)]\!]_{n+1} \subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+1} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+1}$$

if and only if

$$[\![ch^+(t_1)]\!]_{n+2} \cap [\![ch^-(t_2)]\!]_{n+2} \subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+2} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+2} \ .$$

As argued in the previous section, the first condition is equivalent to :

LE. $t_2 \not\leq_n t_1$ or

R1. $\exists h \in H$ such that $t_1 \leq_n t_3^h$ or

R2. $\exists k \in K$ such that $t_4^k \leq_n t_2$ or

CA* the involved condition involving $\leq_n$ and atoms.

The induction hypothesis gives us easily the equivalence of the first three conditions at levels n and $n+1$. For the condition (CA*) note first that

– $t_2 \leq_n t_1$		– $t_2 \leq_{n+1} t_1$
– for all $h \in H$, $t_3^h \geq_n t_2$		– for all $h \in H$, $t_3^h \geq_{n+1} t_2$
– for all $k \in K$, $t_4^k \leq_n t_1$	are equivalent to	– for all $k \in K$, $t_4^k \leq_{n+1} t_1$
– for all $h \in H$, $t_3^h \not\geq_n t_1$		– for all $h \in H$, $t_3^h \not\geq_{n+1} t_1$
– for all $k \in K$, $t_4^k \not\leq_n t_2$		– for all $k \in K$, $t_4^k \not\leq_{n+1} t_2$

because of the induction hypothesis.

We have to check that the condition (CA*) :

> Let $H_{f,n}$ be the set of $h \in H$ such that $|t_1 \backslash t_3^h|_n$ finite. For every $a_h \in Atom_n$, $a_h \leq_n t_1 \backslash t_3^h$, $h \in H_{f,n}$, there must be a t_4^k such that $t_4^k \backslash t_2 \leq_n \bigvee_{h\in H_{f,n}} a_h$.

is equivalent to the same condition where we replace all the n with $n+1$.

Recall that since all singletons are denoted, atoms are exactly the singleton types. We need a lemma to prove that the condition (CA*) at level n works on exactly the same atoms as at level $n+1$:

Lemma C.11.2. *Suppose that for every $t \in \mathcal{T}_n$*
– *$t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;*
– *$|t|_n = l$ if and only if $|t|_{n+1} = l$.*
Pick $t \in \mathcal{T}_n$ and an atom $a \in \mathcal{T}_{n+1}$. If $a \leq_{n+1} t$ and $|t|_n$ is finite, then there exists an atom $a' \in \mathcal{T}_n$ with $a =_{n+1} a'$.

Proof: suppose $|t|_n = l$ with l finite. Since every singleton is denoted, $t =_n a_1 \vee \ldots \vee a_l$ for disjoint n-atoms a_i. Then the same equality is true at level $n+1$. Since $a \leq_{n+1} t$, then $a \leq_{n+1} a_1 \vee \ldots \vee a_l$ from which we derive that $a =_{n+1} a_i$ for some i. Thus $a' = a_i$ satisfies the required condition. □

We are now going to check the equivalence of the conditions.

Suppose it is true for the $n+1$ case. Then pick a choice of n-atoms a_h, $h \in H_{f,n}$. By the induction hypothesis the a_h are $n+1$-atoms, too. Also, by the induction hypothesis $|t_1 \backslash t_3^h|_{n+1}$ is finite if and only if $|t_1 \backslash t_3^h|_n$ is finite. Thus $H_{f,n} = H_{f,n+1}$. Since (CA*) is true at level $n+1$, then there must be a t_4^k such that $t_4^k \backslash t_2 \leq_{n+1} \bigvee_{h\in H_{f,n+1}} a_h$. Which implies $t_4^k \backslash t_2 \leq_n \bigvee_{h\in H_{f,n}} a_h$.

Conversely suppose it is true for n. Pick a choice of $n+1$-atoms a_h, $h \in H_{f,n+1}$. If one of these a_h is not equivalent to an n-atom, then by Lemma C.11.2,

$|t_1 \backslash t_3^h|_{n+1}$ would be infinite. Thus we can assume that all a_h are n-atoms. As above we have $H_{f,n} = H_{f,n+1}$, and since (CA*) is true at level n, there must be a t_4^k such that $t_4^k \backslash t_2 \leq_n \bigvee_{h \in H_{f,n}} a_h$. Which implies $t_4^k \backslash t_2 \leq_{n+1} \bigvee_{h \in H_{f,n+1}} a_h$.

We have now to prove the condition on the cardinality. We start by observing that all the atoms we have described above (when we proved that every singleton is denoted) are atoms independently of the level. They are atoms because of their shape. We now prove the following

- $|t|_{n+1} = l$ implies $|t|_{n+2} = l$;
- $|t|_{n+1} \geq l$ implies $|t|_{n+2} \geq l$.

from which we can conclude $|t|_{n+1} = l$ if and only if $|t|_{n+2} = l$.

Suppose $|t|_{n+1} = l$. Then $t =_{n+1} a_1 \vee \ldots \vee a_l$ for some disjoint atoms. Thus $t =_{n+2} a_1 \vee \ldots \vee a_l$, and since the a_i are still atoms (and they are still disjoint), $|t|_{n+2} = l$.

Suppose $|t|_{n+1} \geq l$, then $t \geq_{n+1} a_1 \vee \ldots \vee a_l$ for some disjoint atoms. Thus $t \geq_{n+2} a_1 \vee \ldots \vee a_l$, and since the a_i are still atoms (and they are still disjoint), $|t|_{n+2} \geq l$.

□

We finally observe that adding the $\mathbb{k}$ to our types is not restrictive, as $\mathbb{k} =_k ch^k(\mathbf{0})$.

Hinging on Proposition C.11.1, we define preorder between types as follows.

Definition C.11.3 (Order)**.** Let $t, t' \in \mathcal{T}_n$, then $t \leq_\infty t'$ if and only if $t \leq_n t'$.

Due to Proposition C.11.1, this relation is well defined and induces an equivalence $=_\infty$ on the set of types T. Let $\widetilde{\mathcal{T}}$ be $\mathcal{T}/_{=_\infty}$, we are finally able to produce a unique pre-model $\mathcal{D}$ defined as :

$$\mathcal{D} = \mathbb{B} + \widetilde{\mathcal{T}} \ .$$

Where

- $[\![ch^+(t)]\!] = \{[t']_{=_\infty} \mid t' \leq_\infty t\}$;
- $[\![ch\overline{(t)}]\!] = \{[t']_{=_\infty} \mid t \leq_\infty t'\}$.

This pre-model defines a new preorder between types that we denote by $\leq$. However, the following proposition proves that $\leq$ is not new but it is the limit of the previous preorders, i.e. $\leq_\infty$.

Proposition C.11.4. *Let $t, t' \in \mathcal{T}$, then $t \leq t'$ if and only if $t \leq_\infty t'$.*

Proof: We prove it by induction on the height of the types. That is we prove by induction on n that if $t \in \mathcal{T}_n$, then

- $t = \mathbf{0}$ if and only if $t =_\infty \mathbf{0}$;
- $|t| = l$ if and only if $|t|_\infty = l$.

Note that to check emptiness of a type in $\mathcal{T}_{n+1}$ we only invoke types in $\mathcal{T}_n$.

The condition at level 0 only requires that the types $\mathbb{k}$ be interpreted into distinct singletons contained in $\widetilde{\mathcal{T}}$, which is the case.

The second statement, and the whole inductive step are proven as in the proof of Proposition C.11.1. □

It is now easy to show the following.

Theorem C.11.5. *The pre-model* $(\mathcal{D}, [\![\,]\!])$ *is a model.*

Proof: Consider the extensional interpretation $\mathbb{E}(\,)$ of types as in Definition C.2.2. We have to check that $[\![t]\!] = \varnothing \Longleftrightarrow \mathbb{E}(t) = \varnothing$. Note that in fact the range of $\mathbb{E}(\,)$ is $\mathcal{P}(\mathbb{B} + [\![\mathcal{T}]\!])$. By proposition C.11.4, we have that $\langle [\![\mathcal{T}]\!], \subseteq \rangle$ is isomorphic to $\langle \widetilde{\mathcal{T}}, \leq \rangle$. Up to this isomorphism, $\mathbb{E}(\,)$ coincides with $[\![\,]\!]$. □

C.11.3 Proof of decidability of finiteness

Given our model of types, we show that we can

1. decide whether a type is finite
2. if it is the case, list all its atoms

To prove our claim we proceed by induction on the height of the types. We strengthen the statement by requiring that all atoms of a finite type t have the same height, or lower, of t. We assume that at height 0, this is the case. It is a reasonable assumption : for example it is the case if we have for base types the type of all integers plus all constant types. Consider a type t of height $n+1$ and assume that for lower heights we can decide whether a type is finite and, if it is the case, list all its atoms. By Theorem C.3.2, this guarantees that we can also decide emptiness of all types of height $n+1$. We ask ourselves which atoms can be proved to belong to t. If we put t in normal form, we obtain the disjunction of terms of the form

$$r = ch^+(t_1) \wedge c\bar{h}(t_2) \wedge \bigwedge_i \neg ch^+(t_3^i) \wedge \bigwedge_j \neg c\bar{h}(t_4^j) \ .$$

(We exclude base types, because they have been considered at height 0, and "mixed types", which can be reduced to one of the "pure" cases.) Only atoms of the form $ch(s)$, can be contained in non-base types. For how many s we can have that $ch(s) \leq t$? A union is finite if and only if all its summands are, thus t is finite if and only if all the r's are finite. When is r finite ? First of all it is finite when it is empty, which we can test it by induction hypothesis.

Otherwise if r is not empty, then r is finite if and only if $ch^+(t_1) \wedge c\bar{h}(t_2)$ is finite, which happens exactly when $t_2 \leq t_1$ and $t_1 \wedge \neg t_2$ is finite. For the "if" part, note that $ch(s)$ belongs to $ch^+(t_1) \wedge c\bar{h}(t_2)$, if and only if $s = t_2 \vee s'$ for some $s' \leq t_1 \wedge \neg t_2$. Since $t_1 \wedge \neg t_2$ is finite and of smaller height, then by induction hypothesis we can list all its atoms, thus all the corresponding s''s, thus all the corresponding $ch(t_2 \vee s')$ that are all the possible candidates of atoms of r. By induction hypothesis we also have that all the s' have at most height n.

For the "only if" part it suffices to prove that if $ch^+(t_1) \wedge c\bar{h}(t_2)$ is infinite, then the whole of r is infinite. Assume that for no i, $t_1 \leq t_3^i$ and for no j, $t_4^j \leq t_2$ (otherwise r is empty). We have to find infinitely many s such that $t_2 \leq s \leq t_1$, $s \not\leq t_3^i$ for all i and $t_4^j \not\leq s$ for all j. Pick atoms $a_3^i \leq t_1 \wedge \neg t_3^i$ and $a_4^j \leq t_4^j \wedge \neg t_2$. Note that no a_3^i can coincide with any a_4^j, because they are taken from disjoint sets. Then for any type s' such that $t_2 \leq s' \leq t_1$, the type

$s := (s' \vee \bigvee_i a_3^i) \wedge \neg \bigvee_j a_4^j$ belongs to r. It is possible that for two different s' the corresponding s coincide. However such "equivalence classes" of s' are finite. Since there are infinitely many s', there are infinitely many s, so r is infinite.

In summary, for every r that forms t we check whether $t_2 \leq t_1$ and $t_1 \wedge \neg t_2$ is finite, and at the end we find either that t is infinite (if one of the r is) or that it is finite. In the latter case we have a finite list of candidates to be the atoms of t (namely all $ch(s)$ for s included in the the various $t_1 \wedge \neg t_2$) and to list all the atoms of t we just to check for each candidate its inclusion in t. Which we can do, since they are at most of height $n+1$.

C.11.4 Proof of Theorem C.4.5

We first show that $(\mathcal{V}, [\![\,]\!]_\mathcal{V})$ is a pre-model. Inspecting the typing rules, it is easy to show that for every value v and every types t_1, t_2

1. $\Gamma \vdash v : \mathbf{1}$;
2. $\Gamma \vdash v : t_1$ if and only if $\Gamma \nvdash v : \neg t_1$;
3. $\Gamma \vdash v : t_1 \wedge t_2$ if and only if $\Gamma \vdash v : t_1$ and $\Gamma \vdash v : t_2$.

Point (1) is a simple application of the subsumption rule. For (2) suppose that there exists t such that $v : t$ and $v : \neg t$. The only rule to deduce a negative type for a value is the subsumption rule. Therefore there must be a type s, such that $v : s$, $s \leq t$ and $s \leq \neg t$. But then $s = \mathbf{0}$, impossible since the empty type is not inhabited. Suppose instead there exists t such that $\nvdash v : t$ and $\nvdash v : \neg t$; if $v = c^s$ then $ch(s)$ is not smaller than t nor than $\neg t$, impossible since $ch(s)$ is atomic. The same can be deduced from the atomicity of b_n for $v = n$. Therefore $(\mathcal{V}, [\![\,]\!]_\mathcal{V})$ is a pre-model.

By the subsumption rule we have that if $v : s$ and $s \leq t$ then $v : t$. Therefore $s \leq t \Rightarrow [\![s]\!]_\mathcal{V} \subseteq [\![t]\!]_\mathcal{V}$. For the other direction, if $s \not\leq t$, there is an atom a in $s \backslash t$. For every atom a there is a value v such that $\Gamma \vdash v : a$ (this is clearly true for channels, while it was an assumption for basic types). By subsumption $\Gamma \vdash v : s$ and $\Gamma \vdash v : \neg t$, which implies $\Gamma \nvdash v : t$. Thus $[\![s]\!]_\mathcal{V} \not\subseteq [\![t]\!]_\mathcal{V}$.

To prove that it is a model we have to check that $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$. Again the range of $\mathbb{E}(\,)$ is $\mathcal{P}(\mathbb{B} + [\![\mathcal{T}]\!]_\mathcal{V})$. By the observation above, we have that $\langle [\![\mathcal{T}]\!]_\mathcal{V}, \subseteq \rangle$ is isomorphic to $\langle \widetilde{\mathcal{T}}, \leq \rangle$. Up to this isomorphism, $\mathbb{E}(\,)$ coincides with $[\![\,]\!]_\mathcal{V}$. □

C.11.5 Proof of the subject reduction

As usual, the crucial step is the substitution lemma C.4.6. We need to prove

– If $\Gamma, t/p \vdash M' : t'$ and $\Gamma \vdash v : t$, then $\Gamma \vdash M'[v/p] : t'$.
– If $\Gamma, t/p \vdash P$ and $\Gamma \vdash v : t$ then $\Gamma \vdash P[v/p]$.

This is done by induction on the typing rules, by making use of Theorem C.4.4. Then consider a well-typed premise of the reduction rule : $\Gamma \vdash \overline{c^t} v \;\|\; \sum_{i \in I} c^t(p_i).P_i$. This means that $\Gamma \vdash v : t$ and $\Gamma, t/p_i \vdash P_i$. Since $t \leq \bigvee_{i \in I} \wr p_i \int$, there must be a j such that $\vdash v : \wr p_j \int$. For all such j, the sub-

stitution v/p_j is defined. By the substitution lemma, for all such j we have $\Gamma \vdash P_j[v/p_j]$.

C.11.6 Proof of Lemma C.4.9

Take a nonempty type $s \leq ch^+(\mathbf{1})$. This means that its disjunctive normal form contains only channel types. Consider first the case where s is composed of only one clause $s = ch^+(t_1) \wedge ch\bar{}(t_2) \wedge \bigwedge_h \neg ch^+(t_3^h) \wedge \bigwedge_k \neg ch\bar{}(t_4^k)$. Since s is not empty we have

- $t_2 \leq t_1$ and
- $\forall h \in H$, $t_1 \not\leq t_3^h$ and
- $\forall k \in K$, $t_4^k \not\leq t_2$ and
- there exists a choice of atoms $a_h \leq t_1 \backslash t_3^h$ for $h \in H_f$ such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$.

Consider now some type t and the inequation $s \leq ch^+(t)$. This is satisfied if an only if $s \wedge \neg ch^+(t) = \mathbf{0}$. We can think of $ch^+(t)$ as an extra $ch^+(t_3^h)$ added to the normal form of s. In order to have that $s \wedge \neg ch^+(t)$ is empty, we only have two possibilities. The first is that $t_1 \leq t$. Therefore the first candidate for least t is precisely t_1. But can it be smaller than this?

First, note that we must have that $t \geq t_2$, as otherwise we cannot have $s \leq ch^+(t)$. Therefore to obtain a smaller t we must remove some atoms in $t_1 \backslash t_2$. Which ones? Consider all possible choices of atoms $a_h \leq t_1 \backslash t_3^h$ for $h \in H_f$ such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$. As noticed, since s is not empty, there must be at least one such choice.

We claim that none of those a_h can be removed from t_1. To show this, consider a choice of atoms a_h as above with $h \in H_f$ and let $a = a_{\bar{h}}$ for some $\bar{h} \in H_f$. Consider $t = t_1 \setminus a$ and recall we can consider t as one extra t_3^h in the normal form of s. Now we must check condition (CA*) for this new clause. Let $H^\bullet = H \cup \{\bullet\}$, with $t_3^\bullet = t$. Note that $t_1 \setminus t = a$ is finite, and thus $H_f^\bullet = H_f \cup \{\bullet\}$. By putting $a = a_\bullet$, we can see the above choice of atoms as a choice of atoms a_h, with $h \in H_f^\bullet$. Indeed the atom a plays the double role of $a_{\bar{h}}$ and $a_\bullet$.

In order for (CA*) to be satisfied, we should be able to find a t_4^k such that $t_4^k \leq t_2 \vee \bigvee_{h \in H_f^\bullet} a_h = t_2 \vee \bigvee_{h \in H_f} a_h$, which it is not possible by hypothesis. Then, such atoms cannot be removed from t_1.

Now, consider an atom a that is not of this form. Reasoning in similar way as above we can show that we can take a out of t_1 if and only if for all possible choices of atoms $a_h \leq H_f$, such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$, there is $\bar{k}$ such that $t_4^{\bar{k}} \setminus (t_2 \vee \bigvee_{h \in H_f} a_h) = a$.

How many such atoms there are? Only finitely many, as the universal quantification above is finite. Therefore we can remove these atoms one by one. The corresponding t is such that $s \leq ch^+(t)$ and moreover we cannot remove any other atom. Finally all such atoms can be computed.

The above proves the statement for types s composed only of one clause. Consider a type s whose disjunctive normal form is $s = s_1 \vee \ldots \vee s_n$, and suppose for each s_i the type t_i is the least such that $s_i \leq ch^+(t_i)$. Then the type

$t = t_1 \vee \ldots \vee t_n$ is the least such that $s \leq ch^+(t)$. Clearly it has the property. To show it is the least such, remove one atom a from it and suppose it still has the property. Therefore no s_i contains a. However a belongs to one of the t_i. Therefore, by removing a from such t_i we would obtain a smaller t'_i such that $s_i \leq ch^+(t'_i)$, contradiction.

Acknowledgements. This work was conceived in the *lingerie* of the École Normale Supérieure, on the white board near the Lavazza coffee machine. I'm not sure the place exists any longer.

Annexe D

Defining fairness for concurrent reactive systems

D.1 Introduction

A mathematical model of a reactive and concurrent system usually comes with one or more *fairness* assumptions. A fairness assumption usually stipulates that if a particular behaviour of the system is sufficiently often possible during a given run of the system, then that behaviour occurs sufficiently often in that run. Depending on what exactly we mean by "behaviour", "sufficiently often" and "possible", many different fairness notions arise.

For example, a run is said to be *weakly fair* with respect to an action A when the following implication is true : If A is eventually always enabled during the run, then it must be taken infinitely many times. Consider a process with two parallel threads. In principle, a scheduler could only give CPU time to one of the two threads. Such a scheduler is clearly not fair, and indeed it violates weak fairness with respect to the first action of the other thread.

In contrast to other important classes of temporal properties, such as *safety* and *liveness* [AS85, Lam77], no general characterisation of fairness has so far been proposed, i.e., there is no agreed definition of what the class of all fairness properties is. Apt, Francez and Katz [AFK88] gave some criteria that must be met by fairness. Following [Lam00], we think that their most important criterion is that a fairness assumption must be *machine closed*[1] with respect to the safety property defined by the underlying transition system. This, basically, means that fairness is imposed in such a way to the transition system that the system "cannot paint itself into a corner" [AFK88] ; i. e. , whatever the system does in finite time, it is possible to continue in such a way that the fairness assumption is met. (To recall the precise definition, see Section D.2.2.) However, machine closure does not exclude some properties that, we think, should not be

1. *Machine closure* was originally called *feasibility* [AFK88]. The term *"machine closed"* was introduced in [AL91].

considered to be fairness properties. For example, given a system M and an action A of the system, consider the two properties :

F_M : Action A is always eventually taken if it is always eventually enabled.

E_M : Action A is eventually henceforth never taken.

Whereas F_M (called *strong fairness* w.r.t. A in M) is a typical fairness assumption that enforces an action to be taken sufficiently often, E_M rather prevents a particular choice, viz. action A, from being taken sufficiently often. E_M is therefore not a fairness property from our point of view. However, both properties are machine closed with respect to any safety property [2].

Another issue is that fairness should be closed under intersection, i.e., the intersection of finitely many, or better, countably many fairness assumptions should be a fairness assumption. This is because fairness assumptions are usually imposed stepwise and componentwise, e.g., with respect to a particular process, state, or transition. The fairness assumption for the system is then the intersection of all fairness assumptions for its components.

Thus, for a given system M or, more generally, for a given safety property S, we want to define when a temporal property F should be called *a fairness property in* S such that

1. When F is a fairness property in S, then F is machine closed with respect to S.
2. Fairness properties are closed under (countable) intersection.
3. Popular fairness notions from the literature such as *strong fairness* (see Section D.3.3) should correspond to fairness properties in our sense.

Machine closure is not sufficient to guarantee closure under intersection : The intersection of F_M and E_M is the empty set in some systems M, and the empty set is not machine closed w.r.t. any nonempty safety property. Kwiatkowska [Kwi91] proposes a definition of fairness [3] that is closed under countable intersection. However, important popular fairness notions, such as strong fairness, are not covered by her definition, as we will show in Section D.8.

In this work, we propose a definition of when a linear-time temporal property is a *fairness property* with respect to a given system, such that the above requirements are met. We give three characterisations for the family of all fairness properties : a language-theoretic, a game-theoretic and a topological characterisation. The language-theoretic characterisation is a general formalisation of the standard intution : if something is sufficiently often possible, it will happen sufficiently often. In the game-theoretic characterisation, a fairness property is a property that can be guaranteed by a scheduler that gets control over the system infinitely often for a finite amount of time. Finally, it turns out that the fairness properties are the "large" sets from a topological point of view, i.e., they are the *co-meager* sets in the natural topology of runs of a given system.

2. E_M also meets the other criteria proposed in [AFK88].

3. [Kwi91] works on the domain of Mazurkiewicz traces. She defines a fairness property for a system to be a G_δ set of maximal traces that is machine closed w.r.t. the safety property of the system.

The topological insight provides a link to probability theory, where a set is "large" when it has measure 1. While these two notions of largeness are very similar, they do not coincide in general. However, we show that they coincide for ω-regular properties and bounded Borel measures. That is, an ω-regular temporal property of a finite-state system has measure 1 under a bounded Borel measure if and only if it is a fairness property with respect to that system.

The characterisation of fairness directly leads to a generic relaxation of when a system is correct with respect to a linear-time specification. This notion of correctness, which we call *fair correctness*, has again a language-theoretic, a topological and a game-theoretic interpretation. We motivate this notion of correctness and discuss how a system can be shown to be fairly correct.

Structure of the chapter. This chapter is essentially a copy of the paper [VV12]. Section D.2 defines basic preliminary notions that are used throughout the chapter. In Section D.3, we review the most popular fairness notions from the literature. That leads us to derive a first general definition of fairness in Section D.4 using language-theoretic terms. In Section D.5, we provide the first equivalent characterisation of fairness in terms of a two-player game, called the *Banach-Mazur game*. The game-theoretic characterisation allows us to prove some important properties of the class of fairness properties in Section D.6. In Section D.7, we identify the relationship of fairness with the Safety-Progress classification of properties proposed by Manna and Pnueli [MP90], which refines our intuition about which temporal properties are fairness properties. In Section D.8, we give another equivalent characterisation of fairness, now in topological terms. It relates to the topological characterisation of safety and liveness that was proposed by Alpern and Schneider [AS85]. The insight that fairness properties are the properties that are "large" from a topological point of view then leads to a link to probabilistic systems and properties that have measure one. This link is explored in Section D.9. Finally, Section D.10 introduces fair correctness and discusses proof and model checking techniques for its verification.

D.2 Basic notations and definitions

In this chapter, we collect basic preliminary definitions, which include sequential runs, temporal properties and transition systems.

D.2.1 Systems and runs

Let Σ be a nonempty set, whose elements will be thought of as *states*. Σ^* and Σ^ω denote the set of finite and infinite sequences over Σ, respectively. The set of all sequences $\Sigma^* \cup \Sigma^\omega$ is denoted as Σ^∞. We use the symbols α, β for denoting finite sequences, and x, y for arbitrary sequences. The empty sequence is denoted by ϵ. The length of a sequence x is denoted by $|x|$ ($= \omega$ if x is infinite). The set of all sequences of length k is denoted by Σ^k. Concatenation

of sequences is denoted by juxtaposition; $\sqsubseteq$ denotes the usual *prefix order* on sequences. As usual, we write $x \sqsubset y$ when $x \sqsubseteq y$ and $x \neq y$.

Two sequences x, y are *compatible* if $x \sqsubseteq y$ or $y \sqsubseteq x$. Given a set $X \subseteq \Sigma^\infty$, we denote by $\max(X)$ the set of maximal elements of X under the prefix order. By $x\uparrow = \{y \mid x \sqsubseteq y\}$ and $x\downarrow = \{y \mid y \sqsubseteq x\}$ we denote the sets of all *extensions* and *prefixes* of a sequence x, respectively. The least upper bound of a sequence $(\alpha_i)_{i=0,1,\ldots}$ of finite sequences, where $\alpha_i \sqsubseteq \alpha_{i+1}$, is denoted by $\sup_i \alpha_i$. For a sequence $x = s_0, \ldots, s_n, \ldots$ and an index i where $0 \leq i < |x|$ of x, x_i denotes the finite prefix $s_0, \ldots, s_{i-1}$ of x and $x(i)$ denotes the state s_i. If $x \neq \epsilon$ and $0 \leq i < |x|$, then i is called a *position* of x.

An *action* or *transition relation* over Σ is a nonempty relation $A \subseteq \Sigma \times \Sigma$. Action A is *enabled* in a state s if there exists an s' such that $(s, s') \in A$; A is *taken* in a sequence x if there is a position i of x such that $(x(i), x(i+1)) \in A$. A *system* is a tuple $M = (\Sigma, R, \Sigma_0)$, where $R \subseteq \Sigma \times \Sigma$ is a transition relation over Σ and $\Sigma_0 \subseteq \Sigma$ is a set of *initial states*. The system is *finite* if Σ is finite. A *path* or *word* of a system M is a sequence x such that $(x(i-1), x(i)) \in R$ for each $i, 0 < i < |x|$. A *run* of a system M is a path of M such that if it is nonempty, then $s_0 \in \Sigma_0$. The set of all runs of M is denoted by S_M.

Remark: A run is thought of as an observation of global states. The empty run stands for the observation in which the initial state has not yet been observed. We could also develop the theory for sequences of alternating states and transitions or of transitions only. Modulo some technical details, these alternatives would differ very little.

Example D.2.1.

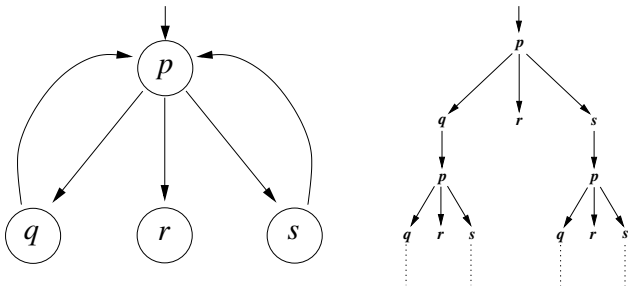


FIGURE D.1 – A four-state system and the set of its runs represented as an infinite tree

Figure D.1 shows a system M over $\Sigma = \{p, q, r, s\}$, where $\Sigma_0 = \{p\}$, $R = \{(p, q), (p, r), (p, s), (q, p), (s, p)\}$. The sequences $x_1 = pq$, $x_2 = (pq)^\omega$, and $x_3 = pr$ are three runs of this system. We have $x_1 \sqsubseteq x_2$ and $x_2, x_3 \in \max(S_M)$. The infinite tree in Fig. D.1, represents S_M, the set of all finite and infinite runs of M.

D.2.2 Temporal properties

A *temporal property* (*property* for short) is a set $E \subseteq \Sigma^\infty$ of sequences of states. We say that some sequence x *satisfies* a property E if $x \in E$; otherwise we say that x *violates* E. We say that E is *finitary* if $E \subseteq \Sigma^*$ and E is *infinitary* if $E \subseteq \Sigma^\omega$. Furthermore : E is *upward-closed* if $x \in E$ and $x \sqsubseteq y$ implies $y \in E$; E is *downward-closed* if $x \in E$ and $y \sqsubseteq x$ implies $y \in E$; E is *complete*, sometimes also called *limit-closed*, if $\alpha_i \in E$ for $i \in \mathbb{N}$ with $\alpha_i \sqsubseteq \alpha_{i+1}$ implies $\sup_i \alpha_i \in E$.

A property S is a *safety property* if for any sequence x violating S, there exists a finite prefix α of x that violates S and each extension of a sequence violating S also violates S, i.e.,

$$\forall x \notin S : \exists \alpha \sqsubseteq x : \alpha\uparrow \cap S = \varnothing.$$

Equivalently, a property is a safety property precisely when it is *downward-closed* and *complete*. We can think of a safety property S as a tree, because $(S \cap \Sigma^*, \sqsubset_1)$ is a tree with root ϵ, where $\alpha \sqsubset_1 \beta$ if there exists a state s such that $\alpha s = \beta$. S corresponds to the set of all (finite and infinite) paths of the tree that start in the root (cf. also Fig. D.1). The set S_M is a safety property for each system M. Therefore, in the following, we will think of a safety property as a generalised system. In particular, we call an element of a safety property also a *run*. The set of all sequences Σ^∞ is also a safety property and can be seen as the set of runs of a "universal" system.

Consider a safety property S and a finite sequence $\alpha \in S$. A property E is *live in α with respect to S* if there exists a sequence x such that $\alpha \sqsubseteq x \in S \cap E$. Intuitively, E is live in a finite run of a system if the system has still a chance to satisfy E in the future. A property E is a *liveness property in S* if E is live in every $\alpha \in S \cap \Sigma^*$. In this situation, we also say that (S, E) is *machine-closed* [AL91, AFK88]. If $S = \Sigma^\infty$, then we simply say that E is a *liveness* property. Hence E is a liveness property if and only if

$$\forall \alpha \in \Sigma^* : \alpha\uparrow \cap E \neq \varnothing.$$

A property is *ω-regular* if it is a property accepted by some Büchi automaton, or, equivalently a property definable in Monadic Second Order logic (see, e.g., [Tho90]).

Example D.2.2. $\Sigma^{\leq k} = \{\alpha \in \Sigma^* \mid |\alpha| \leq k\}$ is a safety property for each $k \in \mathbb{N}$; Σ^* and Σ^ω are examples of liveness properties. Whereas Σ^* is a liveness property with respect to each safety property S, Σ^ω is a liveness property with respect to S only if $\max(S) \subseteq \Sigma^\omega$; $\max(S)$ is always a liveness property with respect to S. Σ^∞ is the only property that is a safety as well as a liveness property.

Remark: A temporal property is often (e.g. [AS85, MP90]) defined to be a subset of Σ^ω. It is then argued that finite runs can be mimicked by infinite ones by repeating the last state infinitely often. This often leads to a simplification of

the presentation. The difference between the two approaches is not really essential. We will indicate when notable differences between the two approaches arise. Including finite runs, as we do here, gives rise to a more natural generalisation to other domains such as non-sequential runs.

D.2.3 Linear time temporal logic

Some temporal properties can be expressed by formulas of a *linear-time* temporal logic such as LTL (see [MP92, Eme90]).

Let Σ be a set of states, AP a countable set of *atomic propositions*, and $v : AP \to \mathcal{P}\Sigma$ a mapping that assigns each atomic proposition the set of states at which it is *satisfied*. We assume here that v is given implicitly and hence we will also refer to an atomic proposition as a *state property*. In particular, we will use in many examples, a state s as an atomic proposition that is satisfied at s and only at s.

The formulae ϕ of the logic *LTL* are defined as follows :

$$\begin{array}{rcl} \phi, \psi & := & \\ & | & \Phi \text{ where } \Phi \text{ is a state property} \\ & | & \top, \neg\phi, \phi \wedge \psi \\ & | & \bigcirc \phi, \phi \,\mathrm{U}\, \psi \end{array}$$

The temporal operators are pronounced as follows : $\bigcirc \phi$ as "next time" ϕ and $\phi \,\mathrm{U}\, \psi$ as ϕ "until" ψ.

