Etude de la programmation logico-fonctionnelle concurrente

Wendelin Serwe

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N ° attribué par la bibliothèque

T H È S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Informatique : Systèmes et Communications

préparée au laboratoire LEIBNIZ – IMAG

dans le cadre de

l'**École Doctorale Mathématiques, Sciences et Technologie de l'Information, Informatique**

présentée et soutenue publiquement

par

Wendelin Bernhard SERWE

le 15 mars 2002

Étude de la programmation logico-fonctionnelle concurrente

Directeur de thèse : M. Rachid ECHAHED

JURY

Mme. Brigitte PLATEAU	Président
M. Mario RODRÍQUEZ-ARTALEJO	Rapporteur
M. Martin WIRSING	Rapporteur
M. Rachid ECHAHED	Directeur de thèse
M. Didier BERT	Examinateur
M. François FAGES	Examinateur
M. Philippe JORRAND	Examinateur

Remerciements

Tout d'abord je tiens à exprimer toute ma gratitude aux membres du jury :

- M. Rachid Echahed, mon directeur de thèse, pour ses conseils, discussions et encouragements précieuses, qui ont été indispensables à l'élaboration de cette thèse.
- Mme. Brigitte Plateau qui m'a fait l'honneur de présider le jury de thèse.
- M. Mario Rodríquez-Artalejo et M. Martin Wirsing pour m'avoir fait l'honneur de juger ce travail en tant que rapporteurs.
- M. Didier Bert, M. François Fages et M. Philippe Jorrand pour avoir accepté de le juger en tant qu'examinateurs.

De plus, je remercie toutes les personnes qui ont participé d'une manière ou d'une autre à l'élaboration de cette thèse, en particulier :

- toute l'équipe Programmation Multi-Paradigme (PMP) du laboratoire Leibniz au sein de laquelle ce travail a été mené, c'est-à-dire Jérémie Blanc, Jean-Christophe Janodet et Frédéric Prost, ainsi que Bruno Galmar et Marc Perache. Ils ont participé directement au travail par leurs relectures, discussions et implantations.
- M. Michael Hanus et son équipe aux universités de Aix-la-chapelle et Kiel, en particulier Bernd Brassel, Klaus Höppner et Frank Steiner pour les différentes discussions que nous avons eues.
- Jérémie Blanc, Olivier et Cécile Lavoisy et Marie-Luise Schneider qui m'ont aidé à améliorer la rédaction du résumé en français du mémoire.
- tous les membres du laboratoire Leibniz pour l'ambiance chaleureuse qu'ils ont su créer.
- toute l'équipe de la médiatheque pour son aide précieuse dans mes recherches bibliographiques.
- mes parents, mes amis et tous les autres qui m'ont soutenu pendant ma thèse.

Contents

Chapitre I

Étude des langages logico-fonctionnels concurrents

I.1 Introduction

Les programmes informatiques sont de plus en plus omniprésents du fait de l'utilisation des moyens de traitement d'information qui se trouvent dans pratiquement tous les appareils de la vie quotidienne, comme par exemple les systèmes de paiement électroniques, les voitures ou même les lave-linges. Néanmoins, l'état de l'art de la programmation ne permet pas d'éviter les problèmes liés à des programmes erronés et non fiables. On peut donc dire que nous sommes toujours au début d'une *science* de la programmation [Dij01]. Un moyen pour pallier ces problèmes consiste à améliorer les méthodes et outils utilisés pour la construction de programmes.

Un outil particulier pour l'expression des programmes est le *langage* de programmation. Il est admis en linguistique et informatique que le langage utilisé pour l'expression d'une pensée a une influence sur l'idée. Même si la conjecture de Church [Chu36] semble suggérer le contraire, la plupart des programmeurs vont admettre que certains langages sont plus au moins adaptés pour l'expression de différents aspects d'un système. Donc, un langage *multiparadigme*, c'est-à-dire un langage qui combine plusieurs styles de programmation, offre aux programmeurs la possibilité de choisir pour chaque partie d'un système le concept le plus approprié.

Dès le début de la programmation, la recherche sur les langages de programmation a été dirigée vers un niveau de description de plus en plus abstrait. Un haut niveau d'abstraction est souhaitable, car il permet au programmeur de se concentrer sur les points essentiels du système en faisant abstraction des détails superflus. Ainsi, un haut niveau d'abstraction permet des programmes concis et proches de la description du système. Une sémantique bien définie est une deuxième propriété souhaitable, car elle est indispensable pour la compréhension et la validation d'un programme et de ses propriétés.

Un langage dit *déclaratif* permet au programmeur de déclarer le problème à résoudre au lieu de préciser pas à pas une possibilité de résolution du problème. Ainsi, les langages déclaratifs sont par leur nature des langages de très haut niveau. Les langages déclaratifs que nous considérons dans la suite de ce mémoire sont les langages logiques,

fonctionnels et logico-fonctionnels. Ces langages sont fondés sur les notions de *fonctions* et de *prédicats* qui ont été utilisées pour la description d'algorithmes avant même l'invention des ordinateurs.

Néanmoins, les concepts théoriques sous-jacents aux langages déclaratifs sont insuffisants pour la description aisée des systèmes complexes qui nécessitent l'interactivité, la concurrence et la distribution [Tur39, Mil93a, Weg98]. L'utilité de la concurrence pour obtenir une meilleure structuration des programmes a été montrée par exemple pour la définition de systèmes de fenêtrage pour les langages déclaratifs [Pik89, GR93a, FPJ95].

La notion de *processus* a été introduite comme concept ou abstraction pour la description de systèmes concurrents. Cette notion est étudiée sous la forme d'*algèbres de processus* ou de *calculs de processus*. D'une manière simplifiée, un processus est défini par les *actions* qu'il est capable d'exécuter. Cependant, les langages fondés uniquement sur les algèbres de processus doivent être enrichis pour offrir les notions de fonction et de prédicat sans le besoin de les coder en termes de processus.

Afin de faciliter la construction de systèmes complexes, la composition d'un système à partir de *composants* (existants) a été proposée. Cette approche permet de réduire la complexité d'un système en permettant la construction séparée de ses différents composants, ainsi que leur (ré-)utilisation dans d'autres systèmes. Toutefois, il n'existe pas de définition formelle généralement admise d'un composant.

Un formalisme qui combine les différentes approches de programmation mentionnées précédemment permettrait l'expression de la plupart des parties d'un système complexe à l'aide du concept le plus approprié. Ainsi, la description d'un tel système serait plus concise et lisible, car il n'y aurait plus besoin d'*encoder* une notion par une autre. De plus, une séparation claire des différentes notions ne saurait qu'augmenter la clarté d'un programme. Évidemment, cette séparation entraîne une perte par rapport à la flexibilité offerte par le langage et oblige le programmeur à suivre une certaine discipline de programmation, ce qui n'est plus perçu comme trop contraignant aujourd'hui.

De nombreux langages et modèles de programmation intégrant les différent notions ont été proposés. Nous donnons un bref aperçu dans le paragraphe I.6. Dans ce mémoire nous explorons un nouveau modèle de calcul, ou un nouveau cadre pour les langages de programmation qui permet la construction de systèmes complexes à l'aide de composants. Ces langages de programmation combinent la programmation déclarative avec la concurrence sous forme de processus mobiles. De plus, le formalisme proposé est une extension conservatrice à la fois des langages déclaratifs et des algèbres de processus.

⋆ ⋆ ⋆

Le reste de ce chapitre est organisé comme suit. Nous présentons notre modèle au paragraphe suivant et donnons sa sémantique formelle au paragraphe I.3. Le paragraphe I.4 présente une méthode pour l'analyse de la confidentialité entre les processus d'un composant. Nous présentons notre prototype d'une plate-forme de programmation multiparadigme au paragraphe I.5. Le paragraphe I.6 donne une brève comparaison avec quelques travaux voisins, c'est-à-dire des langages et modèles de programmation qui visent des objectifs similaires aux nôtres. Nous concluons en paragraphe I.7 en apportant quelques perspectives.

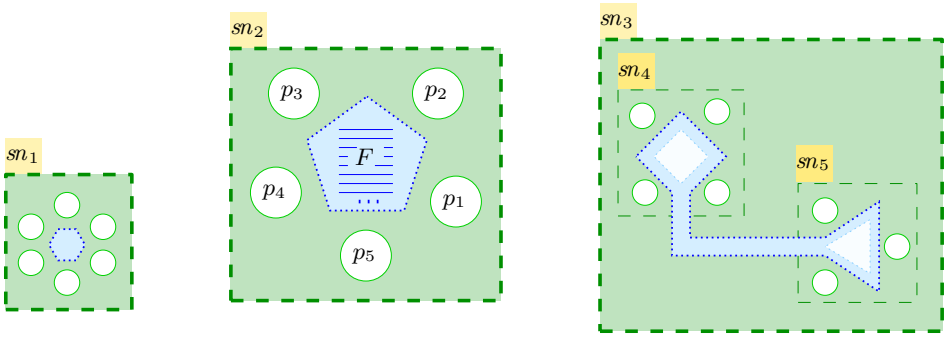

FIG. I.1 – Système en exécution

I.2 Un modèle de calcul pour la programmation déclarative concurrente

Nous modélisons un système par un ensemble de *composants* qui interagissent entre eux par l'envoi de messages. Chaque composant est identifié par son *nom* qui peut être considéré comme l'*adresse* du composant. À l'intérieur, un composant est constitué d'un ensemble de *processus* p_i et d'un *store* F, c'est-à-dire un programme déclaratif classique. Un store peut être vu comme une présentation statique d'une théorie qui décrit une situation. Les processus communiquent par la modification des stores, c'est-à-dire en changeant d'une manière non-monotone la théorie décrite par le store, par exemple, en ajoutant ou en enlevant une formule. Nous appelons ces modifications du store *actions*. Un message envoyé d'un composant à un autre correspond à une action à exécuter sur le store du composant distant. Un système de trois composants, nommés sn_1, sn_2 et sn_3, en exécution est présenté en figure I.1.

Dans ce mémoire (et plus particulièrement dans ce résumé) nous nous concentrons sur la *définition* d'un composant, et nous effleurons uniquement les problèmes liés à la composition des composants. Par exemple, le composant sn_3 présenté en figure I.1 est composé des deux composants sn_4 et sn_5. Cependant, la structure interne de sn_3 peut être masquée en prenant l'union disjointe des stores et des ensembles de processus des composants sn_4 et sn_5.

Afin de clarifier la description d'un composant nous suggérons la stratification en plusieurs niveaux, comme le présente la figure I.2. Le niveau le plus bas correspond à la description du store, c'est-à-dire les sortes, fonctions et prédicats sont définis à ce niveau. Un deuxième (méta-)niveau est nécessaire pour la description des actions, car les actions manipulent les stores comme données. Finalement, les processus utilisent à la fois les stores et les actions, et leur description demande un troisième niveau. Nous reviendrons plus en détail aux différents niveaux au paragraphe I.2.3.

I.2.1 Stores

Comme mentionné précédemment, nous appelons *store* un programme déclaratif classique. Notre approche pour la combinaison de la programmation déclarative et de la concurrence est générique dans le sens qu'elle est indépendante du langage déclaratif

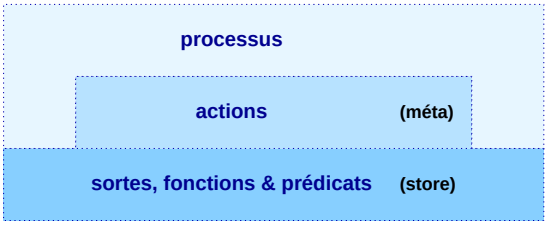

FIG. I.2 – Niveaux d'une description d'un composant (vision simplifiée)

utilisé pour la description d'un store.

I.1 Définition. *Un* store F *est un programme déclaratif classique, i.e.,* $F = \langle \Sigma, \mathcal{R} \rangle$*, (écrit dans le langage* $\mathcal{L}$*), composé d'une* signature Σ *et d'un ensemble de* règles *(aussi appelées* phrases *ou* formules*). Une* signature $\Sigma = \langle S, \Omega \rangle$ *est une paire d'un ensemble de* sortes S *et d'une famille de symboles d'opérateurs, tel que* Σ *contienne au moins la sorte* `Truth` *avec son constructeur* TRUE*. Nous notons la famille d'ensembles des* termes *pour une signature* Σ *et variables* X *par* $T^{\mathcal{L}}(\Sigma, X)$*. De plus, nous disposons d'un prédicat* $eval_{\mathcal{L}}(F, t)$ *(également écrit comme* $F \vdash_{\mathcal{L}} t$*), qui est valide si le terme* t *(de sorte* `Truth`*) peut être réduit vers* TRUE *en utilisant les règles du store* $F = \langle \Sigma, \mathcal{R} \rangle$*.*

Le prédicat $eval_{\mathcal{L}}$ (ou la relation $\vdash_{\mathcal{L}}$) correspond à un test de validité, comme en programmation logique. Notons que la définition d'un store est similaire à la notion d'un système logique dans le contexte des institutions [GB92]. Par la suite, nous omettons le langage $\mathcal{L}$ s'il peut être déduite du contexte.

I.2 Exemple. *Le prédicat* $eval_{\mathcal{TOY}}$ *(respectivement,* $eval_{Curry}$*) est défini par la sur-réduction (conditionnelle), c'est-à-dire la sémantique classique de* $\mathcal{TOY}$ [LFSH99] *(respectivement, Curry* [HAK+00b]*). Notons qu'un programme en* $\mathcal{TOY}$ *(respectivement, en Curry) est un ensemble de règles et satisfait donc la définition I.1.*

I.2.2 Actions

La sémantique d'un processus étant l'ensemble des séquences (en sémantique linéaire) ou le graphe (en sémantique arborescente) d'*actions* qu'il est capable d'exécuter, il est clair que la notion d'action est centrale pour la définition d'un processus. La plupart des algèbres de processus considèrent des actions abstraites, c'est-à-dire les actions sont vues comme des éléments d'un vocabulaire. Or dans notre modèle, l'exécution d'une action a l'effet de modifier le store. Nous devons donc spécifier davantage la notion d'action.

I.2.2.1 Méta-signatures

Comme une action transforme un store en un autre, nous définissons une action comme une fonction totale récursive d'un store vers un store. La définition d'une action nécessite ainsi la représentation d'un programme déclaratif comme un terme ou un élément d'un type abstrait de données particulier. Nous appelons la signature d'un tel type une *méta-signature*, car elle se situe à un niveau méta par rapport au niveau du store (cf. figure I.2).

I.3 Définition. *Soit $\mathcal{L}$ un langage déclaratif. Une* méta-signature *pour $\mathcal{L}$ est une paire $\mathrm{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$ d'un ensemble de* méta-sortes *$M_{\mathcal{L}}$ et d'une famille de symboles de* méta-fonctions *$MO_{\mathcal{L}}$, tel que $M_{\mathcal{L}}$ contienne au moins les sortes correspondant aux entités syntaxiques du langage $\mathcal{L}$.*

Comme exemples de méta-signatures, citons `FlatCurry` [Han, HAK+00a], la représentation intermédiaire pour les langages logico-fonctionnels (en particulier Curry) ou la sorte `Module` utilisée dans le module `META-LEVEL` de Maude [CDE+98, CDE+99] pour la représentation de modules Maude. Un exemple d'une méta-signature pour un langage logico-fonctionnel simple est donné en figure 3.3 à la page 90.

Parce que les différents langages déclaratifs peuvent être implantés en utilisant différents langages de programmation, nous ne spécifions pas la façon dont les méta-fonctions $MO_{\mathcal{L}}$ d'une méta-signature $\mathrm{M}\Sigma_{\mathcal{L}}$ sont définies. En effet, leur définition va dépendre du langage utilisé pour l'implantation du store.

I.2.2.2 Actions élémentaires

À partir de la notion de méta-signature, nous pouvons définir la notion d'une action élémentaire, comme une fonction d'un store vers un autre.

I.4 Définition. *Soit $\mathcal{L}$ un langage déclaratif et soit $\mathrm{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$ la méta-signature associée. Une* action élémentaire *est définie comme une fonction totale récursive dont le profil est de la forme $\mathtt{a} : \mathtt{s}_1 \to \ldots \to \mathtt{s}_n \to \mathtt{store} \to \mathtt{store}$ où $\mathtt{s}_i \in M_{\mathcal{L}}$ $(\forall i \in \{1; \ldots; n\}, n \geq 0)$ et $\mathtt{store}$ $(\in M_{\mathcal{L}})$ est la méta-sorte de programmes écrits en $\mathcal{L}$. Nous demandons également que le store retourné par une action élémentaire soit bien formé, c'est-à-dire un programme acceptable pour le langage $\mathcal{L}$.*

Comme pour les méta-fonctions, nous ne spécifions pas un formalisme particulier pour la définition des actions, car il va dépendre du langage d'implantation des stores. Évidemment, indépendamment du formalisme utilisé pour la définition des actions, nous devons nous assurer que les définitions des actions respectent les conditions de la définition I.4, à savoir la *totalité* et la *récursivité*. La totalité garantie que l'action élémentaire peut être appliquée à tout store, et la récursivité assure que l'exécution de l'action élémentaire se termine dans un temps fini. Ces deux propriétés réunies avec la condition que le store résultant soit bien formé, ces deux conditions assurent qu'il n'y aura pas de problème lors de l'exécution dans le sens que le store d'un composant soit toujours bien formé.

I.2.2.3 Exemples d'actions élémentaires

Nous terminons ce paragraphe en donnant quelques exemples d'actions. L'action élémentaire tell (respectivement, del) permet d'ajouter (respectivement, enlever) une règle du store. L'affectation est probablement l'action la plus répandue. L'exécution de $c := v$ modifie la définition de la constante[1] c vers la valeur v, c'est-à-dire une méta-représentation d'un terme. Figure I.3 donne l'exemple de l'affectation $c := 42$. L'effet de l'exécution de cette action est la transformation du store contenant la règle $c \to 23$ dans un store contenant la règle $c \to 42$.

[1] Dans les langages impératifs classiques, c est habituellement considéré comme une *variable*.

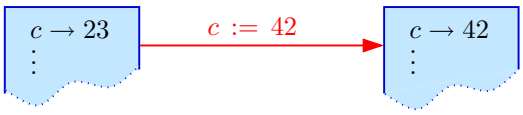

FIG. I.3 – Exemple de l'exécution d'une affectation

Notons qu'il y a plusieurs possibilités pour définir ces actions. Par exemple, si les règles d'un store sont stockées dans une liste, au moins deux possibilités s'offrent pour l'ajout d'une règle, à savoir au début ou à la fin de la liste (cf. les « prédicats » `asserta` et `assertz` de Prolog [DEDC96, pages 44 – 47]). Une autre différence est liée à la multiplicité d'une règle dans le store : en programmation concurrente par contraintes (ccp) classique [Sar93], l'ajout d'une contrainte impliquée par le store n'a pas d'effet. Or elle est importante pour un store fondé sur la logique linéaire [Gir87]. D'une manière similaire, dans le cas d'une affectation, il semble raisonnable de « normaliser » la valeur v avant la modification du store. Afin de permettre au programmeur de disposer toujours de l'action la plus appropriée, notre modèle permet la *définition* des actions utilisées par le programmeur.