Satisfaction is defined as follows (cf. [Eme90, CES86, KP95]). Let x be a sequence, and i a position of x. We define :

1. $x, i \models \Phi$ if $x(i) \in v(\Phi)$,
2. $x, i \models \top$ always ; $x, i \models \neg p$ if $x, i \not\models p$; $x, i \models p \wedge q$ if $x, i \models p$ and $x, i \models q$,
3. $x, i \models \bigcirc \phi$ if $i + 1$ is a position of x and $x, i + 1 \models \phi$,
4. $x, i \models \phi \,\mathrm{U}\, \psi$ if there exists a position $j \geq i$ of x such that $x, j \models \psi$ and for each k, we have $i \leq k < j \Rightarrow x, k \models \phi$.

Then for a system M, we say that ϕ is *satisfied* in M, denoted $M \models \phi$, if for all $x \in S_M$ we have $x, 0 \models \phi$. Furthermore, we define $\mathrm{sat}(\phi) = \{x \in \Sigma^\infty \mid x, 0 \models \phi\}$. A temporal property E is said to be *LTL-expressible* if there exists an LTL formula ϕ such that $E = \mathrm{sat}(\phi)$. It is well known that all LTL-expressible properties are ω-regular [Eme90, Tho90].

Additional boolean operators can be defined as usual. Additional temporal operators are defined as abbreviations as follows.

$\Diamond \phi = \top \,\mathrm{U}\, \phi$ ("eventually" ϕ),

$\Box \phi = \neg \Diamond \neg \phi$ ("always" ϕ).

RLTL (restricted LTL) refers to the subset of those LTL formulas that do not contain the operators $\bigcirc$ and U, but may contain $\Diamond$ and $\Box$.

Example D.2.3. With respect to the system M depicted in Fig. D.1, $\Box(\bigcirc q \Rightarrow p)$ denotes a safety property that is satisfied in M, whereas $\Box(p \Rightarrow \bigcirc q)$ denotes

a safety property that is not satisfied in M; $\Diamond p$ is a liveness property that is satisfied in M, whereas $\Diamond q$ is a liveness property that is not satisfied in M; $\Diamond q$ is live in $\alpha = ps$, but not in pr. $\Diamond r$ is a liveness property in S_M, i.e., machine-closed w.r.t. S_M, whereas $\Box \Diamond p$ is not, because it is not live in pr.

D.3 Survey of fairness notions

The distinction of safety and liveness properties in the specification and verification of reactive systems is also reflected in the operational model of a reactive system : Some sort of state machine or transition system defines the set of all possible runs of the system, which is a safety property. To guarantee something to happen at all and to guarantee that some particular choices will eventually be made, there is an additional liveness property. That liveness property is usually called the *fairness assumption* of the reactive system.

Fairness usually means that a particular choice is taken sufficiently often provided that it is sufficiently often possible [AFK88]. Depending on the interpretation of "choice", "sufficiently often", and "possible", many different fairness notions arise (cf., e.g., [LPS81, Fra86, Kwi89]).

Before we propose a general definition of fairness, we review the most popular and some additional notable fairness assumptions that are used in the literature. To do so, we will represent some example systems as Petri nets, assuming the reader to be familiar with the basic firing rule of a Petri net (see, e.g., [Rei85, Mur89]).

The fairness properties we discuss are actually parametric on the systems at hand. For instance, for any given system and any given transition, there is a corresponding "strong fairness" property. Therefore rather than talking of fairness *properties*, we more precisely present fairness *notions*, that can be defined as maps from systems, transitions, and other parameters, to actual properties.

D.3.1 Sequential maximality

Consider the system in Fig. D.2, represented as a Petri net. As such, the system specification only says what can and what cannot happen, i.e., its semantics is the set of all its sequential runs. It does not say that something must happen at all. To say that something must happen, we can use the *maximality assumption*, which says that the system does not arbitrarily stop the computation.

A run x is *maximal w.r.t. an action* A if it is infinite or if its final state does not enable A. Hence, given a system $M = (\Sigma, R, \Sigma_0)$ and a partition of R into actions $A_1, \ldots, A_n$, a run of M is maximal iff it is maximal w.r.t. to each A_i, $i = 1, \ldots, n$. In the system considered, this means that after every a, there must be a b and that after every b, there must be an a. This leaves only the run $(ab)^\omega$, which is the unique maximal run of the system. Therefore, the system satisfies the property "infinitely often a" under the maximality assumption.

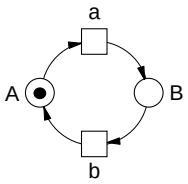


FIGURE D.2 – A simple process

D.3.2 Weak fairness

Consider now the system in Fig. D.3 and assume maximality. Then, that system does not satisfy "infinitely often a" because the maximal run $(cd)^\omega$ does not. Although the overall system does not stop in this run, one of its components does. To rule out such a behaviour, we assume *weak fairness*, also known as *justice* [LPS81].

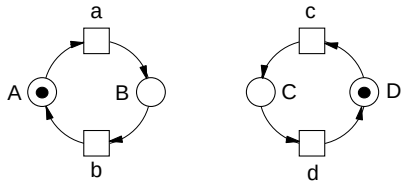


FIGURE D.3 – Two independent processes

A run x is *weakly fair* w.r.t. an action A if A is taken infinitely often or A is always eventually disabled, i.e., for each position i of x there exists a position $j \geq i$ such that A is not enabled in $x(j)$. Therefore, the maximal run $(cd)^\omega$ is not weakly fair with respect to a. The system does in fact satisfy "infinitely often a" under weak fairness with respect to a and b. Weak fairness w.r.t. an action A is obviously strictly stronger than maximality w.r.t. A.

D.3.3 Strong fairness

In the next system, see Fig. D.4, weak fairness is not sufficient to establish "infinitely often a" because the run $(cd)^\omega$ is weakly fair with respect to all transitions of the system. In particular, it is weakly fair with respect to a because a is always eventually disabled.

However, we can consider $(cd)^\omega$ unfair with respect to a because a is infinitely often enabled but never taken. This kind of unfairness is captured by the notion of *strong fairness*, which is also known as *compassion* [LPS81].

A run is *strongly fair* w.r.t. an action A if A is taken infinitely often in x or A is eventually henceforth never enabled in x, i.e., there is a position i of x such that A is not enabled in $x(j)$ for each position $j \geq i$. Strong fairness with

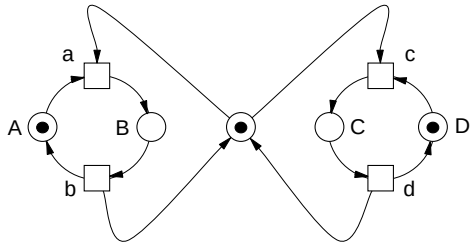


FIGURE D.4 – Two processes sharing a resource

respect to a together with maximality with respect to b and d then establish "infinitely often a" in the system. Strong fairness is obviously strictly stronger than weak fairness.

D.3.4 k-fairness

In the next system, see Fig. D.5, strong fairness with respect to all transitions fails to establish "infinitely often e", because the run $(abcd)^\omega$ violates it but is strongly fair w.r.t. all transitions. In particular, it is strongly fair with respect to e because e is never enabled in that run.

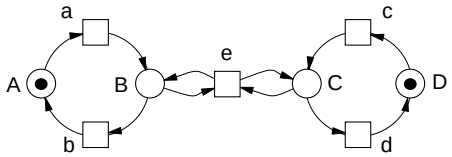


FIGURE D.5 – Two processes sharing an action

Among the fairness notions that establish "infinitely often e", there is the notion of *strong k-fairness* [Bes84] for $k \geq 1$, cf. also *hyperfairness* as discussed by Lamport [Lam00].

Let $M = (\Sigma, R, \Sigma_0)$ be a system. An action A is *k-enabled* in a state s of a system M, $k \in \mathbb{N}$ if there exists a (finite nonempty) word $w = s_0, \ldots, s_n$ of M such that $s_0 = s$, $n \leq k$ and A is enabled in s_n. A run x is *strongly k-fair* w.r.t. action A if A is infinitely often taken in x or A is eventually henceforth never k-enabled, i.e., there is a position i of x such that A is not k-enabled in $x(j)$ for each position $j \geq i$.

Weak fairness for all transitions together with strong 1-fairness for e indeed establish "infinitely often e". Strong $(k+1)$-fairness is clearly stronger than strong k-fairness, and strong 0-fairness coincides with strong fairness.

Remark: The "unfairness" arising in the system in Fig. D.5 is also known from the variant of the Dining Philosophers, in which a philosopher picks up

both his forks at the same time to eat. There, a philosopher may starve because his two neighbours "conspire" against him by eating alternately in such a way that his two forks are never available at the same time. Note that transition e in Fig. D.5 needs two resources (B and C) at the same time. There are more complex fairness notions that better capture the "unfairness" in this example [AFG93, Vol02, Vol05].

D.3.5 ∞-fairness

Consider now the infinite-state system in Fig. D.6. Suppose we are interested here in the property "state 0 is visited infinitely often". This property is not established by strong k-fairness for any k because the diverging run $a_1 a_2 \ldots$ is strongly k-fair with respect to any transition for any $k \geq 0$. However, we can use the stronger notion of *strong ∞-fairness* [Bes84].

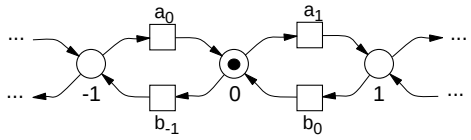


FIGURE D.6 – A nondeterministic walk on the integer line

An action A is *∞-enabled* in a state s of a system M if there exists a word $w = s_0, \ldots, s_n$ of M such that $s_0 = s$, and A is enabled in s_n. A run x is *strongly ∞-fair* w.r.t. action A if A is infinitely often taken in x or A is eventually henceforth never ∞-enabled, i.e., there is a position i of x such that A is not ∞-enabled in $x(j)$ for each position $j \geq i$.

It is easy to see that ∞-fairness w.r.t. a_0 and b_0 establishes the above specification.

D.3.6 Word fairness

While strong ∞-fairness with respect to transitions is very strong, there are still some useful specifications that are not established by it. As an example, consider the system in Fig. D.7 and the specification "the finite word *aba* occurs infinitely often". The run $(abcd)^\omega$ does not satisfy the specification but it is strongly ∞-fair w.r.t. every transition, because every transition is taken infinitely often in this run. In such a case, we can extend the above fairness notions and define them w.r.t. finite words of transitions rather than with respect to a single transition only.

Let M be a system. We say that a word w of M is *enabled* in a state s of M if sw is also a word of M. We say that w is *taken* in x at position i if $x_i w \sqsubseteq x$. A run x of M is *strongly fair* w.r.t. w if w is taken infinitely often in x or w is eventually henceforth never enabled, i.e., there exists a position i of x such that $x(j)$ does not enable w for each position $j \geq i$. A run x of M is *word fair* if it

is strongly fair w.r.t. every finite word w of M ; x is *state fair* (*transition fair*) if it is strongly fair w.r.t. every word of length 1 (length 2) of M.

Clearly, strong fairness w.r.t. the word *aba* establishes the specification considered above.

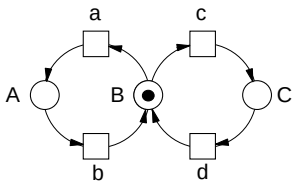


FIGURE D.7 – A recurrent free choice

D.3.7 Other notions of fairness

Another remarkable notion is *equifairness* [Fra86]. A run x is *equifair* w.r.t. a pair (A_1, A_2) of actions if it is strongly fair w.r.t. both and the following holds : If x has infinitely many positions i such that both A_1 and A_2 are enabled in $x(i)$, then x has infinitely many positions j such that A_1 is taken k times in x_j implies that A_2 is taken k times in x_j.

Equifairness with respect to a and c in Fig. D.7 prescribes that each fair run has infinitely many positions such that the number of previous occurrences of a equals the number of previous occurrences of c.

Remark: In the literature, we usually find the additional intuitive assumption that for each state s of M, A_1 is enabled in s iff A_2 is enabled in s.

Other fairness notions found in the literature include those that were developed for the verification of randomised systems, e.g., *extreme fairness* [Pnu83] and α-*fairness* [LPZ85]. There are many more fairness notions in the literature, which we cannot all mention here. Overviews can be found elsewhere [Fra86, Kwi89, AFK88, Jou01, Lam00].

D.4 A language-theoretic characterisation of fairness

All the examples of fairness notions that we have presented in Section D.3 require that some finite behaviour must be taken "sufficiently" often, provided that it is "possible". The most general notion of a finite behaviour is a finitary property $Q \subseteq \Sigma^*$, where Q "is taken" in a finite prefix α of a sequence if $\alpha \in Q$.

The weakest form of "Q is possible" that we encountered is "there exists an extension into Q". The strongest form of "sufficiently often" that we encountered is "infinitely often". We therefore obtain the following generalisation of ∞-fairness as the strongest fairness notion with respect to some finitary property Q :

Definition D.4.1 (∞-Fairness w.r.t. Q)**.** Let S be a safety and Q a finitary property. A run x is ∞-*fair* with respect to Q if
- there are infinitely many $i \in \mathbb{N}$ such that $x_i \in Q$ or
- Q is not *strictly live* w.r.t. S in some finite prefix α of x, i.e., there is no finite run β such that $\alpha \sqsubset \beta \in Q \cap S$.

The set of ∞-fair runs in S w.r.t. Q is denoted as $\mathrm{F}_S(Q)$.

Thus, Definition D.4.1 says that a run x is ∞-fair w.r.t. Q if each finite prefix of x that has a proper extension within S into Q is properly extended along x into Q. This implies that any finite run in $\max(S)$ is ∞-fair because it has no proper extension.

Example D.4.1. If Q is the set of all finite sequences that end with an occurrence of a given action A, i.e., $Q = \{\alpha s s' \mid \alpha \in \Sigma^*, (s, s') \in A\}$, then $\mathrm{F}_S(Q)$ is exactly strong ∞-fairness with respect to A as introduced in Section D.3.5. This is easily generalised to ∞-fairness with respect to a word.

Definition D.4.1 presents the strongest form of fairness we consider with respect to some finitary property Q. Any weaker form of fairness, such as strong and weak fairness, can be obtained by weakening. We thus define that a property is a fairness property if it *contains* all ∞-fair runs with respect to some Q.

Definition D.4.2 (Fairness)**.** Let S be a safety property. A temporal property E is a *fairness property in* S w.r.t. some finitary property Q if $\mathrm{F}_S(Q) \subseteq E$. We say that E is a *fairness property* in S if there exists a Q such that $\mathrm{F}_S(Q) \subseteq E$.

Example D.4.2. Any property weaker than ∞-fairness (such as strong k-fairness etc.) is a fairness property according to Definition D.4.2. Furthermore, let $Q = \{\alpha \mid A_1 \text{ is taken exactly } k \text{ times in } \alpha \text{ implies } A_2 \text{ is taken exactly } k \text{ times in } \alpha\}$. Then $\mathrm{F}_{S_M}(Q)$ is a subset of equifairness in M w.r.t. (A_1, A_2). Hence equifairness in S_M is a fairness property in S_M. Therefore all fairness notions introduced in Section D.3 generate fairness properties with respect to a given system.

The class of ∞-fairness properties w.r.t. some Q (Definition D.4.1) is, in general, not closed under weakening and thus Definition D.4.2 is not redundant. To show this, a simple cardinality argument suffices : the class of ∞-fairness properties on a finite system has at most the cardinality of the continuum c (as there are at most c finitary properties), whereas the class of fairness properties has in general cardinality 2^c. Indeed, consider the complete graph over three states $\{p, q, r\}$, and consider the set P to be ∞-fairness with respect to the set of all finite runs that end with p. P is the set of runs with infinitely many occurrences of p. The complement of P is the set of runs with finitely many occurrences of p and has cardinality c. Therefore there are 2^c distinct supersets of P.

More interestingly, we can show that important fairness notions, such as strong fairness, are not covered without weakening, i.e., by Definition D.4.1 :

Proposition D.4.3. *Consider the complete system over two states $S = \{p, q\}^\infty$. Let F denote strong fairness w.r.t. action $A = \{(p, p)\}$. There is no finitary property Q such that $F = \mathrm{F}_S(Q)$. Let F' denote F intersected with maximality w.r.t. to all other transitions. There is no Q such that $F' = \mathrm{F}_S(Q)$.*

Proof: Deferred to Section D.8.7.

D.5 A game-theoretic characterisation of fairness

The language-theoretic characterisation can be used to prove that a given property E is indeed a fairness property : We display a finitary property Q such that $\mathrm{F}_S(Q) \subseteq E$. However, the language-theoretic characterisation does not give us a useful tool for proving that a given property is not a fairness property. The game-theoretic characterisation of fairness, presented in this section, will give us such a tool. In particular, this characterisation implies that a fairness assumption can never prevent a particular finite behaviour from occurring sufficiently often and thus allows us to prove that the property E_M in Section D.1 is not a fairness property. The game-theoretic characterisation also turns out to be useful for proving some properties of our definition of fairness.

D.5.1 The Banach-Mazur game

Let S be a safety property, and E any property. The game $G(S, E)$ is played by two players called *Alter* and *Ego*. The state of a play is a finite run of S. At every move, one player extends the current run by a finite, possibly empty, sequence. Alter has the first move. The two players move alternately. The play goes on forever, converging to a finite run α or an infinite run x in S. Ego wins if $x \in E$ (resp. $\alpha\uparrow \cap S \subseteq E$), otherwise Alter wins.

Definition D.5.1 (Banach-Mazur game)**.** Let S be a safety property, and E any temporal property. A *play* of the game $G(S, E)$ is an infinite sequence of finite runs $(\alpha_i)_{i\in\mathbb{N}}$ of S such that $\alpha_0 = \epsilon$ and $\alpha_i \sqsubseteq \alpha_{i+1}$ for each $i \geq 0$. Given a play $(\alpha_i)_{i\in\mathbb{N}}$, we say that *Ego wins* if $(\sup_i \alpha_i)\uparrow \cap S \subseteq E$. Otherwise *Alter wins.*

A *partial play* is a finite prefix $(\alpha_i)_{i\leq n}$ of some play. The set of partial plays on S is denoted as PL_S. A *strategy* in S is a mapping $f : PL_S \to \Sigma^* \cap S$ such that if $\alpha = f(\alpha_0, \ldots, \alpha_i)$, then $\alpha_i \sqsubseteq \alpha$. We say that *Ego plays* f in a play $(\alpha_i)_{i\in\mathbb{N}}$ if for every $j \geq 0$, $f(\alpha_0, \ldots, \alpha_{2j+1}) = \alpha_{2j+2}$. Likewise, we say that *Alter plays* f in the play $(\alpha_i)_{i\in\mathbb{N}}$ if for every $j \geq 0$, $f(\alpha_0, \ldots, \alpha_{2j}) = \alpha_{2j+1}$. A strategy f is *winning* for Ego (Alter) in $G(S, E)$ if Ego (Alter) wins each play where he (she) plays f.

The game $G(S, E)$, henceforth called the *Banach-Mazur game*[4], defines an intermediate version of nondeterminism between *daemonic* and *angelic* nondeterminism.

4. The naming will be explained in Section D.8.6.

Suppose that the correctness of a system is specified by a temporal property E. Saying that nondeterminism is *daemonic* means that we define a system M to be correct if each run of M satisfies E. This corresponds to a game in which Alter chooses an arbitrary, possibly infinite run in S—Ego, who wants to prove correctness, is guaranteed to win only if every run of the system S is in E. Saying that nondeterminism is *angelic* means that we define a system M to be correct if there is a run of M that satisfies E. This corresponds to a game in which Ego chooses the entire run. Then, Ego has a winning strategy only if $S \cap E \neq \varnothing$. In the Banach-Mazur game, the nondeterminism is resolved alternately between Alter and Ego. In the following, we will prove that Ego has a winning strategy in the Banach-Mazur game $G(S, E)$ if and only if E is a fairness property for S.

The definition of strategy we have given is the most general one : players play remembering the full history of the play. As long as we are interested only in knowing whether there is a winning strategy for one of the players, we can use a simpler notion of strategy that does not decrease the power of the players. In this simplified version, players only know the current state of the play, but not the moves that led to it.

Definition D.5.2. A strategy f in S is *decomposition invariant* [Gra08] if for any $\alpha_0, \ldots, \alpha_i$ and $\beta_0, \ldots, \beta_j$, we have that $\alpha_i = \beta_j$ implies $f(\alpha_0, \ldots, \alpha_i) = f(\beta_0, \ldots, \beta_j)$.

We denote a decomposition-invariant strategy as a mapping $f : \Sigma^* \cap S \to \Sigma^* \cap S$ such that $\alpha \sqsubseteq f(\alpha)$ for all $\alpha \in \Sigma^* \cap S$.

Proposition D.5.3 ([Gra08])**.** *Given a Banach-Mazur game $G(S, E)$, Ego (Alter) has a winning strategy if and only if he (she) has a decomposition-invariant winning strategy.*

Proof: Grädel [Gra08] gives a proof for a slightly different setting using topological arguments that we will introduce later. We could reproduce his proof for our setting using arguments from below. Instead we provide an alternative, more direct proof here.

Given a winning strategy f, we want to define a decomposition-invariant winning strategy g for the same game. To this end, we need to define $g(\alpha)$ for any finite run α. To apply f, we first need to decompose α into some partial play $\alpha_0, \ldots, \alpha_i$. We do this for Ego, the construction for Alter being essentially the same.

Let $\alpha_0 = \epsilon$. Define α_1 to be the smallest prefix of α such that $f(\alpha_0, \alpha_1)$ is compatible with α. Let $\alpha_2 = f(\alpha_0, \alpha_1)$. Recursively, define α_{2j+1} to be the smallest prefix of α such that $f(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{2j+1})$ is compatible with α. Define $\alpha_{2j+2} = f(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{2j+1})$. Let k be the the first index for which $\alpha \sqsubseteq \alpha_{2k}$. Define $g(\alpha) = \alpha_{2k}$. To prove that g is a winning strategy, consider now a play $(\beta_i)_{i \in \mathbb{N}}$, where Ego plays g. This means that each β_{2j+1} is decomposed into $(\alpha_0^j, \ldots, \alpha_{2h_j+1}^j)$ and $\beta_{2j+2} = g(\beta_{2j+1}) = f(\alpha_0^j, \ldots, \alpha_{2h_j+1}^j)$. We claim that all such decompositions are "compatible" in the sense that for each $j < k$, $(\alpha_0^j, \ldots, \alpha_{2h_j+1}^j)$ is a prefix of $(\alpha_0^k, \ldots, \alpha_{2h_k+1}^k)$. It is sufficient to show this for $k = j + 1$.

Let thus $(\alpha_0, \ldots, \alpha_{2l+1})$ be a decomposition of β_{2j+1} and $(\alpha'_0, \ldots, \alpha'_{2m+1})$ be a decomposition of β_{2j+3}. We show that for $i < 2l+1$, $\alpha_i = \alpha'_i$ by induction on i. By definition $\alpha_0 = \alpha'_0 = \epsilon$. α_1 is defined as the smallest prefix such that $f(\alpha_0, \alpha_1)$ is compatible with β_{2j+1}.

If $f(\alpha_0, \alpha_1)$ is a strict prefix of β_{2j+1}, then $f(\alpha_0, \alpha_1)$ is compatible with β_{2j+3}. Moreover α_1 is the shortest prefix such that $f(\alpha_0, \alpha_1)$ is compatible with β_{2j+3}. In fact, if there were a shorter one, call it $\hat{\alpha}$, then $f(\alpha_0, \hat{\alpha})$ would be also compatible with β_{2j+1}, thus contradicting the definiton of α_1 . Thus $\alpha_1 = \alpha'_1$.

If $\beta_{2j+1} \sqsubseteq f(\alpha_0, \alpha_1)$, then $\beta_{2j+2} = g(\beta_{2j+1}) = f(\alpha_0, \alpha_1)$. Thus $f(\alpha_0, \alpha_1)$ is compatible with β_{2j+3}, and again $\alpha_1 = \alpha'_1$. As $\alpha_0 = \alpha'_0$ and $\alpha_1 = \alpha'_1$, $\alpha_2 = f(\alpha_0, \alpha_1) = f(\alpha'_0, \alpha'_1) = \alpha'_2$. The inductive step is analogous, for both even and odd indices.

The limit of all the decompositions is a play $(\alpha_h)_{h \in \mathbb{N}}$ where Ego plays f, and therefore he wins. But $\sup_i \beta_i = \sup_j \beta_{2j+1} = \sup_j \alpha^j_{2h_j+1} = \sup_h \alpha_h$, and this shows that g is also a winning strategy.

In the following, we will only consider decomposition-invariant strategies, but for brevity not mention it explicitly. We will now give a simplified characterisation of the games in which Ego has a winning strategy. To this end, we will use the following definition :

Definition D.5.4. We consider the game $G(S, E)$ and a strategy f in S. A run $x \in S$ is *f-compliant* (for Ego) if x is the result of a play in which Ego plays f. The set of all f-compliant runs is denoted by R_f.

Hence, a strategy f is winning for Ego iff $R_f \subseteq E$. We provide an alternative, easier characterisation of compliant runs :

Proposition D.5.5. *Let f be a strategy in S. A run $x \in S$ is f-compliant iff for each position i of x there exists a position $j \geq i$ of x such that $f(x_j) \sqsubseteq x$.*

Proof: Trivially f-compliant runs satisfy the condition. Consider now a run x that satisfies the condition. In other words, for every prefix α of x, there exists a prefix β, such that $\alpha \sqsubseteq \beta$ and $f(\beta) \sqsubseteq x$. Also if $\alpha \neq x$, we can take β to be a proper extension of α. Let $h(\alpha)$ be one of such β, say, the shortest. Consider the play $(\alpha_n)_{n \in \mathbb{N}}$ defined as follows :

- $\alpha_{2h+1} = h(\alpha_{2h})$,
- $\alpha_{2h+2} = f(\alpha_{2h+1})$.

Clearly, the limit of the play is x, and this shows that x is f-compliant.

A second, simpler characterisation for infinite runs is the following :

Proposition D.5.6. *Let $x \in S$ be an infinite run and f a strategy. Then x is f-compliant iff for infinitely many i, x_i belongs to the image of f.*

Proof: As above, f-compliant runs satisfy the condition trivially. Consider now a run x that has infinitely many prefixes belonging to the image of f. We show that x satisfies the condition of Proposition D.5.5. For any position i, consider the set $H = \{f(x_h) \mid h \leq i\}$. Take j to be the smallest position such that $f(x_j) \sqsubseteq x$ and $f(x_j) \notin H$. Such j must exists because H is finite, and clearly $j > i$.

We now arrive at the final simplification of the notion of strategy.

Definition D.5.7. A strategy f is *progressive* if $f(\alpha) = \alpha \Rightarrow \alpha \in \max(S)$

Note that a strategy f is progressive if and only if $R_f \subseteq \max(S)$. As Alter can always enforce the result of the play to be a maximal run of S, it is not restrictive to require that Ego plays progressively. More precisely :

Lemma D.5.8. *Ego has a winning strategy in $G(S, E)$ iff he has a progressive winning strategy in $G(S, E \cap \max(S))$.*

Proof: Let f be a winning strategy for Ego in $G(S, E)$. Define f' to be a strategy such that $f'(\alpha) = f(\alpha)$ if $f(\alpha) \in \max(S)$ and otherwise $f'(\alpha) = f(\alpha)s$ for some state s such that $f(\alpha)s \in S$; f' is clearly progressive. Now consider a run $x \in R_{f'}$. If it is finite, it is maximal and then $x \in R_f$. If it is infinite, there are infinitely many indices j such that $f'(x_j) \sqsubseteq x$. But since $f'(\alpha) \sqsubseteq f(\alpha)$, for the same indices, $f(x_j) \sqsubseteq x$, and $x \in R_f$.

Thus we have $R_{f'} \subseteq R_f$, and hence if f is winning for Ego, f' must also be winning for Ego. The converse is trivial. □

D.5.2 The game-theoretic characterisation of fairness

We are now ready to state and prove the main result of the section :

Theorem D.5.9. *Ego has a winning strategy in $G(S, E)$ iff E is a fairness property for S.*

Proof: ($\Leftarrow$) Suppose $\mathrm{F}_S(Q) \subseteq E$. Define a strategy f by $f(\alpha) = \beta$ where $\alpha \sqsubset \beta \in Q \cap S$ if there exists such a β and $f(\alpha) = \alpha$ otherwise. We have $R_f \subseteq \mathrm{F}_S(Q) \subseteq E$, hence f is a winning strategy for Ego in $G(S, E)$.

($\Rightarrow$) Let f be a progressive winning strategy for Ego in $G(S, E)$. Define $Q = f(\Sigma^*)$. Let $x \in \mathrm{F}_S(Q)$. We want to show that $x \in E$. If $x = \alpha$ is a finite maximal run of S, then $f(\alpha) = \alpha$ and $\alpha \in R_f$. By definition of a progressive strategy, any other finite run has a proper extension in Q and thus it cannot be in $\mathrm{F}_S(Q)$. If x is infinite, since any finite prefix $\alpha \sqsubseteq x$ can be properly extended to Q, there must be infinitely many i such that $x_i \in Q$. By Proposition D.5.6, we obtain $x \in R_f$. From $R_f \subseteq E$ follows $x \in E$. □

The intuition behind this game-theoretic characterisation of fairness is that, while fairness restricts the allowed behaviour, it should not restrict it too much. Ego, who wants to produce a fair run, can enforce some (live) choice to be taken infinitely often but he cannot prevent other choices being taken infinitely often (by Alter).

Example D.5.1. Consider $S = \Sigma^\infty$ for this paragraph. Then Σ^ω is a fairness property in S, whereas Σ^* is a liveness but not a fairness property in S because Alter can enforce the outcome of the play to be infinite. Similarly, for any sequence x, the property $\{\alpha x \mid \alpha \in \Sigma^*\}$ is a liveness property but not a fairness

property in S. The property $\Box \Diamond \Phi$ is a fairness property whereas $\Diamond \Box \Phi$ is a liveness but not a fairness property—for any *non-trivial* state property Φ, i.e., $\varnothing \neq \mathrm{sat}(\Phi) \neq \Sigma$. Hence the property E_M in Section D.1 is not a fairness property in general.

More examples for fairness properties in $S = \Sigma^\infty$ are $\Box(\Phi \Rightarrow \Diamond \Psi)$, $\Diamond \Box \Phi \Rightarrow \Box \Diamond \Psi$, and $\Box \Diamond \Phi \Rightarrow \Box \Diamond \Psi$.

Call a run *periodic* if it is of the form $\alpha\beta^\omega$ for $\alpha, \beta \in \Sigma^*$ and *aperiodic* otherwise. The set of aperiodic runs is a fairness property, whereas the set of periodic runs is a liveness but not a fairness property ; f defined by $f(\alpha) = \alpha s^k r$, where $k = |\alpha|, s, r \in \Sigma, s \neq r$ is a winning strategy for Alter w.r.t. aperiodic runs.

It is clear that at most one of the two players has a winning strategy in a given Banach-Mazur game. In the above examples, we have shown that Ego does not have a winning strategy by showing that Alter has a winning strategy. A property E such that either Ego or Alter has a winning strategy in $G(S, E)$ is called *determinate* w.r.t. S. Hence, for a given determinate property E, we have a complete proof strategy for showing whether E is a fairness property or not—either we display a winning strategy for Ego or one for Alter.

The family of determinate properties is quite large. It includes all *Borel properties* [Oxt71, Thms. 4.3 and 6.3], which will be defined and explained later in Section D.8.3. It can be argued that we are usually not interested in properties that are not determinate. In fact, indeterminate properties do exist, but to prove their existence, one needs to invoke the axiom of choice [Oxt71, Ch. 6]. For our purposes, it shall suffice to state that each ω-regular property is a Borel property (see [Tho90]) and hence determinate.

D.6 Properties of fairness properties

In Section D.1, we discussed three requirements for a general definition of fairness. We required popular notions of fairness to be covered by our definition. Furthermore, each fairness property should be machined-closed w.r.t. the system and the class of all fairness properties should be closed under countable intersection. By help of the game-theoretic characterisation, we can now prove that the latter two requirements are met by our definition of fairness.

Proposition D.6.1. *Let S be a safety property and F a fairness property in S. Then (S, F) is machine-closed.*

Proof: Let $\alpha \in S \cap \Sigma^*$. We have to show that there exists a run x such that $\alpha \sqsubseteq x \in S \cap F$, which is clear because Ego has a winning strategy in the game $G(S, F)$, in particular for those plays in which Alter starts with move α.

Proposition D.6.2. *Let S be a safety property and F_i fairness properties in S for each $i \in \mathbb{N}$. Then $\bigcap_i F_i$ is a fairness property in S.*

Proof: Let f_i be a progressive winning strategy for Ego in $G(S, F_i)$ for each $i \in \mathbb{N}$. Define for $\alpha \in \Sigma^k$, $f(\alpha) = f_k(f_{k-1}(\ldots f_0(\alpha)\ldots))$. It is straight-forward

to verify that any f-compliant run is f_i-compliant for every $i \in \mathbb{N}$, and therefre f is a winning strategy for Ego w.r.t. $\bigcap_{i\in\mathbb{N}} F_i$. □

Example D.6.1. Let F_k, $k \in \mathbb{N}$ be strong k-fairness w.r.t. some action of a system as defined in Sect. D.3.4 and let $F = \bigcap_{k\in\mathbb{N}} F_k$. F is a fairness property that in general is clearly strictly stronger than any F_k but strictly weaker than strong ∞-fairness as was shown in Sect. D.3.5.