À part les actions qui modifient l'ensemble des règles d'un store, nous avons besoin d'actions qui modifient la *signature* du store. La création de nouveaux symboles de fonctions est réalisée par l'action élémentaire new. Intuitivement, l'exécution de $\mathsf{new}(f, s)$ enrichit la signature par le symbole f de sorte s. Évidemment, nous avons besoin d'actions similaires à new pour tous les types de symboles d'une signature. L'ensemble de ces actions dépend donc du langage utilisé pour la description du store. Dans le reste de ce mémoire, nous notons $\mathcal{N}\big(\mathsf{a}(t_1, \ldots, t_n)\big)$ la signature de *nouveaux symboles* ajoutés par l'exécution de l'action élémentaire $\mathsf{a}(t_1, \ldots, t_n)$.

I.2.3 Signature de composant

Un composant est défini comme une partie d'un système, et la description d'un composant dépend donc du reste du système. Dans notre modèle, un composant peut être identifié par son *nom de store*, et nous pouvons représenter un système par l'ensemble $\mathcal{SN}$ de tous les noms de store de l'ensemble des composants.

Avant de présenter la définition des processus et d'un composant, nous introduisons dans ce paragraphe la notion de *signature de composant*. Intuitivement, une telle signature déclare tous les symboles utilisés dans la description d'un composant. Elle comporte donc en particulier une signature et une méta-signature comme introduites auparavant. La figure I.4 (un affinement de la figure I.2) montre la stratification des différents symboles d'une signature de composant.

I.5 Définition. *Soit $\mathcal{SN}$ un ensemble de noms de stores. Pour un nom de store $\widehat{sn} \in \mathcal{SN}$ et un langage déclaratif $\mathcal{L}$, nous définissons la* signature de composant $\mathbb{C}\Sigma$ *comme un octuple* $\mathbb{C}\Sigma = \langle \Sigma, \mathbb{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ *où*

- $\Sigma = \langle S, \Omega \rangle$ *est une* signature *d'un store, c'est-à-dire d'un programme écrit en $\mathcal{L}$,*
- $\mathbb{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$ *est une* méta-signature *pour $\mathcal{L}$,*
- *A est une famille ($\overline{(S \uplus M_{\mathcal{L}})}^2$-indicée) de symboles d'*actions*,*

[2] Nous notons $\overline{E}$ l'ensemble de sortes construites à partir de l'ensemble de sortes de base E.

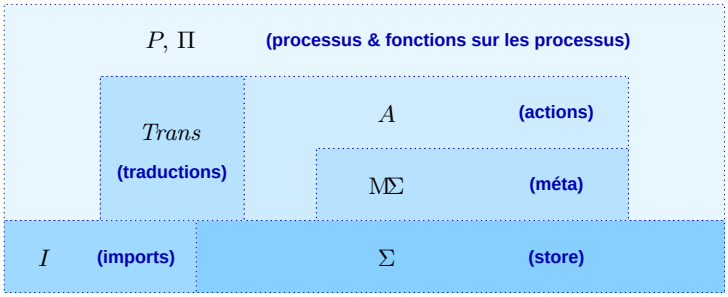

FIG. I.4 – Niveaux d'une description de système

- $I = \left\{ \mathbb{\Sigma}_{sn} = \langle \Sigma_{sn}, \mathbb{M}\Sigma_{\mathcal{L}_{sn}}, A_{sn} \rangle \,\middle|\, sn \in (SN \setminus \{\widehat{sn}\}) \right\}$ *est une famille (SN-indicée) de* signatures importées $\mathbb{\Sigma}_{sn}$*, c'est-à-dire de triplets de signatures* $\Sigma_{sn} = \langle S_{sn}, \Omega_{sn} \rangle$*, méta-signatures* $\mathbb{M}\Sigma_{\mathcal{L}_{sn}} = \langle M_{\mathcal{L}_{sn}}, MO_{\mathcal{L}_{sn}} \rangle$ *et familles* $(\overline{(S_{sn} \uplus M_{\mathcal{L}_{sn}})}$*-indicées) de symboles d'actions* A_{sn} *(*$\mathcal{L}_{sn}$ *est le langage utilisé pour la description du store du composant* sn*),*
- *les* symboles exportés $E = \langle E_\Sigma, E_{\mathbb{M}\Sigma}, E_A \rangle$ *sont un triplet d'une sous-signature* E_Σ*, d'une sous-méta-signature* $E_{\mathbb{M}\Sigma}$ *et d'une sous-famille de symboles d'actions* E_A*, c'est-à-dire* $E_\Sigma \subseteq \Sigma$*,* $E_{\mathbb{M}\Sigma} \subseteq \mathbb{M}\Sigma$ *et* $E_A \subseteq A$*,*
- $Trans = \{ Tr_{sn} \mid sn \in SN \}$ *est une famille (SN-indicée) de familles de symboles de* traduction Tr_{sn}*,*
- P *est une famille (*$\overline{PS}$*-indicée) de symboles de* processus*, contenant au moins le processus sans paramètre* success*,*
- Π *est une famille (*$\overline{PS}$*-indicée) de symboles de* fonctions de processus

et où l'ensemble de sortes PS est défini par

$$PS \;=\; S \uplus M_{\mathcal{L}} \uplus \Big(\biguplus_{sn \in (SN \setminus \{\widehat{sn}\})} (S_{sn} \uplus M_{\mathcal{L}_{sn}}) \Big) \uplus \{\texttt{action}; \texttt{process}; \texttt{storename}\} \qquad \text{(I.1a)}$$

Selon la définition I.5 et la figure I.4, un programmeur doit spécifier les signatures des stores, c'est-à-dire la signature du store initial du composant et les signatures importées des autres composants avec lesquels le composant interagit. Notons que ces signatures ne sont pas nécessairement du même type, car les stores des composants peuvent être décrits en différents langages déclaratifs.

Outre les signatures, une signature de composant contient également les méta-signatures associées qui sont utilisées pour la définition des actions comme mentionné au paragraphe I.2.2. La définition I.5 étend la définition I.4 en admettant comme paramètres d'une action élémentaire des termes du store en plus des méta-termes. Ce nouveau type de paramètre doit être compris comme paramètre de la méta-sorte correspondant aux termes, qui est obtenu par l'application implicite de *reify*. Intuitivement, *reify* associe à un terme sa représentation en tant que méta-terme. L'utilisation des sortes du store dans les profils des actions permet, en plus de faciliter la description des processus, de vérifier statiquement le typage des arguments d'une action. Par exemple, pour l'action $c := v$, il est possible de vérifier statiquement si les types de c et v sont les mêmes, ce qui évite de vérifier le typage du store après chaque exécution d'une action.

Les symboles exportés par un composant sont ceux que d'autres composants peuvent

utiliser. Évidemment, les symboles exportés doivent être un sous-ensemble des symboles définis par le composant.

La communication entre processus est fondée sur la modification des stores. Si un processus veut communiquer une valeur à un processus d'un autre composant dont le store est décrit dans un langage différent, cette valeur doit être traduite dans le langage utilisé dans le store distant. Pour ce faire, un programmeur doit spécifier les *traductions*.

Les deux dernières familles de symboles d'une signature de composant sont les symboles des processus et les fonctions de processus. La différence entre les processus et les fonctions de processus est similaire à la distinction entre les fonctions définies et les constructeurs dans la plupart des langages déclaratifs récents, comme par exemple Curry [HAK+00b] ou $\mathcal{TOY}$ [LFSH99]. Contrairement aux fonctions de processus qui sont définies par des règles de réécriture, les processus sont définis par les définitions de processus.

Afin de définir les expressions d'actions et de processus par la suite, nous introduisons, la notion de *terme de composant*.

I.6 Définition. *Soit* $\mathrm{C}\Sigma$ *une signature de composant et* X *une famille de variables. Comme dans l'équation* (I.1a)*, nous considérons la famille d'opérations suivante :*

$$PO \;=\; \Big(\biguplus_{sn\in SN} (\Omega_{sn} \uplus MO_{\mathcal{L}_{sn}} \uplus A_{sn})\Big) \uplus Trans \uplus P \uplus \Pi \uplus SN \tag{I.1b}$$

Nous définissons la famille de termes de composant $CT(\mathrm{C}\Sigma, X)$ *comme les termes construits à partir de la signature* $\mathrm{P}\Sigma = \langle PS, PO\rangle$ *et les variables* X*, c'est-à-dire*

$$CT(\mathrm{C}\Sigma, X) \;=\; T(\mathrm{P}\Sigma, X) \tag{I.2}$$

I.2.4 Interactions

Dans notre modèle, les processus d'un même composant utilisent le store comme moyen de communication. Tous les processus ont accès au store, et les modifications apportées par l'exécution d'une action sont ainsi visibles pour les autres processus du composant. L'interaction entre les processus des différents composants est fondée sur le même principe : en effet, tout processus peut modifier les stores des autres composants du système. Dans ce paragraphe nous introduisons les notions nécessaires pour cette interaction.

I.2.4.1 Symboles importés et exportés

Nous présentons d'abord les symboles importés d'un autre composant, ce qui est à distinguer des symboles importés au niveau d'un store. En effet, si le langage déclaratif le permet, la description d'un store peut être répartie dans plusieurs modules qui exportent et importent leurs définitions respectives, afin de faciliter la description du store. D'autre part, les symboles importés d'un autre composant sont nécessaires à la description de l'interaction entre composants, car l'exécution d'une action élémentaire sur le store d'un autre composant demande la construction des paramètres correspondants.

La nécessité d'importer les symboles d'un autre composant sn vient de la possibilité d'exécuter des actions sur le store de sn. De ce fait, il est évident que les symboles d'actions exécutables sur le store de sn doivent être importés. Comme les paramètres d'une action peuvent être des termes et des méta-termes, la construction de ces paramètres nécessite d'importer également la signature et la méta-signature de sn.

Les symboles exportés par un composant décrivent les symboles qui peuvent être utilisés par les autres composants du système. Évidemment, les symboles exportés doivent être un sous-ensemble des symboles définis par le composant.

I.2.4.2 Traductions

Dans le cas d'un système dont les composants sont décrits en utilisant des langages déclaratifs différents, la communication d'une valeur d'un store à un autre nécessite la *traduction* de cette valeur. Afin d'assurer une traduction unique, nous définissons une traduction comme une fonction totale récursive. Notons qu'une traduction ne peut être définie dans un des deux stores concernés, car elle représente des associations entre les deux stores. En fait, une traduction peut être définie dans un « langage union » qui combine les deux stores.

Nous proposons de spécifier une traduction en plusieurs étapes. Dans un premier temps, la valeur à traduire est réduite vers une « forme normale » en utilisant la sémantique opérationnelle du store de départ. La deuxième étape concerne la génération d'une expression correspondant à la valeur, à l'aide d'une fonction de traduction. Finalement, la valeur correspondante peut être obtenue en calculant la « forme normale » de l'expression obtenue lors de la deuxième étape. Ainsi, nous permettons au programmeur de ne spécifier que la fonction de traduction et de déléguer les autres étapes à l'implantation de la plate-forme.

La spécification des fonctions de traduction utilise un système de réécriture dédié, dont la signature est l'union des signatures des deux stores. Ainsi, la *signature de traduction* entre un store de signature $\Sigma_1 = \langle S_1, \Omega_1 \rangle$ et un store de signature $\Sigma_2 = \langle S_2, \Omega_2 \rangle$ est définie par $\mathfrak{T}\Sigma_{\Sigma_1, \Sigma_2} = \big\langle (S_1 \uplus S_2), (\Omega_1 \uplus \Omega_2 \uplus Tr) \big\rangle$, où Tr est une famille de symboles de traduction (cf. définition I.5). Les fonctions de traduction peuvent ainsi être définies par des règles de réécriture classiques sur la signature de traduction. Afin de distinguer les règles de traduction des règles des stores, nous appelons les premières *t-rules*.

Une autre possibilité de modéliser l'interaction entre deux composants en différents langages est fondée sur l'hypothèse que les deux composants « comprennent » un troisième langage. Des exemples de cette approche sont, parmi d'autres, UTS [HS87], IDL [COR01] ou l'interface de `ocaml` (respectivement, ADA) vers C [LDG+01, chapitre 17] (respectivement, [Ada95, chapitre B.3]). À part le fait qu'un tel langage n'existe pas dans tous les cas, les traductions des structures de données plus complexes doivent toujours être implantées. De plus, l'utilisation de ces langages n'est pas toujours aisée, à cause des paramètres particuliers nécessaires lors de l'appel du compilateur.

I.2.5 Processus

Les processus d'un composant sont spécifiés dans le style d'une algèbre de processus, cf. par exemple [Fok00, BW90]. Les processus de base peuvent être combinés à l'aide

d'opérateurs sur les processus. De plus, les fonctions de processus permettent la description des processus dans un style fonctionnel. Dans ce paragraphe, nous présentons la définition des processus. Nous commençons avec les processus de base et poursuivons avec les opérateurs sur les processus. Nous terminons avec les règles qui définissent les processus.

I.2.5.1 Expressions d'action et actions gardées

Les processus les plus basiques dans notre modèle sont les *actions gardées*. Intuitivement, une action gardée est une paire composée d'une *garde* et d'une *expression d'action*. Intuitivement, une *expression d'action* est un terme bien formé à partir d'une signature de composant, enrichie de deux constructeurs, à savoir le couplage d'un appel à une action élémentaire avec le nom du composant sur lequel l'action est à exécuter, et la composition séquentielle de deux expressions d'actions.

I.7 Définition. *Soit* $C\Sigma = \langle \Sigma, M\Sigma_{\mathcal{L}}, A, I, E, Trans, P, \Pi \rangle$ *une signature de composant et* X *une famille de variables, où PS est défini par l'équation* (I.1a)*. Nous définissons l'ensemble des* expressions d'action $\mathcal{A}(C\Sigma, X)$ *comme l'ensemble de termes de composant de sorte* `action`*, c'est-à-dire* $\mathcal{A}(C\Sigma, X) = CT_{\texttt{action}}(\widetilde{C\Sigma}, X)$ *où la signature de composant* $\widetilde{C\Sigma}$ *est l'enrichissement de* $C\Sigma$ *avec les deux* constructeurs *de la sorte* `action` *:*[3]

1. $\langle \bullet, \bullet \rangle$ *de profil* $\texttt{storename} \rightarrow (\texttt{store}_{\mathcal{L}_{sn}} \rightarrow \texttt{store}_{\mathcal{L}_{sn}}) \rightarrow \texttt{action}$ *(pour tout nom de composant* $sn \in SN$*) et*
2. $\bullet\,;\bullet$ *de profil* $\texttt{action} \rightarrow \texttt{action} \rightarrow \texttt{action}$*.*

Notons que la définition I.7 implique qu'une expression d'action est l'une des trois formes suivantes : $\langle sn, \mathtt{a}(t_1, \ldots, t_n) \rangle$, $a_1\,;a_2$ ou $pf(a_1, \ldots, a_n)$ (où $pf \in \Pi$ est une fonction de processus dont la sorte du résultat est `action`). Une expression d'action qui ne contient pas de fonction de processus, est dite en *forme normale*. L'ensemble des expressions d'action en forme normale est noté $\mathcal{A}^{\mathcal{N}}(C\Sigma, X)$ ou $\mathcal{A}^{\mathcal{N}}(C\Sigma, X, sn)$, si toutes les actions sont associées au nom du composant sn.

L'extension de la notation $\mathcal{N}$ désignant la signature nouvelle ajoutée par une action élémentaire aux expressions d'actions est immédiate. Nous disons qu'une expression d'action utilise les nouveaux symboles d'une manière raisonnable, si ces nouveaux symboles ne sont pas utilisés avant leur introduction, c'est-à-dire si dans une composition séquentielle de deux expressions d'action, comme par exemple $a_1\,;a_2$, l'action a_2 est définie par rapport à la signature de composant enrichie avec la nouvelle signature introduite par a_1.

Intuitivement, une action gardée est une paire composée d'une garde et d'une expression d'action.

I.8 Définition. *Soit* $C\Sigma = \langle \Sigma, M\Sigma_{\mathcal{L}}, A, I, E, Trans, P, \Pi \rangle$ *une signature de composant (pour un langage déclaratif* $\mathcal{L}$ *et un nom de composant* $\widehat{sn}$*) et* X *une famille de variables. Une* action gardée *est une paire* $[g \Rightarrow a]$ *composée*

- *d'une* garde g *(de sorte)* `Truth` *du store local, c'est-à-dire* $g \in T^{\mathcal{L}}_{\texttt{Truth}}(\Sigma, X)$*, et*
- *d'une expression d'action* $a \in \mathcal{A}(C\Sigma, X)$ *qui utilise les nouveaux symboles d'une manière raisonnable.*

[3] Le symbole « • » indique les positions des paramètres d'un opérateur.

L'ensemble des actions gardées est noté $\mathcal{G}(\mathrm{C}\Sigma, X)$.

Selon la définition I.8, la garde d'une action gardée doit être un terme du store du composant local. Cette restriction est motivée par le fait que l'exécution d'une action gardée est atomique (cf. I.3.1), ce qui serait difficile à mettre en œuvre dans un contexte distribué.

I.2.5.2 Expressions de processus

Les expressions de processus sont définies comme des termes de composant sur une signature de composant enrichie par, d'une part, le constructeur des actions gardées, et, d'autre part, par les opérateurs de composition des processus connus des algèbres de processus, à savoir la composition séquentielle (;) et parallèle ($\|$) ainsi que le choix non-déterministe ($+$) et le choix avec priorité ($\oplus$).

I.9 Définition. *Soit* $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ *une signature de composant et* X *une famille de variables, où PS est défini par l'équation* (I.1a). *Nous définissons l'ensemble des* expressions de processus $\mathcal{P}(\mathrm{C}\Sigma, X)$ *comme l'ensemble de termes de composant de sorte* `process`*, c'est-à-dire* $\mathcal{P}(\mathrm{C}\Sigma, X) = CT_{\texttt{process}}(\widetilde{\widetilde{\mathrm{C}\Sigma}}, X)$ *où la signature de composant* $\widetilde{\widetilde{\mathrm{C}\Sigma}}$ *est l'enrichissement de* $\widetilde{\mathrm{C}\Sigma}$ *(cf. définition I.7) avec les* constructeurs *de la sorte* `process` *suivants :*

- $[\bullet \Rightarrow \bullet]$ *de profil* `Truth` $\rightarrow$ `action` $\rightarrow$ `process` *permet de construire des actions gardées (cf. définition I.8) et*
- $\bullet\,;\bullet$, $\bullet \parallel \bullet$, $\bullet + \bullet$ *et* $\bullet \oplus \bullet$ *de profil* `process` $\rightarrow$ `process` $\rightarrow$ `process`.

Les fonctions de processus sont définies par des règles de réécriture spécifiques que nous appelons *p-rules*. Par la suite nous appelons une expression de processus qui ne contient pas d'appel à une fonction de processus un *terme de processus*. L'ensemble des termes de processus en forme normale est noté $\mathcal{P}^{\mathcal{N}}(\mathrm{C}\Sigma, X)$. Une expression de processus qui ne contient pas d'action gardée, ni d'occurrence de l'opérateur $\oplus$ est appelée une *expression de processus restreinte*. Si les parties droites des p-rules définissant une fonction de processus sont toutes des expressions de processus restreintes, la fonction de processus est dite restreinte. Nous notons ${}^{r}\mathcal{P}(\mathrm{C}\Sigma, X)$ l'ensemble des expressions de processus qui ne contiennent pas d'appel à une fonction de processus non restreinte.