Proposition D.6.2 cannot be generalised to arbitrary intersections : for any $x \in S = \{p, q\}^\infty$, let $F_x = S\backslash\{x\}$. Clearly F_x is a fairness property, but $\bigcap_{x\in S} F_x$ is empty. An interesting subclass of fairness properties that forms a complete lattice is discussed in [VVK05].

We have already argued in Sect. D.5.2 that the third requirement—that most popular fairness notions produce fairness properties—is also satisfied by our definition. While checking several other fairness notions in the literature, we either proved that the notion falls into the class we have defined or violates already the first requirement (machine closure). An example of this is *unconditional fairness*, which prescribes a particular action to be taken infinitely often regardless of the choices the system provides. The literature contains an extensive discussion why those properties should not be considered fairness properties (cf. [AFK88, Lam00]). Another such example is *bounded fairness*, which requires that an action has to be taken within some fixed time after it has been enabled. More precisely, a run x is *k-bounded-fair* for $k \in \mathbb{N}$ if the following is true for each position i of x : If A is enabled at i then A is taken at some position j of x such that $i \leq j \leq i + k$. It is easy to see that k-bounded fairness is a safety property and therefore not a liveness property in general in a given system.

The notion of *finitary fairness* [AH94] is an exception in our survey of 'fairness' notions found in the literature : A run x is *finitary-fair* w.r.t. an action A if there exists a $k \in \mathbb{N}$ such that x is k-bounded-fair w.r.t. A. Finitary fairness is, in contrast to k-bounded fairness, a liveness property (i.e., machine-closed) in a given system. However, it is not a fairness property in general. For instance consider $S = \{p, q\}^\infty$, and finitary fairness w.r.t. the transition (p, q). A winning strategy for Alter is defined by $f(\alpha) = \alpha s^k$, where $k = |\alpha|$ and $s \in \{p, q\}$ is the last state of α. Note that for the system in Fig. D.6, no run is finitary-fair w.r.t. all transitions, i.e., the intersection of the properties finitary fairness w.r.t. a_i and finitary fairness w.r.t. b_i for $i \in \mathbb{N}$ is empty.

However, given an action A, consider the following property $F_k(A)$: If A is infinitely often enabled, then A must be taken infinitely often within k steps. It is not difficult to verify that $F_k(A)$ is a fairness property.

This leads to the question whether we could have defined fairness in a different way and still satisfy our requirements. In particular : could have we defined fairness in a more liberal way, i.e., as a larger family of properties ? Again, thanks to the game-theoretic characterisation, we can answer negatively to this question. Indeed we will now show that we cannot enlarge the class of fairness properties without giving up the first or the second requirement. We do not have a proof for this statement in full generality, but we can prove it if we restrict

it to determinate properties. As argued above, this can be considered as a mild restriction.

Theorem D.6.3. *Let S be a safety property. The family $\mathcal{F}_S = \{F \mid F$ is a fairness property in $S\}$ is a maximal family of determinate properties w.r.t. S such that*

1. *$F \in \mathcal{F}_S$ implies (S, F) is machine-closed and*
2. *$F, F' \in \mathcal{F}_S$ implies $F \cap F' \in \mathcal{F}_S$.*

Proof: Suppose by contradiction that $\mathcal{F}_S$ is not maximal, and consider a strictly larger familiy $\mathcal{F}'$ of determinate properties that satisfies (1) and (2). Take $E \in \mathcal{F}' \setminus \mathcal{F}_S$. Since $E \notin \mathcal{F}_S$, Ego has no winning strategy in $G(S, E)$, but since E is determinate, Alter has a winning strategy f in that game. Let $\alpha = f(\epsilon)$ be its first move. Consider the property

$$F = \neg E \cup \bigcup_{\beta \text{ is incompatible with } \alpha} \beta\uparrow,$$

where $\neg E$ denotes $S \setminus E$.

Ego has a winning strategy in the game $G(S, F)$, defined as follows : If the play is in a state that is compatible with α, then Ego can use f to guarantee $\neg E$. Otherwise, the play is in a state β that is incompatible with α, in which Ego does not have to do anything to win. Thus $F \in \mathcal{F}_S \subseteq \mathcal{F}'$. Since $\mathcal{F}'$ satisfies (2), we have that $E \cap F \in \mathcal{F}'$. But by definition of F, α has no extension into $S \cap E \cap F$. Hence $(S, E \cap F)$ is not machine-closed, contradicting (1).

D.7 Fairness and the dafety-progress hierarchy

With the *safety-progress classification*, Manna and Pnueli [MP90] gave a classification of temporal properties that is in some sense orthogonal to the safety-liveness classification. Whereas safety and liveness define a partition of all temporal properties (safety properties, liveness properties, and those that are neither safety nor liveness), the members of the safety-progress classification form a hierarchy. Manna and Pnueli presented a language-theoretic, a topological, a temporal-logical and an automata-theoretic view of their hierarchy. In this subsection, we study the relationship of fairness properties with the safety-progress hierarchy in the language-theoretic view. In this way, we complement our insights into which properties are fairness properties.

D.7.1 The safety-progress hierarchy

Manna and Pnueli [MP90] define four operators that construct temporal properties from finitary properties. While they consider only infinitary properties as temporal properties, we generalise their operators here to our setting in

a natural way. Let Q be a finitary property. Define

$$\mathrm{A}(Q) = \{x \mid \forall \alpha \sqsubseteq x : \alpha \in Q\} \tag{D.1}$$
$$\mathrm{E}(Q) = \{x \mid \exists \alpha \sqsubseteq x : \alpha \in Q\} \tag{D.2}$$
$$\mathrm{R}(Q) = \{x \mid \forall \alpha \sqsubseteq x : \exists \beta : \alpha \sqsubseteq \beta \sqsubseteq x \wedge \beta \in Q\} \tag{D.3}$$
$$\mathrm{P}(Q) = \{x \mid \exists \alpha \sqsubseteq x : \forall \beta : \alpha \sqsubseteq \beta \sqsubseteq x \Rightarrow \beta \in Q\}. \tag{D.4}$$

Properties of the form $\mathrm{A}(Q)$ are exactly the safety properties. Properties of the form $\mathrm{E}(Q)$, $\mathrm{R}(Q)$, and $\mathrm{P}(Q)$ are called *guarantee*, *recurrence*, and *persistence properties*, respectively.

We have the following dualities :

$$\neg\mathrm{A}(Q) = \mathrm{E}(\neg Q) \text{ and } \neg\mathrm{E}(Q) = \mathrm{A}(\neg Q) \tag{D.5}$$
$$\neg\mathrm{R}(Q) = \mathrm{P}(\neg Q) \text{ and } \neg\mathrm{P}(Q) = \mathrm{R}(\neg Q), \tag{D.6}$$

where $\neg X$ denotes the complement of X w.r.t. the appropriate universe. As $\mathrm{A}(Q) = \mathrm{R}(\mathrm{A}(Q) \cap \Sigma^*)$ and $\mathrm{E}(Q) = \mathrm{R}(\mathrm{E}(Q) \cap \Sigma^*)$, we have that each safety property and each guarantee property is also a recurrence property. Similarly, each safety and each guarantee property is also a persistence property.

Example D.7.1. Let Φ be a state property. Then $\mathrm{sat}(\Box\,\Phi)$, $\mathrm{sat}(\Diamond\,\Phi)$, $\mathrm{sat}(\Box \Diamond \Phi)$ and
$\mathrm{sat}(\Diamond \Box\, \Phi)$ are examples of safety, guarantee, recurrence and persistence properties respectively.

D.7.2 Liveness and the safety-progress hierarchy

We know already that Σ^∞ is the only property that is a safety as well as a liveness property. More generally, a safety property is a liveness property relative to another safety property S if and only if it contains S. The following proposition also clarifies when a guarantee, recurrence or persistence property is a liveness property (relative to a given safety property S).

Proposition D.7.1. *Let Q be a finitary property and S a safety property.*

1. $\mathrm{A}(Q)$ *is a liveness property in S iff* $\Sigma^* \cap S \subseteq Q$.
2. $\mathrm{E}(Q)$ *is a liveness property in S iff Q is a* pseudo-liveness property in S, *where E is a* pseudo-liveness property in S *if for each $\alpha \in \Sigma^* \cap S$ there exists an $x \in E \cap S$ that is compatible with α.*
3. $\mathrm{R}(Q)$ *is a liveness property in S iff Q is a liveness property in S.*
4. $\mathrm{P}(Q)$ *is a liveness property in S iff Q is a liveness property in S.*

Proof: All claims are proved by a straight forward application of the definitions. □

D.7.3 Fairness and the safety-progress hierarchy

Which properties in the safety-progress hierarchy are fairness properties? We know already that a property is a fairness property in S only if it is a liveness property in S. We now show that each live guarantee as well as each live recurrence property is a fairness property. To do this, we use special classes of strategies.

Definition D.7.2. Let S be a safety property, and f be a strategy in S. We say that f is *idempotent* if $f(f(\alpha)) = f(\alpha)$ for all $\alpha \in S \cup \Sigma^*$. We say that f is *stable* if $f(\alpha) \sqsubseteq \beta \Rightarrow f(\beta) = \beta$ for all $\alpha, \beta \in S \cup \Sigma^*$.

Clearly, each stable strategy is idempotent.

Proposition D.7.3. *Let S be a safety property.*

1. *A subset of S is a live guarantee property if and only if it is the set of f-compliant runs of some stable strategy f in S.*
2. *A subset of S is a live recurrence property if and only if it is the set of f-compliant runs of some idempotent strategy f in S.*

Proof: 1. To prove one direction, let Q be a finitary pseudo-liveness property (cf. Proposition D.7.1), define f such that $f(\alpha) = \alpha$ if $\alpha \in \mathrm{E}(Q)$ and let otherwise $f(\alpha)$ be any extension of α into Q; f is then stable and $R_f = \mathrm{E}(Q)$. For the other direction, given a stable strategy f, we have that $f(\Sigma^*)$ is a pseudo-liveness (it actually is a liveness property) and $R_f = \mathrm{E}(f(\Sigma^*))$. 2. For a given finitary liveness property Q, define f such that $f(\alpha) = \alpha$ if $\alpha \in Q$ and let otherwise $f(\alpha)$ be any extension of α into Q; f is then idempotent and $R_f = \mathrm{R}(Q)$. For a given idempotent f, $f(\Sigma^*)$ is a liveness property and $R_f = \mathrm{R}(f(\Sigma^*))$. □

It follows from Proposition D.7.3 that each property that contains a live guarantee property or a live recurrence property is a fairness property. We show now that live persistence properties are not fairness properties in general.

Proposition D.7.4. *Let S be a safety property. A persistence property P is a fairness property in S if and only if there exists a guarantee property $\mathrm{E}(Q)$ that is live in S such that $\mathrm{E}(Q) \cap S \subseteq P$.*

Proof: ($\Leftarrow$) is clear from Proposition D.7.3. For ($\Rightarrow$), let f be a winning strategy for $\mathrm{P}(Q)$ in S and let $\alpha \in \Sigma^*$. Let $\alpha_0 = \alpha$ and let α_{i+1}, for each $i \in \mathbb{N}$, be any extension of $f(\alpha_i)$ into $S \cap \neg Q$, provided that such an extension exists. If such an extension exists for each $i \in \mathbb{N}$, we obtain a run $x = \sup_{i \in \mathbb{N}} \alpha_i$ where $x \in \mathrm{R}(\neg Q)$ and hence $x \notin \mathrm{P}(Q)$. However, $x \in R_f$, which contradicts f being a winning strategy for $\mathrm{P}(Q)$. Therefore, there is an $i \in \mathbb{N}$ such that $\beta_\alpha := f(\alpha_i)$ has no extension into $S \cap \neg Q$, hence all extensions of β_α in S satisfy Q. Define $Q' = \{\beta_\alpha \mid \alpha \in S \cap \Sigma^*\}$; Q' is clearly live in S. Furthermore, we have $\mathrm{E}(Q') \cap S \subseteq \mathrm{P}(Q)$.

We have shown in Proposition D.7.3 that each property that contains a live recurrence property is a fairness property. The converse does not hold. (As a

counterexample consider $S = \Sigma^\infty$ and $X = \Sigma^\omega$. X contains no recurrence property because it contains no finite runs.) However, we can give a characterisation of fairness in terms of recurrence properties.

Proposition D.7.5. *A property F is a fairness property in S iff there exists a finitary property Q that is live in S such that* $\mathrm{R}(Q) \cap \max(S) \subseteq F$.

Proof: ($\Leftarrow$) is due to Lemma D.5.8. For ($\Rightarrow$), let f be a winning strategy for Ego in $G(S, F)$. Let $Q = f(S \cap \Sigma^*)$. We have $\mathrm{R}(Q) \cap \max(S) \subseteq R_f \subseteq F$.

D.8 A topological characterisation of fairness

This section presents a topological characterisation of fairness. It turns out that fairness properties are exactly the sets that are *large* in a topological sense, i.e., they are the *co-meager* sets in the natural topology of runs of the system. This insight will provide us with an important link to probability theory, where a set is large if it has measure 1. We will explore that link in Section D.9.

Furthermore, the topological characterisation liberates us from our concrete choice of semantical domain, i.e., sequences of states, as other semantic domains are equipped with a natural topology, which then immediately provides a notion of fairness.

Of course, the topological characterisation potentially gives us access to a large body of results in topology. This is even more interesting as other important concepts such as safety, liveness, guarantee, persistence and recurrence were given topological characterisations.

Last but not least, the topological characterisation will provide us additional confidence that we have found a natural definition of fairness.

In Section D.8.1, we will briefly introduce the use of topology in our context, which is based on [Smy92]. In Section D.8.2, we recall the observation, made by Alpern and Schneider [AS85], that safety properties correspond to *closed* sets, whereas liveness properties correspond to *dense* sets. In Section D.8.3, we discuss some important topological notions that constitute the *Borel Hierarchy*. In Section D.8.4, we discuss the topological notion of "largeness". In Section D.8.5, we introduce the Scott topology on systems. Finally, in Section D.8.6, we show that fairness properties correspond to large sets in the Scott topology. For a classic introduction to topology, see [Dug66].

D.8.1 Observable properties

Given an observer of a system who can see the entire state and its evolution, what temporal properties are observable ? By this we shall mean that the observer can detect the presence of the property in finite time. Therefore, a temporal property E is *observable* iff

$$x \in E \Rightarrow \exists \alpha \sqsubseteq x : \alpha\uparrow \subseteq E.$$

The observation of the finite prefix α *guarantees* that the observed run satisfies E. In fact, it is easy to see that a property is observable if and only if it is a guarantee property $\mathrm{E}(Q)$ as defined in Section D.7.1. We may think of the observer as having a (possibly infinite) set Q of finite runs, which she uses to detect the property $E = \mathrm{E}(Q)$. Note that the operator $\mathrm{E}(\cdot)$ defines a bijection between observable and finitary properties.

Note that we only require the presence of an observable property to be detectable, but not its absence. In fact, for no non-trivial property E (i.e., $\varnothing \neq E \neq \Sigma^\infty$) it is the case that E and $\neg E$ are both observable. (Then, the empty run would belong to one of the two, which implies that both are trivial.)

Observable properties have two important closure properties :

1. The intersection of finitely many observable properties is observable.
2. The union of arbitrarily many observable properties is observable.

To see this, note that an observable property can be decomposed into *cones*, i.e., properties of the form $\alpha\uparrow, \alpha \in \Sigma^*$:

$$\mathrm{E}(Q) = \bigcup_{\alpha \in Q} \alpha\uparrow.$$

Closure under union follows immediately. Closure under finite intersection follows from the fact that

$$\alpha\uparrow \cap \beta\uparrow = \sup(\alpha, \beta)\uparrow$$

if α and β are compatible and $\alpha\uparrow \cap \beta\uparrow = \varnothing$ otherwise.

These two closure properties say that the family of observable properties is a *topology* on Σ^∞, i.e., a *topology* on a nonempty set Ω is defined to be a family $\mathcal{T} \subseteq \mathcal{P}\Omega$ that is closed under union and finite intersection. This implicitly requires $\Omega, \varnothing \in \mathcal{T}$. The pair $(\Omega, \mathcal{T})$ is called a *topological space*. A member of $\mathcal{T}$ is called an *open set*. Hence observable properties are the open sets of our topology, which is called the *Scott topology* on Σ^∞.

A *base* of a topology $\mathcal{T}$ is a family $\mathcal{B} \subseteq \mathcal{T}$ such that each open set is the union of members of $\mathcal{B}$. Hence the family

$$\mathcal{B} = \{\alpha\uparrow \mid \alpha \in \Sigma^*\}$$

is a base of the Scott topology on Σ^∞. A member of $\mathcal{B}$ is called a *basic open set*.

A set X is called a *neighbourhood* of x if $x \in G \subseteq X$ for some open set G. Hence a neighbourhood of a run x is a property E such that some finite observation of it guarantees E.

D.8.2 Safety and liveness properties

As noted in Section D.7.1, the complement of a guarantee property is a safety property. The complement of an open set of a topology is called a *closed* set. Thus safety properties are the closed sets of the Scott topology. By duality, safety properties are closed under arbitrary intersection and finite union. The

closure of a set X, denoted $\overline{X}$, is the smallest closed set that contains X. For the Scott topology, we will call $\overline{E}$ the *safety closure* of the property E. We have

$$\overline{E} = \mathrm{A}(\{\alpha \mid \alpha \text{ is a finite prefix of some } x \in E\}).$$

For a safety property, the absence of the property is detectable. The safety closure of a set E can be viewed as the smallest tree that contains all runs of E.

Liveness properties are exactly the *dense* sets of the Scott topology : Recall that a set X is *dense* if it intersects every nonempty open set, or equivalently, every nonempty basic open set, i.e., a property E is dense in the Scott topology iff for each finite run α,

$$E \cap \alpha{\uparrow} \neq \varnothing,$$

i.e., precisely iff E is a liveness property. Equivalently, X is dense iff $\overline{X} = \Omega$, i.e., E is a liveness property iff $\overline{E} = \Sigma^\infty$.

For a property E, we define its *liveness extension* $\mathrm{L}(E)$ by

$$\mathrm{L}(E) = E \cup \bigcup_{\alpha \in \Sigma^*, \alpha{\uparrow} \cap E = \varnothing} \alpha{\uparrow}.$$

Clearly, $\mathrm{L}(E)$ is a liveness property for any E.

The correspondence of safety and liveness to closed and dense sets was pointed out in [AS85]. As each set can be written as the intersection of a closed and a dense set, each temporal property can be written as the intersection of a safety and a liveness property. We have, for any temporal property E,

$$E = \overline{E} \cap \mathrm{L}(E).$$

D.8.3 The Borel hierarchy

Guarantee properties are not closed under countable intersection. For example, we have

$$\bigcap_{k \in \mathbb{N}} \mathrm{E}(\Sigma^k) = \Sigma^\omega,$$

and Σ^ω is clearly not a guarantee property. By duality, safety properties are not closed under countable union. A set that can be written as the intersection of countably many open sets is called a G_δ *set*; a set that can be written as the union of countably many closed sets is called an F_σ *set*. More generally, let G denote the family of open sets and F the family of closed sets and define for a family $\mathcal{F} \subseteq \mathcal{P}\Omega$,

$$\mathcal{F}_\delta = \Big\{ \bigcap_{i \in \mathbb{N}} X_i \mid X_i \in \mathcal{F} \text{ for all } i \Big\}$$

and

$$\mathcal{F}_\sigma = \Big\{ \bigcup_{i \in \mathbb{N}} X_i \mid X_i \in \mathcal{F} \text{ for all } i \Big\}.$$

Thus, for example $\mathrm{G}_{\delta\sigma}$ denotes the family of all sets that can be written as the countable union of G_δ sets. The families defined in that way form an infinite

hierarchy, called the *Borel hierarchy*. The union of all families of the Borel hierarchy is called *Borel σ-field* (w.r.t. the Scott topology on Σ^∞). It is the smallest family of sets that contains the open sets and is closed under complementation and countable union. A member of the Borel σ-field is called a *Borel set*.

We recall that ω-regular properties belong to both $G_{\delta\sigma}$ and $F_{\sigma\delta}$ [Tho90]. LTL-expressible properties belong to the same level of the Borel hierarchy [MP90].

We will mainly be interested in G_δ sets. Above, the property Σ^ω was shown to be a G_δ property. To see more examples, define for $k \in \mathbb{N} \cup \{\omega\}$ and a finitary property Q :

$$R^k(Q) = \{x \mid |\{i \mid x_i \in Q\}| \geq k\}.$$

Clearly, for $k \in \mathbb{N}$, $R^k(Q)$ is a guarantee property. For $k = \omega$, we observe

$$R^\omega(Q) = \bigcap_{k \in \mathbb{N}} R^k(Q).$$

Thus, $R^\omega(Q)$ is a G_δ set. Also, each guarantee property is a G_δ set, and hence the union $R^\omega(Q) \cup \mathrm{E}(Q')$ is a G_δ set.

D.8.4 Topologically large sets

As announced, we will prove that fairness properties w.r.t. a safety property S are the "large sets" in a topological sense. This means that *most* runs of S are fair.

In a topological space, we say that a set E is *somewhere dense* if there exists an open set G such that E intersects every nonempty open subset of G. A set is *nowhere dense* if it is not somewhere dense, or equivalently, if its closure does not contain any nonempty open set—or again equivalently, if its complement contains a dense open set [Dug66].

For an intuition on nowhere dense sets, imagine D to be a set of "dirty" points. If D is a dense set, then it pollutes the whole topological space : wherever we go in the topological space, we will have some dirty point in the neighbourhood. If D is a somewhere dense set, then it pollutes part of the space. There are regions in which you will be always near a dirty point, but possibly also clean neighbourhoods. Finally, if D is nowhere dense, then every clean point lives in a clean neighbourhood. Intuitively, a nowhere dense set is small because the remainder of the topological space can stay clear of it. In this geometric sense, it can be thought of as a set that is full of holes.

A set is *meager* (or of *first category*), if it is the countable union of nowhere dense sets. Topologically, a countable union of small sets is still small. This was observed by René-Louis Baire, who proved that the unit interval of the real line cannot be obtained as the countable union of nowhere dense sets. This result can be thought of as a generalisation of Cantor's theorem, which states that the unit interval is not obtained as the countable union of points [Oxt71].

The complement of a "small" set is therefore to be thought of as "large". The complement of a meager set is called *co-meager* (or *residual*).

In some topological spaces, co-meager sets can be equivalently characterised through G_δ sets. A topological space is called a *Baire space* if the intersection of countably many dense open sets is dense. In a Baire space, a set is a dense G_δ set if and only it can be written as the intersection of countably many dense open sets. Therefore, co-meager sets can equivalently be characterised as follows :

Proposition D.8.1. *In a Baire space, a set is co-meager if and only if it contains a dense* G_δ *set.*

Proof: Straight-forward, see [Oxt71].

D.8.5 The natural topology of a system

If $\mathcal{T}$ is a topology on Ω and $Z \subseteq \Omega$ is a nonempty set, then the family $\mathcal{T}_Z = \{X \cap Z \mid X \in \mathcal{T}\}$ is also a topology, called the *relative topology* w.r.t. $\mathcal{T}$ and Z or $\mathcal{T}$ *relativised* to Z. Therefore each system, or more generally each safety property S has a natural topology, viz. the Scott topology relativised to S, which is thus the family

$$\mathcal{T}_S = \{\mathrm{E}(Q) \cap S \mid Q \subseteq \Sigma^*\}.$$

The family $\mathcal{B} = \{\alpha\uparrow \cap S \mid \alpha \in \Sigma^*\}$ is a base for $\mathcal{T}_S$ and likewise we have : E is a member of a particular class of the Borel hierarchy generated by the Scott topology if and only if $E \cap S$ is a member of the same class of the Borel hierarchy generated by the Scott topology relativised to S.

A set $E \subseteq S$ is dense in the relative topology iff it is live w.r.t. S, i.e., if (S, E) is machine-closed. However, if E is a liveness property (i.e., a dense set), then $E \cap S$ is not necessarily live w.r.t. S (a dense set in the relative topology)—and the converse is also not true in general : Consider $\Sigma = \{p, q\}$, $S = p^\omega\downarrow$ and $E = \mathrm{sat}(\Diamond q)$. E is a liveness property but not live w.r.t. S. Furthermore, $S = S \cap S$ is live w.r.t. S but not a liveness property.

Proposition D.8.2. *The Scott topology on* Σ^∞ *(relativised to some safety property* S*) is a Baire space.*

Proof: Let $E = \bigcap_{i\in\mathbb{N}} \mathrm{E}(Q_i) \cap S$ such that Q_i is dense (i.e., live) in S and let $\alpha \in S \cap \Sigma^*$. As Q_0 is live in S, there exists α_0 such that $\alpha \sqsubseteq \alpha_0 \in S \cap Q_0$. Likewise, since Q_1 is live in S, there exists α_1 such that $\alpha_0 \sqsubseteq \alpha_1 \in S \cap Q_1$ and hence $\alpha_1 \in S \cap \mathrm{E}(Q_0) \cap \mathrm{E}(Q_1)$. Doing this for all $i \in \mathbb{N}$, we obtain $x = \sup_i \alpha_i \in S \cap \bigcap_{i\in\mathbb{N}} \mathrm{E}(Q_i)$. Therefore E is dense (i.e., live) in S.

It is essential that S be a safety property. The Scott topology relativised to Σ^* is not a Baire space. $\Sigma^{\geq k} = \{x \mid |x| \geq k\}$ is a dense open set for each k. The intersection is the empty set, which is not dense relative to Σ^*.

D.8.6 The topological characterisation of fairness

In this section, we show that fairness properties are precisely the co-meager sets. First we give a topological characterisation of ∞-fairness w.r.t. some Q.

Theorem D.8.3. *Given a safety property S and a property F, we have $F = \mathrm{F}_S(Q)$ for some finitary property Q (cf. Definition D.4.1) if and only if F is a dense G_δ relative to S.*

Proof: ($\Rightarrow$) Suppose $F = \mathrm{F}_S(Q)$ for some Q. We know it is dense; it remains to show that it is a G_δ set. The property $\mathrm{F}_S(Q)$ is defined to be the union of $R^\omega(Q)$, which was shown above to be a G_δ set, and an open set : the set of all runs that cannot be extended into Q. An open set is also a G_δ set, and the union of two G_δ sets is a G_δ set.

($\Leftarrow$) If F is a dense G_δ set, it can be written as the intersection of dense open sets, i.e., $F = \bigcap_{i\in\mathbb{N}} \mathrm{E}(Q_i) \cap S$, where each $\mathrm{E}(Q_i)$ is dense in S. Now, let $\min X$ denote the set of minimal elements of a set X in the prefix order. Without loss of generality, we can suppose the following :

1. $Q_i = \min \mathrm{E}(Q_i)$. This is because $\mathrm{E}(Q_i) = \mathrm{E}(\min \mathrm{E}(Q_i))$ (the open sets are the upward closure of their minimal elements).
2. $i < j \Rightarrow \mathrm{E}(Q_i) \supseteq \mathrm{E}(Q_j)$. This is because the finite intersection of open sets is an open set, and thus if for some $i < j$, we had $\mathrm{E}(Q_i) \not\supseteq \mathrm{E}(Q_j)$, we could take $Q'_j = \min \mathrm{E}(Q_i) \cap \mathrm{E}(Q_j)$.
3. $Q_0 = \{\epsilon\}$ so that $\mathrm{E}(Q_0) = \Sigma^\omega$.

Define $Q = \bigcup_{i\in\mathbb{N}} Q_i$. We claim that $F = \mathrm{F}_S(Q)$. We prove the double inclusion.

First let $x \in F$. Therefore $x \in \mathrm{E}(Q_i)$ for each $i \in \mathbb{N}$. By assumption 1, for each i, there is exactly one element of Q_i that is a prefix of x. Either there are infinitely many such prefixes, and thus $x \in R^\omega(Q)$, or there are finitely many such prefixes. In the latter case, the longest such prefix, call it α (there is at least one by assumption 3), cannot be strictly extended into Q. To show this, let h be such that $\alpha \in Q_h$. Suppose that there is an α' that is strictly longer than α such that $\alpha' \in Q$. Let k be such that $\alpha' \in Q_k$. By assumption 1, $\alpha \notin Q_k$ and by assumption 2, $k > h$ and $\mathrm{E}(Q_h) \supset \mathrm{E}(Q_k)$, where the inclusion is strict. As $F = \bigcap_{i\in\mathbb{N}} \mathrm{E}(Q_i) \cap S$, there must be an element $\alpha'' \in Q_k$ such that α'' is a prefix of x. But as $\alpha \notin Q_k$ and $\mathrm{E}(Q_h) \supset \mathrm{E}(Q_k)$, α'' must be strictly longer than α, contradicting the fact that α is the longest prefix of x that is in Q. This shows that x has a prefix that cannot be extended into Q, and thus $x \in \mathrm{F}_S(Q)$.

For the other inclusion, let $x \in \mathrm{F}_S(Q)$. Suppose first $x \in R^\omega(Q)$. Therefore for infinitely many i, $x \in \mathrm{E}(Q_i)$, and because of assumption 2, $x \in \mathrm{E}(Q_i)$ for all i.

Suppose now x has a prefix α that does not have a strict extension into Q. Because F is dense, there is an extension x' of α into F. As $x' \in \mathrm{E}(Q_i)$ for each i, there is an $\alpha_i \in Q_i$ that is a prefix of x'. As α_i is compatible with α and α has no strict extension into Q, we have $\alpha_i \sqsubseteq \alpha$ for each $i \in \mathbb{N}$. Hence $\alpha_i \sqsubseteq x$, and therefore $x \in \mathrm{E}(Q_i)$ and $x \in F$.[5]

Theorem D.8.4. *A property F is a fairness property for S if and only if $F \cap S$ is co-meager in the Scott topology relativised to S.*

5. A similar argument was used by Landweber [Lan69] to characterise the G_δ subsets of $\{0,1\}^\omega$.

Proof: By definition, F is a fairness property if and only if it contains some ∞-fairness property w.r.t. some Q. By Theorem D.8.3, this is equivalent to containing a dense G_δ. And by Propositions D.8.1 and D.8.2, this is equivalent to being co-meager.

Remark: The Banach-Mazur game first appeared in the *Scottish Book* [Mau81]—a notebook with an interesting history and contributions from now well-known mathematicians such as S. Banach, S. Mazur, S. Ulam, H. Steinhaus and others. Problem No. 43 in that book was posed by Stanisław Mazur. Given a nonempty set E of real numbers, Alter and Ego play the following game. Alter chooses a nonempty interval d_1, then Ego chooses a nonempty subinterval d_2 of d_1, then Alter chooses a subinterval d_3 of d_2, and so on. Alter wins if the intersection of the intervals $d_i, i \in \mathbb{N}$, intersects E, otherwise Ego wins. Mazur observed that Alter has a winning strategy if E is co-meager in some interval and Ego has a winning strategy if E is meager, and asked whether these conditions are also necessary for the existence of a winning strategy. This was affirmed by Stefan Banach, but no proof was ever published.

Oxtoby [Oxt57] proved this theorem in a more general setting, for a generalisation of the game in any complete metric space. Thus, essentially, the equivalence between co-meagerness and existence of a winning strategy for Ego was known. For an outline of the history and variants of the Banach-Mazur game, we refer to the survey [Tel87]. The Banach-Mazur game was recently studied as a special case of a *path game* in [BGK03] and [PV03].

Remark: If we restrict ourselves to infinitary temporal properties $E \subseteq \Sigma^\omega$, then the natural topology is the Scott topology relativised to Σ^ω, which is known as the *Cantor topology*. The Cantor topology is metrisable; a standard metric over Σ^ω is defined by

$$d(x, y) = \frac{1}{2^n},$$

where n is the length of the longest common prefix of x and y. The Cantor topology has therefore better separation properties than the Scott topology, which does not meet the separation axiom T_0 (cf. [Dug66, Smy92]).

The fairness properties within Σ^ω are also the co-meager sets in a given safety property, and the Banach-Mazur game is also essentially the same because of Lemma D.5.8. Recurrence properties and G_δ sets coincide in that setting [MP90].

D.8.7 On the necessity of weakening

Kwiatkowska [Kwi91] had proposed to define a fairness property for a system to be a dense G_δ set. However, strong fairness with respect to a particular transition is in general not a dense G_δ set, as we will now prove. By Theorem D.8.3, this also proves then Proposition D.4.3 above, for which we postponed the proof.

Proposition D.8.5. *Consider the complete system over two states $S = \{p, q\}^\infty$. Let F denote strong fairness w.r.t. action $A = \{(p, p)\}$ and let F' denote F intersected with maximality w.r.t. to all other transitions. Then F and F' are not G_δ sets.*

Proof: Without maximality, the proof is easy : it is enough to observe that strong fairness by itself is not upward closed, whereas each G_δ set is. However,maximality is a basic implicit assumption for many authors, including [Kwi91].

Suppose, by contradiction, that F' is a G_δ set. Then there exist open sets $G_i, i \in \mathbb{N}$, such that $F' = \bigcap_{i\in\mathbb{N}} G_i$. Observe that we have, by definition of F',

$$pq^\omega \in F'$$

and hence $pq^\omega \in G_0$. As G_0 is open, there exists a $k_0 > 0$ such that

$$pq^{k_0}\uparrow \subseteq G_0.$$

Furthermore, we have

$$pq^{k_0}pq^\omega \in F',$$

and hence $pq^{k_0}pq^\omega \in G_0 \cap G_1$. As G_1 is open, there exists a $k_1 > 0$ such that

$$pq^{k_0}pq^{k_1}\uparrow \subseteq G_1,$$

and hence $pq^{k_0}pq^{k_1}\uparrow \subseteq G_0 \cap G_1$. We repeat the operation for all natural numbers, and obtain a sequence of finite runs that converges to an infinite run

$$x = pq^{k_0}pq^{k_1}pq^{k_2}\ldots$$

that belongs to the intersection of all G_i, and hence is contained in F'. However, x has infinitely many p's, and therefore infinitely many positions in which t is enabled, but t is never taken in x. Therefore x is not strongly fair, that is, it is not in F' – a contradiction.

Interestingly, a similar argument was used by Landweber [Lan69] (Lemma 3.1) to show that the set of sequences of $\{0,1\}^\omega$ that contain finitely many 1's is not a G_δ set of the Cantor topology over $\{0,1\}^\omega$.

D.9 Fairness and probability

In this section, we provide a probability-theoretic view of fairness. In Section D.8, we showed that fairness properties are exactly the sets that are large in a topological sense. In probability theory, a set is large if it has measure 1. Although these two notions of largeness are very similar, they do not coincide in general. However, we will prove that topological largeness and probabilistic largeness of temporal properties coincide for finite-state systems, ω-regular properties and bounded Borel measures. Thus in the finite case and under mild assumptions, a property is a fairness property if and only if it has probability 1.

We start by pointing out some characteristics of the family of fairness properties.

D.9.1 Characteristics of fairness

Let S be a safety property. The family $\mathcal{F}_S$ of all fairness properties in S, i.e., of all—in the topological sense—large sets in S, has the following properties, which appeal to our intuition of largeness.

1. Any superset of a large set is large. Hence the union of arbitrarily many large sets is large.
2. The intersection of countably many large sets is large. Together with 1., this says that $\mathcal{F}_S$ is a *σ-filter*.
3. Every large set is dense (in S) and therefore nonempty.
4. If a set is large, its complement is not. Call a set *small* if its complement is large. A set may be neither large nor small. In that case we call it *medium-sized*.
5. Intersection with a large set preserves size, i.e., if F is large and X is small (medium-sized) [large] then $F \cap X$ is small (medium-sized) [large].
6. If $S = \Sigma^\infty$ and $|\Sigma| \geq 2$, then every countable set is small, but there are also uncountable sets that are small.

Statement 1 follows immediately from the definition of fairness. Statement 2 was proved in Proposition D.6.2. Statement 3 was proven in Proposition D.6.1. Statement 4 follows immediately from statements 2 and 3. Statement 5 follows straight forwardly from statements 1 and 2. For the first part of statement 6, first observe that any singleton set is small because Ego has a strategy to avoid it. By duality, the union of countably many small sets is small. For part 2 of statement 6, consider $\Sigma = \{p, q, r\}$ and $E = \mathrm{sat}(\Diamond \Box (p \vee q)) = \neg \mathrm{sat}(\Box \Diamond r)$. E is clearly uncountable but small because Ego has a winning strategy for $\mathrm{sat}(\Box \Diamond r)$. A similar example can be constructed for $|\Sigma| = 2$. Statement 6 can be generalised to safety properties S such that for each finite $\alpha \in S$, we have that $\alpha\uparrow$ is uncountable.

D.9.2 Probabilistic largeness

To define probabilistic largeness, we first recall the standard setting of how probability is adjoined to systems.

A *σ-field* over a nonempty set Ω is a family $\mathcal{A}$ of subsets of Ω that contains the empty set and is closed under complementation and countable union. The intersection of arbitrarily many σ-fields is again a σ-field. Hence for each family $\mathcal{F} \subseteq \mathcal{P}\Omega$, there exists the smallest σ-field that contains $\mathcal{F}$. Given a topology $\mathcal{T}$ on Ω, the *Borel* σ-field of $\mathcal{T}$ is the smallest σ-field that contains $\mathcal{T}$. A member of the Borel σ-field is called a *Borel set*, which is precisely what we have introduced already in Sect. D.8.3.

A *probability measure* on a σ-field $\mathcal{A}$ over Ω is a function $\mu : \mathcal{A} \to [0, 1]$ such that $\mu(\Omega) = 1$ and for any sequence of pairwise disjoint sets $(X_i)_{i \in \mathbb{N}}$, $\mu(\bigcup_{i \in \mathbb{N}} X_i) = \sum_{i \in \mathbb{N}} \mu(X_i)$. A *Borel* probability measure of a topology is a

probability measure over the Borel σ-field of the topology. Given a probability measure μ on $\mathcal{A}$, and two sets $X, Y \in \mathcal{A}$, the *probability of X conditional to Y* is defined as $\mu(X \mid Y) = \mu(X \cap Y)/\mu(Y)$, provided $\mu(Y) > 0$.

A probabilistic system, or more general a *probabilistic safety property*, is a pair of a safety property S and a Borel measure μ of the Scott topology relativised to $\max(S)$. We can think of it as the tree that S represents in which at each branching a multi-sided coin is flipped to determine how the run is extended. We assume here maximality for convenience, i.e., we assume that an enabled coin flip is eventually executed and each outcome of the coin flip properly extends the current run. We will write $\mu(X)$ for $\mu(X \cap \max(S))$ and $\mu(X \mid Y)$ for $\mu(X \cap \max(S) \mid Y \cap \max(S))$. Let $\alpha \in \Sigma^*$ and $s \in \Sigma$ such that $\alpha s \in S$. The conditional probability

$$p^s_\alpha = \mu(\alpha s\uparrow \mid \alpha\uparrow) = \mu(\alpha s\uparrow)/\mu(\alpha\uparrow) \tag{D.7}$$

is the probability that the outcome of the coin flip at the branching at α is s. Equation D.7 implies $\mu((s_0 \dots s_n)\uparrow) = p^s_\epsilon \cdot \ldots \cdot p^{s_n}_{s_0,\dots,s_{n-1}}$. For each $\alpha \in S$, we have

$$\sum_{\alpha s \in S} p^s_\alpha = 1. \tag{D.8}$$

The Borel measure is determined by the p^s_α, i.e., given $p^s_\alpha \in [0,1]$ for each α and s with $\alpha s \in S$ such that D.8 holds, there is a unique Borel measure μ of the Scott topology relativised to $\max(S)$ such that D.7 holds.

Let (S, μ) be a probabilistic safety property. We say that μ is a *Markov measure* if

$$p^{s'}_{\alpha s} = p^{s'}_{\beta s}$$

for all $\alpha, \beta \in S \cap \Sigma^*$ and $s, s' \in \Sigma$ such that $\alpha s s', \beta s s' \in S$, i.e., if the coin-flip probabilities only depend on the last state. We say that μ is *positive* if all $p^s_\alpha > 0$ and therefore $\mu(\alpha\uparrow) > 0$ for each $\alpha \in S$; μ is said to be *bounded (away from zero)* if there exists a constant $c > 0$ such that $p^s_\alpha > c$ for each $\alpha s \in S$.

Example D.9.1. Consider the system S, represented in Fig. D.8, which corresponds to the Petri net in Fig. D.7. For each $\alpha \in S \cap \Sigma^*$ such that $\alpha B \in S$,

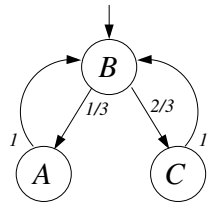


FIGURE D.8 – A three-state system

we define $p^A_{\alpha B} = 1/3$ and $p^C_{\alpha B} = 2/3$. Also, we are forced to have $p^B_{\beta A} = 1$ when $\beta A \in S$ and $p^B_{\beta C} = 1$ when $\beta C \in S$. This defines a Markov measure on S.

Topological and probabilistic largeness are very similar notions. Oxtoby's classic book [Oxt71] is devoted to the study of this similarity. All six statements in Section D.9.1 are also true for probabilistic largeness, where for statement 3, we naturally have to assume that μ is positive. To see that statement 3 is true, let $\mu(F) = 1$ and $\alpha \in S$. If there is no extension of α into $S \cap F$, then $\alpha\uparrow \cap S \subseteq \neg F$. As $\mu(\alpha\uparrow) > 0$, we have $\mu(\neg F) > 0$ and hence $\mu(F) < 1$, a contradiction. All other statements follow from the laws of probability theory.

D.9.3 Separation

Although similar, the two notions of largeness do not coincide in general : there are sets that are large in one sense but not in the other.

Consider an (unrestricted and asymmetric, i.e., biased,) random walk on the integer line starting at 0. At each state the probability of going right is $p \neq 1/2$, and the probability of going left is $1 - p$ (cf. also Fig. D.6). The property X_1 = "The walk returns to 0 infinitely often" has probability 0 (see for instance [Spi01]), but is co-meager (one easily displays a winning strategy for Ego in the Banach-Mazur game). On the other hand, the complement of X_1 has probability 1, but is meager.

A similar set can be displayed in a finite-state system : Consider the system in Fig. D.8. Note that the probability of going from B to C is not the same as the probability of going from B to A. The property X_2 = "The number of previous A's equals the number of previous C's infinitely often" has probability 0, but is co-meager.

Note that in both cases the winning strategies for Ego are unbounded, i.e., the length of the sequences Ego adds is unbounded because he has to be able to compensate for unbounded moves by Alter.

More generally, we have the following result.

Definition D.9.1. Let S be a safety property. A run x of S is said to *have infinitely many choices* if for infinitely many positions i, the prefix x_i has more than one extension in S.

Lemma D.9.2. *Let (S, μ) be a probabilistic safety property such that μ is a bounded Borel measure. Let x be a run of S with infinitely many choices. Then for every $\varepsilon > 0$, there exist an i such that $\mu(x_i\uparrow) < \varepsilon$. In particular $\mu(\{x\}) = 0$.*

Proof: Let $c > 0$ be the bound of the Markov measure. Let $a = (1 - c)$. Let $k > 0$ be such that $a^k < \varepsilon$. Consider a prefix x_i that contains at least k choices. At each choice, the probability of the prefix is multiplied at most by a, and therefore $\mu(x_i\uparrow) \leq a^k < \varepsilon$.

Proposition D.9.3. *Let M be a finite-state system and μ a bounded Borel measure on S_M. Suppose every maximal run of S_M has infinitely many choices. Then S_M can be partitioned into a co-meager set and a set of measure 1.*

Proof: The proof is essentially an adaptation from [Oxt71, Thm. 1.6]. Take a dense countable subset X of S_M, for instance the set of periodic runs. Let

x^i be the i-th run of X for $i \in \mathbb{N}$. For every $i, j \in \mathbb{N}$, consider the prefix $x^i_{k_j}$ such that $\mu(x^i_{k_j}\uparrow) < (1/2)^{i+j+1}$. Let $G^i_j = x^i_{k_j}\uparrow$. Let $G_j = \bigcup_{i\in\mathbb{N}} G^i_j$, and let $G = \bigcap_{j\in\mathbb{N}} G_j$. Each G_j is open (as union of basic open sets) and contains the dense set X. Thus G is a dense G_δ, and therefore co-meager. On the other hand, $\mu(G_j) \leq \sum_{i\in\mathbb{N}} \mu(G^i_j) < \sum_{i\in\mathbb{N}} (1/2)^{i+j+1} = (1/2)^j$. Therefore $\mu(G) = 0$ and the complement of G has measure 1.

D.9.4 Coincidence

We now prove that for bounded Borel measures on finite systems and ω-regular properties, the two notions of largeness coincide. Note that the property X_2 described in the counterexample in Section D.9.3 is not accepted by any finite-state automaton.

Proposition D.9.4. *Let M be a finite-state system, μ a bounded Borel measure on S_M, and E an ω-regular property. If E is co-meager in S_M, then $\mu(E) = 1$.*

Proof: If E is co-meager, Ego has a progressive winning strategy for E in the Banach-Mazur game $G(S_M, E)$. Berwanger, Grädel and Kreutzer [BGK03] have shown that Ego then has also a *positional* winning strategy, i.e., a strategy f such that

$$f(\alpha s) = \alpha s w \Rightarrow f(\beta s) = \beta s w$$

for all $\alpha, \beta \in \Sigma^*$. We will now show that $\mu(R_f) = 1$, and because $R_f \subseteq E$, we will have $\mu(E) = 1$.

As there are only finitely many states, the positional strategy f is also *bounded*, i.e., there exists a $k \in \mathbb{N}$ such that

$$|f(\alpha)| - |\alpha| \leq k$$

for each $\alpha \in \Sigma^*$.

Let $c > 0$ be the bound on probabilities of transitions, and let E_i be the event : "at position $i \cdot k$, the run follows the strategy f", i.e.,

$$E_i = \big\{x \in S \mid f(x_{i\cdot k}) \sqsubseteq x\big\}.$$

Clearly the probability of each E_i is at least c^k.

Then $\sum_n \mu(E_n) = +\infty$. If the E_n were independent, by the second Borel-Cantelli lemma (see for instance [Wil91]), we would have that the set of infinitely many occurrences of E_n :

$$\limsup E_n = \bigcap_{n=1}^{\infty} \bigcup_{k=n}^{\infty} E_k$$

has probability 1. And clearly $\limsup E_n \subseteq R_f$.

Unfortunately the E_n are not independent, as the exact probability of playing the strategy at some position may depend on whether the strategy had been played before or not. However, while the exact probability may vary, we still

know a lower bound for it : independently on whether the strategy has been played elsewhere, at position $i \cdot k$ the strategy is played with probability at least c^k. This form of "lower bound independence" is enough for our purposes. Indeed we use the following variant of the second Borel-Cantelli Lemma (see also [Sch07], page 29)

Lemma D.9.5. *Let E_n be events in some probability space. Denote by E'_n the complement of E_n. Suppose there exist a sequence $c_n > 0$ such that $\sum_{n=0}^{\infty} c_n = +\infty$ and such that the events are "lower bound independent" with respect to c_n, in the following sense : for each pair of finite disjoint sets of natural numbers I, J, for each $n \notin I \cup J$ if $\mu(E_n \mid \bigcap_{i\in I} E_i \cap \bigcap_{j\in J} E'_j)$ is defined then $\mu(E_n \mid \bigcap_{i\in I} E_i \cap \bigcap_{j\in J} E'_j) \geq c_n$.*

Then the probability that infinitely many E_n occur is 1.

We recall that $\mu(A \mid B)$ is defined as $\mu(A \cap B)/\mu(B)$. Therefore it is defined only if $\mu(B) > 0$.

Define $a_n = 1 - c_n$. Then $\mu(E'_n \mid \bigcap_{i\in I} E_i \cap \bigcap_{j\in J} E'_j) \leq a_n$, if it is defined. By induction on k we prove that for each $n \geq 0$ $\mu(\bigcap_{n\leq i\leq n+k} E'_i) \leq \prod_{n\leq i\leq n+k} a_i$. The basis is $\mu(E'_n) \leq a_n$ which is a special case of lower bound independence. For the step, if $\mu(\bigcap_{n\leq i\leq n+k} E'_i)$=0, then trivially $\mu(\bigcap_{n\leq i\leq n+k+1} E'_i) = 0 \leq \prod_{n\leq i\leq n+k+1} a_i$. Otherwise,

$$\begin{aligned}
\mu(\bigcap_{n\leq i\leq n+k+1} E'_i) &= \mu(E'_{n+k+1} \mid \bigcap_{n\leq i\leq n+k} E'_i)\mu(\bigcap_{n\leq i\leq n+k} E'_i) \\
&\leq a_{n+k+1} \cdot \prod_{n\leq i\leq n+k} a_i \\
&= \prod_{n\leq i\leq n+k+1} a_i
\end{aligned}$$

The first equality is by definition of conditional probability, while the inequality is by induction hypothesis and by lower bound independence.

Using logarithms one shows that $\sum_{n\leq i} c_i = +\infty$ implies $\prod_{n\leq i} a_i = 0$ and thus $\mu(\bigcap_{n\leq i} E'_i) = 0$ for any n. Therefore $\mu(\bigcup_{n=1}^{\infty} \bigcap_{k=n}^{\infty} E'_k) = 0$ which implies $\mu(\bigcap_{n=1}^{\infty} \bigcup_{k=n}^{\infty} E_k) = 1$.

The converse also holds.

Proposition D.9.6. *Let M be a finite-state system, μ a bounded Borel measure on S_M, and E an ω-regular property. If $\mu(E) = 1$, then E is co-meager in S_M.*

Proof: If E is not co-meager, then Alter has, owing to determinacy of ω-regular properties, a winning strategy f in the Banach-Mazur game $G(S_M, E)$. Let $\alpha_0 = f(\epsilon)$ be the first move of Alter in that strategy. We have $\mu(\alpha_0\uparrow) > 0$. As f is a winning strategy for Alter, f is also a winning strategy for Ego in the Banach-Mazur game $G(S_M, \neg(\alpha_0\uparrow \cap E))$. From Proposition D.9.4 now follows $\mu(\neg(\alpha_0\uparrow \cap E)) = 1$, hence $\mu(\alpha_0\uparrow \cap E) = 0$. Because of $\mu(\alpha_0\uparrow) > 0$, we obtain $\mu(E) < 1$.

We obtain that topological and probabilistic largeness coincide under the conditions discussed :

Theorem D.9.7. *In a finite system M under a bounded Borel measure μ, we have, for each ω-regular property E, $\mu(E) = 1$ if and only if E is co-meager in the Scott topology on S_M .*

In particular, topological and probabilistic largeness also coincide for properties expressible in LTL.

We observe that the requirement on boundedness could be further weakened. For instance, we could consider *slowly vanishing* measures, i.e., measures that are not bounded but such that there exists a function $h(n)$ such that $\prod_n(1 - h(n)) = 0$ and for any prefix α of length n $p^s_\alpha > h(n)$ if $\alpha s \in S$.

D.10 Fair correctness

A system is *correct* w.r.t. a linear-time specification, i.e., a temporal property E, if each run of the system satisfies E. If the system comes with a fairness assumption, we only have to check that each fair run satisfies the specification. The fairness assumption can be seen as a rely specification for the system that is guaranteed by the environment.

Traditionally only simple, well-understood fairness assumptions are employed, such as maximality, weak fairness and, occasionally, strong fairness. Proof calculi and model-checking algorithms have been optimised to deal with such simple fairness assumptions.

However, sometimes, a system does not satisfy a desired specification even under the assumption of strong fairness. As an example, we can consider again Dijkstra's dining philosophers and the "conspiracy" scenario mentioned in Section D.3.4. For this particular example, there is also a starvation-free, although more complex, solution under strong fairness. However, for many problems, a system satisfying the actual specification is impossible, too difficult, or too expensive to obtain, cf. [FR03]. Still, simple systems may exist for those problems that satisfy the specification "well enough" and which actually work in practice.

However, in such cases, an appropriate fairness assumption, or more generally, an appropriate rely specification may be more difficult to find, more expensive to specify, and proof calculi and model-checking algorithms may not work well under them. For these cases, we can use a generic relaxation of correctness that allows us to verify a system that satisfies the specification "well enough", i.e., for most, but not necessarily all, runs. To formalize "most" runs, we use topological largeness, i.e., fairness :

Definition D.10.1. A system M is *fairly correct* w.r.t. a temporal property E if E is co-meager in S_M.

Equivalently, M is fairly correct w.r.t. E if there exists a fairness assumption for M under which it is correct w.r.t. E, i.e., there exists a fairness property F for S_M such that $S_M \cap F \subseteq E$. Thus, we abstract from the concrete fairness

assumption that is required to guarantee the specification E. Note that the difference between traditional and fair correctness is small (in a topological sense). In particular, fair correctness guarantees that the system does not violate safety, i.e., the safety property implied by the specification : M is fairly correct w.r.t. E implies that M is correct w.r.t. $\overline{E}$, that is, $S_M \subseteq \overline{E}$, where $\overline{E}$ denotes the safety closure of E introduced in Sect. D.8.2. This implication follows from the fact that a co-meager set is dense.

Fair correctness occurs implicitly in the literature, in game-theoretic, topological and probabilistic form as will be explained below. In particular, it turns out that fair correctness is one of the seven relaxations of linear-time correctness considered by Pistore and Vardi [PV03] and by Berwanger, Grädel and Kreutzer [BGK03]. [PV03] provide an independent motivation of these relaxations of correctness in planning scenarios. They argue that such intermediate versions of correctness are adequate for specifying goals for automated task planning in nondeterministic domains.

We will now discuss how fair correctness can be verified.

D.10.1 Proving fair correctness

If ϕ is an LTL formula, then $\mathsf{A}\,\phi$ is a formula of the branching-time logic CTL*, cf. [Eme90], where A stands for "for all paths". If we re-interpret A in CTL* as "for most paths" in the topological sense, we obtain a language that can express fair correctness for LTL and which has been studied by Ben Eliyahu and Magidor [BEM96]. They present a sound and complete proof system for that language. The same proof system has been introduced before by Lehmann and Shelah [LS82] for a version of CTL* in which the path quantifier A is interpreted as "for almost all paths" in the probabilistic sense. Lehmann and Shelah showed soundness and completeness of the proof system for finite probabilistic systems with bounded measures.

Alternatively we can prove fair correctness, using deduction in linear-time temporal logic, by proving that the specification is implied by some well-known fairness property in the system – such as those presented in Sect. D.3. One can then ask whether there is a fairness notion that is *complete* for proving fair correctness in this way, i.e., a fairness notion F that implies all temporal properties that are fairly correct in the system, i.e., M is fairly correct w.r.t. to E if and only if $F \cap S_M \subseteq E$. It is easy to see that, if such property existed, it should be obtained as the intersection of all fairness properties. This intersection may be the empty set in some systems as was shown in Section D.6.

However, if one is interested in a particular class of specifications, then it is possible to find a fairness property that is complete *for that class.*

Definition D.10.2. Let S be a safety property and $\mathcal{F} \subseteq \mathcal{P}\Sigma^\infty$ a family of linear-time properties. A fairness property F in S is $\mathcal{F}$-*complete* w.r.t. S if for each property $E \in \mathcal{F}$, we have : If E is co-meager in S, then $F \cap S \subseteq E$.

A complete fairness property, or a corresponding winning strategy for Ego, can be used to prove fair correctness w.r.t. each specification in $\mathcal{F}$. The following

is a trivial consequence of the definition.

Proposition D.10.3. *Given a family $\mathcal{F}$, there is a $\mathcal{F}$-complete fairness property w.r.t. S if and only if*

$$\bigcap\{F \in \mathcal{F} \mid F \text{ is a fairness property in } S\} \tag{D.9}$$

is a fairness property in S.

Note that if F is complete for a family $\mathcal{F}$, then it is also complete for every subfamily of $\mathcal{F}$. Also note that if F is $\mathcal{F}$-complete and F' is a fairness property in S such that $F' \subseteq F$, then F' is also $\mathcal{F}$-complete.

Even if a $\mathcal{F}$-complete fairness property exists, it need not be a member of the family $\mathcal{F}$.

Definition D.10.4. Let F be a fairness property. We say that F is *initial* in $\mathcal{F}$ w.r.t. S if F is $\mathcal{F}$-complete and $F \in \mathcal{F}$.

If $\mathcal{F}$ has an initial fairness property, then this is essentially unique :

Proposition D.10.5. *If F and F' are initial in $\mathcal{F}$, then $F \cap S = F' \cap S$.*

Proof: From the definition of initiality follows $F \cap S \subseteq F'$ and $F' \cap S \subseteq F$.

We are usually interested in countable families $\mathcal{F}$, such as the family of all ω-regular properties or the family of all LTL-expressible properties. The characterisation of completeness in D.9 then implies that there is a complete fairness property because fairness is closed under countable intersection.

In particular, completeness w.r.t. ω-regular and LTL-expressible properties can be characterised through word fairness (as defined in Sect. D.3.6).

Theorem D.10.6. *Let $\mathcal{F}_1$ denote the family of all ω-regular properties or LTL-expressible properties. Let M be a finite system. A fairness property F is $\mathcal{F}_1$-complete w.r.t. M if and only if each run in $F \cap S_M$ is word fair. In particular, word fairness is $\mathcal{F}_1$-complete w.r.t. M.*

Proof: ($\Leftarrow$) If E is an ω-regular property that is co-meager in S_M, then it follows—as in the proof of Proposition D.9.4—that Ego has a positional winning strategy f in the Banach-Mazur game $G(S_M, E)$. Let $x \in S_M$ be word fair. If x is finite, then it must be maximal because of word fairness. As each finite maximal run is fair, we have $x \in E$. If x is infinite, some state s is repeated infinitely often. It follows that the word $w = sf(s)$ is enabled infinitely often in x. As x is word-fair, w is taken infinitely often in x. This in turn implies that $x \in R_f$. As f is winning for Ego, we have $x \in E$. ($\Rightarrow$) It is easy to see that word fairness w.r.t. a particular word w is LTL-expressible. As word fairness is indeed a fairness property in S_M, as argued in Section D.4, the claim follows from the characterisation in D.9.

The assumption in Proposition D.10.6 that M is finite is essential. Consider the nondeterministic walk on the integer line in Fig. D.6 and the run $x = 0, 1, \ldots$; x is word fair but violates $\mathrm{sat}(\Diamond -1)$, which is topologically large.

Another fairness notion that is complete for ω-regular properties is *α-fairness*, which was introduced by Lichtenstein, Pnueli and Zuck [LPZ85] to prove probabilistic largeness. They also proved completeness w.r.t. probabilistic largeness in finite-state systems.

Although there exist complete fairness notions for ω-regular properties and therefore for LTL-expressible properties, those are not ω-regular themselves :

Proposition D.10.7. *Let $\mathcal{F}_1$ denote the family of all ω-regular properties or the family of LTL-expressible properties. There are finite systems M such that there is no initial fairness property in $\mathcal{F}_1$ w.r.t. S_M.*

Proof: Consider $M = (\Sigma, R)$, with $\Sigma = \{p, q\}$ and $R = \Sigma \times \Sigma$. Suppose there were an ω-regular fairness property F that is $\mathcal{F}_1$-complete w.r.t. $S_M = \Sigma^\infty$. Then Ego has a positional winning strategy f in $G(S_M, F)$. Let $f(p) = pw_p$ and $f(q) = qw_q$. Let $x = (w_p w_q)^\omega$. Since f is winning and $x \in R_f$, we have $x \in F$. Let $k = |w_p| + |w_q|$ and let $w = p^k q$. This word w does not occur in $x : p^k$ occurs in x only if $w_p w_q = p^k$. In that case $p^k q$ cannot occur in x. It follows that x is not strongly fair w.r.t. w. However, strong fairness w.r.t. w can be expressed in LTL. Hence F is not $\mathcal{F}_1$-complete, which contradicts our supposition.

Proposition D.10.7 shows that largeness of an LTL formula ϕ can in general not be checked by expressing a complete fairness property as LTL formula ψ and then checking the formula $(\psi \to \phi)$.

However, there are natural families that have an initial fairness property :

Proposition D.10.8. *Let $\mathcal{F}_2$ be the family of all RLTL-expressible properties and M a finite system. Then state fairness (as defined in Sect. D.3.6) is initial in $\mathcal{F}_2$ w.r.t. S_M.*

Proof: Lichtenstein, Pnueli and Zuck [ZPK02] point out that state fairness is complete for showing probabilistic largeness of properties that are expressible in RLTL. A direct, detailed proof for this result is given by Matthias Schmalz [Sch07]. State fairness for a finite system $M = (\Sigma, R)$ can be expressed by the following RLTL formula :

$$\bigwedge_{(s,s') \in R} \Box \Diamond s \to \Box \Diamond s' \tag{D.10}$$

D.11 Conclusions

We have proposed a definition of fairness that is in line with the well-known formalizations of safety and liveness. We have given three equivalent characterisations of fairness : one that generalises the standard notion of strong fairness, one in terms of the Banach-Mazur game, and one in terms of topological largeness. The game-theoretic characterisation stresses the intuition that fairness guarantees that some finite behaviour will always eventually happen while fairness can never prevent any finite behaviour from happening recurrently. The topological characterisation confirms the intuition that *most* runs of a system

are fair. To strengthen this intuition, we have shown that in some cases, topological largeness coincides with probabilistic largeness. In these cases, the set of unfair runs has probability 0.

The definition of fairness led to a generic relaxation of linear-time correctness, which we called fair correctness. Verifying fair correctness of a system can be useful whenever the specification is satisfied only under some, possibly strong, fairness assumption and the fairness assumption is either unknown, expensive to specify, or impossible to specify in the temporal logic. We have shown that the model checking problem for fair correctness has the same complexity for LTL and ω-regular specification as the corresponding problem for classical correctness. This also answered an open question posed by Pistore and Vardi.

One prominent topic in the literature on fairness is the *implementability* of a fairness notion (cf. e.g.[Jou01]), in which one asks whether a scheduler exists that can be superimposed onto the system to realize a given fairness assumption. In this work, we have chosen the simplest possible setting to study fairness, aligned with earlier related contributions [AS85, MP90], where a system has only one source of nondeterminism, which is then restricted by the fairness assumption. In the corresponding setting of a scheduler that implements a fairness assumption, the scheduler has access to all nondeterministic choices of the system. Besides our abstract study of fairness, such a simple setting can be adequate, for example, for studying some non-distributed multitasking operating systems. Then, Thm. D.9.7 implies that any ω-regular fairness assumption on a finite system can be generically implemented with probability 1 through coin flipping. This includes the implementation of very strong fairness assumptions such as ∞-fairness over finitary properties (Def. D.4.1). Alternatively, one can use a deterministic scheduler for word fairness, e.g., repeatedly scheduling all enabled finite words of the system, to implement any ω-regular fairness notion in a finite system (cf. Thm. D.10.6). One could then still ask for our setting whether some fairness assumption can be implemented in some sense more efficiently in a given finite system or at all in a given infinite system.

However, often the question of implementability arises in a specific system model, e.g., a finite set of distributed processes that communicate asynchronously by message passing. In such models, we often have multiple sources of nondeterminism, such as the uncertain order of message receipt or the unknown inputs from the environment. These sources of nondeterminism then need to be distinguished in the system model because usually not all of them can be influenced by a superimposed scheduler. Studying implementability then requires a more complex setting to reflect this. A natural generalized setting to study implementability are *open systems* with two kinds of nondeterministic choices, one controlled by a scheduler and one by an adversary, i.e., the environment. A direction for further work is to formalize fairness in that setting, which is tightly related to the notion of *realizability* as introduced by Abadi, Lamport and Wolper [ALW89] and to two-player games, similar to the ones presented in [ACV10].

It is clear that some of the fairness notions discussed above cannot be implemented in such a generalized setting as they stand. For example, if an action is

k-enabled, it does not necessarily mean that a scheduler can ensure the execution of the action in the future because it may not have access to all the actions that lead to the enabledness. Further complications can arise in distributed systems where the scheduler may require non-trivial distributed computations to obtain global knowledge that is needed to guarantee a certain fairness notion [Jou99]. Studying such issues requires further specialization of the system model to reflect distribution and the fact that the scheduler has only partial knowledge of the system state.

We mentioned related work throughout the chapter. Some authors used the notion that we have described as fairness in different contexts without actually attempting to define fairness : Ben Eliyahu and Magidor [BEM96] observed that some popular fairness notions describe co-meager sets. Alur and Henziger [AH95] argue that for the compositional modeling of reactive systems, machine closure should be strengthened to what they call *local liveness*, which is the same as what we have defined as fairness. They gave the game-theoretic definition. As mentioned above, a generalised form of the Banach-Mazur game had been considered in [PV03] and [BGK03]. Thus, our results link different strands of research that were developed independently.

In addition, we would like to mention two related works, whose relations with ours should be further studied. One is recent work by Manfred Jaeger [Jae09] that also finds connections between a more intensional notion of fairness and Martin-Löf randomness. The second one is the work by Darondeau et.al. [D$^+$92], where another, rather different, point of view on fairness is provided. For them, fairness properties are characterised as the Π^0_3 sets in Kleene's hierarchy. Thus the class of fairness properties is not closed under superset. No topological characterisation is provided, although there are analogies between Π^0_3 sets and $F_{\sigma\delta}$ sets.

Some follow-up work on our proposal has already appeared. Together with Schmalz [SVV07], we show, for various subclasses of LTL, that model checking fair correctness has either the same complexity as checking classical correctness or that it is even strictly less expensive. Furthermore we show elsewhere [SVV09] that the game-theoretic characterisation can be used to present counterexamples in probabilistic model checking. Asarin, Chane-Yack-Fa and Varacca [ACV10] study the notion of fairness in the context of two-player games. A generalisation of the Banach-Mazur game is proposed, and the equivalence between two-player games with fairness and Markov decision processes is shown. Baier et.al. [B$^+$08] apply our framework to timed automata. There, a coincidence result between topological and probabilistic largeness is also shown.