I.2.5.3 Définitions de processus

Un processus est défini par un ensemble de « *clauses* » ordonnées par priorité. Chaque clause est composée d'une garde et d'une expression de processus restreinte.

I.10 Définition. *Soit* $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ *une signature de composant. Une* définition du processus $\mathrm{q} \in P$ *est une phrase de la forme suivante :*

$$\mathrm{q}(x_1, \ldots, x_m) \Leftarrow \bigoplus_{i=1}^{n} \big([g_i \Rightarrow a_i]\,;{}^{r}p_i\big) \tag{I.3}$$

où (pour tout $i \in \{1; \ldots; n\}$*, tel que* $n > 0$*) :*

- $[g_i \Rightarrow a_i] \in \mathcal{G}\big(\mathrm{C}\Sigma, \{x_1; \ldots; x_m\}\big)$ *est une action gardée pour le composant local et*

– $^{\mathrm{r}}p_i \in {}^{r}\mathcal{P}\big(\widehat{\mathrm{C}\Sigma}, \{x_1; \ldots; x_m\}\big)$ *est une expression de processus restreinte, où la signature de composant* $\widehat{\mathrm{C}\Sigma}$ *est l'enrichissement de* $\mathrm{C}\Sigma$ *avec la nouvelle signature* $\mathcal{N}([g_i \Rightarrow a_i])$ *introduite par l'action gardée.*

Intuitivement, la sémantique opérationnelle d'un appel à un processus $\mathrm{q}(t_1, \ldots, t_m)$ est similaire à la construction des alternatives dans le langage des commandes gardées de [Dij75], c'est-à-dire la garde valide la plus prioritaire détermine la clause choisie. L'exécution d'une clause signifie l'exécution atomique de la séquence d'actions élémentaires obtenue par l'évaluation de l'expression d'action. Ensuite l'exécution continue par l'exécution du terme de processus obtenu en évaluant l'expression de processus restreinte.

I.2.6 Composants et systèmes

La définition d'un composant regroupe une signature de composant avec l'ensemble des définitions introduites précédemment.

I.11 Définition. *Soit* $\mathcal{SN}$ *un ensemble de noms de composant. Un* composant *est un octuple* $\mathcal{C} = \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \Pi\mathcal{R}, p^{\mathtt{i}} \rangle$ *où*

– $\widehat{sn} \in \mathcal{SN}$ *est le* nom de composant *du composant,*
– $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ *est une* signature de composant *pour* $\mathcal{SN}$ *et* $\widehat{sn}$*,*
– $F = \langle \Sigma, \mathcal{R} \rangle$ *est un* store*, également appelé le* store initial*,*
– $\mathfrak{A}$ *est un ensemble de* définitions d'actions *définissant les actions* A*,*
– $\mathcal{T}r$ *est un ensemble de* t-rules *définissant les* traductions Tr*,*
– $\mathcal{R}^{\mathrm{p}}$ *est un ensemble de* définitions de processus *définissant les processus* P*,*
– $\Pi\mathcal{R}$ *est un ensemble de* p-rules *définissant les fonctions de processus* Π *et*
– $p^{\mathtt{i}} \in {}^{r}\mathcal{P}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing)$ *est un terme de processus restreint et clos, également appelé* terme de processus initial*.*

On peut distinguer plusieurs parties dans la définition d'un composant. D'abord, nous avons les définitions qui sont *statiques*, dans le sens qu'elles ne changeront pas lors de l'exécution du composant. Ce sont le nom du composant $\widehat{sn}$, les symboles importés I et exportés E, mais aussi les actions A, traductions Tr, processus P et fonctions de processus Π avec leurs définitions respectives (c'est-à-dire $\mathfrak{A}$, $\mathcal{T}r$, $\mathcal{R}^{\mathrm{p}}$ et $\Pi\mathcal{R}$). D'autre part, le store F est *dynamique* et évolue lors de l'exécution du système. Finalement, le terme de processus initial ainsi que le store initial forment l'*initialisation* du composant.

La combinaison d'un composant à partir de deux composants $\mathcal{C}_1$ et $\mathcal{C}_2$ (cf. le composant sn_3 en figure I.1) correspond à prendre l'union disjointe de toutes les parties des composants (et la composition parallèle des termes de processus initiaux) :

$$\mathcal{C}_1 \,|||\, \mathcal{C}_2 = \langle \widehat{sn}, \mathrm{C}\Sigma_1 \uplus \mathrm{C}\Sigma_2, \mathcal{R}_1 \uplus \mathcal{R}_2, \mathfrak{A}_1 \uplus \mathfrak{A}_2, \mathcal{T}r_1 \uplus \mathcal{T}r_2, \mathcal{R}_1^{\mathrm{p}} \uplus \mathcal{R}_2^{\mathrm{p}}, \Pi\mathcal{R}_1 \uplus \Pi\mathcal{R}_2, p_1^{\mathtt{i}} \parallel p_2^{\mathtt{i}} \rangle$$

où les composants de départ sont définis par $\mathcal{C}_i = \langle sn_i, \mathrm{C}\Sigma_i, \mathcal{R}_i, \mathfrak{A}_i, \mathcal{T}r_i, \mathcal{R}_i^{\mathrm{p}}, \Pi\mathcal{R}_i, p_i^{\mathtt{i}} \rangle$ (pour $i \in \{1; 2\}$) et $\widehat{sn}$ est un *nouveau* nom de composant. Ainsi le composant $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ est défini par rapport à l'ensemble de noms de store $\big(\mathcal{SN} \setminus \{sn_1;\ sn_2\}\big) \cup \{\widehat{sn}\}$. L'utilisation de l'union disjointe pour la combinaison de composants permet de définir aisément un « adaptateur » qui transmet les actions arrivant au composant aux sous-composants concernés.

$$\mathsf{success}\,;p \;\equiv\; p \qquad\qquad \mathsf{success} \parallel p \;\equiv\; p \qquad\qquad (Unit_{\equiv})$$
$$p_1 \parallel p_2 \;\equiv\; p_2 \parallel p_1 \qquad\qquad p_1 + p_2 \;\equiv\; p_2 + p_1 \qquad\qquad (Comm_{\equiv})$$

FIG. I.5 – Schémas d'axiomes pour la congruence structurelle $\equiv$

Nous modélisons un système comme un ensemble de composants, qui doivent être définis par rapport au même ensemble de noms de store.

I.12 Définition. *Soit $\mathcal{S}$ un ensemble de composants $\mathcal{S} = \{\mathcal{C}_{sn_1}; \ldots; \mathcal{C}_{sn_n}\}$ et soit $\mathcal{SN}$ l'ensemble de noms de stores associés, c'est-à-dire $\mathcal{SN} = \{sn_1; \ldots; sn_n\}$. Nous appelons $\mathcal{S}$ un* système *si le composant $\mathcal{C}_{sn} = \langle sn_{sn}, \mathrm{C}\Sigma_{sn}, \mathcal{R}_{sn}, \mathfrak{A}_{sn}, \mathcal{T}r_{sn}, \mathcal{R}^{\mathrm{p}}_{sn}, \Pi\mathcal{R}_{sn}, p^{\mathtt{i}}_{sn}\rangle$ avec la signature de composant $\mathrm{C}\Sigma_{sn} = \langle \Sigma_{sn}, \mathrm{M}\Sigma_{sn}, A_{sn}, I_{sn}, E_{sn}, Trans_{sn}, P_{sn}, \Pi_{sn}\rangle$ est défini par rapport à $\mathcal{SN}$ ($\forall sn \in \mathcal{SN}$) tel que pour toute paire de noms de store sn_1, $sn_2 \in \mathcal{SN}$ avec $sn_1 \neq sn_2$ nous avons $\left(I_{sn_1}\right)_{sn_2} \subseteq E_{sn_2}$.*

I.3 Sémantique

Dans ce paragraphe nous exposons la sémantique du modèle présenté dans le paragraphe précédent. Nous détaillons d'abord la sémantique opérationnelle et donnons dans un deuxième temps une sémantique compositionnelle pour les processus d'un composant.

I.3.1 Sémantique opérationnelle

Nous présentons la sémantique de notre modèle en plusieurs étapes. En plus de la sémantique opérationnelle d'un processus que nous exposons d'abord, la sémantique opérationnelle d'un composant permet la résolution interactive de buts. Nous décrivons l'intégration de ces deux sémantiques dans un deuxième temps. Nous terminons par la sémantique opérationnelle d'un système.

I.3.1.1 Exécution des processus d'un composant

La sémantique des processus d'un composant $\mathcal{C}$ est définie par un système de transitions $\mathsf{T}_{\mathcal{C}} = \langle \mathsf{Q},\ \longrightarrow,\ \langle F, p^{\mathtt{i}}\rangle\rangle$. Les états de $\mathsf{T}_{\mathcal{C}}$ sont des couples d'un store et d'un terme de processus, l'état initial étant le couple du store initial et du terme de processus initial. La relation de transition $\longrightarrow$ est définie dans le style de la machine abstraite chimique [BB92] à l'aide d'une relation de congruence $\equiv$. Les schémas d'axiomes de $\equiv$ sont montrés en figure I.5 et $\longrightarrow$ est définie par les règles d'inférence de la figure I.6.

L'exécution d'une action gardée est décrite par la règle (R_{action}). La prémisse indique que la garde g doit être valide. La fonction *sel* prend un nom de store *sn* et une séquence d'actions élémentaires (c'est-à-dire une expression d'action en forme normale) et retourne la sous-séquence d'actions élémentaires qui sont destinées au store nommé *sn*. La modification du store est décrite à l'aide de la fonction *exec* définissant l'application (séquentielle) d'une séquence d'actions élémentaires sur un store. Notons que

$$\frac{p \equiv p' \qquad \langle F,\, p'\rangle \longrightarrow \langle F',\, p''\rangle \qquad p'' \equiv p'''}{\langle F,\, p\rangle \longrightarrow \langle F',\, p'''\rangle} \tag{$\mathsf{R}_{\equiv}$}$$

$$\frac{F \vdash g}{\begin{array}{l}\big\langle F,\ \big[g \Rightarrow \langle sn_1,\, a_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots ; \langle sn_n,\, a_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big]\big\rangle \longrightarrow \\ \quad \Big\langle exec\Big(sel\big(\widehat{sn},\ \langle sn_1,\, a_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots ; \langle sn_n,\, a_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big),\ F\Big),\ \mathsf{success}\Big\rangle\end{array}} \tag{R_{action}}$$

$$\frac{\big(\mathrm{q}(x_1,\, \ldots\, ,\, x_n) \Leftarrow \bigoplus_{i=1}^{m}([g_i \Rightarrow a_i]\,;p_i)\big) \in \mathcal{R}^{\mathrm{p}} \qquad \big\langle F,\ \big(\bigoplus_{i=1}^{m} rename([g_i \Rightarrow a_i\Downarrow]\,;p_i)\big)[v_j/x_j]\big\rangle \longrightarrow \langle F',\, p'\rangle}{\langle F,\, \mathrm{q}(v_1,\, \ldots\, ,\, v_n)\rangle \longrightarrow \langle F',\ p'\Downarrow\rangle} \tag{R_{call}}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1\,;p_2\rangle \longrightarrow \langle F',\, p_1'\,;p_2\rangle} \tag{$\mathsf{R}_{;}$}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 \parallel p_2\rangle \longrightarrow \langle F',\, p_1' \parallel p_2\rangle} \tag{$\mathsf{R}_{\parallel}$}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 + p_2\rangle \longrightarrow \langle F',\, p_1'\rangle} \tag{R_{+}}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \longrightarrow \langle F',\, p_1'\rangle} \tag{$\mathsf{R}_{\oplus}$}$$

$$\frac{\langle F,\, p_2\rangle \longrightarrow \langle F',\, p_2'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \longrightarrow \langle F',\, p_2'\rangle} \quad \text{si } \nexists\, p_1'(\not\equiv p_1),\ \nexists\, F'',\ \text{tel que } \langle F,\, p_1\rangle \longrightarrow \langle F'',\, p_1'\rangle \tag{$\mathsf{R}'_{\oplus}$}$$

FIG. I.6 – Règles d'inférence définissant la relation de transition $\longrightarrow$

l'évaluation de la garde et l'exécution de toute la séquence d'actions forment une seule transition, ce qui implique l'atomicité de l'exécution d'une action gardée.

Lors d'un appel de processus, l'expression d'action est évaluée dans le même store sur lequel les gardes sont évaluées, et l'expression de processus est évaluée dans le store après l'exécution de l'action (cf. la règle (R_{call})). Cette distinction traduit la composition séquentielle de l'action gardée et de l'expression de processus.

Les règles restantes sont classiques, à part les deux règles définissant l'opérateur $\oplus$. Ces dernières expriment que pour le terme de processus $p_1 \oplus p_2$, l'exécution de p_2 est possible uniquement si l'exécution de p_1 ne l'est pas.

I.3.1.2 Sémantique opérationnelle d'un composant

La sémantique opérationnelle d'un composant $\mathcal{C}$ comporte deux parties, à savoir l'exécution des processus (cf. le paragraphe précédent) et la résolution de buts, en utilisant la sémantique opérationnelle classique du langage utilisé pour la description du store. Nous décrivons l'intégration de ces deux facettes par un nouveau système de transitions $\mathcal{T}_{\mathcal{C}} = \big\langle \mathcal{Q},\ \longmapsto,\ \langle F,\, p^{\mathtt{i}},\, \mathtt{g}^{\mathtt{i}},\, \mathtt{g}^{\mathtt{i}}\rangle\big\rangle$, où $\mathtt{g}^{\mathtt{i}}$ dénote le but initial (qui peut être TRUE). Les états (ou configurations) de $\mathcal{T}_{\mathcal{C}}$ sont des quadruplets $\langle F,\, p,\, \mathtt{g}^{\mathtt{i}},\, \mathtt{g}\rangle$ d'un store F, d'un terme de processus p, du but initial $\mathtt{g}^{\mathtt{i}}$ et du but courant $\mathtt{g}$. Classiquement, les configurations d'un langage concurrent n'utilisent que les deux premières parties F et p,

$$\frac{\langle F, \mathtt{g}\rangle \rightsquigarrow \langle F, \mathtt{g}'\rangle}{\langle F, p, \mathtt{g}^{\mathtt{i}}, \mathtt{g}\rangle \longmapsto \langle F, p, \mathtt{g}^{\mathtt{i}}, \mathtt{g}'\rangle} \quad (\mathsf{G})$$

$$\frac{\langle F, p\rangle \longrightarrow \langle F', p'\rangle}{\langle F, p, \mathtt{g}^{\mathtt{i}}, \mathtt{g}\rangle \longmapsto \langle F', p', \mathtt{g}^{\mathtt{i}}, \mathtt{g}^{\mathtt{i}}\rangle} \quad (\mathsf{P})$$

FIG. I.7 – Règles d'inférence pour le système de transitions $\mathcal{T}$

qui sont suffisantes afin de décrire l'exécution des processus. D'autre part, les langages déclaratifs classiques n'utilisent que les parties F et $\mathtt{g}$ nécessaires pour la description de la résolution de buts. Notre combinaison de ces deux modèles ajoute la possibilité d'évaluer des buts tout en décrivant l'exécution de processus concurrents.

La relation de transition de $\mathcal{T}_{\mathcal{C}}$ est définie par les deux règles d'inférence montrées en figure I.7. La règle (G) décrit la résolution de buts, par l'utilisation de la sémantique opérationnelle du langage déclaratif utilisé pour le store. Les modifications du store sont décrites par la règle (P) en utilisant le système de transitions du paragraphe précédent. Notons que la modification du store peut invalider certaines étapes dans la résolution de $\mathtt{g}$, ce qui motive le fait de relancer la résolution à partir de $\mathtt{g}^{\mathtt{i}}$, le but initial. Évidemment, la règle (P) invite à une étude de son amélioration afin de garder le plus possible de l'arbre de résolution construit avant la modification.

I.3.1.3 Sémantique opérationnelle d'un système

La sémantique opérationnelle d'un système composé de plusieurs composants peut être définie par un système de transitions global. Un exemple d'un tel système est le deuxième niveau de la sémantique opérationnelle de KLAIM [NFP98]. Par contre, nous préférons ne pas suivre cette approche car elle requière l'hypothèse simplificatrice qu'il est possible de connaître l'état de tous les composants d'un système distribué à un moment donné.

Nous définissons la sémantique opérationnelle d'un système $\mathcal{S}$ comme un ensemble de systèmes de transitions, chacun de ces systèmes modélisant un composant du système $\mathcal{S}$. L'interaction entre les composants ou systèmes de transitions est définie à l'aide de méta-règles qui expriment les relations entre les événements liés à la communication des différents composants. Un *événement* correspond donc soit à l'envoi soit à la réception d'un message, c'est-à-dire d'une séquence d'actions à exécuter. L'association de l'ensemble de tous les événements correspondant à une transition d'un composant permet de définir l'ensemble de tous les événements qui ont lieu pendant une exécution du système. Pour spécifier le système de communication, nous définissons une relation de correspondance entre les envois et réceptions respectifs. Les propriétés du système de communication correspondent ainsi aux propriétés de cette relation. Par exemple, si la relation de correspondance est bijective, le moyen de communication utilisé est fiable, dans le sens où tout message envoyé est reçu, et tout message reçu a été envoyé.

I.3.2 Sémantique compositionnelle

La sémantique opérationnelle telle qu'elle est présentée au paragraphe précédent n'est pas *compositionnelle*, c'est-à-dire qu'il y a deux processus qui ont la même séman-

tique, mais qui se comportent différemment s'ils sont exécutés dans un contexte particulier. Intuitivement, la sémantique n'est pas compositionnelle car elle ne prend pas en compte les actions que l'environnement du processus peut exécuter.

En nous inspirant d'une sémantique compositionnelle proposée pour la programmation concurrente par contraintes [dBP91], nous avons défini une sémantique compositionnelle pour les processus d'un composant en étiquetant les actions exécutées par les processus avec trois étiquettes. De plus nous permettons la simulation de l'environnement à l'aide d'actions hypothétiques [Ser98]. Ainsi, la nouvelle sémantique incorpore tous les comportements possibles de l'environnement et sa compositionnalité peut être prouvée (cf. chapitre 5, théorème 5.22).

I.4 Analyse de la confidentialité

Comme de plus en plus de données secrètes ou confidentielles circulent sur les réseaux informatiques, il devient de plus en plus important d'assurer la confidentialité de ces données. Nous définissons la confidentialité d'un point de vue de non-interférence, c'est-à-dire nous associons à chaque information un niveau de confidentialité. Dans ce cadre, nous disons qu'un programme respecte la confidentialité si une information ne dépend pas d'information de confidentialité plus élevée. À notre connaissance, cette approche a été introduite par [SVI96] dans le contexte des langages impératifs, et étendue pour les langages concurrents par la suite en [SV98, BC01a, BC01b]. Dans ce paragraphe, nous donnons une analyse de la confidentialité pour les processus d'un seul composant qui de plus est limité aux seules actions tell, del, :=, new et skip.