Annexe E

Modelling the handshake protocol

E.1 Introduction

Asynchronous circuits are used to design systems where the local activity of each subunit is not restrained by some global condition, like the long time intervals imposed by a system clock. When designing such systems, one has to face several questions. How do we know when a message we sent has reached its destination so that we can use the same channel again, i.e. how can we avoid *transmission interference* ? How can we ensure the correct behavior regardless of computational speeds of single modules and propagation delays over wires, i.e. how can we enforce *delay-insensitivity* ?

Handshake protocols try to answer these questions by imposing an interactive communication discipline. In particular, the protocols require that after a circuit has sent a message on a channel, it has to wait for a confirmation that the message was received before sending again on the same channel. This requirement alone is enough to rule out transmission interference. For their simplicity and efficiency, handshake protocols have been employed by enterprises like Philips and Sun in the development of a series of VLSI chips [Hso].

The first attempts to formalize delay-insensitive protocols and their properties employed trace sets [Udd84]. In particular, the first model to specifically address the handshake case was given in [VB93]. Trace models have been able to neatly formalize the properties of handshake protocols which ensure delay-insensitivity, but so far they have failed in representing correctly their composition [Fos09]. To solve this problem, Fossati [Fos07] proposed a game semantics for handshake circuits which describes their composition correctly for the first time. Technically, the result was accomplished by representing handshake "behaviors" as sums of *deterministic handshake strategies*. The price to pay is that there are behaviors which do not fit in this representation. The crucial example is the mixer component, MIX, which will be described in Section E.4.5.

To overcome these limitations, we propose two alternative approaches to modeling handshake protocols : a process calculus, inspired by Robin Milner's Calculus of Communicating Systems (CCS) [Mil89], and a Petri net [Rei85] model.

Similarly to CCS, our calculus defines concurrent processes that communicate via rendez-vous channels. However, in order to ensure the handshake discipline, the calculus features another synchronization mechanism, by means of shared resources, reminiscent of coordination languages like Linda [Gel85]. Also, the calculus is endowed with a linear typing system, inspired by [KPT99, YHB02]. These design choices allow to express the external behavior of a handshake protocol, along with a more complex internal behavior.

We claim that this is the first independent syntactical description of the handshake behavior, as our *handshake configurations* are independent from any semantical interpretation while the handshake behavior is ensured by the typing system. This was not the case in previous process algebras for handshake protocols [VB93, JUY93], where only processes whose trace semantics satified the handshake behavior were considered, thus processes were trace sets and inevitably suffered from the compositionality problems observed in the underlying trace model.

We then compare our calculus and the trace model by defining, for each configuration, the corresponding set of *quiescent* traces, i.e. the traces corresponding to computations that may not be extended with an output event. We show that this quiescent trace semantics is sound w.r.t. Van Berkel's definition [VB93]. By means of an example, we show that quiescent trace equivalence is not a congruence w.r.t. parallel composition. This confirms the intuition that trace models of handshake protocols are not informative enough, and their branching structure needs to be taken into account. Indeed weak bisimilarity is a congruence for our calculus.

A graphical representation is also a very natural choice for dealing with asynchronous circuits : in graphs as in circuits, composition is easy when everything else works properly. Several works have taken a similar perspective ([AGN96],[Mac95],...). In particular Dan Ghica developed a language for asynchronous hardware design by taking inspiration from the Geometry of Interaction and handshake circuits [Ghi07]. However his goal was to improve previous hardware design languages [VB93, Bar98] and not to capture all handshake behaviors.

The second model we present is based on Petri nets [Rei85]. Petri nets are widely used as models of asynchrony, and are close to the context in which the handshake communication protocol originated [CS72a]. However, the properties of delay-insensitivity and absence of transmission interference had not yet been formalized under a graphical representation. We call our model *handshake Petri nets*. We show that handshake Petri nets capture precisely the handshake protocol, in the sense that the behavior of every net is a handshake language and that every handshake language is the behavior of some net.

We finally relate our two models, by giving the process calculus a Petri net semantics. In particular we show that there is a strict correspondence between

handshake processes and *finite* Petri nets, in the sense that for each finite handshake Petri net, there is a weakly bisimilar process.

Structure of the chapter This chapter is a fusion of the papers [FV08, FV09].

In Section E.2 we define the notion of handshake language as set of traces (taking inspiration from [VB93] and [Fos07]). In In Section E.3, we present syntax, operational semantics and type system of our calculus. We show that the set of quiescent traces of a typed configuration is a handshake language. We show that weak bisimilarity is a congruence, while quiescent trace semantics is not. Section E.4 we introduce handshake Petri nets and some of their subclasses. We provide an interpretation of handshake Petri nets into handshake languages and we prove the correctness and completeness of this interpretation. Completeness of deterministic handshake Petri nets with respect to deterministic handshake languages follows as a corollary. In Section E.5, we present the interpretation of the calculus into handshake nets, and we show that it is fully abstract with respect to weak bisimilarity. To conclude, we show the universality of the semantics, by showing that every bisimilarity class of finite handshake nets is denoted by a process.

E.2 The handshake protocol

In this section we characterize the handshake protocol in terms of languages obeying its communication discipline. We do not exactly give another trace model as, for instance, we do not define composition. We just need a yardstick against which to measure the correctness of our model. Moreover, we are only interested in the communication discipline, so we assume circuits have *nonput ports* (no data is exchanged in a communication).

Definition E.2.1. A *handshake structure* is a pair $\langle P, d\rangle$, where P is a finite set of *ports* and the function $d : P \to \{act, pas\}$ determines a direction for each port, *active* or *passive*.

As we shall see, active ports are allowed to start a communication, while passive ports are initially waiting.

For the rest of this section let $\langle P, d\rangle$ be a handshake structure and let $\cup_{p\in P}\{p, \bar{p}\}$ be the alphabet of messages on $\langle P, d\rangle$. In particular, p and $\bar{p}$ are both messages on some port p [1]. Two messages are *independent* when they are not on the same port. The function λ_P is defined on $\cup_{p\in P}\{p, \bar{p}\}$ so that $\lambda_P(p) = -$ (input message) and $\lambda_P(\bar{p}) = +$ (output message), for all $p \in P$. We may write λ instead of λ_P when P is redundant or clear from the context.

Let t be a trace on the alphabet of messages $\cup_{p\in P}\{p, \bar{p}\}$. t is a *handshake trace* on $\langle P, d\rangle$ if for all $p \in P$:

1. The same name p is used for both the message and the port. The context will always make clear which p we are referring to.

– $t \restriction \{p, \bar{p}\} = \bar{p}p\bar{p}p \ldots$ when $d(p) = act$;
– $t \restriction \{p, \bar{p}\} = p\bar{p}p\bar{p} \ldots$ when $d(p) = pas$;

We call *thread* each such restriction and we call *request* (*acknowledge*) the message appearing in the odd (even) positions in each thread of p.

Threads induce an equivalence on traces, the *homotopy* relation $\sim_P$. Given two handshake traces s and t, we say that $s \sim_P t$ when they have the same set of threads. As usual, we denote by $[s]_\sim$ the equivalence class of trace s with respect to $\sim$, we call $[s]_\sim$ the *position* of s.

Given a set of traces σ we write $\sigma^{\leq}$ for its prefix-closure. Let σ be a set of handshake traces, $s \in \sigma^{\leq}$ is *passive* in σ if and only if there is no message σ can output after s :

$$\forall s \cdot m \in \sigma^{\leq}, \lambda(m) = -.$$

We write $Pas(\sigma)$ for the set of passive traces in $\sigma^{\leq}$.

We define $\mathbf{r}_P$ as the smallest binary relation which is closed by reflexivity, transitivity and concatenation, and such that for any distinct ports $p, q \in P$:

1. $p\bar{q}\ \mathbf{r}_P\ \bar{q}p$;
2. $\bar{p}\bar{q}\ \mathbf{r}_P\ \bar{q}\bar{p}$;
3. $pq\ \mathbf{r}_P\ qp$

We say that s *reorders* t in P if $s\ \mathbf{r}_P\ t$. Note that the relation $\mathbf{r}_P$ is *not* symmetric.

Let s be a handshake trace and $p \in P$. We write $p\ \mathbf{x}_P\ s$ if sp is still a handshake trace. We are now ready for the definition of handshake language.

Definition E.2.2. A *(handshake) language* σ on $\langle P, d\rangle$ is a non-empty set of finite handshake traces on $\langle P, d\rangle$ such that :

1. $Pas(\sigma) \subseteq \sigma$ (closed under passive prefixes) ;
2. $(t \in \sigma \wedge s\ \mathbf{r}_P\ t) \Rightarrow s \in \sigma$ (reorder closed) ;
3. $(s \in \sigma^{\leq} \wedge p\ \mathbf{x}_P\ s) \Rightarrow s \cdot p \in \sigma^{\leq}$ (receptive).

Note that the traces of a handshake language are finite, but the language itself may contain an infinite number of traces.

Definition E.2.3. Let σ be a handshake language. We say that σ is *positional* if, for finite $s, s' \in \sigma^{\leq}$, with $s \sim_P s'$, we have :

1. $s \cdot t \in \sigma^{\leq} \Rightarrow s' \cdot t \in \sigma^{\leq}$;
2. $s \in \sigma \Rightarrow s' \in \sigma$;

We say that σ is *deterministic* if for any distinct $p, q \in P$:

1. $s \cdot \bar{p} \in \sigma^{\leq} \Rightarrow s \notin \sigma$ (progress) ;
2. $s \cdot \bar{p} \in \sigma^{\leq} \wedge s \cdot \bar{q} \in \sigma^{\leq} \Rightarrow s \cdot \bar{p} \cdot \bar{q} \in \sigma^{\leq}$ (absence of conflict).

Positionality means that the only thing relevant in a choice is the position we are at and not the way we reached it. As for determinism : when a deterministic language σ is able to produce an output, waiting is not an option ; when there is a choice of two outputs, one choice must not exclude the other. It is not difficult to prove the following fact.

Proposition E.2.4. *A deterministic language is positional.*

Examples

Consider the handshake structures $\mathcal{P} = \langle\{p\}, \{p \mapsto pas\}\rangle$ and $\mathcal{A} = \langle\{p\}, \{p \mapsto act\}\rangle$, corresponding respectively to a passive and to an active port. Then, $p\bar{p}p\bar{p}p\bar{p}$ is a handshake trace on $\mathcal{P}$ but not on $\mathcal{A}$. The set

$$\{p\bar{p}, p\bar{p}p\bar{p}, p\bar{p}p\bar{p}p\bar{p}, \ldots\}$$

is not closed under passive prefixes as it does not contain the empty string, then it is not a handshake language. Whereas both sets

$$RUN_p = \{\bar{p}, \bar{p}p\bar{p}, \bar{p}p\bar{p}p\bar{p}, \ldots\} \qquad \text{and} \qquad \{\varepsilon, \bar{p}, \bar{p}p\bar{p}, \bar{p}p\bar{p}p\bar{p}, \ldots\}$$

are handshake languages on $\mathcal{A}$. In particular RUN_p is deterministic, the other is not. The set

$$\{\bar{p}, \bar{p}p\bar{p}, \bar{p}p\bar{p}p\bar{p}\}$$

is not a handshake language on $\mathcal{A}$, because it is not receptive : after the last trace the environment is still supposed to send an acknowledge, but the language is not ready to receive it. Even the receptive RUN_p becomes not receptive if we extend its structure with a passive port, as in $\mathcal{B} = \langle\{p, q\}, \{p \mapsto act, q \mapsto pas\}\rangle$. A process which is receptive with respect to $\mathcal{B}$ is the following :

$$REP_{p,q} = \{\varepsilon, q\bar{p}, q\bar{p}p\bar{p}, q\bar{p}p\bar{p}p\bar{p}, \ldots\}$$

this process is also called repeater since, after reception of a request on its passive port it "handshakes" indefinitely on the active. Now look at the following sets on $\mathcal{B}$:

$$\{\varepsilon, q\bar{q}\bar{p}, q\bar{q}\bar{p}q, q\bar{q}\bar{p}qp\}$$

$$\{\varepsilon, q\bar{q}\bar{p}, q\bar{p}\bar{q}, q\bar{q}\bar{p}q, q\bar{p}\bar{q}q, q\bar{q}\bar{p}p, q\bar{p}\bar{q}p, q\bar{q}\bar{p}qp, q\bar{q}\bar{p}pq, q\bar{p}\bar{q}pq, q\bar{p}\bar{q}qp\}$$

Neither of them is reorder-closed, then neither of them is a handshake language. For example, $q\bar{q}q\bar{p}\ \mathbf{r}_{\{p,q\}}\ q\bar{q}\bar{p}q$ but $q\bar{q}\bar{p}q$ is in the prefix-closure of both of the above sets, while $q\bar{q}q\bar{p}$ is in the prefix-closure of none. We leave it to the reader to figure out the reorder-closures of the above two sets and to show that the second's is a handshake language while the first's is not.

Finally, consider yet another set on $\mathcal{B}$:

$$\{\varepsilon, q\bar{p}, q\bar{q}, q\bar{p}p\bar{q}, q\bar{q}q\bar{p}, q\bar{p}p\bar{q}q, q\bar{q}q\bar{p}p\bar{p}, q\bar{q}q\bar{p}p\bar{p}p\}$$

The reader can verify that it satisfies all the properties of a handshake language, we show that it does not satisfy those of positionality. Note that $q\bar{p}p\bar{q}q$ and $q\bar{q}q\bar{p}p$ are two traces with the same position and that both are in the prefix-closure of the above set. However, while the first is actually an element of the set, the second is not and, conversely, while the second can be extended with $\bar{p}$, the first cannot. The language is not deterministic either since after the initial q there is a mutually exclusive choice between $\bar{p}$ and $\bar{q}$.

E.3 The calculus

In this section we provide the formal definition for our *Calculus of Handshake Configurations (CHC)*. We stress that we do not model data-passing as we are only interested in the communication protocol. The calculus is endowed with two communication mechanisms. Besides the external communication via rendez-vous channels, there is also a form of internal communication, invisible to the outside, where actions may require resources in order to be performed and may release resources for other actions to use. This is necessary to model internal synchronizations between different ports of the same system. However, different systems shall communicate only through channels.

E.3.1 Syntax and operational semantics

We consider a set A of *channels* denoted by a, b [2], and a set R of *resources* denoted by r, k. The syntax of the calculus uses three syntactic categories : *threads*, *processes* and *handshake configurations*. Threads are purely sequential and allow prefixing while processes are parallel compositions of threads. The prefixes are *input* and *output* actions on a finite set of resources. As we will see later, input actions *release* resources and output actions *use* resources. Let $\Delta \subseteq A$:

$$\begin{array}{rcll} act & ::= & a^{\{r_1,\ldots,r_n\}} \mid \bar{a}_{\{r_1,\ldots,r_n\}} & \text{Actions} \\ T & ::= & \mathbf{0} \mid act.T \mid \mathbf{Rec}\ T & \text{Threads} \\ P,Q & ::= & T \mid P \mid Q \mid P \setminus \Delta & \text{Processes} \end{array}$$

A handshake configuration is composed of a process P along with a *multiset* of resources S for internal synchronization. A configuration can be *open* or *closed* :

$$M ::= \wr P, S \int \mid \langle P, S \rangle \quad \text{Open and closed configurations}$$

Intuitively, open configurations represent systems under construction, whose resources are still accessible to the environment. Closed configurations represent completed systems and can only communicate via handshake channels.

The operational semantics is given in terms of an LTS over handshake configurations. Labels are channels with their polarity, plus the unobservable label :

$$e ::= \bar{a} \mid a \mid \tau \qquad a \in A$$

Given an observable label, the function ch returns the channel on which it occurred. Formally : $ch(\bar{a}) = ch(a) = a$ for any channel a. The definition of the operational semantics is simplified thanks to the congruence ($\equiv$) between processes :

$$P \mid Q \equiv Q \mid P \qquad \mathbf{Rec}\ \mathbf{0} \equiv \mathbf{0}$$

2. We will use channels to model ports, but we prefer to keep the conceptual difference between the two notions

$$\frac{}{\wr a^{\{r_1,\ldots,r_n\}}.T, S\int \xrightarrow{a} \wr T, S + \{r_1,\ldots,r_n\}\int} \text{(inev)}$$

$$\frac{}{\wr \bar{a}_{\{r_1,\ldots,r_n\}}.T, S + \{r_1,\ldots,r_n\}\int \xrightarrow{\bar{a}} \wr T, S\int} \text{(outev)}$$

$$\frac{M \xrightarrow{e} M'}{M \mid N \xrightarrow{e} M' \mid N} \text{(par1)} \qquad \frac{M \xrightarrow{\bar{a}} M' \qquad N \xrightarrow{a} N'}{M \mid N \xrightarrow{\tau} M' \mid N'} \text{(par2)}$$

$$\frac{M \xrightarrow{e} M' \quad ch(e) \notin \Delta}{M \setminus \Delta \xrightarrow{e} M' \setminus \Delta} \text{(res)}$$

$$\frac{\wr P, S\int \xrightarrow{e} \wr Q, S'\int}{\langle P, S\rangle \xrightarrow{e} \langle Q, S'\rangle} \text{(closure)} \qquad \frac{\wr T \cdot \mathbf{Rec}\ T, S\int \xrightarrow{e} \wr T', S'\int}{\wr \mathbf{Rec}\ T, S\int \xrightarrow{e} \wr T', S'\int} \text{(rec)}$$

$$\frac{P \equiv P' \quad \wr\ P', S\int \xrightarrow{e} \wr Q', S'\ \int \quad Q \equiv Q'}{\wr P, S\int \xrightarrow{e} \wr Q, S'\int} \text{(struct)}$$

FIGURE E.1 – Labeled transition semantics

Let $res(P)$ be the set of resources of a process P. As meta-notation, we define sequential composition of threads $T \cdot T'$:

$$(act.T)\cdot T' = act.(T\cdot T') \quad T\cdot T' = T' \ (T \equiv \mathbf{0}) \quad (\mathbf{Rec}\ T)\cdot T' = \mathbf{Rec}\ T \ (T \not\equiv \mathbf{0})$$

and we extend process operators to configurations :

$$\wr P_1, S_1\int \mid \wr P_2, S_2\int = \wr P_1 \mid P_2, S_1 + S_2\int \qquad \wr P, S\int\backslash\Delta = \wr P\backslash\Delta, S\int$$

$$\langle P_1, S_1\rangle \mid \langle P_2, S_2\rangle = \langle \mu_1(P_1) \mid \mu_2(P_2), \mu_1(S_1) + \mu_2(S_2)\rangle \qquad \langle P, S\rangle\backslash\Delta = \langle P\backslash\Delta, S\rangle$$

where $+$ denotes the union of multisets and $\mu_1 : res(P_1) \cup S_1 \to R_1$ and $\mu_2 : res(P_2) \cup S_2 \to R_2$ are injective functions between resources such that R_1 and R_2 are disjoint and all the resources they contain are fresh. Moreover $\mu(S)$ is the point-to-point application of the function μ to the multiset S, while $\mu(P)$ is the process obtained from P by renaming any label occurrence according to μ. This guarantees that two closed configurations can only communicate via channels.

Note that we do not define the parallel composition of an open and a closed configuration. The idea is that we may combine two different parts of a system (in the construction stage) or two completed systems (for interaction), but we may not combine a system under construction with a completed one.

The derivation rules for the operational semantics are shown in Figure E.1. When an input occurs, a set of resources becomes available ; while an output requires a set of resources in order to occur, then the used resources disappear.

$$\frac{a \in A}{\mathbf{0} \triangleright !a} \text{ (ax)} \qquad \frac{T \triangleright !a}{\mathbf{Rec}T \triangleright !a} \text{ (rec)}$$

$$\frac{T \triangleright ?a}{\bar{a}_{\{r_1,\dots r_n\}}.T \triangleright !a} \text{ (outpref)} \qquad \frac{T \triangleright !a}{a^{\{r_1,\dots r_n\}}.T \triangleright ?a} \text{ (inpref)}$$

$$\frac{P \triangleright \Gamma' \qquad Q \triangleright \Gamma'' \qquad \forall a \in Dom(\Gamma') \cap Dom(\Gamma''),\ \Gamma'(a) \neq \Gamma''(a)}{(P \mid Q) \setminus (Dom(\Gamma') \cap Dom(\Gamma'')) \triangleright \Gamma' \odot \Gamma''} \text{ (par)}$$

$$\frac{P \triangleright \Gamma}{\wr P, S \int \triangleright \Gamma} \text{ (oconf)} \qquad \frac{P \triangleright \Gamma}{\langle P, S \rangle \triangleright \Gamma} \text{ (cconf)}$$

FIGURE E.2 – Handshake types

The other rules are quite standard. Note however that the operational distinction between open and closed configurations comes from the two distinct cases of composition given above. In the parallel composition of open configurations, one side may influence the other by modifying a shared resource, as no renaming takes place. This is not possible for closed configurations, as renaming prevents the sharing of resources.

A sequence of transitions $M_0 \xrightarrow{e_0} M_1 \dots M_n \xrightarrow{e_n} M$ is denoted $M_0 \xrightarrow{t}\!\!\!\!\rightarrow M$, where $t = e_0 \dots e_n$. The string t is called the *strong* trace of the sequence, while the *weak* trace is the restriction of t to the labels other than τ. Strong ($\sim$) and weak ($\approx$) bisimilarity are also defined as usual [Mil89] on the labeled transition system for CHC.

E.3.2 Typing system

A *type* Γ is a partial function from channel names to $\{!, ?\}$. We will use the shorthand notation $!a$ or $?a$ to describe a type defined on channel a, and commas to join types. We say that a is *active* in Γ when $\Gamma(a) =!$ and we say it is *passive* when $\Gamma(a) =?$.

Let Γ' and Γ'' be two types and let a be a channel. Let us define the function $\Gamma' \odot \Gamma'' : (Dom(\Gamma') \backslash Dom(\Gamma'')) \cup (Dom(\Gamma'') \backslash Dom(\Gamma')) \to \{!, ?\}$ such that :

- $\Gamma' \odot \Gamma''(a) = \Gamma'(a)$, when $a \in Dom(\Gamma') \backslash Dom(\Gamma'')$;
- $\Gamma' \odot \Gamma''(a) = \Gamma''(a)$, when $a \in Dom(\Gamma'') \backslash Dom(\Gamma')$.

Typing judgements are of the form $T \triangleright \Gamma,\ P \triangleright \Gamma,\ M \triangleright \Gamma$, where T is a thread, P a process, M a configuration and Γ a type. The typing rules are shown in Figure E.2. The empty thread is active : this models receptiveness, because a thread of passive type must always be able to perform another input. The following three rules guarantee that threads are alternating on each channel. The parallel composition of two processes is allowed only if threads on the same channel have dual types. These channels must then be restricted so that

no other process can communicate on them. This models the point-to-point communication discipline of handshake protocols. Note that resources do not play any role in the typing.

The following results show the intuition behind the typing system.

Lemma E.3.1 (Reduction)**.** *Let M be a configuration such that $M \rhd \Gamma$. Then :*

- $M \xrightarrow{a} \;\Leftrightarrow\; \Gamma(a) =?$;
- $M \xrightarrow{\bar{a}} \;\Rightarrow\; \Gamma(a) =!$;
- $M \xrightarrow{e} M' \wedge e \neq \tau \;\Rightarrow\; M' \rhd \Gamma'$ *s.t.* $Dom(\Gamma') = Dom(\Gamma) \wedge \forall b \in Dom(\Gamma), b \neq ch(e) \leftrightarrow \Gamma(b) = \Gamma'(b)$;
- $M \xrightarrow{\tau} M' \;\Rightarrow\; M' \rhd \Gamma$;

Corollary E.3.2 (Subject Reduction)**.** *Let $M \rhd \Gamma$ and $M \xrightarrow{s}\!\!\!\!\rightarrow M'$ then there is a type Γ' such that $M' \rhd \Gamma'$.*

E.3.3 Examples

As a first example, we show a configuration representing the OR handshake protocol.

$$OR = \langle a^{\{r_1\}}.\mathbf{Rec}\ \bar{a}_{\{r_2\}}.a^{\{r_1\}}.\mathbf{0} \mid \mathbf{Rec}\ \bar{b}_{\{r_1\}}.b^{\{r_2\}}.\mathbf{0} \mid \mathbf{Rec}\ \bar{c}_{\{r_1\}}.c^{\{r_2\}}.\mathbf{0}, \varnothing\rangle$$

We have that $OR \rhd ?a, !b, !c$. When a request on the passive port a arrives, the resource r_1 becomes available and this enables OR to send a request on either active port b, c. An acknowledge to this last request enables an acknowledge to the first one. The second configuration represents the MIX protocol.

$$MIX = \langle b^{\{k_1\}}.\mathbf{Rec}\ \bar{b}_{\{k_2\}}.b^{\{k_1\}}.\mathbf{0} \mid c^{\{k_1\}}.\mathbf{Rec}\ \bar{c}_{\{k_2\}}.c^{\{k_1\}}.\mathbf{0} \mid \mathbf{Rec}\ \bar{d}_{\{k_1\}}.d^{\{k_2\}}.\mathbf{0}, \varnothing\rangle$$

We have that $MIX \rhd ?b, ?c, !d$. Each time an environment request arrives (on either passive port b, c), the component MIX handshakes on its active port d and after completion it acknowledges to the first request. If, by the time the handshake on the active port is complete, the environment has sent a request on the other port, MIX chooses nondeterministically which request to acknowledge first.

The two protocols can be composed in parallel, communicating on the common ports. We have $(OR \mid MIX) \setminus \{b, c\} \rhd ?a, !d$.

E.3.4 Soundness

In this section we show that typed CHC configurations indeed define handshake languages, using the weak traces of the labeled transition semantics.

Each feature of the calculus plays a role in modeling the handshake discipline. Let us see informally how. First of all, handshake languages alternate input and output on the same port. This is enforced by the typing system. The reorder closure is guaranteed by the fact that different ports are on different parallel threads. The only reordering that is in general not allowed is when an

input blocks an output. An input can block an output because an output may need resources that will only be provided by the input. Finally, receptiveness is guaranteed by the fact that inputs do not need resources, and can always occur, provided that the alternation with the corresponding outputs is respected.

In order to denote handshake languages we consider the weak traces of the transition sequences of a configuration. If we considered the traces of all transition sequences, the denoted languages would always be prefix closed and some handshake languages would excape us. To characterize the larger class of languages closed under passive prefixes, we consider only the traces of the *quiescent* transition sequences.

A configuration M is *quiescent* if it cannot (weakly) perform an output, i.e. if there is no transition sequence of the form $M \stackrel{(\tau)^*}{\longrightarrow\!\!\!\!\!\rightarrow} \stackrel{\bar{a}}{\longrightarrow}$, for any channel a. A transition sequence $M \stackrel{t}{\longrightarrow\!\!\!\!\!\rightarrow} M'$ is *quiescent* if M' is.

Definition E.3.3. Let M be a handshake configuration. We define $HL(M)$ to be the set of weak traces of all the quiescent transition sequences which start from M.

Let M be a configuration and Γ a type such that $M \rhd \Gamma$. The handshake structure $HS(\Gamma) = \langle Ports_\Gamma, d_\Gamma \rangle$ is defined by setting $Ports_\Gamma = Dom(\Gamma)$ and $d_\Gamma = \Gamma$.

Proposition E.3.4 (Soundness)**.** *Let M be a handshake configuration, such that $M \rhd \Gamma$. Then $HL(M)$ is a handshake language on the handshake structure $HS(\Gamma)$.*

We observe, however, that the other direction, the fact that each language is the denotation of some configuration, cannot be established. This is due to the presence of non recursive handshake languages which could never be captured by finite configurations. It would still be interesting to characterize the class of handshake languages that correspond to CHC configurations. We leave this as future work.

E.3.5 Compositionality

Open configurations can communicate via shared resources, but this is not directly observable in the labeled transition semantics. Thus we cannot expect a labeled equivalence to be fully congruent for them. However weak bisimilarity is a congruence with respect to composition of closed configurations :

Proposition E.3.5. *Let M_1, M_2, N be closed handshake configurations such that $M_1, M_2 \rhd \Gamma$, $N \rhd \Gamma'$ and $(M_1 \mid N) \setminus \Delta \rhd \Gamma \odot \Gamma'$, $(M_2 \mid N) \setminus \Delta \rhd \Gamma \odot \Gamma'$, where $\Delta = Dom(\Gamma) \cap Dom(\Gamma')$. Then $M_1 \approx M_2$ implies $(M_1 \mid N) \setminus \Delta \approx (M_2 \mid N) \setminus \Delta$.*

This is consistent with our intepretation of resources as *internal* means of communication. Our main goal was to describe the *externally observable* behavior of a system and we do so by considering only those configurations whose resources cannot be accessed by the environment.

In Section E.2 we talked about the difficulty of finding a good definition of composition for handshake languages. This intuition is confirmed as "quiescent trace equivalence" is not a congruence. Consider the following processes :

$$P_1 = \bar{c}_{\{r_1,r_2\}}.c^{\{\}}.\mathbf{0} \mid \bar{b}_{\{r_3,r_1\}}.b^{\{r_1\}}.\mathbf{Rec}\,\bar{b}_{\{r_1\}}.b^{\{r_1\}}.\mathbf{0} \mid (\bar{d}_{\{r_3,r_2\}}.d^{\{r_3\}}.\mathbf{0} \mid d^{\{\}}.\bar{d}_{\{\}}.d^{\{\}}.\mathbf{0})\backslash\{d\}$$

$$P_2 = \bar{c}_{\{r_1\}}.c^{\{\}}.\mathbf{0} \mid \mathbf{Rec}\ \bar{b}_{\{r_1\}}.b^{\{r_1\}}.\mathbf{0}\ .$$

Consider the closed configurations $M_1 = \langle P_1, \{r_1, r_2, r_3\}\rangle$ and $M_2 = \langle P_2, \{r_1, r_2, r_3\}\rangle$. They are both interpreted as the same handshake language :

$$HL(M_1) = \{\bar{c}, \bar{c}c, \bar{b}, \bar{b}b\bar{c}, \bar{b}b\bar{c}c, \bar{b}b\bar{b}, \ldots\} = HL(M_2)$$

however, if we compose them with $N = \langle b^{\{\}}.\mathbf{Rec}\,\bar{b}_{\{\}}.b^{\{\}}.\mathbf{0} \mid c^{\{k\}}.\mathbf{0} \mid \bar{a}_{\{k\}}.a^{\{\}}.\mathbf{0},\ \varnothing\rangle$ we obtain two configurations with different interpretations :

$$HL((M_1 \mid N) \setminus \{b, c\}) = \{\varepsilon, \bar{a}, \bar{a}a\} \qquad HL((M_2 \mid N) \setminus \{b, c\}) = \{\bar{a}, \bar{a}a\}$$

Therefore the parallel composition of CHC configurations cannot be used to define the composition of handshake languages. In order to compose handshake protocols, some more knowledge on the branching structure is needed. CHC provides a suitable formalism to study this structure.

E.4 Handshake Petri nets

We now present an alternative model of the Handhsake protocol, based on Petri nets. We will assume some basic knowledge on Petri nets, which we will present in their standard graphical representation [Rei85]. We will consider Petri nets in their *unsafe* version, where places are allowed to contain several tokens at the same time. This is not just for convenience. Unsafe nets are necessary to carry out our construction. We also stress that the nets we consider are in general not finite, in the sense that they may have infinitely many places and/or transitions. However, the correspondence with the process calculus will be valid only for finite nets.

E.4.1 Definition

Handshake Petri nets are Petri nets with a special "external interface", reflecting the structure of handshake ports. We define handshake ports in two phases. We first define the static structure of ports, and then we specify the markings.

Let G be a Petri net and let N_O and N_I be a partition of its nodes (places and transitions). The elements of N_O will be called *output transitions / places* while the elements of N_I will be called *input transitions / places*. We give an inductive definition of a *static handshake port* $a = \langle G, N_O, N_I\rangle$ as follows :

- (Basic cases) N_O and N_I contain no transition ;
- (Inductive cases) let $a' = \langle G', N'_O, N'_I\rangle$ be a static port :

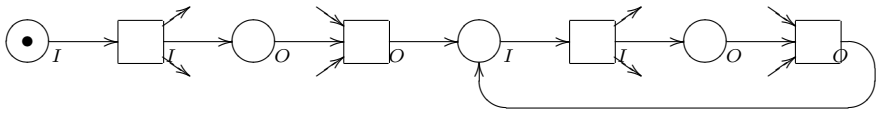


FIGURE E.3 – A passive port (I is for input and O is for output)

- (input prefixing) given a place $p \in N'_I$ with no outgoing arcs, a is obtained from a' by adding an input transition t and an arc from p to t ;
- (output prefixing) given a place $p \in N'_O$ with no outgoing arcs and a place $p' \in N'_I$ with exactly one outgoing arc, a is obtained from a' by adding an output transition t, an arc from p to t and an arc from t to p' ;
- (alternation) given a place $p \in N'_O$ and a transition $t \in N'_I$ with no outgoing arcs, a is obtained from a' by adding an arc from t to p.

Let a' be a static port and let p be a place of (the Petri net of) a' such that if p is an input place, p has an outgoing arc. Let a be the net obtained from a' either by adding one token to p or by keeping a' with no tokens, then a is a *handshake port*. Moreover, if a is as a' (no tokens) or if p is an output place we say that a is an *active port*, otherwise we say that a is a *passive port*.

Let G be a Petri net, G' a subgraph of G and $a = \langle G', N_O, N_I \rangle$ a port s.t. :

- a place of G' may only be connected to transitions of G' ;
- a transition $t \in N_O$ of G' may only have outgoing arcs to places of G' ;
- a transition $t \in N_I$ of G' may only have incoming arcs from places of G'.

then a is a *handshake port of* G. Figure E.3 shows an example of a passive handshake port of some net. The arrows without source or target indicate the way the port may connect to the rest of the net. The statical structure imposes alternation between the firings of input and output transitions. It is also ensured that, if an input place may ever contain a token, then it must have an outgoing arc to an input transition (receptiveness). Finally, by allowing a port to contain several input and output transitions we are able to model each event separately. For instance, two distinct input transitions may connect differently to the rest of the net, thus providing different resources.

Definition E.4.1. The pair $H = \langle G_H, Ports_H \rangle$ is a *handshake Petri net (hpn)* just when G_H is a Petri net and $Ports_H$ a set of disjoint handshake ports of G_H.

Let $H = \langle G_H, Ports_H \rangle$ and let t (p) be a transition (place) of G_H. Then t (p) is *internal* of H if it is neither of input nor of output (in some port a of H).

E.4.2 Composition

A *linkage* between an active port $a \in Ports_H$ and a passive port $b \in Ports_H$ of an hpn $H = \langle G_H, Ports_H \rangle$ is the hpn $L(H, a, b) = \langle G_{H,a,b}, Ports_H \backslash \{a, b\} \rangle$,

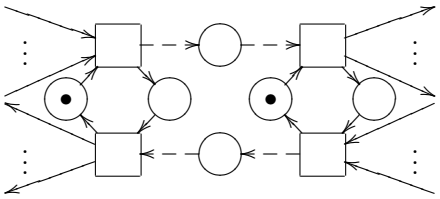


FIGURE E.4 – Example of composition of ports

where $G_{H,a,b}$ is the net obtained by adding two fresh places p_1 and p_2 to G_H and arcs from each output transition of a to p_1, from p_1 to each input transition of b, from each output transition of b to p_2 and from p_2 to each input transition of a.

We call *link* of $L(H, a, b)$, denoted $link(H, a, b)$, the graph consisting of the graphs of a and b plus p_1, p_2 and any arc connecting them to transitions of a or b. Figure E.4 shows an example of link between an active port (left) and a passive port (right).

Definition E.4.2. Let $H_1 = \langle G_1, Ports_1 \rangle$ and $H_2 = \langle G_2, Ports_2 \rangle$ be two handshake Petri nets. Let $\{a_1, \dots a_n\} \subseteq Ports_1 \cup Ports_2$ be a set of active handshake ports and let $\{b_1, \dots b_n\} \subseteq Ports_1 \cup Ports_2$ be a set of passive handshake ports, such that for $1 \leq i \leq n$, $a_i \in Ports_1$ if and only if $b_i \in Ports_2$. Then :

$$H_1 \parallel_{\{(a_1,b_1),\dots(a_n,b_n)\}} H_2$$
$$=$$
$$L(\dots L(\langle G_1 + G_2, Ports_1 \cup Ports_2 \rangle, a_1, b_1), \dots a_n, b_n)$$

is the composition of H_1 with H_2 by linking the pairs (a_1, b_1), $\dots (a_n, b_n)$.

It is easy to see that the composition of two hpns is well-defined and associative.

E.4.3 Handshake marked graphs

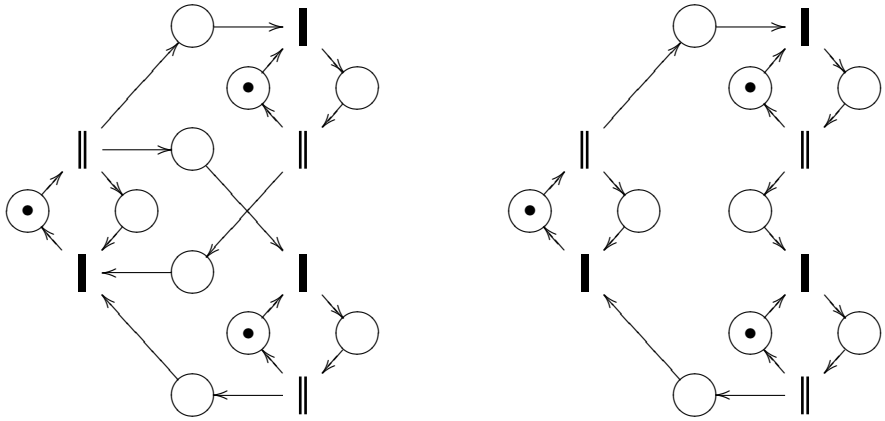
In the first stage we focus on marked graphs [Pet81], which are Petri nets where each place has at most one incoming and at most one outgoing arc.

Marked graphs are significant as they allow to identify places with communication channels and in turn to represent all and only circuits which can be built out of channel, synchronization and fork operations. Moreover they have a special historical importance for the handshake protocol [CS72b]. We call *handshake marked graph* a handshake Petri net which is also a marked graph.

Examples

Marked graphs represent the core of determinism. In particular they allow the representation of most deterministic handshake components : $STOP$, RUN,

CON, SEQ, PAR, PAS, $JOIN$ (in the notation of [VB93]). Two of these components are represented below [3].



PAR (left) waits for a request on its passive port and then starts two handshakes in parallel on its active ports. Only after successful termination of both it acknowledges to the first request. SEQ (right) also waits for a request on its passive component, but then it starts its active ports in sequence, before finally acknowledging to the initial request.

The examples show that handshake marked graphs (or marked graphs in general) always react in the same way to a given stimulus. For example, SEQ always sends a request on its first active port after the reception of a request on its passive port. It can be shown that handshake marked graphs embed a particular subclass of handshake languages where each pair stimulus/response can be seen as a couple of brackets in the language and each trace becomes well-bracketed with respect to any of these couples, after a fixed number of closing brackets.

E.4.4 Deterministic extensions

Marked graphs express only deterministic behaviors but not all deterministic behaviors are captured by marked graphs. As far as we know, no structural characterization of determinism in Petri nets exists in the literature. We propose a definition that completely characterizes determinism in the context of handshake nets.

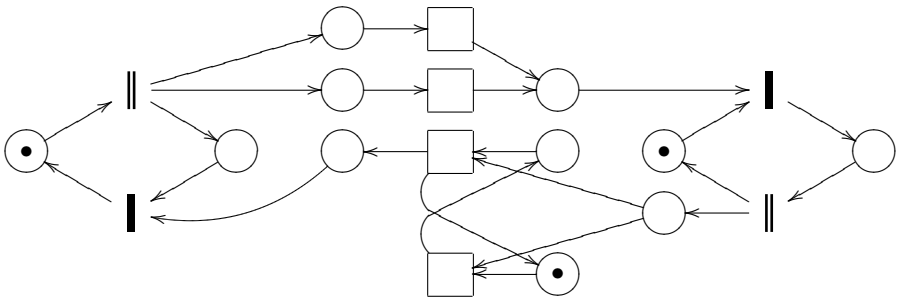
Definition E.4.3. A *handshake deterministic-branching net* (a *handshake DB net*, or just a *DB net*, for short) is a handshake Petri net in which every place p with several outgoing arcs is such that :

3. Although handshake Petri nets are formalized here for the first time, a similar representation for both components had already been given as far back as [CS72a]. The pictures there were an inspiration for our work.

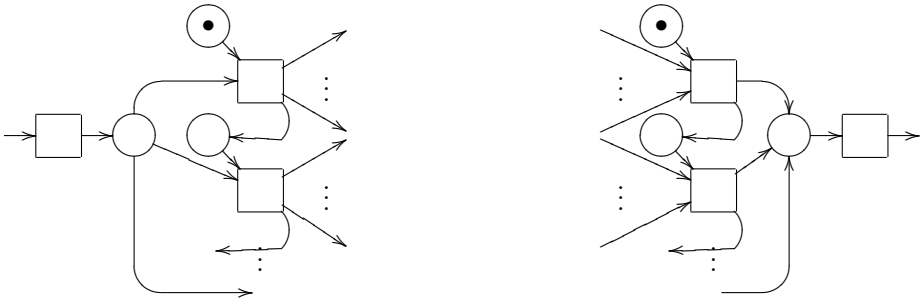
- Each post-transition t of p has a "guard", a place whose only post-transition is t;
- Exactly one of the guards of p's post-transitions initially contains a token, so that at most one post-transition may initially be enabled;
- Each of p's post-transitions has exactly one outgoing arc to some guard of a post-transition of p, so that each time one post-transition has fired one post-transition may be enabled;
- Each guard of a post-transition of p may have incoming arcs only from p's post-transitions, so that no more than one post-transition may ever be enabled.

Examples

As an example, consider $COUNT_N$ which, after reception of a request on its passive port, handshakes N times on its active port. Then it acknowledges to the first request and returns to wait for an activation. In this case, the circuit needs to decide (deterministically, of course) when to acknowledge to a passive request (after N handshakes on the active port). Here is the circuit $COUNT_2$, also known as DUP :



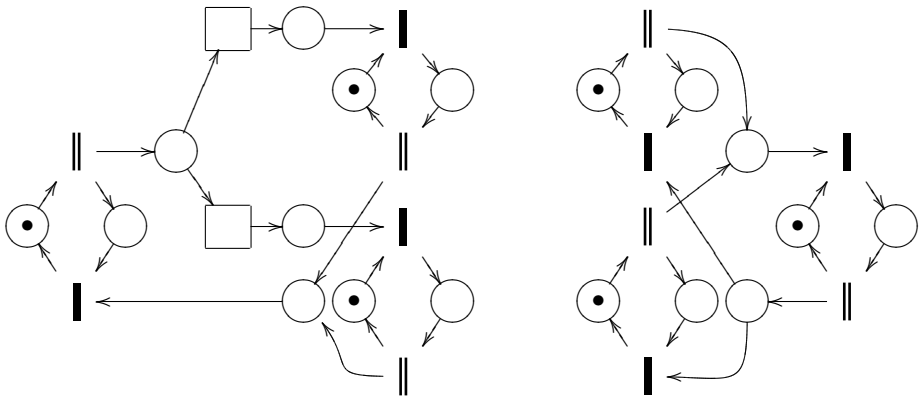
A DB structure allows us to select each firing of a given transition and associate it to a brand new dedicated transition, as shown below :



We call the above *input* (left) and *output* (right) *occurrence selecters*.

E.4.5 General nets

In this subsection we present two examples of nondeterministic nets, the OR (below left) and the MIX (below right) handshake components :



OR has a passive and two active ports. When a request on the passive port arrives a request on either active port is sent. Acknowledge to this last request enables an acknowledge to the first one. As the picture shows, this example can be modeled by a *free choice* net (transitions with a shared precondition do not have any preconditions other than that).

Conversely, MIX has two passive and one active port. Each time an environment request arrives (on either passive port) MIX handshakes on its active port and after completion it acknowledges to the first request. If by the time the handshake on the active port completes the environment had sent a request on the other port, MIX chooses nondeterministically which request to acknowledge first.

This situation could not be described with a free-choice net since the choice of which request to acknowledge may not be a choice at all if the environment only sent one request.

E.4.6 Soundness and completeness

Consider an hpn $H = \langle G, I, O \rangle$, name its ports and let P_H be the set of these names. Now take $d_H : P_H \to \{act, pas\}$, the function which maps each port name p to the appropriate label, *act* if port p is active in H, *pas* if it is passive. This allows us to define the handshake structure $HS(H) = \langle P_H, d_H \rangle$.

Then for any port p, name p ($\bar{p}$) its input (output) transition, and name τ any internal transition. An execution t of H is *quiescent* when for no $p \in P_H$ it can be extended as $H \xrightarrow{t} \xrightarrow{(\tau)^*} \xrightarrow{\bar{p}}$. We define $HL(H)$ as the set of strings consisting of the external restriction of each quiescent execution of H.

The soundness and completeness of the Petri nets model, can be respectively formalized by the following theorems.

Theorem E.4.4. *Let H be a handshake Petri net, then $HL(H)$ is a handshake language on the handshake structure $HS(H)$.*

Theorem E.4.5. *Let σ be a handshake language on a handshake structure $\langle P, d\rangle$. Then there is an hpn H_σ such that $HS(H_\sigma) = \langle P, d\rangle$ and $HL(H_\sigma) = \sigma$.*

The proof of soundness is a rather straightforward verification of the properties defining a handshake language. In the remaining of this section, we try to hint the proof of completeness (theorem E.4.5). We must warn that the construction of H_σ we propose may lead to an infinite net, but let us also point out that an infinite representation is in general unavoidable. An example of language with no finite representation is the one which contains an infinite chain of (finite) traces where outputs are chosen according to a non recursive function.

Let us focus first on positional handshake languages. These languages make their choices according to the reached position, regardless of the particular interleaving followed in the execution. In the following we write p_i ($\bar{q}_i$) for the ith occurrence of input p (output $\bar{q}$). Then we can represent a *choice* as a pair made of a position $[s]_\sim$ and an output occurrence or a special symbol $*$, where $\langle [s]_\sim, \bar{q}_i\rangle$ expresses the choice of "playing" $\bar{q}_i$ at $[s]_\sim$ and $\langle [s]_\sim, *\rangle$ the choice of doing nothing at $[s]_\sim$.

Let σ be a handshake language and c a choice in σ. Let $t \in \sigma^{\leq}$, we say that t *allows* c in σ ($t \rightarrow_\sigma c$) when $[t]_\sim = fst(c)$ and :

- $t \in \sigma$ if $snd(c) = *$;
- $t \cdot snd(c) \in \sigma^{\leq}$ otherwise.

We say that t *prevents* c in σ ($t \nrightarrow_\sigma c$) when it does not allow it.

If we consider positional strategies we see immediately that positions, rather than traces, allow choices. Moreover, since we only consider data-less communications and since outputs do not affect choices (by reordering), a position can be represented by a set of occurrences of distinct input messages, taking the last input occurrence of each thread.

Starting from the above observations and systematically using the selecter structures introduced in Section E.4.4 to select occurrences of input and output messages we are able to construct a handshake Petri net which corresponds to the given positional handshake language.

Proposition E.4.6. *Let σ be a positional handshake language on $\langle P, d\rangle$. Then there is an hpn H_σ such that $HS(H_\sigma) = \langle P, d\rangle$ and $HL(H_\sigma) = \sigma$.*

Since the construction associates single occurrences to transitions and since a move may occur infinitely many times, the constructed graph H_σ is in general infinite.

In the non-positional case, reshuffling the threads of a trace may affect a choice. We first define an *atomic reshuffling* of a trace $t(= t' \cdot m \cdot n \cdot t'')$ as a trace of the form $t' \cdot n \cdot m \cdot t''$, for m and n independent messages.

Definition E.4.7. Let S be a set of pairs of the form $\langle p_i, \bar{q}_j\rangle$. S is *critical* for a choice c in a handshake language σ just when, for all $t \in \sigma^{\leq}$ such that

$[t]_\sim = fst(c)$,

$$\forall \langle p_i, \bar{q}_j \rangle \in S, t = t' \cdot \bar{q}_j \cdot t'' \cdot p_i \cdot t''' \;\Rightarrow\; t \nrightarrow_\sigma c.$$

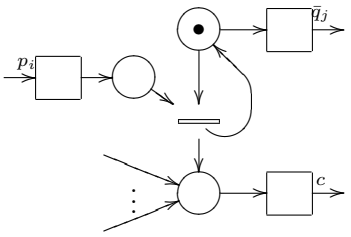
We write $crit_\sigma(c)$ for the set of minimal critical sets for c in σ.

A similar notion is that of *critical pair* (the above being critical set of pairs) for c in σ : a pair $\langle p_i, \bar{q}_j \rangle$ for which there is $s \in \sigma$ such that $s = s' \cdot \bar{q}_j \cdot p_i \cdot s'' \nrightarrow_\sigma c$ while $s' \cdot p_i \cdot \bar{q}_j \cdot s'' \rightarrow_\sigma c$. If $\langle p_i, \bar{q}_j \rangle$ is a critical pair for c in σ, we say that it is *inverted* in t if and only if $t = t' \cdot \bar{q}_j \cdot t'' \cdot p_i \cdot t'''$.

Lemma E.4.8. *Let c be a choice in a handshake language σ. Let $t \in \sigma^{\leq}$, $[t]_\sim = fst(c)$:*

$$t \nrightarrow_\sigma c \;\Leftrightarrow\; \exists S \in crit_\sigma(c), \forall \langle p_i, \bar{q}_j \rangle \in S, t = t' \cdot \bar{q}_j \cdot t'' \cdot p_i \cdot t'''$$

The construction of the net is a modification of the one for positional handshake languages, as we briefly sketch here. For any minimal critical set S for c in σ and for all $\langle p_i, \bar{q}_j \rangle \in S$ we connect transitions p_i, $\bar{q}_j$ and c as follows :



Suppose that the jth firing of $\bar{q}$ occurs before the ith firing of p, and similarly for all the other pairs in S (represented in the picture by the other incoming arcs in the precondition of transition c). Then c is clearly prevented. Note that this scheme works for both a choice to output and a choice not to, as each choice corresponds to a transition in the graph.

The completeness theorem, specializes to several classes of nets and languages. For instance to the deterministic case.

Theorem E.4.9. *Let H be a handshake DB net, then $HL(H)$ is a deterministic handshake language on the handshake structure $HS(H)$. Conversely, if σ is a deterministic handshake language on $\langle P, d \rangle$, there is a DB net H_σ such that $HS(H_\sigma) = \langle P, d \rangle$ and $HL(H_\sigma) = \sigma$.*

As we mentioned, marked graphs correspond to a particular class of languages too, the *well-bracketed* ones. In this case there is even a construction yielding finite graphs. We still do not know any independent characterization of the nets that correspond to positional languages.

E.5 Comparing the two models

In this section we relate the calculus CHC with its Petri nets model. We show a strict correspondence between the processes of the calculus and the finite Handshake nets.

E.5.1 Petri Net semantics of the calculus

Let M be a handshake configuration, such that $M \rhd \Gamma$. We can assume $M = \langle P, S\rangle$ and define the hpn $[\![M]\!]_\Gamma$ by cases of P. The definitions for open configurations are identical, $[\![\wr P, S \wr]\!]_\Gamma = [\![\langle P, S\rangle]\!]_\Gamma$.

Let $P = \mathbf{0}$. Then $\Gamma = !a$ for some channel a. Now, define a port which contains a single output place p holding a token and call it a as the channel. Let G be the Petri net which contains p plus an internal place q, labeled r, for each $r \in S$, where q contains as many tokens as there are occurrences of r in S. Then $[\![M]\!]_\Gamma$ is the hpn $\langle G, \{a\}\rangle$.

Let $P = a^{\{r_1,\dots r_n\}}.P'$. Then the last applied typing rule is (inpref), then $\Gamma = ?a$ and $P' \rhd !a$. Let $[\![\langle P', S\rangle]\!]_{!a} = \langle G', Ports'\rangle$. By construction, $Ports' = \{a\}$ where a is an active port. Let p' be the place of a with a token. Let's extend a by adding a fresh input place p, a fresh input transition t labeled a and arcs from p to t and from t to p', then by removing the token from p' and putting a token into p. Finally let's add arcs from t to any place labeled r_i, for $1 \le i \le n$. If any of these places does not exist yet, add it anew. We thus obtain a graph G. Then $[\![M]\!]_\Gamma = \langle G, \{a\}\rangle$. The case of the output prefix is dual.

Let $P = \mathbf{Rec}P'$. Then $\Gamma = !a$ and $P' \rhd !a$, for some channel a. Let $[\![\langle P', S\rangle]\!]_{!a} = \langle G', Ports'\rangle$. By construction, $Ports' = \{a\}$ where a is also the active port associated to channel a. Let p be the place of a which holds a token. If p has an incoming arc or if any other place in a has two incoming arcs, $[\![\langle P, S\rangle]\!]_\Gamma = [\![\langle P', S\rangle]\!]_{!a}$. Otherwise a must contain a place p' with no outgoing arcs, by construction. Note that both p and p' must be output places, also by construction. Then replace p and p' by a place q obtained by "joining" them. In particular, q must be the new source of p's outgoing arc and the new target of p''s incoming arc. Call G the graph so obtained. Then $[\![\langle P, S\rangle]\!]_\Gamma = \langle G, \{a\}\rangle$.

Let $P = (P' \mid P'') \setminus \Delta$. We construct $[\![\langle P, S\rangle]\!]_\Gamma$ in three steps. First let $[\![\langle P', S\rangle]\!]^\Delta_{\Gamma_1}$ be obtained from $[\![\langle P', S\rangle]\!]_{\Gamma_1}$ by renaming each port a such that $(a, a) \in \Delta$, as $a^!$ when $\Gamma_1(a) = !$ and as $a^?$ when $\Gamma_1(a) = ?$. Define analogously $[\![\langle P'', S\rangle]\!]^\Delta_{\Gamma_2}$. Then let $\langle G', Ports\rangle = [\![\langle P', S\rangle]\!]^\Delta_{\Gamma_1} \parallel_{\{(a^!,a^?)\mid a\in\Delta\}} [\![\langle P'', S\rangle]\!]^\Delta_{\Gamma_2}$. Then, for any two distinct places p and p' of G' labeled by the same resource r do the following : substitute p and p' by a single place also labeled by r, having all the arcs of both p and p' ; note also that by construction, p and p' contained the same number of tokens k, then put k tokens in the new place as well. Let G be the net so obtained. Then $[\![M]\!]_\Gamma = \langle G, Ports\rangle$.

E.5.2 Full abstraction and definability

The semantics is well defined and fully abstract with respect to weak bisimilarity :

Lemma E.5.1. *Let M be a configuration such that $M \rhd \Gamma$. Then $[\![M]\!]_\Gamma$ as defined above is a handshake Petri net and $M \approx [\![M]\!]_\Gamma$.*

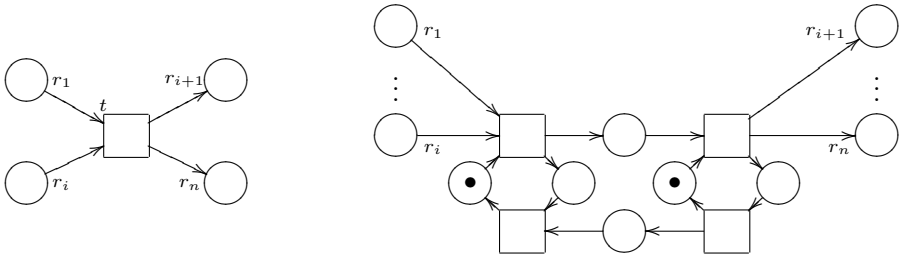
Theorem E.5.2 (Full Abstraction)**.** *Let M and M' be two configurations such that $M \rhd \Gamma$ and $M' \rhd \Gamma'$. Then $M \approx M' \Leftrightarrow [\![M]\!]_\Gamma \approx [\![M']\!]_{\Gamma'}$.*

For each finite hpn there is a weakly bisimilar handshake configuration :

Theorem E.5.3 (Definability)**.** *Let $H = \langle G, Ports\rangle$ be a handshake Petri net. Then there are a closed handshake configuration M and a handshake type Γ, such that $M \rhd \Gamma$ and $[\![M]\!]_\Gamma \approx H$.*

We present here a simplified construction of the configuration associated to H. The idea is that each port a of the net H can be modeled by a thread $Proc(a, H)$, inductively on the structure of the port.

Each internal transition t is first *unfolded* as a link between two ports and then associated to a process. Let t have incoming arcs from internal places labeled $r_1, \dots r_i$ and outgoing arcs to internal places labeled $r_{i+1}, \dots r_n$. Then t is unfolded as follows :



where $r_1, \dots r_n$ are place labels. . Then let H be a hpn, $u(H)$ is the hpn obtained from H by unfolding each of its internal transitions. It can be shown that $H \approx u(H)$.

For each internal transition t, let l_t be a fresh label associated to it. Consider the following process.

$$Proc(t, H) = (\mathbf{Rec}\ \bar{l}_{t\{r_1,\dots r_i\}}.l_t.\mathbf{0} \mid l_t^{\{r_{i+1},\dots r_n\}}.\mathbf{Rec}\ \bar{l}_t.l_t^{\{r_{i+1},\dots r_n\}}.\mathbf{0}) \setminus \{l_t\}$$

Then we define

$$Proc(H) = Proc(a_1, H) \mid \dots \mid Proc(a_n, H) \mid Proc(t_1, H) \mid \dots \mid Proc(t_m, H)$$

where $a_1, \dots a_n$ are the ports of H and $t_1, \dots t_m$ are the internal transitions of H. Then let $Conf(H) = \langle Proc(H), S_H\rangle$, where S_H is the multiset of labels of internal places of H with a token and a label appears in S_H as many times as the number of tokens in the corresponding place. Finally let $ch(Ports)$ be the set of names of ports in $Ports$, then $\Gamma_H : ch(Ports) \to \{!, ?\}$ is the function which associates ! to its active ports' names and ? to its passive ports' names. It can be shown that $[\![Conf(H)]\!]_{\Gamma_H} \approx u(H)$. Thus $[\![Conf(H)]\!]_{\Gamma_H} \approx H$.

E.6 Proofs

E.6.1 Proofs from Section E.2

We state two definitions which will be useful in the proofs which follow.

Definition E.6.1. Let s and t be two handshake traces on a given handshake structure, such that $s \sim t$. We define $d_\sim(s,t)$, the *homotopy distance* between s and t as follows :

- $d_\sim(s,t) = 0 \Leftrightarrow s = t$
- $d_\sim(s,t) = 1 \Leftrightarrow (s = s' \cdot m \cdot n \cdot s'') \wedge (t = s' \cdot n \cdot m \cdot s'')$, for two messages m and n ;
- In general $d_\sim(s,t) = k > 0$ if and only if $s \neq t$ and there is a sequence of $k+1$ (and no less) traces $s_0, s_1, \ldots s_k$ such that $s = s_0$, $s_k = t$ and for all $0 \leq i < k$, $d_\sim(s_i, s_{i+1}) = 1$.

The second definition is less significant, but still useful. Let H be an hpn and t and t' two distinct external transitions of H. We call t' the *complement* of t, and viceversa, when t and t' belong to the same port.

Proof of Proposition E.2.4 Let σ be a deterministic handshake language and let $s, t \in \sigma^{\leq}$, such that $s \sim t$. Suppose that $s \cdot \bar{s} \in \sigma^{\leq}$. We show that $t \cdot \bar{s} \in \sigma^{\leq}$ by induction on $d = d_\sim(s,t)$:

- $d = 0$. $s = t$, then $t \cdot \bar{s} \in \sigma^{\leq}$;
- $d = l+1$. There is a sequence $s = t_0, \ldots t_{l+1} = t$ such that $d_\sim(t_i, t_{i+1}) = 1$ and $t_i \cdot \bar{s} \in \sigma^{\leq}$, $\forall 0 \leq i \leq l$. Let $t_l = t' \cdot m \cdot n \cdot t''$ and $t_{l+1} = t' \cdot n \cdot m \cdot t''$. If m is an output or n an input, $t_{l+1} \cdot \bar{s}$ **r** $t_l \cdot \bar{s}$, which implies $t_{l+1} \cdot \bar{s} \in \sigma^{\leq}$. Let m be an input and n an output, then we prove $t_{l+1} \cdot \bar{s} \in \sigma^{\leq}$ by induction on the length of $\bar{s}$.
  - $\bar{s} = \varepsilon$. Then $t_{l+1} \cdot \varepsilon = t_{l+1} \in \sigma^{\leq}$ by hypothesis ;
  - $\bar{s} = \bar{s}'a$. $t_{l+1} \cdot \bar{s}' \in \sigma^{\leq}$ by hypothesis. If a is an input, $t_{l+1} \cdot \bar{s}' \cdot a \in \sigma^{\leq}$ by receptivity. Then let a be an output and let $a_1, \ldots a_k$ be all the outputs that σ can send at $t_{l+1} \cdot \bar{s}'$. Then determinism implies $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k \in \sigma^{\leq}$. Note also that if an output was possible at $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$, it would also be possible at $t_{l+1} \cdot \bar{s}'$, by reordering. Then $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$ is passive in σ. Then also $t_l \cdot \bar{s}' \cdot a_1 \ldots a_k \in \sigma$, as it reorders $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$. Then $a \in \{a_1, \ldots a_k\}$, as a is an output and $t_l \cdot \bar{s}' \cdot a \in \sigma^{\leq}$. Then $t_{l+1} \cdot \bar{s}' \cdot a \in \sigma^{\leq}$.

The proof that $s \in \sigma \Rightarrow t \in \sigma$ is even simpler, as the outer induction alone will do. □

E.6.2 Proofs from Section E.3

Proof of Lemma E.3.1 Let $M = \langle P, S \rangle$. We will prove it by cases of P :

- Let $P = \mathbf{0}$. Then P is not reducible and the premises of the left to right implications are false. Note also that if $\Gamma(a)$ is defined, it must be !. Then even in the right to left implication (first statement) the premise is false ;

- Let $P = act.P'$. Let act be an input action $act = a^{\{r_1,\dots r_n\}}$. Then $\langle P, S\rangle \xrightarrow{a} \langle P', S+\{r_1,\dots r_n\}\rangle$. The last applied derivation rule is *(inpref)*, so we have $\Gamma = ?a$ as type for P and $P' \rhd !a$ as premise. The case where act is an output action is just symmetric, so we skip it. On the other hand, let $\Gamma(a) = ?$. Then act is an input action $act = a^{\{r_1,\dots r_n\}}$ and $\langle P, S\rangle \xrightarrow{a}$;
- Let $P = (P' \mid P'') \setminus \Delta$. The last applied derivation rule is *(par)*. Then the premises are $P' \rhd \Gamma'$ and $P'' \rhd \Gamma''$, where $\Delta = Dom(\Gamma') \cap Dom(\Gamma'')$ and $\Gamma = \Gamma' \odot \Gamma''$. Let $\Gamma(a) = ?$, then by definition of $\odot$, either $\Gamma'(a) = ?$ or $\Gamma''(a) = ?$. Let $\Gamma'(a) = ?$, the other case is symmetric. By induction hypothesis $\langle P', S'\rangle \xrightarrow{a}$, for any multiset of labels S'. Since $\Gamma(a) = ?$, $a \notin \Delta$ and $\langle P, S\rangle \xrightarrow{a}$. Now let $\langle P, S\rangle \xrightarrow{a}$. Either $\langle P', S'\rangle \xrightarrow{a}$ or $\langle P'', S''\rangle \xrightarrow{a}$. Say $\langle P', S'\rangle \xrightarrow{a}$, the other case is symmetric. By induction hypothesis $\Gamma'(a) = ?$. Also, since $\langle P, S\rangle \xrightarrow{a}$, a may not be in Δ. Then $\Gamma(a) = ?$. Moreover Q (the reduct of P) must be of the form $(Q' \mid P'') \setminus \Delta$, where $\langle P', S'\rangle \xrightarrow{a} \langle Q', S'_Q\rangle$. By induction hypothesis $Q' \rhd \Gamma'_Q$, where $Dom(\Gamma') = Dom(\Gamma'_Q)$ and for all $b \in Dom(\Gamma')$, $b \neq a \leftrightarrow \Gamma'(b) = \Gamma'_Q(b)$. Then $\Gamma_Q = \Gamma'_Q \odot \Gamma''$ is defined and such that for all $b \in Dom(\Gamma)$, $b \neq a \leftrightarrow \Gamma(b) = \Gamma_Q(b)$. The proofs of the third and fourth points follow the same reasoning. So, let $\langle P, S\rangle \xrightarrow{\tau} \langle Q, S_Q\rangle$. There are four cases but they come in symmetric pairs, so we are really left with two of them. Let $Q = (Q' \mid P'')\backslash$, where $\langle P', S'\rangle \xrightarrow{\tau} \langle Q', S'_Q\rangle$. By induction hypothesis $Q' \rhd \Gamma'$, which implies $Q \rhd \Gamma$. Finally let $Q = (Q' \mid Q'') \setminus \Delta$, where $\langle P', S'\rangle \xrightarrow{a} \langle Q', S'_Q\rangle$ and $\langle P'', S''\rangle \xrightarrow{\bar{a}} \langle Q'', S''_Q\rangle$. Let $Q' \rhd \Gamma'_Q$ and $Q'' \rhd \Gamma''_Q$. By induction hypothesis, for all $b \in Dom(\Gamma')$ ($b \in Dom(\Gamma''_Q)$), $b \neq a$ if and only if $\Gamma'(b) = \Gamma'_Q(b)$ ($\Gamma''(b) = \Gamma''_Q(b)$). Then $\Gamma'_Q \odot \Gamma''_Q = \Gamma$ and $Q \rhd \Gamma$;
- Let $P = \mathbf{Rec}\ T$. The last applied derivation rule is *(rec)*, so that the type in the conclusion is $\Gamma = !a$ and the premise is $T \rhd !a$. P may reduce to Q if and only if $T \cdot \mathbf{Rec}\ T$ may reduce to Q, if and only if T may reduce to T', where $Q = T' \cdot \mathbf{Rec}T$. By induction hypothesis, this must be an output reduction $T \xrightarrow{\bar{a}} T'$, since $T \rhd !a$. Then $P \xrightarrow{\bar{a}} Q$. As we said $\Gamma(a) = !$. Also, it is easy to show that Q must have the same type T' has, which by induction hypothesis is $?a$. Then $\Gamma_Q = ?a$.