Appliqué à notre modèle, nous associons (à l'aide d'une association de niveaux de confidentialité ℓ) à tout symbole f d'une signature un niveau de confidentialité $\ell(f)$ parmi un treillis de niveaux de confidentialité $\mathfrak{L}$. Le niveau de confidentialité d'un terme est défini comme la plus petite borne supérieure des niveaux de confidentialité de tous les symboles apparaissant dans le terme. Intuitivement, un store respecte la confidentialité si l'évaluation d'un terme (ou la résolution d'un but) ne dépend pas d'une information d'un niveau de confidentialité (strictement) supérieur à celui du terme. Donc, une règle respecte la confidentialité si le niveau de confidentialité du membre gauche est supérieur aux niveaux du membre droit et des conditions. Un store respecte la confidentialité si toutes ses règles respectent la confidentialité.

Nous formalisons le respect de la confidentialité d'un processus en deux étapes. Intuitivement, un (terme de) processus respecte la confidentialité si l'effet de son exécution à un niveau de confidentialité π ne dépend pas des valeurs initiales d'un niveau plus élevé que π. Donc nous définissons $\approx$, une bisimulation particulière, et demandons qu'un processus respectant la confidentialité soit bisimilaire à lui-même. Deux paires d'un processus p_i et d'un store F_i ($i \in \{0; 1\}$) sont bisimilaires pour un niveau de confidentialité π et une association de niveaux de confidentialité ℓ si et seulement si d'une part, les deux stores F_i sont identiques au renommage des variables près pour toute règle dont le niveau de confidentialité du membre gauche est inférieur (ou égale) à π, et, d'autre part, si un des processus peut exécuter une action a, alors soit le deuxième processus peut exécuter la même action de telle sorte que les nouvelles paires de processus et stores sont de nouveau bisimilaires, soit le deuxième processus ne peut pas

exécuter la même et dans ce cas toute modification du store par le premier processus ne concerne que des règles d'un niveau de confidentialité strictement supérieur à π.

Contrairement aux analyses fondées sur le typage [SV98, BC01a, BC01b], nous définissons une exécution abstraite dont le but est de calculer un ensemble de contraintes portant sur les niveaux de confidentialité des symboles d'un composant. Le store initial correspond au système de contraintes exprimant le respect de la confidentialité par les règles du store initial. Ensuite, l'exécution abstraite d'une action ajoute les contraintes nécessaires au store. Les contraintes associées à une action expriment deux choses. Premièrement, les règles ajoutées par l'action doivent respecter la confidentialité. Ceci évite les mauvaises actions, c'est-à-dire les actions qui explicitement rendent publique les informations confidentielles. Deuxièmement, l'action ne doit pas modifier une règle d'un niveau inférieur à la plus petite borne supérieure de toutes les gardes testées jusque là pendant cette exécution. Intuitivement, toute action doit être d'un niveau supérieur à la garde, car autrement il serait possible d'obtenir une information sur une valeur confidentielle.

Lors d'un appel à un processus pendant l'exécution abstraite, seul le niveau de confidentialité est intéressant, ce qui permet d'abstraire la structure des paramètres. Comme le treillis des niveaux de confidentialité et l'ensemble de symboles d'une signature sont finis, l'ensemble des appels à un processus est fini. Ainsi nous pourrons explorer toutes les exécutions possibles, car il est suffisant d'exécuter un appel à un processus une seule fois, en coupant les appels récursifs déjà traités. Ainsi nous prouvons la terminaison de notre analyse.

Le résultat de l'analyse pour un store F et un terme de processus p est le système de contraintes calculé, noté Δ_F^p. Nous prouvons que le store F et le terme de processus utilisés pour calculer ce système de contraintes respectent la confidentialité par rapport à toute association de niveaux de confidentialité ℓ telle que ℓ respecte Δ_F^p, c'est-à-dire telle que toute les contraintes de Δ_F^p soient valides.

Notons que notre analyse accepte des programmes plus généraux que ceux de [SV98, BC01b], car nous permettons la création dynamique de processus (ce qui n'est pas possible en [SV98]) ainsi que des termes de processus de la forme $(p_1 \parallel p_2)\,;\, p_2$ (qui ne sont pas considérés en [BC01b]).

I.5 Description d'un prototype

Afin de montrer la faisabilité de notre modèle, nous avons implanté un prototype d'une plate-forme pour la programmation multiparadigme selon le modèle proposé. Ce prototype, implanté en `ocaml` [LDG+01], est un interpréteur pour un seul composant. Ainsi, l'exécution d'un système nécessite de lancer une instance du prototype par composant, de manière possible sur différentes machines connectées, par exemple par l'internet.

La description d'un composant est séparée en cinq parties ou fichiers, à savoir le store, les actions, les processus, l'interface (c'est-à-dire les symboles importés) et les traductions. Ainsi la signature d'un composant est repartie sur les fichiers de telle sorte que les définitions et déclarations des différents symboles soient regroupées.

Le schéma général du processus de l'interprétation d'un composant est donné sur la

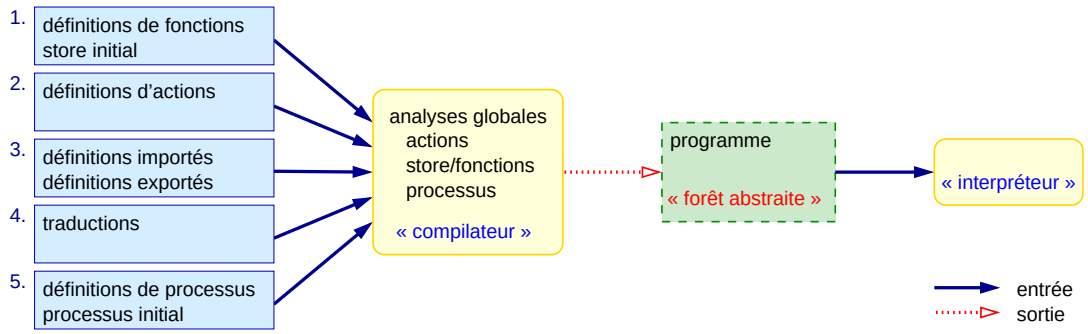

FIG. I.8 – Vision globale du processus d'interprétation d'un composant

figure I.8. L'interpréteur prend le nom du composant comme paramètre et commence à analyser (dans l'ordre indiqué par les numéros sur la figure I.8) les cinq fichiers correspondants aux cinq parties de la description d'un composant. En utilisant l'information contenue dans ces fichiers, le « compilateur » produit une « forêt abstraite », c'est-à-dire une représentation interne du composant. Finalement, la forêt abstraite est passée à un « interpréteur » qui prend en charge l'exécution proprement dite du composant.

En plus de l'exécution du processus initial autour du store initial, l'interprétation d'un composant nécessite l'exécution d'un interpréteur pour la résolution interactive de buts (comme dans les langages déclaratifs classiques) ainsi qu'un processus qui gère l'exécution des messages reçus par les autres composants du système. Notre prototype offre de plus un interpréteur optionnel qui permet de modifier le store par l'exécution interactive d'actions, similaire à la possibilité d'échanger le programme dynamiquement en Erlang [AVWW96].

I.6 Comparaison

La recherche sur les langages de programmation est un domaine vaste, et une comparaison détaillée avec tous les travaux voisins ne pourrait être présenté dans ce mémoire. Dans ce paragraphe, nous comparons donc brièvement notre modèle à quelques travaux voisins qui nous semblent être les plus proches. La structuration de cette section ainsi que l'organisation ont été choisies de manière arbitraire, dans le sens où d'autres possibilités auraient été tout aussi possibles. En effet, le caractère multiparadigme des travaux rend subjective toute classification linéaire.

I.6.1 Programmation déclarative

Contrairement à notre modèle, la plupart des intégrations de la concurrence aux langages déclaratifs ne distinguent pas clairement les notions sous-jacentes aux langages déclaratifs des processus, mais *encodent* les processus, par exemple en tant que prédicats ou fonctions.

La programmation logique est considérée comme intrinsèquement concurrente dans le sens où la recherche de preuve pour une conjonction d'atomes peut être effectuée en

parallèle. À notre avis, cette forme de concurrence *implicite* visant à améliorer le temps d'exécution d'un programme est à différencier de la concurrence *explicitement* spécifiée par le programmeur afin de mieux modéliser un problème. De plus, cette vision de la programmation logique ne distingue pas les prédicats des processus.

Notre modèle d'exécution est proche du modèle de la programmation concurrente par contraintes (ccp) [Sar93]. Dans ce modèle, un ensemble de processus communique par accumulation de contraintes dans un store. Contrairement à notre modèle, l'évolution du store est *monotone* en ccp. De plus, la plupart des sémantiques proposées pour ccp définissent la sémantique d'un processus par le store *final* de l'exécution du processus, et ne sont donc pas conçues pour la modélisation de processus qui ne terminent pas. Les extensions de ccp avec des stores non-monotones qui ont été proposées, définissent des actions prédéfinies spécifiques [dBKPR93, CR95] ou utilisent des logiques non-monotones [SL92, SJG96, GJS96, BdBP97, RF97, FRS98].

Les deux « prédicats » prédéfinis `assert` et `retract` de Prolog [DEDC96] permettent d'ajouter et d'enlever des formules au programme. Dans notre modèle, ces « prédicats » correspondent plutôt à des actions.

La programmation logique par contraintes réactive [FFS95, FFS98] permet la modification dynamique de la théorie utilisée pour la résolution de contraintes. La technique est fondée sur une réorganisation de l'arbre de recherche afin que cette réorganisation soit aussi minimale que possible. Il semble intéressant d'adapter ces techniques en vue d'améliorer la règle (P).

La plupart des extensions concurrentes de langages fonctionnels encodent les processus sous forme de fonctions. En conséquence, un processus doit nécessairement retourner une valeur, même si, à notre connaissance, cette valeur n'est pas considérée dans la plupart des langages [Rep91, AVWW96, PJGF96, TLK96a, Rep99, LDG$^+$01]. Une des motivations amenant à considérer les processus comme des fonctions semble être la possibilité d'utiliser un style fonctionnel dans la description de processus. Notons que ce style est également possible dans notre modèle grâce aux fonctions de processus.

La conception du langage Concurrent Haskell (CH) [PJGF96] a été guidée par la recherche d'une extension minimale de Haskell [PJHA$^+$99], la définition d'abstraction plus commodes étant possible en Haskell. La seule différence entre un processus en CH et une fonction classique est le type (ou la sorte) : un processus en CH est une valeur du type monadique `IO t` qui dénote une fonction transformant l'état (un argument *implicite*) et retourne une valeur de type `t`. L'extension concurrente de cette modélisation de l'interaction nécessite la définition d'une sémantique opérationnelle à deux niveaux, similaire à celle proposée au paragraphe I.3.1.

Le langage CML [Rep91, Rep99] diffère de notre modèle dans deux choix fondamentaux : La communication entre processus en CML est fondée sur l'échange de messages et la synchronisation des évènements de communication. De plus, une fonction peut être définie à l'aide de processus (et vice versa), ce qui contredit la séparation des différents concepts.

En Erlang [AVWW96], les processus sont définis à l'aide de fonctions non typées. Ces processus communiquent à l'aide de primitives spécifiques qui ont pour effet d'envoyer des messages. Comme dans notre modèle (et son prototype Sabir), l'environnement d'exécution d'Erlang permet l'interaction avec le système. Ainsi, un module peut être

remplacé par un autre sans avoir à arrêter son exécution.

Les seuls langages logico-fonctionnels pour lesquels nous connaissons une extension concurrente sont Curry [HAK+00b] et Escher [Llo95]. Les processus sont modélisés par les contraintes ou prédicats en Curry [Han99]. La notion de *port* de Curry est une extension de celle de [JMH93] qui permet le passage de nom de port, et ainsi la modélisation de systèmes distribués et mobiles. L'extension concurrente de Escher [Llo] utilise, tout comme notre modèle, un store commun comme moyen de communication entre processus. Néanmoins, Escher ne permet pas de grouper plusieurs actions d'une manière atomique.

I.6.2 Programmation concurrente

Les algèbres de processus, comme par exemple CSP [Hoa87], CCS [Mil89], ACP [Fok00, BW90] et le π-calcul [Mil99], permettent la description aisée de processus. Néanmoins, les langages fondés uniquement sur ces algèbres nécessitent le codage des notions de fonctions et prédicats en termes de processus, contrairement à notre modèle.

Le langage LOTOS [LOT00, ELO01] combine la spécification algébrique des types de données avec les algèbres de processus. Contrairement à notre modèle, les spécifications des fonctions ne peuvent pas être modifiées lors de l'exécution. De plus, la communication en LOTOS utilise la notion de synchronisation de « portes », ce qui peut être simulé dans notre modèle. Par contre, la diffusion, qui est immédiate dans notre modèle, est plus difficile à mettre en œuvre en LOTOS.

Le π-calcul [Mil99] permet la modélisation de processus mobiles dans le sens où la structure de communication peut évoluer grâce au passage de canaux de communication. Notre action élémentaire `new` en conjonction avec le constructeur de type `Name` permet de modéliser la mobilité d'une manière similaire. Contrairement à notre modèle, la réception de messages dans le π-calcul est limitée à un seul canal en même temps. Des extensions du π-calcul (asynchrone) sans cette limitation sont le join-calculus [FG96] et $\mathcal{L}_\pi$ [CM98]. Intuitivement, les processus dans le join-calculus et $\mathcal{L}_\pi$ communiquent à l'aide d'un multi-ensemble de messages, ce qui est une forme particulière de store.

I.6.3 Coordination

Afin de faire coopérer plusieurs processus (ou programmes) concurrents, différents modèles et langages de *coordination* ont été proposés, les plus proches de notre formalisme étant ceux fondés sur un espace de données commun. La plupart des langages de coordination sont conçus pour la coordination de langages impératifs, dans le sens qu'ils définissent un ensemble restreint d'actions sur un espace de données partagé. Notons que ce modèle implique que tous les processus doivent utiliser le même langage afin de communiquer.

Le premier langage de coordination en tant que tel, Linda [Gel85], permet à des processus de communiquer à l'aide d'un *espace de n-uplets* partagé. Linda offre au processus essentiellement trois opérations, à savoir `out`, `in` et `read`. `out`(t) met le n-uplet t dans l'espace de n-uplets. `in`(t) attend jusqu'à ce que l'espace de n-uplets contienne un n-uplet t' qui soit filtré par le n-uplet t, puis établie les liaisons des variables libres de t vers les valeurs filtrées de t' et enfin enlève t' de l'espace de n-uplets. `read`(t) a

le même comportement que $\mathsf{in}(t)$ sauf que le n-uplet t' n'est pas enlevé de l'espace de n-uplets. Comme la communication en Linda implique la génération d'un n-uplet, elle est dite « générative ».

Donc, tout modèle de coordination fondé sur Linda suppose que tous les composants du système partagent le langage utilisé pour la description des n-uplets. Les propositions de combiner les langages de coordination à la Linda avec la programmation déclarative enrichissent le modèle en considérant l'espace de n-uplets comme une théorie logique. Mais contrairement à notre modèle, ces théories restent limitées à des formules atomiques (des n-uplets).

I.6.4 Spécifications (exécutables)

La différence entre les langages de programmation et les langages de spécification est que les premiers visent à l'exécution du système, alors que les seconds à une description abstraite. Les spécifications exécutables sont donc similaires à notre modèle.

Les spécifications algébriques avec état implicite [DG94, Kho96] utilisent la notion de « modificateur élémentaire » pour décrire les modifications des fonctions. Si ces modificateurs sont similaires à nos actions élémentaires, ils ne peuvent pas être définis par le programmeur et sont spécifiques à une seule fonction. De plus, les spécifications algébriques avec état implicite sont plus orientées vers la description des changements d'état, contrairement à notre modèle qui prend plus en considération la description des processus qui provoquent ces changements.

Le formalisme des ASM[4] [Gur97] utilise des algèbres pour la description des états. Ces algèbres peuvent être considérées comme les modèles associés aux présentations de théorie que représentent nos stores. Les changements d'état sont exprimés à l'aide de mises à jour élémentaires qui sont similaires à nos actions élémentaires. Finalement, comme les spécifications algébriques avec état implicite, les ASM ne prévoient pas l'utilisation des opérateurs des algèbres de processus pour la description de processus.

Les machines à états algébriques [BW00] utilisent plusieurs niveaux pour la description d'un système et distinguent, comme notre modèle, les parties statiques des parties dynamiques d'un système. En effet, les états d'une machine à états algébriques sont des spécifications algébriques. Les transitions entre ces états sont spécifiées par des règles (ou axiomes) de transition particulières, c'est-à-dire des formules logiques liant les deux états. Ainsi, les machines à états algébriques sont plus proches des *spécifications* que notre modèle, car elles ne sont pas nécessairement exécutables. Une autre différence est que les machines à états algébriques sont composées selon une approche fondée sur le flot de données, contrairement à nos processus qui sont construits à l'aide d'opérateurs issus des algèbres de processus.

I.6.5 Programmation multiparadigme

Les motivations de notre travail sont les mêmes que ceux des langages multiparadigmes, à savoir de permettre l'utilisation de différents styles de programmation dans un formalisme unifié. Néanmoins, la plupart des langages multiparadigmes ne possèdent pas de base formelle couvrant la totalité du langage [Bud95, CLSM96, Con88, Cop98,

[4] ASM est l'acronyme de l'anglais « Abstract State Machine » (machine à états abstraits).

NL95, Pla91, Spi94, Zav89, Zav91, ZJ96], ou nous ne connaissons pas d'extension concurrente [ABPS98, LP96, LP97, AKP93].

Le langage Maude [CDE+99] est fondé sur la logique de la réécriture, ce qui permet l'intégration de différents styles de programmation en les considérant comme des instances particulières de la logique de la réécriture. De plus, la réflexivité de la logique de la réécriture permet la représentation de programmes écrits en Maude sous forme de termes de Maude. Ainsi, si les fonctions et processus sont représentés de la même manière en Maude, ces concepts sont distincts car ils se trouvent à des niveaux de méta-représentation différents. Par contre, nous ne connaissons pas de modèle théorique pour l'interaction d'un programme Maude avec son environnement.

I.7 Conclusion

Dans ce mémoire nous avons présenté un modèle de programmation qui combine la programmation déclarative en général, et la programmation logico-fonctionnelle en particulier, avec la concurrence sous forme de processus mobiles. Une particularité de notre modèle est que nous faisons clairement la distinction de différentes notions, comme par exemple fonction, processus et action. Ainsi nous évitons le besoin d'encoder un concept par un autre et nous permettons l'expression de chaque notion par le formalisme le plus approprié. D'autres aspects de notre modèle sont la possibilité de définir des actions et de spécifier explicitement des traductions. Ces deux traits permettent au programmeur de séparer ces différentes parties d'un programme. Nous avons présenté une sémantique compositionnelle pour les processus d'un composant ainsi qu'une analyse de la confidentialité d'un point de vue de la non-interférence. Cette analyse est fondée sur une exécution abstraite et assure qu'une information secrète ne peut pas influencer une information accessible publiquement.

Néanmoins, le modèle proposé n'est qu'une première étape dans un projet de recherche à plus long terme. En conséquence, de nombreuses possibilités d'améliorations et de questions restent ouvertes. D'une part, le modèle de calcul peut être étudié davantage, comme par exemple par la définition d'une algèbre de processus (dans la suite des travaux de [Ser98]) et d'une sémantique dénotationnelle (fondée sur une structure de Kripke ou la logique linéaire). D'autre part, plusieurs extensions du modèle semblent intéressantes. Premièrement, l'intégration de la notion du temps est nécessaire pour la plupart des systèmes de contrôle. Ainsi il serait intéressant d'étudier l'intégration de la programmation déclarative avec des notions temporelles comme définie en [BE01] dans notre modèle. Deuxièmement, notre modèle considère uniquement la mobilité dans le sens du π-calcul, c'est-à-dire par passage de liens. L'étude de l'extension du modèle avec des processus migrants semble donc intéressante. Troisièmement, la relation de notre modèle avec la notion de réflexivité mérite d'être étudiée davantage, afin de mieux comprendre les relations entre les différents niveaux de la description d'un composant.

Finalement, le prototype actuel offre de nombreuses possibilités d'amélioration. Par exemple, le langage déclaratif utilisé peut être amélioré par l'inclusion de moteurs dédiés à la résolution de contraintes, par des fonctions d'ordre supérieur, par une analyse de types polymorphes, *etc.* Concernant l'implantation des processus, un mécanisme de verrouillage du store plus fin que le store entier devrait améliorer les performances,

d'une part parce que plus de processus pourraient être exécutés en parallèle, et d'autre part parce que la sélection des processus à réveiller après une modification du store pourrait être plus fine. Mise à part les améliorations du prototype proprement dit, le développement d'outils associés comme un environnement de programmation ou d'outils d'analyse est nécessaire.

Chapter 1

Introduction

Programs have become more and more ubiquitous with the utilisation of information processing units in nearly all appliances of every day life, such as, for instance, electronic payment systems, access control systems, (mobile) telephones, cars or even washing machines. On the other hand, the state of the art in programming cannot safeguard us from problems related to incorrect and unreliable programs or software, even when the consequences are very costly or dangerous, as for example the failure of the first launch of a rocket of the Ariane-5 family, the delays in putting the new airport of Denver into operation caused by malfunctioning software controlling the automated baggage handling system, *etc.* Thus, we are still at the beginning of a *science* of programming [Dij01], a state of affairs that is witnessed by the fact that there is currently no consensual definition of a program, if there is a definition at all. One possibility to prevent the problems related to incorrect and unreliable programs is the improvement of the methods used for the construction and maintenance of programs. These methods should help ensure properties of the programs such as correctness, and should be accompanied by tools that allow to put the methods into practice.

One particular tool for the construction and maintenance of programs is the programming *language* used for the expression of the program. Linguists and computer scientists[1] have long recognized that the language in which thoughts are expressed colors in a fundamental way the nature of the idea. In its most extreme form, this statement is called the Sapir-Whorf hypothesis[2] and asserts that it may be possible for an individual thinking in one language to imagine thoughts or utter ideas which cannot in any way be translated, or even be understood by individuals using a different linguistic framework. Although in the domain of computer science Church's conjecture [Chu36] suggests the exact opposite, namely that the choice of the programming language is not important, since all computable functions can be expressed in any sufficiently powerful formalism, as for instance Turing Machines [Tur36] or the λ-calculus [Chu32, Bar84], most programmers will agree that different languages are more or less convenient for the expression of different aspects of systems. Depending on the problem, a solution might be expressed in straightforward manner in a certain language whereas other languages may require a significantly more difficult description of the solution, mostly due

[1] Consider the following citation: "The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore on our thinking abilities" [Dij82].

[2] For more details on the Sapir-Whorf hypothesis, see for instance [Cha95].

to the need of *encoding* the problem in terms of less adapted concepts. Hence, without claiming an equivalent of the Sapir-Whorf hypothesis for programming languages, we believe that the choice of the language influences the structure (and properties) of the resulting program. This has already been described in the literature, see for instance [HJ94, Zei95, Arm96]. Clearly, a *multiparadigm language*, *i.e.*, a language combining different programming styles or paradigms[3], offers the programmer the possibility to choose for each part of a system description the most appropriate programming style.

Since the beginning of programming, research on programming languages has mostly been directed in the sense of providing higher-level languages. For instance, the machine code and assembly languages have been replaced by compiled languages, once the necessary tools, *i.e.*, compilers, were available. A high-level of abstraction is a desirable property for a programming language, since it allows a programmer to focus on the essential parts of a system without the need of considering all of the technical details. Hence a program written in a really high-level language will be concise and clear, since it is organised similarly to the structure of the system which it models. Therefore, programs of a higher abstraction level are more easily written, understood and maintained than programs of lower levels of abstraction. Besides a high-level of abstraction, a programming language should also provide a well-defined semantics, since the precise definition of the meaning of a program is essential in understanding and reasoning about the program and its properties. Consequently, a precise semantics is mandatory for the construction and maintenance of safety-critical programs.

A particular kind of programming languages providing a high-level of abstraction are languages that allow a programmer to rather declare *what* the program is about instead of explicitly writing down *how* to achieve the goal. These languages are called *declarative programming languages*. Besides providing naturally a high-level of abstraction (since all the "how" is omitted), declarative programs have the advantage that they are rather close to a *specification* or description of the problem. They are nevertheless considered as programs, since they are executable and can be used, at least, as a first prototype.

Examples of declarative programming languages are functional, logic or functional-logic programming languages. These languages are based on the notions of *functions* and *predicates* (or relations), *i.e.*, well mastered mathematical concepts which have been successfully used for description of algorithms even before the invention of computers. Classically, functions are described in such a declarative language by means of equations (rewrite rules) or λ-abstractions, whereas Horn clauses (with constraints) are a classical means for the description of predicates. For the rest of this thesis, "declarative programming" is used as a synonym for functional, logic or functional-logic programming.

However, the theoretical concepts of functions and predicates that constitute the basis of classical declarative programming languages, are insufficient to capture the whole complexity of real-world applications where interactivity, concurrency and distributivity are needed [Mil93a, Weg98][4]. An early example where the use of concurrency

[3]For more details on the notion of programming paradigms, refer to section 2.5 where we present some related work on multiparadigm languages.

[4]Notice in this context, that A. Turing admits the existence of computing machines more powerful than "Turing Machines" [Tur36], when he states in [Tur39]: "We shall not go any further into the

yields a better structure of a program are the *coroutines* used for the implementation of a one-pass compiler for COBOL [Con63], where the different phases of the compilation are described as parallel executing processes communicating via directed streams. Another, more recent kind of application that requires interactivity and concurrency are window systems. The usefulness of concurrency in the concise design of window systems has been pointed out by a number of researchers, as for instance in [Pik89] and some concurrent extensions of declarative languages were even motivated by the design of an adequate window system, written in these languages, as for example eXene [GR93a] for CML [Rep99] and Haggis [FPJ95] for Concurrent Haskell [PJGF96].

The notion of *processes* has been introduced as abstraction for the expression of concurrency. Processes have been studied in form of *process algebras* and *process calculi*, *e.g.*, [Hoa85, Mil89, BW90, Mil99, Fok00]. Informally, a process is characterised by the actions that it can execute, or the interactions it has with its environment, *i.e.*, other processes that are executing concurrently. These formalisms, *i.e.*, process algebras and calculi, allow the description of a system as a set of processes that can be executed on a single computer or distributed over a network. However, similar to classical declarative programming languages, languages purely based on process calculi need to be extended in order to provide the notions of functions and predicates without encoding them in terms of processes.

Last, but not least, the construction of complex systems through the assembly of existing *components* has been suggested, inspired by similar approaches that are current practice in other engineering disciplines, as for instance the assembly of a bicycle from a frame, two wheels, a saddle, *etc.* The construction of programs or software systems by the composition of components naturally favors the reuse of existing software and allows to manage the complexity of the overall system by enabling the construction, verification and maintenance of the components separately. However, these approaches still lack a consensual definition of a component, which would ensure that the semantics of a system assembled from several components is clearly defined.

A well-defined combination of the different programming styles mentioned above, namely declarative, concurrent and components, would allow to express most parts of a system using the most appropriate concepts. Hence there would be no need to *encode* one concept by another, and the different aspects of the problem could be expressed directly and, in consequence, more clearly. Furthermore, a clear separation and structuring of these different concepts, abstractions or notions, namely functions, predicates and processes, should lead to well structured, readable and understandable programs which are consequently easy to maintain. Clearly, this enforced separation comes at the price of a loss of flexibility, since its enforces a certain discipline of programming which a programmer has to follow. However, the most flexible programming languages are machine-code or assembly languages, since they do not enforce any discipline whatsoever, and it has been recognised for long that the benefits of adherence to a more structured style of programming are worth paying the price of less flexibility.

Numerous programming languages and models have been suggested in order to accommodate the integration of the different notions mentioned before, namely functions, predicates, processes and components. We refer the reader to the subsequent chapter

nature of this oracle apart from saying that it cannot be a machine."

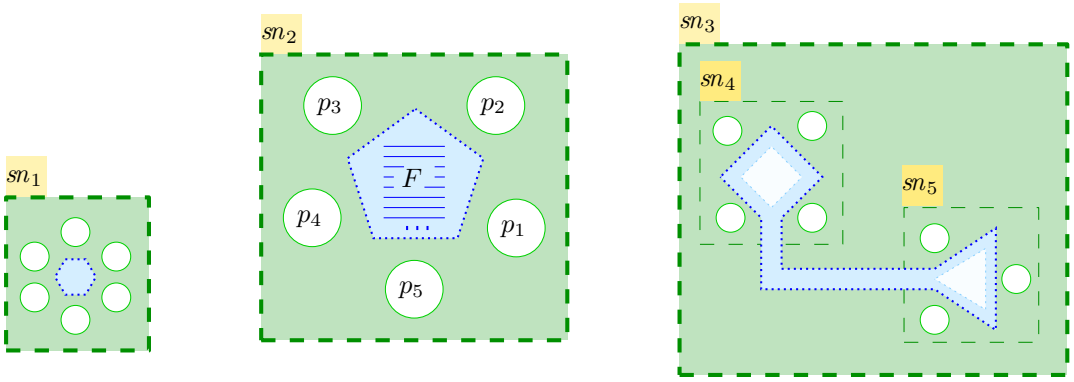

Figure 1.1: Execution Model of a System

for a presentation of those we are aware of and that are related to our proposal.

In this thesis we explore a new computation model, or a new general framework of programming languages, providing a component based approach for constructing systems. These programming languages are a combination of declarative, *i.e.*, either functional, logic or functional-logic, programming with concurrency, expressed in form of mobile processes. The resulting computation model is a conservative extension of, on the one hand, declarative programming languages and, on the other hand, process calculi or process algebra.

In the rest of this chapter, we present the broad outlines of our computation model which is formally presented in chapter 3.

1.1 Overview of the Computation Model

We suggest to model a system as a set of interacting *components*, which may be distributed over a network or reside on a single computer [ES00, ES01a]. These components capture the different, clearly distinguishable entities of a system, that interact with each other by exchanging messages. Each component is identified by a component-name (also called storename) sn which can be seen as the address of the component. Internally, a component is organised as a set of *processes* p_i, *i.e.*, a concurrent program, and a *store* F, *i.e.*, a traditional declarative program, which can be seen as a pair of a *signature* and a set of formulæ describing a (static) theory [BES98, ES99]. The processes p_i communicate by modifying the stores, *i.e.*, by altering, in a possibly non-monotonic way, the current theories described by the stores, for example by simply redefining constants (*e.g.*, adding a message in a queue) or more generally by adding or deleting formulæ in F. We call these modifications of the stores *actions* and allow their defintion by the programmer [ES01b]. Interaction between components is based on asynchronous message passing, where the messages correspond to actions to be executed on the remote store. Figure 1.1 shows the execution of a system of three components with storenames sn_1, sn_2 and sn_3.

In this thesis, and in particular in this section, we focus on the definition of a component, and only scratch the problems related to the composition of components.

For instance, the component sn_3 of figure 1.1 is the composition of the two components sn_4 and sn_5. Nevertheless, the internal structure of sn_3 can be hidden by taking the disjoint union of both, the stores as well as the sets of processes of the components sn_4 and sn_5, and by adding the name of the original component as a suffix to all symbols exported by the component sn_3.

1.1.1 Stores

Our approach to combine declarative programming and concurrency is generic in the sense that it is independent from the actual declarative language used for the description of the store. In fact, we only require that programs in a declarative language can be seen as a pair of a signature Σ and a set of formulæ or rules $\mathcal{R}$, *i.e.*, $F = \langle \Sigma, \mathcal{R} \rangle$, and that the operational semantics (of the declarative language) allows the test for *validity* of a term (of the special sort `Truth`). Most (declarative) programming languages fit into this framework, which is similar to the notion of a logical system in the framework of institutions [GB92].

1.1.2 Actions

Actions are the principal constituents of processes, as it is witnessed by the fact that the semantics of a process is, loosely speaking, defined by the sequences (in linear time semantics) or graph (in branching time semantics) of actions that the process might execute. Most classical process algebras consider *abstract* actions, in the sense that actions are simply elements of a vocabulary (of actions) without further specification, besides a relation which associates pairs of actions in order to model synchronisation and communication. In our model, the execution of an action has the observable effect of modifying the store of the component. Thus we need to specify the notion of actions further to take into account these modifications.

Roughly speaking, an action $\langle sn, \mathsf{a} \rangle$ is a pair of an elementary action a and a store-name sn denoting the store on which the action is to be executed. Examples of actions are $\langle sn, c := v \rangle$, to change the definition of the (variable) constant c (which is defined in the store of the component sn) to the value v and $\langle \mathit{Display}, \mathsf{print}(\texttt{'hello'}) \rangle$ to print the string `'hello'` on the display *Display* (which is considered as a component). Notice that this approach to actions integrates smoothly actions defined on external, physical machines, by considering simply the available commands as actions executable on these machines. For a given action, different possible semantics exists. For instance the deletion of a formula from the store might remove exactly the formula or all similar formulæ. Thus we suggest to integrate the *definition of actions* into our computation model. Consequently, a programmer can extend the set of actions provided by the language by defining application specific actions, as well as change or precise the semantics of the predefined actions.

In this section, we introduce the notion of *elementary actions* by means of some examples. The formal definition can be found in section 3.2, where we also present how to define application specific actions.

Since an elementary action transforms a store into another one, we are led to define elementary actions as total recursive functions from (well-formed) stores to (well-formed) stores. We require an elementary action to be a *recursive* function in order

to ensure the termination of its execution. The requirement of *totality* gives us the guarantee that the elementary action can be applied to any store. We require further, that all stores in the range of an elementary action (which is applied to a well-formed store) should be well-defined stores, *e.g.*, they should be well-typed. These conditions together ensure that the execution of an elementary action will never produce an error during the runtime of the system. That is to say, all actions can be executed and terminate, and also that the store of a component corresponds to a well-formed program at any moment during the execution of the program.

A similar view of (elementary) actions can be found for instance in Concurrent Haskell (CH), where the monadic I/O operations, *i.e.*, functions of type `IO t`, are considered as *state transformers* and are called *actions* [PJGF96, section 2.1]. A noteworthy difference[5] is that actions in CH are functions, and as such they return a value, in addition to the implicit "world" parameter. On the other hand, our (elementary) actions are the basic elements of processes and describe only the effect on the store without returning any value.

In order to clarify the structure of a component, we suggest to stratify the description of a component in several levels. For instance, actions describe the modification of the store, *i.e.*, a (declarative) program. Hence it seems natural to define action using a *meta-language* with respect to the language, say $\mathcal{L}$, used for the description of the store, since the action manipulates programs as data. The use of a meta-language, where an ADT of programs can be defined straightforwardly, avoids the need of integrating the program modifications inside the language $\mathcal{L}$. Therefore, while a component in our framework can be seen as a self-modifying program, all modifications are restricted in the sense that only modifications of program parts at a strictly lower level are allowed. In fact, since on a higher level, the program parts on lower levels are considered as data, this restriction is similar to the discipline in classical imperative languages, which consists in avoiding the modification of the part of the memory where the program *code* is stored.

Figure 1.2 shows a first representation of the basic levels needed for the description of a system in our computation model. The stores are described at the lowest level. As mentioned before, the meta-level above the store is used for the definition of the actions. The definition of processes involves both levels. Indeed processes use guards, which are expressed at the level of the store, and execute actions on the meta-level[6]. On the right of figure 1.2 we also show a term, namely *succ*(*zero*), of the store together with its meta-representation, which can be obtained thanks to the *reification* mapping *reify*[7]. Seen as a meta-term, *succ*(*zero*) corresponds to an *application* of the function *succ* to the constant (*i.e.*, parameterless term) *zero*. In a reflective programming language, as for instance Maude [CDE+99], the meta-level can be integrated in the language itself, *e.g.*,

[5] Besides the fact that the (sort) *state* used in CH is not further specified, whereas in our model a state is partly defined as a store, *i.e.*, a declarative program (for more details, see the definition of the operational semantics in chapter 4).

[6] Even if figure 1.2 might suggest it, we do not consider the modification of actions by processes. However, we picture the level of processes *above* the meta-level (for actions) since the definition of processes *uses* actions.

[7] "To regard or treat (an abstraction) as if it had concrete or material existence." [The American Heritage Dictionary of the English Language, Fourth Edition, 2000, `http://www.bartleby.com/61/6/P0130600.html`]

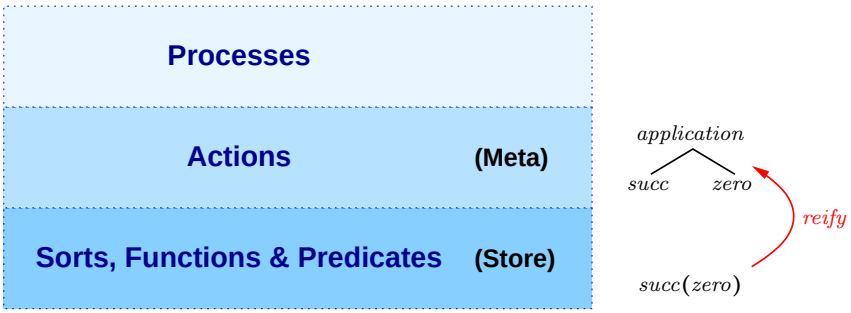

Figure 1.2: Basic Levels of a System-Description

in Maude by means of the system module `META-LEVEL`. Nevertheless, the distinction between the levels is still present and enables a clear structure of the program. We present these issues in more detail in section 3.2.

In analogy to well-formed calls to (or applications of) functions (see definition 3.7), we say that $a(t_1, \ldots, t_m)$ is a well formed call to the elementary action a if for all $i \in \{1; \ldots; m\}$ the term t_i is a well-formed term corresponding to the sort required by the profile of the elementary action a. Notice that, since an action is defined on the meta-level with respect to the store, the layering shown in figure 1.2 suggests that its arguments should also be on the meta-level. We suppose further that we dispose of a mapping *reify* which maps a term t of the store to its meta-representation $\mathit{reify}(t)$, as shown on the right of figure 1.2. However, for convenience of notation, we often omit the explicit calls to *reify* in the sequel, and write meta representations of terms in the same way as usual terms. In section 3.2, we give a more detailed definition of well formed action (parameters).

For the rest of this chapter, we suppose that we are given a set A of predefined (elementary) actions, namely tell, del, $:=$ and new. Intuitively, $\mathsf{tell}(f)$ (respectively, $\mathsf{del}(f)$) add (respectively, remove) a rule (or more generally, a formula) f to (respectively, from) the store (or program). Thus the profile of tell (respectively, *del*) is $\mathsf{tell} : \mathit{rule} \to \mathit{store} \to \mathit{store}$. Furthermore, we abbreviate for the rest of this chapter $\mathsf{tell}(t \to \text{TRUE})$ (where t is a (meta-representation of a term of sort `Truth`) to $\mathsf{tell}(t)$.