□

Proof of Proposition E.3.4 we could give a direct proof, but It also follows from the full abstraction result (Theorem E.5.2) and from the analogous result proven for hpns (Theorem E.4.4). □

Proof of Proposition E.3.5 Let us define the relation

$$\mathcal{R} = \{((M' \mid N)\backslash\Delta, (M'' \mid N)\backslash\Delta) \mid M' \approxeq M'' \wedge (res(M') \cup res(M'')) \cap res(N) = \varnothing\}$$

We claim $\mathcal{R}$ is a bisimulation. Let $M_1 \xrightarrow{e} M'_1$, where $M_1 \approxeq M_2$ and $(res(M_1) \cup res(M_2)) \cap res(N) = \varnothing$. We will show by cases of e that $(M_1 \mid N) \setminus \Delta \xrightarrow{e} (M'_1 \mid N') \setminus \Delta$ implies $(M_2 \mid N) \setminus \Delta \stackrel{e}{\Rightarrow} (M'_2 \mid N') \setminus \Delta$ and $(M'_1 \mid N') \setminus \Delta \approxeq (M'_2 \mid N') \setminus \Delta$ (the other direction is symmetric) :

- Let e be an external event. Then either $M_1 \xrightarrow{e} M_1'$ (and $N' = N$, since the resources of N and of M_1 are separated) or $N \xrightarrow{e} N'$ (and $M_1' = M_1$). In the first case, $M_1 \cong M_2$ implies $M_2 \stackrel{e}{\Rightarrow} M_2'$ and $M_1' \cong M_2'$. Then $(M_2 \mid N) \setminus \Delta \stackrel{e}{\Rightarrow} (M_2' \mid N) \setminus \Delta$ and $(M_1' \mid N) \setminus \Delta \cong (M_2' \mid N) \setminus \Delta$. In the second case, $(M_2 \mid N) \setminus \Delta \stackrel{e}{\Rightarrow} (M_2 \mid N') \setminus \Delta$ and $(M_1 \mid N') \setminus \Delta \cong (M_2 \mid N') \setminus \Delta$;
- Let $e = \tau$. Then either e is an internal action of M_1 or of N, or it is a communication between M_1 and N. The former case is analogous to the case of the external event treated above and we skip it. Then let $M_1 \stackrel{\bar{a}}{\Rightarrow} M_1'$ and $N \stackrel{a}{\Rightarrow} N'$ for some channel name $a \in A$. (the dual case where N sends and M_1 receives follows the same reasoning, so we skip it). $M_1 \cong M_2$ implies $M_2 \stackrel{\bar{a}}{\Rightarrow} M_2'$ and $M_1' \cong M_2'$. Since M_2 and N do not share resources, this cannot prevent the synchronization between $\bar{a}$ and a, then the firing a in N is not affected in any way by $\bar{a}$ and the previous internal firings of M_2. Then $(M_2 \mid N)\setminus\Delta \stackrel{\tau}{\Rightarrow} (M_2' \mid N')\setminus\Delta$ and $(M_1' \mid N')\setminus\Delta \cong (M_2' \mid N')\setminus\Delta$. □

E.6.3 Proofs from Section E.4

Proof of Theorem E.4.4 We proceed by successive steps :

- $HL(H)$ *is a set of finite handshake traces on* $HS(H)$. For any port a of H. By definition each node of a has at most one outgoing arc to another node of a. Moreover, only one place p of a contains a token and a is passive if p is an input place and active if it is an output place. Let a be active. Since the place p containing a token is an output place and since by definition an output place may only have an outgoing arc to an output transition, the first firing of a (if any) shall be an output $\bar{a}$. Following the same reasoning, if a is passive, the first firing of a shall be an input a. Again by definition, if t is an output (input) transition of a and has an outgoing arc to a node of a, this outgoing arc goes to an input (output) place. Hence alternation is ensured. Finally, executions are finite sequences of firings, then their external restrictions are also finite ;
- $HL(H)$ *is non-empty.* By definition an execution is a sequence of transitions, then the empty sequence is also an execution and its external restriction is an external trace. If the empty sequence is not quiescent in H it is the prefix of a quiescent execution (as we will show in the next point) ;
- $HL(H)$ *is closed with respect to passive prefixes.* $HL(H)$ is the set of external traces of all the quiescent executions of H. By contradiction, suppose there is $s \in Pas(HL(H))$ which does not come from a quiescent execution of H. Then there is an extension of this execution which, after a sequence of internal firings, lets an output transition $\bar{a}$ fire. If this is still not quiescent we can do the same thing over again. Note however that H only contains a finite number of external ports and could not continue to output indefinitely, eventually it shall stop and wait for an input, thus reaching a quiescent execution. Then $s \cdot \bar{a} \in HL(H)^{\leq}$ and s is not passive

(contradiction) ;

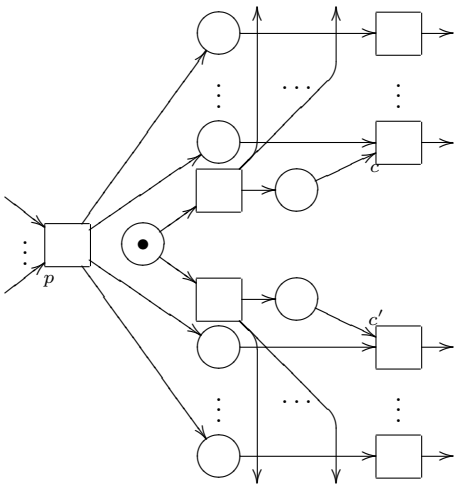
- $HL(H)$ *is reorder-closed.* Reorder-closedness comes as a consequence of the fact that, for any port a of H and for any $n > 0$: if the n-th transition t of a is an output transition, by definition it has only one outgoing arc in H and this arc goes to an input place of a, then the only transition that has to wait for t in H is the $(n+1)$-th transition of a ; conversely, if t is an input transition, by definition it has only one incoming arc in H and this arc comes from an input place of a, then the only transition that t has to wait for in H is the $(n-1)$-th transition of a ;
- $HL(H)$ *is receptive.* By definition, the underlying Petri net of a hpn is unsafe, that means that places may contain an unlimited number of tokens. So, every time an input transition is enabled to fire it can. Note also that an input transition has one incoming arc in H and this comes from an input place. An easy proof by induction can show that whenever an input place of a port contains a token, it must have an outgoing arc to an input transition. Note finally that a passive port is a port in which the place containing a token is an input place. This allows us to conclude that when a port is passive an input transition can fire and thus that $HL(H)$ is receptive.

□

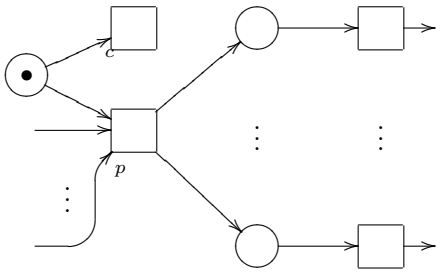
Proof of Proposition E.4.6 We set up H_σ's external structure by providing both an input and an output transition for each port $p \in P$ and by pairing them together by means of a port structure, as showed in definition E.4.1. In particular, the choice of an active or of a passive port structure is taken according to the label $d(p)$. Then we can already state $HS(H_\sigma) = \langle P, d \rangle$. Note also that we have a specific external transition for each message in the alphabet.

Now the internal structure. The occurrence selecters defined in Section E.4.4 allow us to associate a new (internal) transition to each occurrence of message : we use specific selecters for inputs as for outputs. The next step is to associate a transition to each position. Recall that a position can be represented as a set containing the last input occurrence of each thread. Then we take a new transition and we link each transition associated to any of these input occurrences into it : the link is a direct arc-place-arc one. We also add a transition for each choice c allowed at a given position $[s]_\sim$. In particular, if c does not stand for the choice to do nothing, we link $[s]_\sim$'s transition to c's transition, again by a direct arc-place-arc link. Note however that c might be in mutual exclusion with another choice c' at $[s]_\sim$, then we need a shared precondition before the corresponding transitions. But the choice of which one to fire should be made once and for all, then this same precondition should be used in any position where the two choices are allowed and mutually exclusive(we draw several outgoing

arcs from each choice to mean this) :



A special treatment is reserved for the do-nothing choice. In this case we add a transition with no outgoing arcs and put it in mutual exclusion directly with p's transition :



When the choice to do nothing is taken, H_σ has to wait for another input (thus moving to a new position) before doing anything else.

Let us now move to the output side, where each output occurrence $\bar{p}_i$ might be enabled in several positions. Then we make the arcs coming from the choice of $\bar{p}_i$ at each of these positions converge into a unique place, which will have an outgoing arc towards $\bar{p}_i$'s transition. One might object that so doing the same output occurrence might fire twice. But this is prevented by the selecter structure (Section E.4.4) which ensures that each transition associated to an occurrence of $\bar{p}$ may fire at most once.

The construction is finally complete, now we prove by induction that $s \in \sigma^{\leq}$ if and only if $s \in HL(H_\sigma)^{\leq}$.

- $s = \varepsilon$. Trivial since both σ and $HL(H_\sigma)$ are handshake languages.
- $s = s' \cdot a$. Let a be an input. Both σ and $HL(H_\sigma)$ are handshake languages on the same handshake structure $\langle P, d \rangle$. Then any direction we look, $s'a$ must be a handshake trace on $\langle P, d \rangle$. Since s' is a prefix of both languages, $s'a$ is too (receptivity). Now let a be the ith occurrence of output $\bar{p}$.

$s\bar{p} \in \sigma^{\leq}$ means that $\bar{p}_i$ is allowed by the position $[s]_\sim$ in σ and that no mutually exclusive choice has been chosen yet. Then in H_σ, $[s]_\sim$'s transition enables the transition associated to the choice of $\bar{p}_i$ at $[s]_\sim$. Plus, if ever there was a shared precondition among the choice of $\bar{p}_i$ and another choice at $[s]_\sim$, we may assume that it still contains a mark since the other choice has not produced any effect so far. Then $\bar{p}_i$'s transition may fire because it has not yet fired and because all the transitions associated to previous occurrences of $\bar{p}$ have already fired in s. Then $s\bar{p} \in HL(H_\sigma)^{\leq}$. On the other hand, $s\bar{p} \in HL(H_\sigma)^{\leq}$ implies that the transition associated to the choice of $\bar{p}_i$ at position $[s]_\sim$ is enabled by the transition associated to $[s]_\sim$ in H_σ. Then $[s]_\sim$ allows $\bar{p}_i$ in σ, by definition of H_σ. Moreover, if there was another choice excluding $\bar{p}_i$ at $[s]_\sim$ in σ, this was not chosen inside s. Then $s\bar{p} \in \sigma^{\leq}$.

Now, s is passive in σ if and only if the transition $[s]_\sim$ does not have any outgoing arcs in H_σ, that is if and only if s is passive in $HL(H_\sigma)$. s is a non-passive trace in σ if and only if the transition $[s]_\sim$ has a shared precondition with a transition which has no outgoing arcs in H_σ, that is if and only if s is a non-passive trace in $HL(H_\sigma)$. □

Proof of Theorem E.4.9 The proof of the completeness part of the theorem is a simplification of the proof of Proposition E.4.6 [4]. As for the the soundness part, the only properties left to prove are determinism's two, given that we already proved the preliminary properties for Theorem E.4.4. For both of them, the proof is based on the following simple observation. In a DB net, even if a place may have several outgoing arcs, only one of its post-transitions is actually enabled at any given time : each one has a guard and only one guard contains a mark in the initial state ; successively, the firing of the enabled post-transition takes away a mark from its guard and puts it into another guard. This prevents any situation of confusion, so that once a transition is enabled it will stay enabled until it fires.

Also, given two executions ex' and ex'', we can define an execution ex which completes ex' with those firings which occur in ex'' and not in ex' itself. We show how to do this by providing a constructive algorithm which gradually deletes the two original strings ex' and ex'' while writing ex. We initially set ex to the empty string. If at a given time $ex'' = a \cdot u''$ and $ex' = u' \cdot a \cdot v'$, where a does not appear in u', we append a to ex while removing it from the two original strings. So that ex' becomes $u'v'$ and ex'' becomes u''. If a does not appear in ex' we remove it from ex'' and we append it at the end of ex'. If $ex'' = \varepsilon$, we append what is left of ex' at the end of ex. Eventually, ex will consist of all the firings of ex' (possibly in a different order) followed by the firings of ex'' that were not there in ex'. About the order of the firings, note that if ex' and ex'' had the same external trace, ex would still have that external trace.

Now, let $s\bar{p} \in HL(H)^{\leq}$. Recall that $s\bar{p}$ is the prefix of an external trace of a quiescent execution of H. Then $s\bar{p}$ is also the external trace of a prefix of a quiescent execution, then the external trace of an execution of H. For the

4. Recall also that all deterministic handshake languages are positional (Prop. E.2.4).

first property we need to prove that there is no quiescent execution of H whose external trace is s. Since $s \cdot \bar{p} \in HL(H)^{\leq}$, there is an execution ex of H whose external trace is s and which can be extended by $\bar{p}$. Then any execution ex' of H whose external trace is s can be completed with those firings which occur in $ex \cdot \bar{p}$ and not in ex'. Note that the external trace of the execution we obtain is $s \cdot \bar{p}$. Then no execution of H whose external trace is s is quiescent. For the second property, let $s\bar{p}, s\bar{q} \in HL(H)^{\leq}$. Let also $ex'\bar{p}$ be an execution whose external trace is $s\bar{p}$ and $ex''\bar{q}$ be an execution whose external trace is $s\bar{q}$. Just as above we can interleave $ex'\bar{p}$ and $ex''\bar{q}$, so to obtain execution ex whose external trace is $s\bar{p}\bar{q} \in HL(H)^{\leq}$.□

Proof of Lemma E.4.8 The direction right-to-left is almost immediate : if there exists such a critical set, by definition $t \nrightarrow_\sigma c$. Then suppose that $t \nrightarrow_\sigma c$. Let S be the set of all critical pairs for c in σ which are inverted in t and let s be a handshake trace with the same threads as t [5] and in which all pairs of S are inverted. We prove by induction on $d = d_\sim(s,t)$ that $s \nrightarrow_\sigma c$:

- $d = 0$. $s = t$, then $s \nrightarrow_\sigma c$;
- $d = l+1$. There is a sequence $t = t_0, \ldots t_{l+1} = s$ such that $d_\sim(t_i, t_{i+1}) = 1$ and $t_i \nrightarrow_\sigma c$, $\forall 0 \leq i \leq l$. By contradiction assume $t_{l+1} \rightarrow_\sigma c$. Let $t_l = t' \cdot m \cdot n \cdot t''$ and $t_{l+1} = t' \cdot n \cdot m \cdot t''$. If m is an input or n an output, $t_l \mathbf{r} t_{l+1}$. Then since $t_{l+1} \rightarrow_\sigma c$, also $t_l \rightarrow_\sigma c$. Contradiction. Then m is an output, say the jth occurrence of $\bar{b}$, and n is an input, say the ith occurrence of a. By definition $\langle a_i, \bar{b}_j \rangle$ is a critical pair and since the sequence $t_0, \ldots t_{l+1}$ is minimal by definition of $d_\sim$, $\langle a_i, \bar{b}_j \rangle \in S$. But $\langle a_i, \bar{b}_j \rangle$ is not inverted in s : contradiction !

Then S is critical for c in σ. If S is not minimal we just need to take the minimal critical set contained in S and we are done. □

Proof of Theorem E.4.5 We already described the general construction of H_σ for σ positional (proof of proposition E.4.6) as well as its extension to the non-positional case (end of section E.4.6). Lemma E.4.8 justifies this extended construction by telling us that an "exception" to positionality has all the pairs of a critical set inverted and, viceversa, if a trace has all the pairs of a critical set inverted, then it is an exception to positionality. Then the proofs that $s \in \sigma^{\leq} \Leftrightarrow s \in HL(H_\sigma)^{\leq}$ and $s \in \sigma \Leftrightarrow s \in HL(H_\sigma)$ are just adaptations of the corresponding proofs that we gave for proposition E.4.6. □

E.6.4 Proofs from Section E.5

Let e be a label. We write $H \stackrel{e}{\Rightarrow} H'$ when $H \stackrel{(\tau)^*}{\longrightarrow\!\!\!\!\!\rightarrow} \stackrel{e}{\longrightarrow} \stackrel{(\tau)^*}{\longrightarrow\!\!\!\!\!\rightarrow} H'$. We also write $H \stackrel{\hat{e}}{\longrightarrow} H'$ when $H \stackrel{e}{\longrightarrow} H'$ or when $e = \tau$ and $H = H'$.

5. To be more precise we should take s from a larger set, where the number of input occurrences in each thread of s is equal to the number of input occurrences in the corresponding thread of t. This allows a thread of s to differ from the corresponding thread of t by an output occurrence. However reorder-closedness implies that outputs do not affect choices, so that we can assume s and t have exactly the same threads.

Definition E.6.2 (Expansion,$\lesssim$)**.** $\mathbf{E}$ is an *expansion* if $H_1\ \mathbf{E}\ H_2$ implies, for all e :

1. whenever $H_1 \stackrel{e}{\longrightarrow} H_1'$, there is H_2' s.t. $H_2 \stackrel{\hat{e}}{\Rightarrow} H_2'$ and $H_1'\ \mathbf{E}\ H_2'$;
2. whenever $H_2 \stackrel{e}{\longrightarrow} H_2'$, there is H_1' s.t. $H_1 \stackrel{\hat{e}}{\longrightarrow} H_1'$ and $H_1'\ \mathbf{E}\ H_2'$.

We say that H_2 *expands* H_1, written $H_1 \lesssim H_2$, if $H_1\ \mathbf{E}\ H_2$, for some expansion $\mathbf{E}$.

Our definition of expansion is as in [MS], although the same relation originally appeared in [AKH92] with a different terminology. For the above definitions we used the hpn notation, but clearly they hold for any transition system and in particular they hold for configurations.

All throughout these proofs, we will give node labels a special meaning. Transition labels are important in the operational semantics of hpns. While using hpns as a denotational semantics, not only transition but also place labels become meaningful. In particular, a place with a label r in the hpn will correspond to the resource r in the calculus. Not all places in an hpn have a label though, and definitely not all transitions have a unique label, then sometimes we will refer to a node by its name.

We say that an occurrence of the operator **Rec** is *significant* in the thread where it appears, if the sub-thread which follows it contains at least some action.

Ports represent the external structure of hpns, they are internalized once they are linked together. Let us recall the definition of linkage.

A *linkage* between an active port $a \in Ports_H$ and a passive port $b \in Ports_H$ of an hpn $H = \langle G_H, Ports_H\rangle$ is the hpn $L(H,a,b) = \langle G_{H,a,b}, Ports_H \backslash \{a,b\}\rangle$, where $G_{H,a,b}$ is the net obtained by adding two fresh places p_1 and p_2 to G_H and arcs from each output transition of a to p_1, from p_1 to each input transition of b, from each output transition of b to p_2 and from p_2 to each input transition of a.

We call *link* of $L(H,a,b)$, denoted $link(H,a,b)$, the graph consisting of the underlying graphs of a and of b plus p_1, p_2 and all the arcs connecting them to transitions of a or b.

We say that a and b are *internal ports* of $L(H,a,b)$, while p_1 and p_2 are the communication places of $link(H,a,b)$. After the linkage, all places and transitions of a and b become internal. Nonetheless, given one such place p, we still say that p is an *internal input place* or an *internal output place*, if p was an input or an output place, respectively, prior to the linkage. Similarly, given a transition t in $link(H,a,b)$, we say that t is an *internal input transition* or an *internal output transition*, if t was an input or an output transition, respectively, prior to the linkage. The definitions of *internal active* and *internal passive port* are straightforward adaptations of those of passive and active port, respectively. In the following, and with respect to all the above definitions, we may sometimes omit the adjective "internal", if no confusion arises. Note however that when we will say "internal place" or "internal transition", we will implicitly refer to places and transitions which are not contained in any port, internal or external.

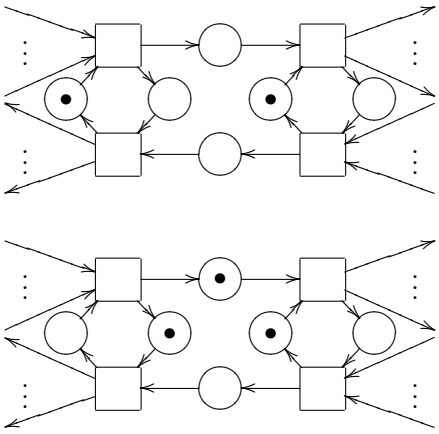


FIGURE E.5 – Examples of ready (above) and transmitting (below) links

For the sake of distinction, from now on we will call *external ports* those ports which are not internal. The above set up allows us to treat internal ports just as we treat external ones. In particular we can extend hpns with a set of internal ports.

Definition E.6.3. The triple $H = \langle G_H, ExtP_H, IntP_H \rangle$ is a *transparent hpn* just when $\langle G_H, ExtP_H \rangle$ is an hpn, $IntP_H$ is a set of disjoint internal ports of G_H and each port in $IntP_H$ is connected to exactly one other port in $IntP_H$ by a link.

Immediately after a linkage, the two communication places in the link are empty (fig. E.5, above). However, we need to consider the markings obtained when a transition in the link fires (fig. E.5, below). We say that a link is *ready* when both communication places are empty, one internal port is active and one internal port is passive. Thus, in a ready link, at most one transition is enabled. If this transition fired, both ports would become passive and exactly one communication place would contain a token. When a link has such a marking we say that it is *transmitting*. In a transmitting link, exactly one internal input transition is enabled. The firing of this transition would remove the token from the communication place and the link would be ready again.

Let H_1 and H_2 be transparent hpns. Let a be a port of H_1. H_2 is a *harmless extension* of H_1 just when it satisfies one of the following conditions :

- H_2 is obtained from H_1 by adding to a a fresh empty input place p, a fresh input transition t, an arc from p to t and possibly an arc from t to some output place of a. Possibly, H_2 could also be provided with some fresh internal places and arcs from t to these places or from t to other internal places which were already in H_1 ;
- H_2 is obtained from H_1 by adding to a a fresh empty output place p, a fresh output transition t, an arc from p to t and an arc from t to some

input place of a which has an outgoing arc. Possibly, H_2 could also be provided with some fresh internal places and arcs from these places to t or from other internal places which were already in H_1 to t ;
- H_2 is obtained from H_1 by adding to a a fresh disconnected empty place p, either of input or of output.

Let a contain a loop and let G be a connected subgraph of this loop starting at a place and ending at a transition, where this transition t has an outgoing arc to a place p of a. Then H_2 is a *loop unraveling* of H_1 just when it is obtained from H_1 by copying all of G and adding it to a, by providing each copied transition with an incoming (outgoing) arc from (to) an internal place of H_1 if and only if its original has an incoming (outgoing) arc from (to) that place, by adding an arc from the copy of t to p, by removing a token from a place of G and by putting a token into its copy whenever there is a place of G with a token.

Let L be a transmitting link of H_2 and let t be the enabled transition of L. H_1 is a *communication completion* of H_2 just when H_1 is obtained from H_2 by letting t fire.

Definition E.6.4. Let $\mathcal{R}_{\overline{\leq}}$ be a relation between transparent hpns such that $H_1 \; \mathcal{R}_{\overline{\leq}} \; H_2$ if and only if either H_2 is a harmless extension of H_1 or H_1 is a loop unraveling of H_2 or H_1 is a communication completion of H_2. Then we define the *port preorder* $\overline{\leq}$ on hpns as the reflexive and transitive closure of $\mathcal{R}_{\overline{\leq}}$.

Lemma E.6.5. $H_1 \overline{\leq} H_2 \Rightarrow H_1 \lesssim H_2$.

Proof: We are going to show that $\overline{\leq}$ is an expansion. Let $H_1 \overline{\leq} H_2$. By definition of $\overline{\leq}$ there is $n \geq 0$ such that $H_1 \; \mathcal{R}^n_{\overline{\leq}} \; H_2$, where :
- $\mathcal{R}^0_{\overline{\leq}}$ is the equality and
- $H' \; \mathcal{R}^{k+1}_{\overline{\leq}} \; H''$ iff there is H such that $H' \; \mathcal{R}_{\overline{\leq}} \; H \; \mathcal{R}^k_{\overline{\leq}} \; H''$.

Both directions of the proof are done by induction on n. Let first $H_1 \stackrel{e}{\longrightarrow} H_1'$:
- Let $H_1 = H_2$. Then $H_2 \stackrel{e}{\longrightarrow} H_1'$ and $H_1' \overline{\leq} H_1'$;
- Before we show the most general case, let us show that if $H_1 \; \mathcal{R}_{\overline{\leq}} \; H_2$ then $H_2 \stackrel{e}{\Rightarrow} H_2'$ and either $H_1' \; \mathcal{R}_{\overline{\leq}} \; H_2'$ or $H_1' = H_2'$:
  - Let H_2 be a harmless extension of H_1. Then every transition of H_1 is also in H_2 so that $H_2 \stackrel{e}{\longrightarrow} H_2'$ and H_2' extends H_1' just in the same way as H_2 extends H_1. Then $H_1' \; \mathcal{R}_{\overline{\leq}} \; H_2'$;
  - Let H_1 be a loop unraveling of H_2. The case where $H_1 \stackrel{e}{\longrightarrow} H_1'$ is the firing of a transition which is also in H_2 is analogous to the previous case. Then let it be the firing of a transition t in a port a, where t is not in H_2 but is the copy of a transition t' which is. By definition of loop unraveling, t' is enabled in H_2 since t was enabled in H_1. Then let $H_2 \stackrel{e}{\longrightarrow} H_2'$ be the firing of t' (note that t' has the same label as t, by definition of port). Let us do a few remarks about the correspondences among the two firings, all of them follow directly from the definition of loop unraveling. The firing of t removes a token from a place p of a in

H_1 if and only if the firing of t' removes a token from a place p' in H_2, where p' is the copy of p. The firing of t removes (adds) a token from (to) a place outside a in H_1 if and only if the firing of t' removes (adds) a token from (to) the same place in H_2. Finally, the firing of t adds a token to a place q of a in H_1 if and only if the firing of t' adds a token to a place q' of a in H_2, where q' is either the copy of q or q itself. In any case H_1' is a loop unraveling of H_2', as H_1 is a loop unraveling of H_2. Then $H_1' \mathcal{R}_{\leq} H_2'$;

- Let H_1 be a communication completion of H_2. Then there is an internal transition t whose firing changes H_2 into H_1, $H_2 \xrightarrow{\tau} H_1$. Then $H_2 \stackrel{e}{\Rightarrow} H_1'$.

- Let $H_1 \mathcal{R}^{k+1}_{\leq} H_2$. Then there is H such that $H_1 \mathcal{R}^{k}_{\leq} H \mathcal{R}_{\leq} H_2$. By induction hypothesis, $H \stackrel{e}{\Rightarrow} H'$ and $H_1' \leq H'$. The sequence of firings $H \stackrel{e}{\Rightarrow} H'$ can be decomposed in three subsequences : $H \xrightarrow{\tau}\!\!\!\!\rightarrow H''$, $H'' \xrightarrow{e} H'''$ and $H''' \xrightarrow{\tau}\!\!\!\!\rightarrow H'$. By induction, and using the previous case as inductive case, we can prove that there is H_2'' such that $H_2 \stackrel{\tau}{\Rightarrow} H_2''$, where either $H'' \mathcal{R}_{\leq} H_2''$ or $H'' = H_2''$. Analogously, there are also H_2''' and H_2' such that $H_2'' \stackrel{e}{\Rightarrow} H_2'''$ and $H_2''' \stackrel{\tau}{\Rightarrow} H_2'$, where either $H''' \mathcal{R}_{\leq} H_2'''$ or $H''' = H_2'''$, and $H' \mathcal{R}_{\leq} H_2'$ or $H' = H_2'$. Then by transitivity we obtain $H_1' \leq H_2'$.

Now let $H_2 \xrightarrow{e} H_2'$:

- Let $H_1 = H_2$. Then $H_1 \xrightarrow{e} H_2'$ and $H_2' \leq H_2'$;
- Let $H_1 \mathcal{R}^{k+1}_{\leq} H_2$. Then there is H such that $H_1 \mathcal{R}_{\leq} H \mathcal{R}^{k}_{\leq} H_2$. By induction hypothesis, $H \stackrel{\hat{e}}{\Rightarrow} H'$ and $H' \leq H_2'$. The case $H = H'$ is trivial. Then let $H \xrightarrow{e} H'$. We do the proof by cases of $H_1 \mathcal{R}_{\leq} H$:
  - Let H be a harmless extension of H_1. Let t be the transition of H which is not in H_1. By definition t has an incoming arc from an empty place, then $H \xrightarrow{e} H'$ cannot be the firing of t. Then the same firing can be performed by H_1 as well, $H_1 \xrightarrow{e} H_1'$. Moreover, H' extends H_1' just in the same way as H extends H_1. Then $H_1' \leq H'$ and by transitivity $H_1' \leq H_2'$;
  - Let H_1 be a loop unraveling of H. Let t be the transition whose firing is $H \xrightarrow{e} H'$. Again every transition of H is also in H_1 however, as a result of the unraveling, t may not be enabled in H_1. If t is enabled in H_1 the proof is analogous to the one given in the previous case, then let us consider the case where t is not enabled in H_1. Let t' be the copy of t in H_1. By definition of loop unraveling, t' is enabled in H_1. Moreover, the same correspondences among the firing of the copy of a transition and that of the original that we remarked in the other direction still hold here. Then if we let $H_1 \xrightarrow{e} H_1'$ be the firing of t', we have that H_1' is a loop unraveling of H'. Then $H_1' \leq H'$ and by transitivity $H_1' \leq H_2'$;
  - Let H_1 be a communication completion of H. Then there is an internal transition t whose firing changes H into H_1, $H \xrightarrow{\tau} H_1$. If the firing of t is also the firing $H \xrightarrow{e} H'$, $e = \tau$ and $H' = H_1$ and H_1 does not need to

do anything. Then let $H \xrightarrow{e} H'$ be the firing of a transition t' different from t. For a communication completion to take place, the link must be transmitting and both ports in a transmitting link are passive. Then t is the input transition of an internal port, so that it can never be in conflict with another transition. Then t' can fire in H_1, $H_1 \xrightarrow{e} H_1'$, but also t can fire in H' and since t and t' are not in conflict, the result of this firing is still H_1'. Then H_1' is a communication completion of H'. Then $H_1' \overline{\leq} H'$ and by transitivity $H_1' \overline{\leq} H_2'$.

As we are now considering transparent hpns, we need to modify the interpretation function accordingly. For the interpretation of threads, it is enough to equip the previously defined denotation with an empty set of internal ports. While in the case of a parallel composition of processes, we need first to extend the definition of linkage to transparent hpns. In particular, a *linkage* between an external active port $a \in Ports_H$ and an external passive port $b \in Ports_H$ of a transparent hpn $H = \langle G_H, ExtP_H, IntP_H \rangle$ is the hpn $L(H, a, b) = \langle G_{H,a,b}, ExtP_H \backslash \{a, b\}, IntP_H \cup \{a, b\} \rangle$, where $G_{H,a,b}$ is the net obtained by adding two fresh places p_1 and p_2 to G_H and arcs from each output transition of a to p_1, from p_1 to each input transition of b, from each output transition of b to p_2 and from p_2 to each input transition of a. Having adapted this definition, we can now keep the definitions of parallel composition of hpns and of denotation of configurations where the process is a parallel composition of subprocesses as they are.

Let H_1 and H_2 be two hpns in which place labels are unique and such that if a place p has label r in H_1 and q has label r in H_2, then p and q contain the same number of tokens. Then we define $H = H' \|_C^{merge} H''$ by the usual composition, followed by the merging of places with the same label. In particular the merging preserves all incoming and outgoing arcs of each place. From now on we will always consider compositions plus mergings, unless otherwise stated.

It is almost immediate to see that if $H_1 \overline{\leq} H_1'$ and $H_2 \overline{\leq} H_2'$ then $H_1 \|_C^{merge} H_2 \overline{\leq} H_1' \|_C^{merge} H_2'$.

Proof of Lemma E.5.1 Let $\mathcal{R}$ be the symmetric closure of the following relation between configurations and hpns :

$$\mathcal{R} = \{(M, [\![M]\!]_\Gamma) \mid M \rhd \Gamma\}$$

$\mathcal{R}$ is not a weak bisimulation. However we will show that if $M \xrightarrow{e} M'$, then $[\![M]\!]_\Gamma \stackrel{\epsilon}{\Rightarrow} H$ and $[\![M']\!]_{\Gamma'} \overline{\leq} H$, where $M \rhd \Gamma$ and $M' \rhd \Gamma'$; and if $[\![M]\!]_\Gamma \xrightarrow{e} H$ then $M \xrightarrow{\hat{e}} M'$ and $[\![M']\!]_{\Gamma'} \overline{\leq} H$, where $M \rhd \Gamma$ and $M' \rhd \Gamma'$. Since $\overline{\leq}$ is stronger than $\lesssim$ (lemma E.6.5) the above argument will allow us to state that $\mathcal{R}$ is an expansion up to $\lesssim$ and hence that $M \lesssim [\![M]\!]_\Gamma$ (Theorem 3.5 of [MS]). Since $\lesssim$ is stronger than $\approx$ (Theorem 3.3 of [MS]), $M \approx [\![M]\!]_\Gamma$.