Certainly the most common elementary action is assignment $:=(c, v)$, also written more familiarly in infix-notation as $c := v$, changes the definition of the constant[8] c to the value v, *i.e.*, a (meta-representation of a) term. A reasonable requirement on the assignment action is to normalise the new value (with respect to the current store) of the constant before adding the new definition. Figure 1.3 gives an example of the execution of the action $c := 42$. The effect of this action is to transform the store containing the rule (or equation) $c == 23$ into a store containing the rule $c == 42$.

Besides actions modifying the rules or formulæ of a store, we need also actions for the modification of the *signature*. The creation of new operator (or function) symbols is handled by the elementary action new. The intuitive meaning of an elementary action $\mathsf{new}(f, s)$ is to enrich the signature of the store with the operator symbol f of sort s. Besides the new-action creating new function symbols, we may need similar actions

[8] In imperative languages, c is traditionally considered as a *variable*.

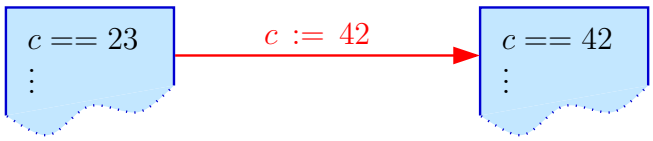

Figure 1.3: Execution of an Assignment Action

for the other kinds of symbols in a signature, *i.e.*, sorts, constructors, *etc.* Obviously, the exact set of needed new-actions depends on the declarative language used for the description of the store.

1.1.3 Processes

The processes of a component are specified in the style of a process algebra, see for instance [BW90, Fok00]. In this section we give a brief overview on the definition of processes in our computation model. To simplify the presentation, we restrict ourselves to systems which contain a single component. The complete definition of the computation model can be found in chapter 3.

1.1.3.1 Guarded Actions

As we have already mentioned before, actions are essential for the definition of processes. In fact, the basic processes in our computation model are *guarded actions*. A guarded action is a pair, consisting of a guard and a sequence of (calls to elementary) actions (with parameters):

$$[g \Rightarrow \mathsf{a}_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; \mathsf{a}_n(t_{n,1}, \ldots, t_{n,m_n})]$$

such that the guard g is a (conjunction of) expression(s) of sort `Truth` and, for all $i \in \{1; \ldots; n\}$, $\mathsf{a}_i(t_{i,1}, \ldots, t_{i,m_i})$ is a well formed call to the elementary action a_i.

To shorten the notation, we sometimes omit the parameters of the (elementary) actions and abbreviate a guarded action to $[g \Rightarrow \mathsf{a}_i]$.

Similar to the "guarded commands" of [Dij75], the execution of the sequence of (elementary) actions a_i in a guarded action $[g \Rightarrow \mathsf{a}_i]$ is only possible if the current theory described by the store allows to prove that the guard g is valid.

1.1 Example. *The following guarded action increments the value of the constant (the name of which is) c under the condition that the constraint, i.e., a (0-ary) function of sort* `Truth`*, increment is* TRUE*:*

$$[increment \Rightarrow c := (c\uparrow + 1)]$$

Recall that we require the assignment action to normalise the new value before adding the definition. By $c\uparrow$ we denote the current value of the constant c.

As the execution of actions modifies the store, the order in which the actions are executed may influence the resulting store. To obtain a *deterministic* execution of a

guarded action, we follow the tradition in imperative programming and require the sequential execution (from left to right) of the elementary actions of a guarded action.

To simplify the notation, we call in the sequel both, guarded and elementary actions, just "actions", whenever there is no risk of confusion.

1.1.3.2 Process Terms

Recall that processes in our computation model are defined in process algebraic style. The elementary (or basic) process terms are the execution of (guarded) actions and process calls. The predefined process success represents the process which executes no action and terminates successfully. As usual in process algebra (see, *e.g.*, [BW90, Fok00]), we provide some operators for combining processes: parallel ($\|$) and sequential ($;$) composition, nondeterministic choice ($+$) and choice with priority ($\oplus$). Hence we define a *process term* p as a "well-formed" expression according the following grammar:

$$
\begin{array}{rcll}
p & ::= & \mathsf{success} & \\
 & | & [g \Rightarrow \mathsf{a}_i] & (\textit{guarded action}) \\
 & | & \mathrm{q}(t_1, \ldots, t_m) & (\textit{process call}) \\
 & | & p\,;p & (\textit{sequential composition}) \\
 & | & p \parallel p & (\textit{parallel composition}) \\
 & | & p + p & (\textit{nondeterministic choice}) \\
 & | & p \oplus p & (\textit{choice with priority})
\end{array}
$$

In the case of a process call $\mathrm{q}(t_1, \ldots, t_m)$, we require that the parameters $t_1, \ldots, t_m$ of the process q are of sorts corresponding to the profile of q (and that the arity of q is m).

The operator of choice with priority $\oplus$ is not very common, but we found it necessary to model critical applications where nondeterminism is not acceptable [Abr96b]. The intended meaning of the process term $\mathrm{q}_1 \oplus \mathrm{q}_2$ is: "execute the process q_2 only if the process q_1 cannot be executed", *i.e.*, the process q_1 has a higher priority than the process q_2.

In order to get clearer and more readable programs by enforcing a "good programming style", we introduce the notion of *restricted process terms*. Roughly speaking, a restricted process term is a process term syntactically restricted neither to contain any occurrence of the priority operator, nor a guarded action. Thus, a restricted process term $\ulcorner p \urcorner$ is a well formed expression according to the following grammar:

$$\ulcorner p \urcorner ::= \mathsf{success} \mid \mathrm{q}(t_1, \ldots, t_m) \mid \ulcorner p \urcorner ; \ulcorner p \urcorner \mid \ulcorner p \urcorner \parallel \ulcorner p \urcorner \mid \ulcorner p \urcorner + \ulcorner p \urcorner$$

As for process terms, we require process calls to be well-sorted.

The principal motivation for the introduction of the notion of restricted process terms is that we allow the use of the operator of choice with priority only between the different clauses inside process definitions . Since the use of this operator is restricted, it can be implemented efficiently (see section 4.1.3).

1.2 Example. *Consider a process* counter *and the guarded action of example 1.1. Then the following is an example of a (restricted) process term, representing a process*

that increments the constant c (if increment is TRUE*) and then behaves as the process* counter*:*

$$[\mathit{increment} \Rightarrow c := (c\uparrow + 1)]\,;\ \text{counter}$$

1.1.3.3 Process Definitions

The behaviour of processes is defined by means of *process definitions*. However, the recursive definition of the behaviour of a process requires some care in order to avoid pathological cases, as for example processes with an infinite branching degree.

1.3 Example. *Consider the following recursive definition of a process* q*:*

$$\mathrm{q} = (\mathrm{q}\,;a) + a \tag{1.1}$$

In [BW90, exemple 2.4.13, page 33] *it is shown that equation* (1.1) *has no unique solution, if we suppose that a solution to the equation does exist. In, fact,* $\mathrm{q}_0 \stackrel{\mathsf{def}}{=} \sum_{i>0} a^i$ *and* $\mathrm{q}_1 \stackrel{\mathsf{def}}{=} \mathrm{q}_0 + a^\infty$ *are two different solutions of equation* (1.1)*. The problem with this definition of the process* q *is, that we do not know, if a call to* q *always terminates (as does the process* q_0*) or if there might be an execution where it does not terminate (as does* q_1*, when choosing the branch* a^∞*). Further examples of recursive definitions that do not uniquely determine processes are the following equations (see* [Fok00, example and exercise 4.1.1, page 32]*):* $\mathrm{q} = \mathrm{q} \parallel a$ *and* $\mathrm{q} = \mathrm{q}$.

To avoid this kind of problems, process definitions are usually required to be *guarded*[9], that is to say, a recursive call to a process has to be preceded by the execution of an action. Notice, that this notion of guard is to be distinguished from the guards in guarded actions, since the guard in a guarded process is an action, whereas the guard in a guarded action is a term (of sort `Truth`) of the store.

A process is defined by a set of rules, clauses[10] or *guarded commands*, ordered by priority. Each guarded command consists of a guarded action and a restricted process term. Using the previously introduced notations, a process can thus be defined by a phrase of the following form:

$$\mathrm{q}(x_1, \ldots, x_m) \ \Leftarrow\ \bigoplus_{i=1}^{n} \left([g^i \Rightarrow \mathsf{a}^i_{j_i}]\,;\ r p_i \right) \tag{1.2}$$

Notice that we have omitted some technical conditions on the use of free variables for the sake of a readable presentation.

Intuitively, the operational behaviour of a process call $\mathrm{q}(t_1, \ldots, t_m)$ is similar to the alternative construct of the guarded command language of [Dij75]. That is to say, we have to evaluate which of the guards of the commands defining q are valid, and then to choose among them the command with the highest priority. Choosing a guarded command means to atomically execute the sequence of elementary actions associated with the guard and afterwards to behave like the associated restricted process term.

[9] In [Mil89, page 101] this is called *weakly guarded*; a *guarded* process may not be below a $\parallel$ operator.

[10] In analogy with the definition of predicates by a set of clauses in logic programming; for the same reasons we use the symbol $\Leftarrow$ for the definition of a process.

1.4 Example (A simple counter). *Extending the preceding examples of this section, we consider a process* counter *modeling a simple counter that can be controlled by means of two constraints, namely increment and reset. Whenever the former is set to* TRUE, counter *increments the value of the constant c and when the latter holds,* counter *resets the value of c to zero. In both cases, the definition of the constraint is removed from the store, and the process is called recursively. Hence, the store on which* counter *acts, has to define a theory about natural numbers and the two constraints increment and reset.*

$$\begin{array}{rcl} \text{counter} & \Leftarrow & \big[\mathit{reset} \quad \Rightarrow c := 0\,; \quad \mathsf{del}(\mathit{reset})\big]; \quad \text{counter} \\ & \oplus & \big[\mathit{increment} \;\Rightarrow c := (c{\uparrow} + 1)\,; \; \mathsf{del}(\mathit{increment})\big]; \; \text{counter} \end{array} \tag{1.3}$$

In a complete system, the constraints increment and reset could be set to TRUE *by another process, controlling for instance a graphical user interface. Notice that the definition of* counter *gives reset a higher priority than increment. Thus, in case both are* TRUE*, the counter c will be first reset to zero, and incremented afterwards (if the constraint increment is then still* TRUE*).*

1.1.4 Operational Semantics

The operational semantics of a component integrates two orthogonal aspects, namely the use of the store as a classical declarative program (*i.e.*, evaluation of expressions and goal solving) and the execution of processes. Informally, the operational semantics of a component is defined by means of a transition system the states of which consist of a store and a process term. Execution of actions corresponds to state transitions, and in every state the (current) store can be used as a classical declarative program.

Hence, our computation model is a conservative extension of declarative programming, since a component without any processes corresponds to a declarative program in the classical sense. On the other hand, when abstracting from the effects of the actions, *i.e.*, the modifications of the stores, and considering only the names of actions that are executed, our computation model is a classical process algebra.

1.1.5 Example of the Dining Philosophers

We conclude this brief overview of our computation model by the complete presentation of a small example.

Consider the "Dining Philosophers" problem inspired from the example presented in [Dij71]. The problem concerns the life of some (Chinese) philosophers, which alternate between thinking and eating. These philosophers live in a same room, and are seated around a round table, in the middle of which stands a large bowl of rice (with the enjoyable property of containing always a sufficient amount of well-prepared rice). Figure 1.4 shows such a table for six philosophers. As usual, a philosopher needs two chop sticks to eat. Unfortunately, there are only as many chop sticks as philosophers, so that the philosophers have to share the sticks with their neighbours. Furthermore, the philosophers are not allowed to exchange sticks across the table.

We model the situation with two predicates on natural numbers, *i.e.*, functions with the profile $\mathit{Nat} \rightarrow \texttt{Truth}$, which have to be added to a signature of natural numbers (as for instance the signature of example 3.6), namely stick(x) and is_eating(y). The former

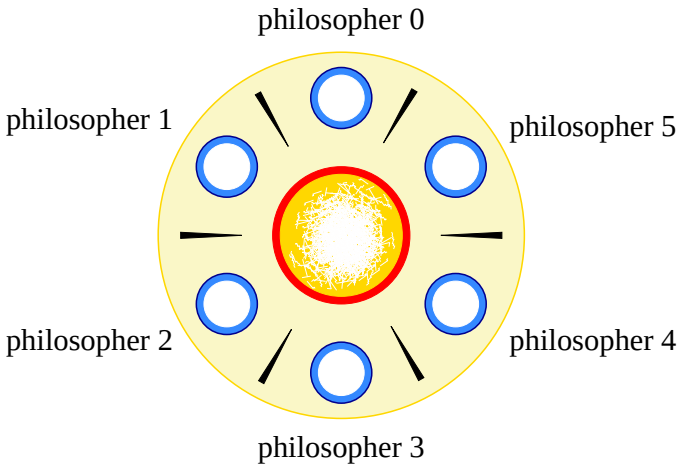

Figure 1.4: Dining Table for Six Philosophers

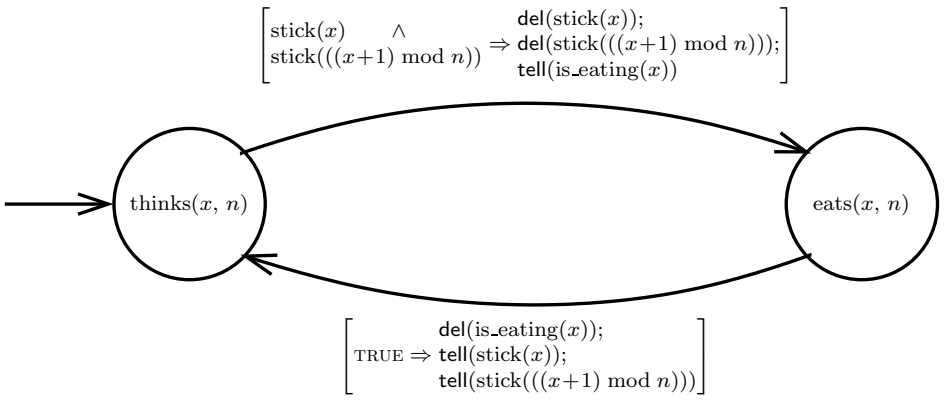

Figure 1.5: Automaton Describing the Behaviour of a Philosopher

represents the fact that stick x is lying on the table, and the latter is true whenever philosopher y is eating. Hence the initial store is an extension of a theory for natural numbers (as for instance the program presented in example 3.10) with n rules stating the presence of the n sticks on the table, *e.g.*, stick$(i) \rightarrow$ TRUE, for $i \in \{0; \ldots; n-1\}$.

The behaviour of a philosopher can be modeled by a simple automaton as shown in figure 1.5. Since a philosopher either thinks or eats, the automaton has two states. The transitions between the states are labeled with guarded actions executed along with the transition. These guarded actions use the elementary actions tell(r) (respectively, del(r)) which add (respectively, remove) an (unconditional) rule "$r \rightarrow$ TRUE" to (respectively, from) the current store, *i.e.*, the current declarative program.

The automaton shown in figure 1.5 translates immediately into the definitions of the two processes shown in table 1.1. We use two processes to model the two states for a philosopher: either the philosophers thinks or eats. These processes take two arguments

$$\text{thinks}(x,\, n) \Leftarrow \left[\begin{array}{l} \text{stick}(x) \quad \wedge \\ \text{stick}(((x{+}1) \bmod n)) \end{array} \Rightarrow \begin{array}{l} \mathsf{del}(\text{stick}(x)); \\ \mathsf{del}(\text{stick}(((x{+}1) \bmod n))); \\ \mathsf{tell}(\text{is\_eating}(x)) \end{array}\right]; \text{eats}(x,\, n)$$

$$\text{eats}(x,\, n) \quad \Leftarrow \left[\text{TRUE} \Rightarrow \begin{array}{l} \mathsf{del}(\text{is\_eating}(x)); \\ \mathsf{tell}(\text{stick}(x)); \\ \mathsf{tell}(\text{stick}(((x{+}1) \bmod n))) \end{array}\right]; \qquad \text{thinks}(x,\, n)$$

Table 1.1: Process Definitions for the Dining Philosophers

of sort *Nat*, corresponding to the number of the philosopher and to the total number of philosophers in the system. In order to start to eat, a thinking philosopher has to execute a guarded action. The guard $\text{stick}(x) \wedge \text{stick}(((x{+}1) \bmod n))$ ensures that the needed sticks are both available, and the three elementary actions modify the theory by removing the two sticks, and by adding the eating philosopher. The guard TRUE in the action of the process eats reflects that an eating philosopher can decide to stop eating at any moment.

The initial process term is the parallel composition of n thinking philosophers, numbered from 0 to $n-1$, *i.e.*,

$$\text{thinks}(0,\, n) \;\parallel\; \ldots \;\parallel\; \text{thinks}((n-1),\, n)$$

Thus, at the beginning of the execution of this system, we observe the situation shown in figure 1.4. Thus, the solving of the goal is_eating(x) is initially impossible: There exists no substitution for x such that is_eating(x) could be reduced to TRUE. However, during the execution, the philosophers start and stop eating (respectively, thinking), such that we might observe the sequence of situations shown in figure 1.6. Together with the images representing the table of the philosophers, figure 1.6 also shows the essential parts of the current theory (or store).

First, philosopher number 3 starts eating (situation 1). In situation 2, philosopher number 0 joins him. Thus, in situation 2, the current theory has been changed such that the solving of the goal is_eating(x) returns the two answer substitutions $\{x \mapsto 0\}$ and $\{x \mapsto 3\}$. Philosopher number 0 has stopped eating in situation 3, whereas philosopher number 3 is still hungry and continues to eat. He is joined by philosopher number 1 in situation 4 and by philosopher number 5 in situation 5. Hence, the solving of the goal stick(y) in situation 5 is impossible, since all sticks are in use, and no stick is available on the table.

Notice that our solution of the Dining Philosophers is straightforward. Indeed, the store is a simple description of the situation, and the two processes are obtained immediately from the formalisation of the behaviour of a philosopher by an extended automaton. Notice further, that our solution does not need an additional semaphore in order to avoid dead-locks, and that we provide a generic description of a philosopher as a process.

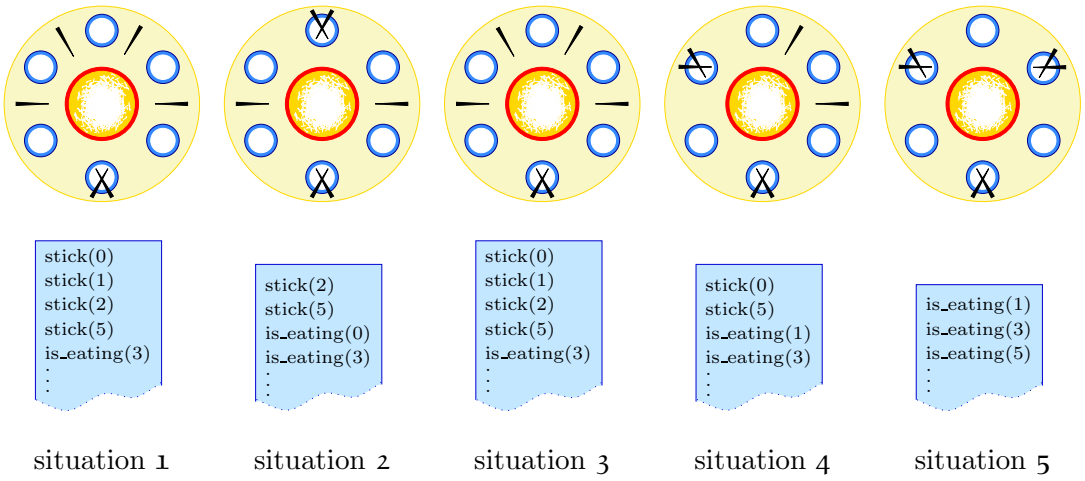

Figure 1.6: Possible Execution Sequence for Six Philosophers

1.2 Plan of the Thesis

The rest of the thesis is organised as follows. In the following chapter we present some of the work on related programming styles or paradigms, programming languages, or research that pursues similar goals, *i.e.*, the integration of several programming paradigms, in particular declarative programming and concurrency.

The complete formal definition of our computation model is the subject of chapter 3. After a recall of the basic notions and definitions of declarative, *i.e.*, functional and logic, programming, chapter 3 introduces the notions of actions, processes and components and gives the corresponding definitions. The chapter ends with a presentation of the construction of a system from a set of components. Since chapter 3 is rather long, we give a small table of its contents on page 75.

We present the operational semantics of our computation model in chapters 4. As already mentioned, the operational semantics of a component is composed of two parts, namely the execution of processes and the classical operational semantics of the store. In addition to these two parts, chapter 4 briefly presents the semantics of a system composed of several components.

In chapter 5 we show that a semantics for the processes of a component based on traces of execution according to the semantics presented in chapter 4 is not compositional and suggest an extension, namely the use of labeled traces, for which we prove compositionality.

To show that our computation model facilitates the understanding of and reasoning about programs, we present in chapter 6 an analysis of non-interference for the processes of a component, applied to the problem of ensuring the confidentiality of information. This analysis is based on an abstract execution of a component.

We have validated our proposal of a programming framework by the implementation of a prototype, which we present in chapter 7. This prototype is composed of an interpreter for a component, defined according to the operational semantics presented in chapter 4. Since our computation model is a conservative extension of declarative

programming, this interpreter can be used for both, the execution of component as well as the execution of declarative programs. Along with the broad outlines of the architecture of the prototype, we give also further examples of programs.

In chapter 8 we give a comparison of our computation model with some of the related work presented in chapter 2, in particular other extensions of declarative languages with concurrency. To avoid a double presentation of the related frameworks, chapter 8 does not present the related work. Therefore, we recommend the lecture of chapter 2 prior to chapter 8. Finally, chapter 9 concludes, giving some directions for future work.

Chapter 2

Related Programming Styles

Research on programming languages and styles is a large and diversified field. In this chapter, we present the different programming languages and related models we are aware of and that are rather closely related to our work, in the sense that they either use similar methods or pursue similar objectives. Obviously, in view of the large number of existing works, this chapter can and does not aim to be complete; we rather focus on the presentation of some examples illustrating the different approaches we are aware of. Furthermore, we limit ourselves in this chapter on a presentation of the different approaches; a comparison with related work is subject of chapter 8.

We present first some extensions of declarative programming languages with concurrency. In a second step, we describe some languages based on theoretical models for concurrency, namely process calculi. Then we overview some research which investigates the interaction and coordination between several concurrent programs. We present also some approaches of executable specifications, and, last, but not least, some approaches aiming at the combination of different programming styles, or paradigms, in a single, multiparadigm computation model. Note that the structure of this chapter is based on an arbitrarily chosen classification of the different approaches. Another classification would have equally well suited the purpose, and some of the work presented would have had its place equally well in a different section (of this chapter). In fact, the multi-paradigm character of most approaches renders any linear, one-dimensional classification very subjective.

2.1 Declarative Programming

Classical declarative languages, *i.e.*, functional, logic and functional-logic languages, aim to provide high-level descriptions of applications or systems. They are called declarative since their aim is to allow to describe or declare the problem, instead of giving a solution of how to resolve it. Since they are based on well mastered mathematical concepts, namely functions and predicates, which have been successfully used in describing algorithms even before the invention of computers, these languages come equipped with a precisely defined semantics. Due to these theoretical foundations, they exhibit well-known nice features, as for example a high level of abstraction, readability due to concise programs, efficient compilation techniques, "optimal" execution

strategies, proof methods, type systems, denotational semantics, *etc.* [Hug90, HJ94].

However, the theoretical concepts of functions and predicates that constitute the basis of classical functional-logic languages, are not sufficient to capture the whole complexity of real-world applications [Weg98] where interactivity, concurrency and distributivity are needed.

To overcome these limitations, quite a few concurrent extensions for declarative languages have been proposed. Most of them do not distinguish clearly between processes and the concepts underlying the declarative language, but rather *encode* or *simulate* processes in terms of the latter. Since this makes these languages rely on the possibilities offered by the operational semantics of the declarative language at hand, these approaches for integration of concurrency seem to be tailored for specific languages, and the generalisation of the method to a general computation model is not straightforward.

In the following, we look more closely at some concurrent extensions of declarative programming languages, considering the logic, functional and functional-logic languages separately. We conclude with a brief presentation of linear logic programming.

2.1.1 (Constraint) Logic Programming

In *Logic Programming* [Kow74, vEK76], a program is a description of a first-order theory by set of Horn Clauses. The execution of a logic program corresponds to solve a *goal*, *i.e.*, to check the satisfiability of a conjunction of atoms using an operational semantics based on SLD-resolution [KK71]. *Constraint Logic Programming* replaces unification in logic programming by the more general concept of constraint solving [JL87]. In fact, unification can be seen as solving a particular kind of constraint, namely to compute the unifier of two terms.

One of the main advantages of logic programming is its well-defined semantics (the least Herbrand model). If we consider a Horn clause such as $p(x) \Leftarrow q, p(x)$, the denotation of the predicate p is empty according to the classical semantics of logic programs (*i.e.*, for any x, $p(x)$ does not hold). But if we see this clause as the description of a process, the semantics of p is the infinite sequence (of actions) q^∞.

Neglecting this difference, most concurrent extensions of (constraint) logic programming are based on the so called "process interpretation" of (constraint) logic programming [vEdLF82, Sha83, Sha87]. According to this view, (constraint) logic programming can be *seen* as "*inherently parallel*", in that a goal can be *seen* as a set of parallel processes, using shared (logical) variables as communication channels. Hence, in interpreting conjunction as parallelism, the benefit of parallel execution can be made easily useful for logic programming languages [CdKC94]. However, this *implicit* parallelism which increases the performance of a logic program by using more than one processor is, in our opinion, to be distinguished from concurrency *explicitly* specified by the programmer.

In this section we give a short glance on some of concurrent (constraint) logic programming languages. A more detailed review of the historic development leading to concurrent constraint programming can be found, for instance in [dBP94] or in [Tic95]. The latter paper points at the "deevolution" of concurrent logic programming languages, that is to say the abandon of high-level features in favor of more practically implementable ones.

Viewing goals as processes, the implementation of "don't know nondeterminism" using backtracking, used for example in Prolog, becomes too inefficient (or even impossible, since a printout, for example, cannot be undone) to implement. In fact, in order to maintain consistency, the whole system should backtrack, since the actions of the process needing to backtrack may have already influenced other processes. Hence, most concurrent logic programming languages provide mechanisms to control the choices made by processes, in order to avoid the need for backtracking. The most popular of such mechanisms are *guards*: a clause can only be selected if the associated guard is satisfied or *entailed* by the current set of constraints, also called the constraint store which is shared by all processes. In this model, communication between processes is directed and asynchronous, since some processes are seen as producers (of bindings) on a certain variable and others as consumers, and the action of the producers and consumer may be performed at different times, see for instance Concurrent Prolog [Sha83, Sha86], Guarded Horn Clauses [Ued85] or PARLOG [CG86].

Another approach to synchronisation is to suspend a process until enough bindings have been produced (by other concurrent processes) so that the number of consistent alternatives for the next step reduces to one, as for instance in AKL [HJ90, FHJ91, JH94, Jan94]. The commitment to a rule in ALPS [Mah87] introduces an additional condition, namely that no further restriction on the global variables can invalidate the guard of the rule and the commitment to the rule. Stated otherwise, in ALPS the current constraint store has to *entail* the guard of a rule.

The idea of entailment for commitment to a rule of ALPS combined with the expression of processes using extra-logical combinators as in process algebras lead to the family of concurrent constraint programming (ccp) languages [SR90, Sar93]. The execution model of the family of ccp consists of a set of agents (or processes) communicating via a common (constraint) store. The only actions executable on this store are telling a new constraint and asking if the current store entails a constraint. Hence, a language of the ccp family is parametric with respect to the constraint system used for the store. An agent (or process) A in ccp is defined according to the following grammar:

$$\begin{array}{lcl} g & ::= & \mathsf{tell}(c) \;\mid\; \mathsf{ask}(c) \\ A & ::= & \mathsf{Stop} \;\mid\; \sum_{i=1}^{n} g_i \rightarrow A_i \;\mid\; A \parallel A \;\mid\; \exists_x A \;\mid\; p(\tilde{x}) \end{array}$$

where the agent Stop represents successful termination and the *guarded choice* agent $\sum_{i=1}^{n} g_i \rightarrow A_i$ behaves like one of the A_i for whom the associated guard g_i is entailed by the store. $A_1 \parallel A_2$ is the parallel composition of the agents A_1 and A_2, $\exists_x A$ behaves like A, but with x considered to be *local*, and procedure calls $p(\tilde{x})$ are defined by declarations, *i.e.*, sentences of the form: $p(\tilde{x})$:- A.

The members of the ccp family differ in the semantics of the different operators, namely tell and ask. As an example, while tell always succeeds and possibly leads to an inconsistent constraint store, an *atomic* atell(c) blocks on a constraint store d if the conjunction of the constraint c with constraint store $c \wedge d$ would lead to a inconsistent constraint store. The semantics of the ccp family of languages has been well investigated, see for instance [SRP91, dBP91] and a process algebra (see section 2.2.1) for ccp has been suggested in [dBP92]. An implementation of the ccp framework is Janus [SKL90].

The original model of ccp is limited to a monotonic evolution of the store, since constraints cannot be removed. Suggestions for non-monotonic extensions either provide new built-in actions [dBKPR93, CR95] or use non-monotonic logics, as logic with defaults [SJG95, SJG96, GJS96] or linear logic [SL92, BdBP97, RF97, FRS98].

For example, [dBKPR93] introduces an operator $update_x$ used to hide the name of a variable (and thus to erase all constraints on it): executing $update_x(c)$ transforms the store d to $(\exists_x d) \wedge c$. [CR95] add two new operators. $remove(c)$ removes the constraint c from the store, and $enforce(c)$ forces the store to entail the constraint c. This enforcement of c may require the removal of all constraints present in the store which are inconsistent with c. In order to ensure a deterministic computation of the result, *i.e.*, the final store, [CR95] introduce the notion of constraint dependencies. For instance, considering the process $\mathsf{ask}(c) \rightarrow \mathsf{tell}(d)$, the constraint d depends on the constraint c. These dependencies have to be kept along with the current store, in order to enable the removal (respectively, addition in the case of a negative dependence) of the dependent constraints.

Using the logic for default reasoning of [Rei80], testing the *absence* of information becomes possible [SJG95, SJG96, GJS96]. More concretely, the process (or agent) if c else A checks that the constraint c is not entailed by the current store and then behaves as the process A, making the assumption (or the "guess") that c will not be entailed by any future store. This strong and restrictive demand on stability of the result (store) assures that the result computed does not depend on vagaries of parallel execution of the processes, as for instance the differences in the execution speed of processors.

Using linear logic[1], the consumption or removal of resources, *i.e.*, constraints, from the store can be modeled. [SL92] is based on a sub-logic of simply typed higher-order linear logic, namely HLL. Processes are considered as formulas of a particular sub-class of HLL, where the tell primitive of (standard) ccp is replaced by (concurrent) agents consisting only of the corresponding constraint. In contrary to the approach of [SL92] which can be considered as a linear logic programming language, [BdBP97] and [RF97, FRS98] extend the model of ccp by considering *linear* stores (or constraint systems). [BdBP97] defines a linear constraint systems as a multiset of constraints, equipped with an entailment relation (extending a basic entailment relation over constraints to multisets). Besides the actions known from ccp, namely (a)tell and ask, [BdBP97] introduces the operator get. Informally, $\mathsf{get}(c)$ corresponds to $\mathsf{ask}(c)$ of classical ccp, where a multiset of constraints, *e.g.*, $\{d_1, \ldots, d_2\}$, which entails c is removed from the store. The (operational) semantics of processes is defined by means of so-called *histories*, which can be considered informally as dependence graphs between (executions of) actions. The constraint stores of LCC [RF97, FRS98] are axiomatised in (intuitionistic) first-order linear logic (ILL) [Gir87]. Correspondingly, the entailment relation used for checking the guards is the entailment in ILL. Thus, testing of a guard *always* removes the constraints needed for the proof[2]. Using a translation of LCC into ILL and the phase semantics of linear logic [Gir87], it is possible to prove safety properties of programs written in LCC and ccp (since ccp programs are a particular sub-class

[1] See section 2.1.4 for a brief presentation of linear logic and some associated programming languages.

[2] In contrary to [BdBP97], LCC does no longer provide an ask-action. In fact, $\mathsf{ask}(c)$ can be simulated by means of $\mathsf{get}(c) \longrightarrow \mathsf{tell}(c)$.

of LCC programs).

The logic programming language Prolog provides two "predicates" that allow one to modify the logic program by adding and deleting clauses, namely `assert` and `retract` [DEDC96]. In fact, the predicates forming a program are considered as being stored in a "database" which can be updated by means of the mentioned built-in "predicates", or "extra-logical operators". This view of logic programming is adopted for the description of deductive databases, where rules or clauses allow to infer new information from a database. But in addition to the fact that the logical semantics of Prolog does not account for these updates operations, Prolog does not support *atomicity*[3], a basic feature of database transactions. [BK98b] surveys research on these update operations in logical databases from the point of view of languages for the description of transactions.

Transaction Logic ($\mathcal{TR}$) [BK94, BK98a] is a logic for the description of database transactions. $\mathcal{TR}$ is equipped with a model and a proof theory and contains a logic programming language as subset. Transaction are described in $\mathcal{TR}$ by means of clauses (as in Prolog), but instead of a single operator of conjunction, $\mathcal{TR}$ introduces a second form of conjunction, called sequential conjunction $\otimes$. Informally, the rule $b \leftarrow a_1 \otimes a_2$ states that the transaction b consists of the two (sub-)transactions a_1 and a_2 which have to be executed both, and in the sequential order, *i.e.*, (the update) a_2 is executed after successful completion of (the update) a_1. To take into account the evolution of the database, models of a program in $\mathcal{TR}$ are *sequences* of database states. A concurrent extension of $\mathcal{TR}$ is $\mathcal{CTR}$ [BK96], introducing an additional concurrent conjunction operator $\|$. Both, $\mathcal{TR}$ as well as $\mathcal{CTR}$, do not distinguish between actions (or updates) and predicates.

Another recent language for the specification of updates and transactions in a logic programming language is ULTRA (Updates in a Logic language with TRAnsactions) [WFF98]. As $\mathcal{CTR}$, ULTRA allows the modeling of concurrent updates (if they are consistent). The main difference between ULTRA and $\mathcal{CTR}$ is that ULTRA gives a precise semantics to the parallel execution of *isolated*, *i.e.*, non-interacting, transactions. But as $\mathcal{CTR}$, ULTRA does not distinguish between predicates and actions (or updates).

One of the main advantages of CIAO, the extension of Prolog described in [CH99], is that CIAO does not need large modifications of the underlying execution model, namely the WAM [AK91]. Similar to the coordination language ESP [BC91, Cia94] which we describe in section 2.3.2, the operators `assert` and `retract` of Prolog are interpreted as send and receive operations on a multiset of atoms, called "blackboard". CIAO provides (a family of) new primitives for launching the concurrent solving of goals (in a (local) copy of the current logic program). Additional primitives allow to control this new goal-solving process, *e.g.*, to force backtracking or termination, or to synchronise on its result. Communication between these processes is achieved by `assert`ing and `retract`ing (marked) atoms.

The most popular communication medium in concurrent logic programming are streams, *i.e.*, lists of infinite length. To send a message `m` on a stream `S`, which is represented by a free, *i.e.*, uninstantiated, (logical) variable, it is sufficient to bind `S` to the value `[m|S_1]`, where `S_1` is a fresh variable. Thus the next message has to be sent to the stream `S_1`. On the side of the receiver, waiting for `S` to become

[3]Informally, atomicity of a transaction ensures that the transaction either executes to completion or not at all.

constrained to a pair `[M|R]` (where `M` and `R` are free logical variables) binds the message `m` to `M` and the remaining stream, namely `S_1`, to `R`. In order to extend this scheme of communication to several senders to the same stream, especially when the number of senders is susceptible to change dynamically, the introduction of additional merger processes becomes necessary.

To avoid the inconveniences associated with additional mergers, the notion of *ports* as a many-to-one communication medium has been introduced in AKL [JMH93]. Instead of providing a complete formal definition of ports, it is argued in [JMH93] that the introduced port primitives have a "logical reading" as a constraint and preserve the monotonicity of the constraint store. Informally, a port is a connection between a multiset of messages and a corresponding stream, and by abuse of the notation the multiset is usually identified with the port. The operation open_port(P, S) creates a new port P together with its associated stream S. open_port is considered as a constraint that states that all members of the multiset P are members in the stream S and vice versa, with an equal number of occurrences. On the other hand, to send a message M on a port P, a process has to use the operation send(P, M) which is interpreted as a constraint stating that the message M is a member of the multiset P.

Oz [Smo95b] is an extension of ccp with first-class procedures and "stateful" data-structures. As in basic ccp, the evolution of the store is monotonic. But in order to cope with the stateful features (which are necessarily non-monotonic) the store of Oz is portioned in three compartments: the constraint, the procedure and the cell store. A cell represents a "mutable binding of a name to a variable" [Smo95b] and can thus be updated to contain a binding to another variable. Surprisingly, this update is considered to be monotone, as the update to the (constraint) store stays monotone. The notion of ports of AKL has been embedded as a means for communication between processes in Oz, as well as in its distributed extension Distributed Oz [VRHB+97], where the store is transparently distributed over several sites. Unsurprisingly, the "imperative features" of Oz allow to define a semantics for ports [Smo95b, section 9, pages 333 – 334]: instead of a multiset, there is a function (or procedure) managing the stream, *i.e.*, all sending goes to this (stateful[4]) function, and only this function is allowed to modify the tail of the stream.

In most of the current constraint logic programming systems, the process of goal solving has to be restarted whenever a change in the data or underlying theory occurs. In order to improve the operational semantics in dynamic, *i.e.*, frequently changing environments, [FFS95, FFS98] investigate the reordering of reduction steps in the context of CLP, based on an incremental model of execution. It has been shown that this approach leads to better execution times than re-execution of the goal from scratch. In the particular context of constraint logic programming over finite domain constraints (CLP(FD)), [GCR99] considers the efficient retraction of constraints, in such a way that there is negligible overhead in the case that no retraction occurs.