Let $M = \langle P, S \rangle$. We can conveniently set $[\![\wr P, S \int]\!]_\Gamma = [\![\langle P, S \rangle]\!]_\Gamma$. The proof is done by cases of P :

– Let $P = \mathbf{0}$. Then M does not reduce. By construction $[\![M]\!]_\Gamma$ contains an active port which consists of an output place with a token, and a bunch of scattered internal places. Then it does not reduce either ;

- Let $P = a^{\{r_1,\dots r_n\}}.T$. Then $\Gamma = ?a$. The only possible reduction for M is $M \xrightarrow{a} \langle T, S + \{r_1, \dots r_n\}\rangle$, where $T \triangleright !a$. On the other hand note that $[\![M]\!]_\Gamma$ also has only one possible reduction, since it does not contain internal transitions and the only port it contains is the passive a. By the definition of the interpretation, exactly one transition is enabled in a port of $[\![M]\!]_\Gamma$. Let t be the enabled transition of a. Being an input transition, t is labeled a. Let H be the result of the firing of t ($[\![M]\!]_\Gamma \xrightarrow{a} H$). By construction, the firing of t removes a token from a single input place p and puts a token in each internal place labeled r_i, for $1 \leq i \leq n$, and one in an output place p' of a. Let a' be the obtained port, statically identical to a but with the token in p' and not in p. Let a'' be the port obtained from a' by cutting p, t and all their incoming and outgoing arcs. Note that a'' is the active port in the hpn $[\![\langle T, S+\{r_1, \dots r_n\}\rangle]\!]_{!a}$, by construction of a. Still by construction p does not have any incoming arc in a, then H is a harmless extension of $[\![\langle T, S + \{r_1, \dots r_n\}\rangle]\!]_{!a}$. Then $[\![\langle T, S + \{r_1, \dots r_n\}\rangle]\!]_{!a} \trianglelefteq H$;
- The case $P = \bar{a}_{\{r_1,\dots r_n\}}.T$, the output prefixing, is very much similar to the input prefixing case treated above and we skip it ;
- Let $P = \mathbf{Rec}\ T$. The operational semantics tells us that M and $\langle T \cdot \mathbf{Rec}\ T, S\rangle$ reduce exactly to the same configurations. Then consider $[\![M]\!]_\Gamma$. By construction it contains a port a in which there is a place p with a token. Then :
  - Let p have no incoming arcs. By definition, $[\![M]\!]_\Gamma = [\![\langle T, S\rangle]\!]_\Gamma$. Also, by the way we defined the interpretation, either p is the only place of a or there is some place of a with two incoming arcs. In the former case $T \equiv \mathbf{0}$, then $\mathbf{Rec}\ T \equiv T$. In the latter case T must contain some significant **Rec**, then $T \cdot \mathbf{Rec}\ T = T$. In both cases we can assume as hypothesis that if $\langle T, S\rangle \xrightarrow{e} M'$, then $[\![\langle T, S\rangle]\!]_\Gamma \stackrel{\hat{e}}{\Rightarrow} H$ and $[\![M']\!]_{\Gamma'} \trianglelefteq H$; and if $[\![\langle T, S\rangle]\!]_\Gamma \xrightarrow{e} H$, then $\langle T, S\rangle \xrightarrow{\hat{e}} M'$ and $[\![M']\!]_{\Gamma'} \trianglelefteq H$. Then we are done ;
  - Let p have an incoming arc from a transition t of a. By construction, no place of a may have two incoming arcs, so that a forms a loop of nodes, starting from p and passing through t just before the loop's closure. This means that T does not contain any significant **Rec** and that $T \not\equiv \mathbf{0}$. Then at most one reduction is possible for $T \cdot \mathbf{Rec}\ T$ and this depends only on the resources available in S. $[\![M]\!]_\Gamma$ also has at most one possible reduction as it contains only one external port and no internal transition. Moreover, let p have an outgoing arc to a transition t' of a. If $[\![M]\!]_\Gamma$ can reduce, its reduction must correspond to the firing of t' as the other transitions are not enabled. By definition of the interpretation relation, t' is an output transition if and only if the first action in T is an output action, and t' has an incoming arc from the (unique) internal place labeled r if and only if the resource r is in the set of resources attached to the first action in T. Then $[\![M]\!]_\Gamma \xrightarrow{e}$ if and only if $\langle T \cdot \mathbf{Rec}\ T, S\rangle \xrightarrow{e}$.
    So, let $\langle T \cdot \mathbf{Rec}\ T, S\rangle \xrightarrow{e} M'$. Since T does not contain any significant

Rec and $T \not\equiv \mathbf{0}$, $T \cdot \mathbf{Rec}\ T$ consists of the sequence of actions appearing in T, in the same order, followed by $\mathbf{Rec}\ T$. Then M' is of the form $\langle T' \cdot \mathbf{Rec}\ T, S' \rangle$, where T' is the thread resulting from the reduction $\langle T, S \rangle \xrightarrow{e} \langle T', S' \rangle$. Let t' have an outgoing arc to a place p' of a and let H be the hpn obtained by letting t' fire in $[\![M]\!]_\Gamma$. If p' and p are the same place, the thread T must consist of a single action prefixing $\mathbf{0}$, by construction. However we run in a contradiction as the typing system imposes that this action be both an input action ((ax) and (inpref) rules) and an output action ((rec) rule). Then p' is different from p and we can obtain $[\![\langle T' \cdot \mathbf{Rec}\ T, S' \rangle]\!]_{\Gamma'}$ by applying the loop unraveling operation to H, with a as the concerned port and the chain which goes from p' to t as the concerned connected subgraph of its loop. By definition E.6.4, $[\![\langle T' \cdot \mathbf{Rec}\ T, S' \rangle]\!]_\Gamma \trianglelefteq H$.

– Let $P = (P_1 \mid P_2) \setminus \Delta$, where $P_1 \rhd \Gamma_1$ and $P_2 \rhd \Gamma_2$, $\Gamma = \Gamma_1 \odot \Gamma_2$ and $\Delta = Dom(\Gamma_1) \cap Dom(\Gamma_2)$. Let $M \xrightarrow{e} \langle (P_1' \mid P_2) \setminus \Delta, S' \rangle$. Then $\langle P_1, S \rangle \xrightarrow{e} \langle P_1', S' \rangle$, where $P_1' \rhd \Gamma_1'$ and $\Gamma' = \Gamma_1' \odot \Gamma_2$. By assumption, $[\![\langle P_1, S \rangle]\!]_{\Gamma_1} \stackrel{e}{\Rightarrow} H_1$ and $[\![\langle P_1', S' \rangle]\!]_{\Gamma_1'} \trianglelefteq H_1$. Then we have that $[\![\langle P_1', S' \rangle]\!]_{\Gamma_1'} \parallel^{merge}_{\{(a,a)|a\in\Delta\}} [\![\langle P_2, S' \rangle]\!]_{\Gamma_2} \trianglelefteq H_1 \parallel^{merge}_{\{(a,a)|a\in\Delta\}} [\![\langle P_2, S' \rangle]\!]_{\Gamma_2}$. By definition, the hpn on the left is $[\![\langle (P_1' \mid P_2) \setminus \Delta, S' \rangle]\!]_{\Gamma'}$, while the one on the right is H, the hpn obtained from $[\![M]\!]_\Gamma$ by letting e fire. Then $[\![\langle (P_1' \mid P_2) \setminus \Delta, S' \rangle]\!]_{\Gamma'} \trianglelefteq H$, and we are done.
Now let $M \xrightarrow{\tau} \langle (P_1' \mid P_2') \setminus \Delta, S'' \rangle$. This internal reduction corresponds to the synchronization of reductions $\langle P_1, S \rangle \xrightarrow{\bar{a}} \langle P_1', S' \rangle$ and $\langle P_2, S' \rangle \xrightarrow{a} \langle P_2', S'' \rangle$, where $a \in \Delta$, $P_1' \rhd \Gamma_1'$, $P_2' \rhd \Gamma_2'$ and $\Gamma' = \Gamma_1' \odot \Gamma_2'$ [6]. By hypothesis, $[\![\langle P_1, S \rangle]\!]_{\Gamma_1} \stackrel{\bar{a}}{\Rightarrow} H_1$ and $[\![\langle P_2, S' \rangle]\!]_{\Gamma_2} \stackrel{a}{\Rightarrow} H_2$, where $[\![\langle P_1', S' \rangle]\!]_{\Gamma_1'} \trianglelefteq H_1$ and $[\![\langle P_2', S'' \rangle]\!]_{\Gamma_2'} \trianglelefteq H_2$. Since $[\![M]\!]_\Gamma = [\![\langle P_1, S \rangle]\!]_{\Gamma_1} \parallel^{merge}_{\{(b,b)|b\in\Delta\}} [\![\langle P_2, S \rangle]\!]_{\Gamma_2}$, holds by definition, the same firings can also occur in $[\![M]\!]_\Gamma$, but since $a \in \Delta$, they are going to be hidden there. So, $[\![M]\!]_\Gamma \stackrel{\tau}{\Rightarrow} H_1' \|^{merge}_{\{(b,b)|b\in\Delta\}} H_2$, where H_1' is like H_1 except for the addition of some tokens to some internal places, so that $[\![\langle P_1', S'' \rangle]\!]_{\Gamma_1'} \trianglelefteq H_1'$. Then $[\![\langle P_1', S'' \rangle]\!]_{\Gamma_1'} \parallel^{merge}_{\{(b,b)|b\in\Delta\}} [\![\langle P_2', S'' \rangle]\!]_{\Gamma_2'} \trianglelefteq H_1' \parallel^{merge}_{\{(b,b)|b\in\Delta\}} H_2$ and $[\![\langle P_1', S'' \rangle]\!]_{\Gamma_1'} \parallel^{merge}_{\{(b,b)|b\in\Delta\}} [\![\langle P_2', S'' \rangle]\!]_{\Gamma_2'} = [\![\langle (P_1' \mid P_2') \setminus \Delta, S'' \rangle]\!]_{\Gamma'}$, by definition.
Let $[\![M]\!]_\Gamma \xrightarrow{e} H$. Recall that $[\![M]\!]_\Gamma = [\![\langle P_1, S \rangle]\!]_{\Gamma_1} \parallel^{merge}_{\{(a,a)|a\in\Delta\}} [\![\langle P_2, S \rangle]\!]_{\Gamma_2}$. We distinguish the case where $[\![M]\!]_\Gamma \xrightarrow{e} H$ is the firing of a transition in a port $a \in \Delta$ (on either side) from the case where it is the firing of a transition in a port $b \notin \Delta$. Consider the former case and let t be the transition which fired. We can assume the firing corresponds to the reduction $\langle P_1, S \rangle \xrightarrow{\bar{a}} \langle P_1', S' \rangle$, as the first event of an internal communication locally is always an output (the case where the reduction is on P_2 is symmetric). However this reduction cannot be performed directly

6. Actually there is also the dual situation where the right side sends and the left side receives, but the proof substantially would not change.

by M, instead it shall be synchronized with an input reduction on the other side, namely $\langle P_2, S'\rangle \xrightarrow{a} \langle P_2', S''\rangle$. As a result of the synchronization we have $\langle (P_1 \mid P_2) \setminus \Delta, S\rangle \xrightarrow{\tau} \langle (P_1' \mid P_2') \setminus \Delta, S''\rangle$. By hypothesis, the reduction $\langle P_2, S'\rangle \xrightarrow{a} \langle P_2', S''\rangle$ corresponds to the firing $H \xrightarrow{\tau} H'$, where $[\![\langle (P_1' \mid P_2') \setminus \Delta, S''\rangle]\!]_{\Gamma'} \overline{\leq} H'$. By definition H' is a communication completion of H, so that $H' \overline{\leq} H$. By transitivity of $\overline{\leq}$ we conclude $[\![\langle (P_1' \mid P_2') \setminus \Delta, S''\rangle]\!]_{\Gamma'} \overline{\leq} H$.
Finally consider the case where $[\![M]\!]_\Gamma \xrightarrow{e} H$ is the firing of a transition in a port $b \notin \Delta$. Then the firing occurs in either of the two subnets, say in $[\![\langle P_1, S\rangle]\!]_{\Gamma_1}$. From the subnet's point of view, the firing is $[\![\langle P_1, S\rangle]\!]_{\Gamma_1} \xrightarrow{e} H_1$. By hypothesis, $\langle P_1, S\rangle \xrightarrow{\hat{e}} \langle P_1', S'\rangle$, where $P_1' \rhd \Gamma_1'$, $\Gamma' = \Gamma_1' \odot \Gamma_2$ and $[\![\langle P_1', S'\rangle]\!]_{\Gamma_1'} \overline{\leq} H_1$. Note that $H = H_1 \parallel^{merge}_{\{(b,b)\mid b\in\Delta\}} [\![\langle P_2, S'\rangle]\!]_{\Gamma_2}$, $M \xrightarrow{\hat{e}} \langle (P_1' \mid P_2)\setminus\Delta, S'\rangle = M'$ and $[\![M']\!]_{\Gamma'} = [\![\langle P_1', S'\rangle]\!]_{\Gamma_1'} \parallel^{merge}_{\{(b,b)\mid b\in\Delta\}} [\![\langle P_2, S'\rangle]\!]_{\Gamma_2}$. Then $[\![M']\!]_{\Gamma'} \overline{\leq} H$.

□

Proof of Theorem E.5.2
By Lemma E.5.1, $M \approx [\![M]\!]_\Gamma$ and $M' \approx [\![M']\!]_\Gamma$. Then $M \approx M' \Leftrightarrow [\![M]\!]_\Gamma \approx [\![M']\!]_\Gamma$. □

We first repeat in details the construction sketched in Section E.5.2.

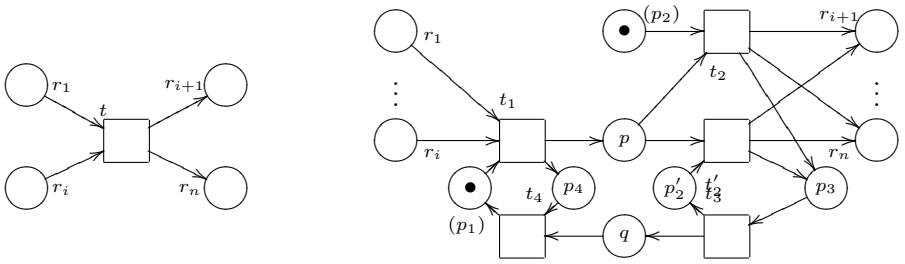
Now let $H = \langle G, ExtP, IntP\rangle$ be a transparent hpn where each internal place has a distinct label. Let also each internal port be named either $a^!$ or $a^?$ for some port name a, such that if an internal port is named $a^!$ then it is an internal active port and is linked to an internal passive port named $a^?$; and viceversa. Finally, let $Ports = ExtP \cup IntP$.

Now, let $a \in Ports$. We define the *unfolding* of a which we denote as $u(a)$, as follows :

- For any transition t of a with no outgoing arcs, add a fresh output place p and an arc from t to p. Note that t must be an input transition, by definition of handshake port. Then a', the net so obtained, is a port;
- For any input place p of a' with more than one incoming arc or with one incoming arc and a token, add another fresh empty input place p' and replace one of p's incoming arcs with another arc with the same source but target p'. Now, if p has an outgoing arc to t, add another fresh input transition t' and an arc from p' to t'. If t has outgoing arcs to some internal places (of G), provide t' with outgoing arcs to the same internal places. Finally, if t has an outgoing arc to an output place p'' (of a'), provide t' with an outgoing arc to p''. Do this over again many times until each input place of the port has either no tokens and at most one incoming arc or one token and no incoming arcs.

Internal transitions require a different treatment. Let t be an internal transition of H with incoming arcs from internal places labeled $r_1, \ldots r_i$ and outgoing arcs to internal places labeled $r_{i+1}, \ldots r_n$. Then t is extended to its *unfolding*

$u(t)$ as follows [7] :



where $r_1, \ldots r_n$ are place labels while $p_1, p_2, p_2', p_3, p_4, p$ and q are place names. As a notation, when a place has a label we write its label next to it, when a place does not have a label but for some reason we want to distinguish it from other places, we associate the place to a name and write the name inside the place or next to the place but in parenthesis (when the place contains a token). However, we always write a transition name next to the transition, as we will not use transition labels in any figure.

Now we can define $u(H)$ as the hpn obtained from H by unfolding all of its ports and internal transitions.

Let $\mathcal{R}_p^u$ be a relation between hpns such that $H\ \mathcal{R}_p^u\ H'$ if and only if H is like H' except that there is a port a of H which is substituted by a port a' in H', where a' is statically as the unfolding of a (including transition labels) and if a place p contains a token in a then either p or one of its copies contains a token in a', if no place contains a token in a then either no place or a place with no outgoing arcs contains a token in a'.

Let $\mathcal{R}_t^u$ be a relation between hpns such that $H\ \mathcal{R}_t^u\ H'$ if and only if H is like H' except that there is a transition t of H which is substituted by a link L^t in H', where L^t is statically as $u(t)$ and L^t is ready.

Definition E.6.6. We define the *unfolding preorder* $\preccurlyeq$ as the smallest preorder which contains $\mathcal{R}_p^u \cup \mathcal{R}_t^u$.

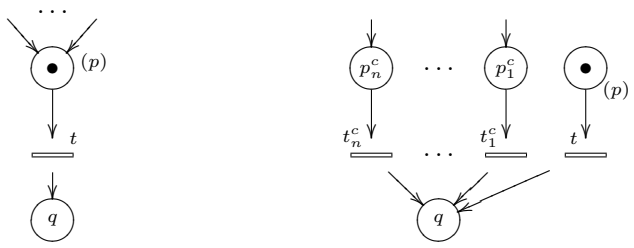
Lemma E.6.7. $H_1 \preccurlyeq H_2 \Rightarrow H_1 \approx H_2$.

Proof: The proof can be divided in three parts :

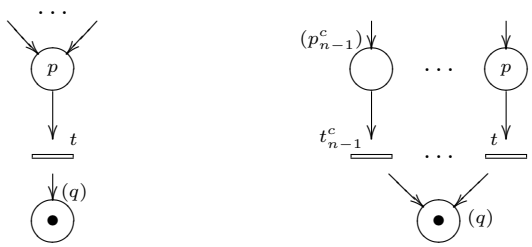
1. *$\mathcal{R}_p^u$ is an expansion up to $\lesssim$.*
   Let $H_1\ \mathcal{R}_p^u\ H_2$, let a be the port of H_1 which is expanded in H_2 and let a' be the corresponding port in H_2. Let $H_1 \xrightarrow{e} H_1'$. If this is the firing of a transition which is not in a then the same transition can fire in H_2, $H_2 \xrightarrow{e} H_2'$, and clearly $H_1'\ \mathcal{R}_p^u\ H_2'$. Then let $H_1 \xrightarrow{e} H_1'$ be the firing of a transition t of a. By definition of $\mathcal{R}_p^u$, there is an enabled transition t' of a' and t' can either be a copy of t or t itself. Let us remark a few correspondences between the firing of t in H_1 and that of t' in H_2, all

7. The unfolding presented in the main text is simpler. The one presented here, while bisimilar, corresponds precisely to the thread.

these remarks follow directly from the definition of $\mathcal{R}_p^u$. The firing of t removes a token from a place p of a if and only if the firing of t' removes a token from a place p' of a' and p' is a copy of p if t' is a copy of t, it is p itself otherwise. The firing of t in H_1 adds (removes) a token to (from) an internal place if and only if the firing of t' in H_2 adds (removes) a token to (from) the same place. The firing of t adds a token to a place q of a in H_1 if and only if the firing of t' adds a token to either q or one of its copies in H_2. Finally, if the firing of t in H_1 does not add any token to any place of a, then the firing of t' adds a token to a place of a with no outgoing arc in H_2. Given the above premises, there is one case where H_1' and H_2' are not in the relation $\mathcal{R}_p^u$ and that is when t is an input transition and the input place p with an outgoing arc to t has at least an incoming arc in H_1. As in the following pictures, for example :



where the picture on the left shows the critical subgraph of the original port and the picture on the right shows what becomes of this subgraph after the unfolding operation is applied. Also, p_i^c and t_i^c, for $1 \leq i \leq n$, are the names we gave to the copies of p and of t, respectively. Note that p contains a token in H_1, as t is enabled. As a result of the unfolding, p does not contain any incoming arc in H_2. But after the firing p no more contains a token, so that the unfolding of a in H_1' produces an hpn H (H_1' $\mathcal{R}_p^u$ H) where p has an incoming arc, differently from H_2'. Note however that, in H_2, t and all its copies have each an outgoing arc to the same place q of a', and so does the transition t in H_1. Then H_2' is a harmless extension of the hpn H. By lemma E.6.5, $H \lesssim H_2'$. The following pictures show the critical subgraph in H_1' (left) and what becomes of it in H, after the unfolding (right).



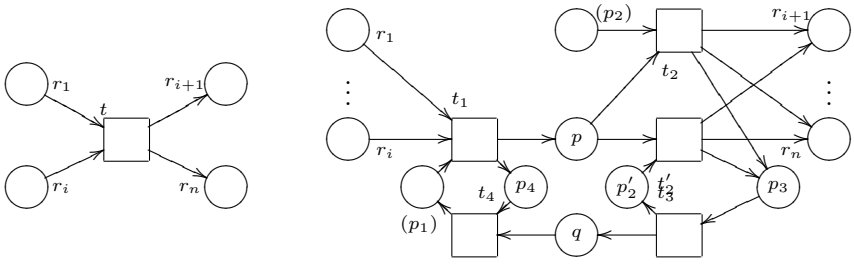
Now let $H_2 \xrightarrow{e} H_2'$. Let this be the firing of a transition t of H_2. Again, if t is a transition which is not in a' the same transition can fire in H_1,

$H_1 \xrightarrow{e} H_1'$, and clearly $H_1' \ \mathcal{R}_p^u \ H_2'$. Then let t be a transition of a'. By definition of $\mathcal{R}_p^u$, there is an enabled transition t' of a in H_1. Moreover t can either be a copy of t' or t' itself. Note that all the correspondences between the two firings that we remarked in the other direction still hold here, except for the last one. However, now we can say that if the firing of t in H_2 adds a token to a place p with no outgoing arcs, then the firing of t' in H_1 either adds a token to p or does not add any token to any place of a. Anyway, as above we have either $H_1' \ \mathcal{R}_p^u \ H_2'$ or $H_1' \ \mathcal{R}_p^u \ H$, for some hpn H such that H_2' is a harmless extension of H.

We conclude that the relation $\mathcal{R}_p^u$ is an expansion up to $\lesssim$ and that is thus contained in $\lesssim$ (Theorem 3.5 of [MS]).

2. *$\mathcal{R}_t^u$ is an expansion up to $\lesssim$.*

Let $H_1 \ \mathcal{R}_t^u \ H_2$ and let t be the internal transition of H_1 which is expanded to a link L^t in H_2.



where the net on the right is not L^t but its underlying static graph. The reason why we did not put the tokens is that the actual marking of L^t may vary. However, by the definition of $\mathcal{R}_t^u$ we know that L^t is ready and this means that both ports are passive and the communication places do not contain any token. Then there are only three possible markings for L^t : a token in p_1 and one in p_2 or p_2', or a token in p_3 and one in p_4. Let $H_1 \xrightarrow{e} H_1'$. If this is the firing of a transition other than t the same transition can fire in H_2, $H_2 \xrightarrow{e} H_2'$, and clearly $H_1' \ \mathcal{R}_t^u \ H_2'$. Then let $H_1 \xrightarrow{e} H_1'$ be the firing of t, so that $e = \tau$. In any of the three cases, we can match the firing of t in H_2 by executing a sequence of firings of transitions in L^t, until t_2 or t_2' has fired. Let this execution be $H_2 \stackrel{\tau}{\Rightarrow} H_2'$. As a result, one token is removed from each place with an outgoing arc to t_1 and a token is added to each place with an incoming arc from t_2. Since H_2' is also ready, $H_1' \ \mathcal{R}_t^u \ H_2'$. As an example, let L^t contain a token in p_3 and one in p_4. In this case, the execution $H_2 \stackrel{\tau}{\Rightarrow} H_2'$ consists of the firing of t_3, t_4, t_1 and t_2'.

Let $H_2 \xrightarrow{e} H_2'$. If this is the firing of a transition outside L^t the same transition can fire in H_1, $H_1 \xrightarrow{e} H_1'$, and clearly $H_1' \ \mathcal{R}_t^u \ H_2'$. Otherwise by definition of $\mathcal{R}_t^u$, $H_2 \xrightarrow{e} H_2'$ is the firing of either t_1 or t_3, both of which are internal firings. Then $e = \tau$. In both cases, after the firing L^t is transmitting and thus H_2' is no more in the relation $\mathcal{R}_t^u$ with H_1. Still,

let us consider the two cases separately. If $H_2 \xrightarrow{\tau} H_2'$ is the firing of t_3, no token is removed from an internal place and after the firing t_4 is the enabled transition in L^t. Then let $H_2' \xrightarrow{\tau} H_2''$ be the firing of t_4. This does not add any token to any internal place, so that $H_1 \ \mathcal{R}_t^u \ H_2''$. Moreover, H_2'' is a communication completion of H_2' and by lemma E.6.5, $H_2'' \lesssim H_2'$. Now let $H_2 \xrightarrow{\tau} H_2'$ be the firing of t_1. The firing of t_1 in H_2 removes a token from an internal place if and only if the firing of t in H_1 removes a token from that internal place. Then let $H_1 \xrightarrow{\tau} H_1'$ be the firing of t in H_1. Note that in H_2' either t_2 or t_2' is enabled. Anyway, the two transitions have outgoing arcs to exactly the same places. Then let $H_2' \xrightarrow{\tau} H_2''$ be the firing of the transition of L^t which is enabled in H_2'. This firing adds a token to an internal place if and only if the firing of t in H_1 adds a token to that place, so that $H_1' \ \mathcal{R}_t^u \ H_2''$. Moreover, H_2'' is a communication completion of H_2' and by lemma E.6.5, $H_2'' \lesssim H_2'$.

We conclude that the relation $\mathcal{R}_t^u$ is an expansion up to $\lesssim$ and that is thus contained in $\lesssim$.

3. We have shown that $(\mathcal{R}_p^u \cup \mathcal{R}_t^u) \subseteq \lesssim$. By definition, $\lesssim$ is a preorder and $\preccurlyeq$ is the smallest preorder which contains $\mathcal{R}_p^u \cup \mathcal{R}_t^u$, then $\preccurlyeq \ \subseteq \lesssim$. Finally by Theorem 3.3 of [MS], $\preccurlyeq \ \subseteq \approx$.

Consider an internal transition t of H. Let t have incoming arcs from internal places labeled $r_1, \ldots r_i$ and outgoing arcs to internal places labeled $r_{i+1}, \ldots r_n$. Then define

$$Proc(t, H) = (\mathbf{Rec}\ \bar{a}_{t\{r_1,\ldots r_i\}}.a_t.\mathbf{0} \mid a_t^{\{r_{i+1},\ldots r_n\}}.\mathbf{Rec}\ \bar{a}_t.a_t^{\{r_{i+1},\ldots r_n\}}.\mathbf{0}) \setminus \{a_t\}$$

where a_t is the channel name we associate to t.

Let $a = u(b)$, where $b \in Ports$, let H' be the hpn obtained from H by unfolding b and let p be a place of a. We define $Proc(a, p, H')$ by induction :

- Let p be included in a directed cycle inside a and have either more than one incoming arc or one incoming arc and a token. Let t be the transition of a which is part of the cycle and whose outgoing arc in the cycle ends in p. Let a' be obtained from a by adding an output place p', by removing the arc from t to p and by adding an arc from t to p'. Then $Proc(a, p, H') =$ **Rec** $Proc(a', p, H')$;
- Let p not be included in a directed cycle inside a, then :
  - if p has no outgoing arcs, $Proc(a, p, H') = \mathbf{0}$;
  - if p has an outgoing arc to a transition t, t has to have an outgoing arc to a place p', since the port is unfolded. Then :
    - $Proc(a, p, H') = \bar{a}_{\{r_1,\ldots r_n\}}.Proc(a, p', H')$, if t is an output transition with incoming arcs from the internal places labeled $r_1, \ldots r_n$;
    - $Proc(a, p, H') = a^{\{r_1,\ldots r_n\}}.Proc(a, p', H')$, if t is an input transition with outgoing arcs to the internal places labeled $r_1, \ldots r_n$;
- $Proc(a, p, H')$ is undefined otherwise.

We define $Proc(a, H') = Proc(a, p, H')$, where p is the place of a holding a token. Just a note, if a is an internal port and hence $a = b^!$ or $a = b^?$, for

some port name b, forget the primes ! and ? while defining the process. So that in $Proc(b^!, p, H')$, for example, output actions are $\bar{b}[\ldots]$ and input actions are $b[\ldots]$, instead of $\bar{b^!}[\ldots]$ and $b^![\ldots]$, respectively.

Lemma E.6.8. *Let H be a hpn whose ports are all unfolded and let a be a port of H. Then $Proc(a, H)$ is defined.*

Proof: $Proc(a, H) = Proc(a, p, H)$, where p is the place of a holding a token. Note that, by definition of port, p exists and is unique. Let p be included in a directed cycle and let t be the transition of a which is part of the cycle and whose outgoing arc in the cycle ends in p. Let also a' be obtained from a by adding an output place p', by removing the arc from t to p and by adding an arc from t to p'. Note that since a is unfolded, p is an output place and so t is an input transition. Then a' is still a port. Moreover the additions of p' and of its incoming arc preserve unfoldedness. Finally, since each node has at most one outgoing arc, a node can be contained in at most one cycle. In particular, p is not contained in a cycle inside a'. Then $Proc(a, p, H)$ is defined if $Proc(a', p, H)$ is defined.

Now take a place q of an unfolded port b (of H) and suppose q is not contained in a cycle inside b. If q has no outgoing arc, $Proc(b, q, H)$ is defined. Then let q have an outgoing arc to a transition t which has an outgoing arc to a place q'. Then $Proc(b, q, H)$ is defined if $Proc(b, q', H)$ is defined. Note that if q' is included in a cycle inside b, the incoming arc of q' which closes the cycle cannot be the one which comes from t since t cannot have more than one incoming arc, by construction of ports. Then q' has more than one incoming arc. Then $Proc(b, q', H)$ is defined by induction hypothesis.

We are now able to define

$$Proc(H) = Proc(a_1, u(H)) \mid \ldots \mid Proc(a_n, u(H)) \mid Proc(t_1, H) \mid \ldots \mid Proc(t_m, H)$$

where $a_1, \ldots a_n$ are the ports of $u(H)$ and $t_1, \ldots t_m$ are the internal transitions of H. Then let $Conf(H) = \langle Proc(H), S_H \rangle$, where S_H is the multiset of labels of internal places of H with a token such that a label appears in S_H as many times as the number of tokens in the corresponding place. Finally let $ch(Ports)$ be the set of names of ports in $Ports$, then $\Gamma_H : ch(Ports) \to \{!, ?\}$ is the function which associates ! to its active ports' names and ? to its passive ports' names.

Lemma E.6.9. *Let H be a transparent hpn. Then $[\![Conf(H)]\!]_{\Gamma_H} \approx u(H)$.*

Proof: Let S_H be the multiset of labels of internal places containing as many occurrences of each label as the number of tokens in the corresponding place. Let us do a few remarks.

For each internal transition t of H, both $[\![Conf(H)]\!]_{\Gamma_H}$ and $u(H)$ contain the associated link $u(t)$.

For each port a of H, $u(H)$ contains the unfolding $u(a)$ and $[\![Conf(H)]\!]_{\Gamma_H}$ contains a port a' such that $u(a)$ is a harmless extension of a'. In order to see this, note that to any transition t which is found on the maximal directed path inside $u(a)$ which starts from the place of $u(a)$ with a token, $Proc(u(a), u(H))$

associates an action of the same type (input or output) and with a set of labels which is just the set of labels of internal places connected to t. The interpretation function inverses this operation. Then what is left out are just some "harmless leaves".

Neither $u(H)$ nor $[\![Conf(H)]\!]_{\Gamma_H}$ contain internal transitions, as all of them have been unfolded to a link of two internal ports. However $u(H)$ may contain some disconnected empty internal places which go inevitably lost when $Proc$ is applied, but since they are disconnected they do no harm.

Then by Lemma E.6.5 and by Theorem 3.3 of [MS], $[\![Conf(H)]\!]_{\Gamma_H} \approx u(H)$.

Proof of Theorem E.5.3

Note that $H \lessdot u(H)$. Then by Lemma E.6.7 we have $H \approx u(H)$. By Lemma E.6.9 we have $[\![Conf(H)]\!]_{\Gamma_H} \approx u(H)$. Then $H \approx [\![Conf(H)]\!]_{\Gamma_H}$. □