2.1.2 Functional Programming

The fundamental operation in functional programming is the application of functions to arguments. A program in a (pure) functional programming language forms a set

[4]It has to remember the current tail of the stream.

of function definitions, where each function itself may be defined as a composition of other functions (defined in the program). Since the notion of functions is a well defined mathematical concept, a functional program is rather close to a mathematical specification of the function to be realised by the program. Functional programming has well-defined theoretical foundations, such as for instacne the λ-calculus [Chu32, Bar84] or equational logic [O'D85]. Execution of a (purely) functional program consists of evaluating an expression to its *normal form*, that is to say until no more functions can be applied.

2.1.2.1 Input/Output in Functional Programming

Similar to logic programming, in functional languages, there is no standard way for the integration of input and output (I/O), as for example interaction with the user or accessing the file-system. A first approach is to allow "impure" features. In this approach, which is followed for instance by Common Lisp [Ste90], Scheme [ADH$^+$98], SML [MTHM97] and `ocaml` [LDG$^+$01], the evaluation of an expression does not only yield the normal-form of the expression, but has also an ("side-") effect, as for instance printing something on a screen or blocking until something read from an input device can be returned as a value. This approach is called "impure", since it destroys desirable properties of functional programming, as for instance the possibility for equational reasoning, also known as "referential transparency" [Wad97].

Monadic I/O which is the approach chosen, for instance, in the lazy[5] functional language Haskell [PJHA$^+$99], separates the actual execution of actions from its denotation [Wad97][6]. Several combinators allow to combine the actions in a sequential manner. Hence a program can be seen as a specification of the command to be performed upon execution. These commands are also called state transformers [PJGF96]. One parameter of the command is the environment (or state) which is passed from action to action during execution. A semantics for a simple form of monadic I/O can be expressed within the functional language, allowing the use of standard techniques for verification of functional programs using monadic I/O [Gor94].

In other languages, as for example FL [BWW90] and Clean [PvE98], this "environment passing" is explicit and not hidden by monads. A problem with explicit environment passing is that, since the environment cannot be duplicated, sharing or duplication has to be avoided. FL [WW88] introduces new primitive operations on a history, that is to say an additional argument (to every function) representing the interaction with the environment so far. Since the history argument can only be manipulated by the new primitives, the access to the environment is ensured to take place in a controlled manner. The quite complex Uniqueness Types System [AP95b] of the lazy functional language (Concurrent) Clean [PvE98] ensures that at most one reference to the environment exist at a time. Therefore, the actions on the environment of

[5] In a lazy functional language the evaluation of a subexpression is effectuated at most once and only if its value is necessary for the evaluation of the final result. Hence the control of the moment of execution of side-effects would be rather elaborate, rendering less interesting the "impure" approach to I/O.

[6] "A program in a pure functional language is an expression that denotes the effect that the program should have on the outside world when the program is executed." [CH98, section I.4, page 11]

Clean are encapsulated in special functions which have access to the environment. An interesting feature of this approach is the possibility to handle multiple environments.

Another class of approaches is based on the notion of streams, which might be represented in a *lazy* language as (lazy) lists. In this approach, an interactive program transforms an element (or response) of the input stream into an element (or request) of the output stream, see for instance [Hen82, BW88]. A semantics of stream based I/O (in the lazy language Miranda) has been presented in [Tho90]. The model of stream processors has been used to define a framework for the construction of graphical user interfaces (GUI's), where the basic stream processors are called Fudgets [CH93, CH98]. Finally, the operations of continuation passing I/O (CPS[7]) can be derived from (synchronous) stream I/O and implemented using a top-level interpreter [Gor94].

2.1.2.2 Concurrent Functional Programming

Similar to logic programming, the execution of "pure" functional programs can also easily take advantage of several processors: Due to the absence of side-effects when evaluating expressions, the arguments of a function can be evaluated in parallel. But this *implicit* parallelism that aims at improving the (run-time) performance of the program, is to be distinguished from *explicitly* specified concurrency that allows the programmer to better structure the program. The former can be seen as an improvement of the operational semantics of functional programming languages (and could be hidden from a programmers point of view), whereas the latter extends the possibilities for structuring interactive applications [Pik89]. Several concurrent extensions have been suggested for functional languages, but most of them do not distinguish between processes and functions.

Concurrent Haskell (CH) [PJGF96] introduces new primitives for, on the one hand, starting processes and, on the other hand, atomically mutable state into the functional language Haskell [PJHA+99]. Since these primitives correspond to state transforming actions, they are introduced in addition to the available monadic I/O actions. The first of theses primitives, namely `forkIO`, has the type `IO () -> IO ()` and takes an action (*i.e.*, a state transforming function, or process) as parameter which is launched concurrently as a side-effect of the execution of the action `forkIO`. These processes can communicate and synchronise via mutable variables, a built-in type called `MVar`, corresponding to a binary semaphore than can contain a value. These `MVar` variables can be created by executing `newMVar`, (destructively) read[8] by `takeMVar` and set by `putMVar`.

These new primitives are voluntarily "low-level", since they are meant as "raw iron from which more friendly abstractions can be built" [PJGF96], using the standard abstraction facilities provided by a functional programming language. However, the difference between functions, actions and processes is not quite clear and may confuse a programmer.

Concurrent ML (CML) [Rep91, PR96, Rep99] introduces new primitives into the functional programming language SML [MTHM97]. First of all, processes are created by execution of the primitive `spawn` which takes as argument an expression and the

[7] CPS is the acronym of Continuation passing style.

[8] The read operation is blocking if there is no value.

side-effect of which is to evaluate the expression concurrently as a separate process (or thread). Processes communicate in a message passing style using buffered `channel`s. To allow an abstract expression of synchronisation between processes, CML introduces the notion of *events*. Informally, an event corresponds to a possible synchronisation which has to be made effective by executing the primitive `sync`. A process executing `sync` is blocked until another (concurrent) process `sync`ronises on a corresponding event, so that the communication and synchronisation can take place. The basic events correspond to the emission and reception of a message, and are provided by means of the functions `recvEvt` and `sendEvt`[9]. More elaborate events can be constructed using further combinators. The combinator `wrap` allows to specify a function which is to be applied to the value returned by the event immediately *after* the synchronisation on the event occurs. A nondeterministic choice between a list of events to synchronise on is provided by the combinator `choose`. Finally, the combinator `guard` allows to associate a function that is called *before* every synchronisation on the event.

Similar to CH, the notions of processes, actions and functions are not clearly distinguished in CML. One of the main motivations behind this fact seems to be the possibility to use the powerful standard abstraction of functional programming (*e.g.*, higher-order polymorphic functions) for the description of processes, allowing a programmer to use the same abstraction facilities as usual in functional programming languages. However, while the type system of CH inhibits the use of processes inside the "purely functional" part such that a CH-program is stratified in two layers, all functions in CML are allowed to have side-effects.

As CML, Facile [TLK96a, TLK96b] extends the functional language SML with "behaviour-expressions" (defining the behaviour of processes), synchronous communication channels and the notion of nodes, also called sites or locations. Behaviour-expressions can be transformed into a value, called a process "script", which can be manipulated by functions as usual. The evaluation of the expressions `(spawn s)` and `(r_spawn n s)` for a location `n` and a script `s` have the side-effect of spawning a new process whose behaviour is defined by the script `s` (on the node `n`). Since `spawn` is considered as a function, Facile allows functions and processes to call each other mutually, similar to CML. Communication between processes in Facile is restricted to message passing through channels.

The untyped concurrent functional language Erlang [AV90, AVWW96] models systems as sets of processes the behaviour of which is described by means of functions. Similar to CML and CH, process creation is the side-effect of a special primitive, namely `spawn`, which returns an identifier or name of the created process. Erlang-processes communicate via message passing using special built-in functions with side-effects, namely the sending of messages to other processes. An interesting feature of Erlang are the built-in functions `delete_module`, `load_module` and `purge_module` which allow users to replace a module (*i.e.*, a part of a program) by a new corrected one without stopping the entire application. Erlang has been (designed and) used for the development of large, fault tolerant telephone switching systems [EPd92, Arm96, Art01].

The parallel functional programming language Eden [BLOMP98] combines a lazy functional language with a (coordination) language for processes, in order to allow

[9]These "functions" were initially called `receive` and `transmit` in [Rep91].

the explicit expression and control of the parallel evaluation structure of a functional program. Eden provides primitives for the definition and creation of processes, communication channels and interaction and the specification of interconnection topologies [BLOM95]. As a consequence, the processes in Eden are distinguished from functions and other static definitions.

Concurrent Clean [NSvEP91, PvE98] is based on term graph rewriting, *i.e.*, a program is specified by a set of graph rewriting rules. In Clean, the programmer can influence (the efficiency of) the code generated by the compiler by means of annotations. For instance, cyclic data structures can be created due to explicit control sharing. Other annotations allow to change the default lazy evaluation strategy locally into a more efficiently implementable eager one. Furthermore, annotations allow a programmer to explicitly specify which parts of the term (or graph) should be reduced concurrently. These annotations distinguish between *parallel* evaluation on another processor and *interleaved* evaluation on the same processor. Thus, similar to Eden, this form of concurrency in (Concurrent) Clean serves mainly the specification of a more efficient operational behaviour of a program by explicitly organising its parallel execution. In our opinion, this form of parallelism is to be distinguished from concurrency which aims at providing a better structured program (that might be executed sequentially by simulation of the parallel execution of processes).

[AP95a] investigates an extension of concurrent Clean (written entirely in Clean) for the simulation of concurrent interactive systems. (Interactive) processes are modeled by means of state transition functions, where the state is a four-tuple composed of a local and a shared state, and the current state of the file-system and GUI. As in CML and CH, the creation of a new process is the side-effect of the execution of a primitive action. The simulation of the execution of (interactive) processes ensures the interleaving execution of the different concurrent processes.

Functional Parallel Programming (FP2) [Jor84, Jor85] allows the description of systems by means of networks of processes communicating by means of ports, *i.e.*, directed channels. The values sent over these communication ports are defined as standard algebraic data types. Processes are defined by, on the one hand, the functions which describe the computation of the values sent by processes in response to received values, and, on the other hand, by transition rules describing the ordering among the communication actions of the process. The states of a process are represented by means of closed atoms, and a transition rule consists of a precondition, a set of events and a postcondition. Informally, the semantics of a transition rule is to transform the state into the postcondition, if the state is an instance of the precondition and the set of events is possible.

2.1.3 Functional Logic Programming

LogLisp [RS81] was one of the earliest attempts to create a language integrating functional and logic programming. The motivation for amalgamating functional and logic programming is to combine the advantages of both paradigms, namely the efficient operational behaviour of functional programming and the expressive features of logic programming, as function inversion, partial data-structures and logical variables. Thus functional-logic programming is more efficient than pure logic programming while being

more expressive than pure functional programming, and has like both, functional as well as logic programming, a well-defined semantics. Background material on the integration of functional and logic programming, including a comprehensive list of references up to 1994, can be found in the survey paper [Han94]. In this paragraph, we present some recent proposals for functional logic programming languages.

One widespread operational semantics for functional-logic programming is based on *narrowing*, which combines a rewrite step with the instantiation of free variables using unification. Narrowing is *sound*, *i.e.*, it computes only correct answers, and *complete*, *i.e.*, all possible answers are effectively computed. Several strategies for narrowing have been proposed, some of which have been proved optimal in the sense that a minimal number of narrowing steps is performed [AEH97, AEH00]. *Residuation* is another operational semantics for functional-logic programming which avoids the instantiation of variables, that is to say, residuation does apply a rewrite rule only if all parameters are sufficiently instantiated, and otherwise suspends. Thus, residuation is interesting in a context where goal are solved in parallel, since suspending only makes sense if some concurrent processes may instantiate the variable. In order to represent infinite data structures and to speed up the operational semantics, the use of graphs (instead of first-order terms) has been suggested [EJ99].

As for purely logic or purely functional programming languages, functional-logic programming languages need to be extended to cope with interactive concurrent applications.

$\mathcal{TOY}$ [GHLR96, ASRA97, GHLR99, LFSH99] is a programming language based on a constructor based conditional rewriting logic (CWRL). The core notion of $\mathcal{TOY}$ are lazy non-deterministic functions, *e.g.*, the term rewriting systems corresponding to programs in $\mathcal{TOY}$ are not required to be confluent, *i.e.*, a term might be rewritten to several, different constructor terms. To represent goals and conditions, $\mathcal{TOY}$ uses the notion of *joinability* (in symbols, $a \bowtie b$), which means that a and b can be rewritten to a common constructor term t. Informally, a program in $\mathcal{TOY}$ consists of a set of equations specifying polymorphic algebraic data-types and conditional rewrite rules defining the operational behaviour of functions defined over these data types. However, we are not aware of a concurrent extension of $\mathcal{TOY}$.

The functional logic language Curry [Han97, HAK+00b] is based on a combination of narrowing with residuation. Processes in Curry are represented by constraints, that is to say that they are introduced in the style of logic programming. On the other hand, the interaction with the external environment is limited to the "functional part", since Curry uses monadic I/O as, *e.g.*, Haskell (see section 2.1.2.1). Hence, there are two different kinds of operators for sequential composition: the monadic operators concerning I/O and the sequential conjunction of constraints (or processes).

Recently, the idea of ports introduced in AKL [JMH93] has been extended and integrated into the functional-logic language Curry in order to cope with distributed applications [Han99]. In Curry, ports can be named, that is to say it is possible to associate a symbolic name (or string) to a port and to use this name to identify the port in a distributed application. Since the symbolic name is a value of a (built-in) data type of Curry, it can also be communicated between processes, allowing the communication structure to vary.

The functional logic language Escher [Llo95, Ede99, Llo99] is not based on narrowing

but on rewriting (with residuation). Escher extends the functional language Haskell in several ways. First, the all variables in the right hand sides of rules that do not appear in the left hand side of the rule have to be explicitly quantified. Second, the answers computed by Escher may contain (besides constructors as in Haskell) variables and some functions which are defined in system modules. The basics of logic programming are explicitly defined by a number of well-chosen rewrite rules defining the functions in the system module `Booleans`. Thus Escher responds to a goal, *i.e.*, a term containing variables, with a term corresponding to the disjunction of all possible solutions, *i.e.*, equations defining values for the variables.

In a concurrent extension of Escher [Llo], processes communicate by means of a common store, called *blackboard*[10]. As in CH, processes are represented as functions of type `IO ()`, and the actions processes may execute are the primitive functions of this same type, as for instance `putChar` which interacts with the outside world of the program, and `(store ref value)` which modifies the blackboard by assigning the value `value` to the reference `ref`. A further difference to CH is that concurrent Escher provides a symmetric operation for parallel composition by means of a special primitive function called `ensemble` which models a set of concurrently executing processes.

2.1.4 Linear Logic Programming

The main difference between classical and linear logic [Gir87, Gir95] is that, contrary to classical logic, the number of occurrences of a formula in linear logic matters. For instance, linear logic distinguishes between the formula "A" and the formula "A and A". The intuition between this distinction is that linear logic considers formulæ as *resources*, and in the former formula, the formula or resource A occurs only once, whereas A occurs twice in the latter. Due to this differentiation linear logic is sometimes called "resource sensitive". Linear logic has two kinds of conjunction (and, dually, of disjunction), namely *multiplicative* ("times", $\otimes$) and *additive* ("with", &). Intuitively, the proposition $A \otimes B$ means that A and B hold *both*, whereas $A \,\&\, B$ stands for an (internal) choice between A and B. The intuitive meaning of the additive disjunction ("plus", $\oplus$) is an (external) choice. Informally $A \oplus B$ means that we may have A or B, but that someone else decides for us which, as it occurs if we toss a coin. Since the meaning of the multiplicative disjunction ("par", $\parr$) is "not that easy" [Gir95][11], we refer the reader to introductions on linear logic, *e.g.*, [Sce90, Wad93, Gir95], for more detailed presentation, and restrict ourselves to notice that $\parr$ can be (and is) used to represent parallel composition of communicating processes. The linear negation of the formula A is written as $A^{\perp}$, and linear implication $A \multimap B$ which means informally that the consumption of A yields B, is defined by $A^{\perp} \parr B$. A formula A of classical logic can be written in linear logic using the two exponentials $!A$ ("of course A"), asserting a infinite number of occurrences of A, and its dual $?A$ ("why not A").

As for logic programming, in linear logic programming the execution of a program is seen as proof search (in linear logic). Similar to the restriction of Prolog to a fragment of classical logic, namely definite Horn clauses, existing linear logic programming

[10] This terminology and communication scheme is similar to some coordination languages, *e.g.*, Linda [Gel85], which we present in more detail in section 2.3.

[11] The double-quotes are from [Gir95].

languages differ in the set of formulæ or fragments of linear logic that are allowed as programs. In this subsection we present some of the linear logic programming languages we are aware of; for a more complete survey, the interested reader may consider for instance [Mil95, Win97]. Most of these languages are based on proof theoretic considerations in sequent calculi [Gen35]. In such a setting, sequents are used for the representation of a computation *state* and the actions and dynamics of a computation are modeled as transformations of sequents occurring during the incremental construction of cut-free proofs. In these languages the proof-theoretic behaviour of logical connectives gives a rigorous semantics to the various aspects of concurrent programming, exploiting the fact that the inference rules defining the operators of linear logic have strong similarities to the inference rules defining the operational semantics of process calculi (which is presented in the following section).

The programming language LO[12] [AP91, AP92] generalises the paradigm of logic programming using definite Horn clauses by, on the one hand, allowing multiplicative linear disjunction ("par", $\parr$) in the heads of clauses and, on the other hand, using *linear* implication. These generalised clauses are interpreted as *methods* describing state transitions, *i.e.*, the formulæ in the head of a clause are consumed when the clause is applied. Using the proof theoretic behaviour of the connectives of linear logic, in particular of $\parr$ and $\multimap$, the operational semantics of LO has a precise definition.

Proofs in a sequent calculus may be redundant, in the sense that several syntactical representations for a same proof exists, for instance by reordering some of the inference rules. In [And92] it is shown that the class of "focusing" proofs is complete, *e.g.*, that every provable formula of linear logic has a focusing proof. The programming language LinLog [And92] is based on the same fragment of linear logic as LO, since focusing proofs have a more compact form for this fragment. However, this syntactical restriction does not come at the expense of loss of expressive power, since an encoding, similar to the encoding of classical logic to clausal form, of full linear logic to LinLog can be defined [And92].

The completeness of uniform proofs for intuitionistic logic depends on the fact that in intuitionistic logic sequents are single conclusion, *i.e.*, they contain a single goal to be proved. The straightforward extension of the notion of uniform proofs to intuitionistic linear logic was used for the design of the linear logic programming language Lolli [HM94].

There are essentially two distinct possibilities to extend the notions of uniform proof and goal directed search to multiple conclusion as needed in full linear logic: by requiring that either *some* or *all* of the components of the goal are reduced. The multiple conclusion specification logic Forum [Mil96] follows the second approach. Forum is a particular presentation of linear logic which allows to see all of linear logic as a logic programming language. Consequently, Forum extends is an extension of Lolli and LO[13]. Different models of concurrent programming have been successfully encoded in Forum, see for instance [Mil96] for an encoding of CML and [Mil92a] for one of the π-calculus.

[12] LO stands for "*L*inear *O*bjects".

[13] As already mentioned, full linear logic can be represented into LinLog (of which LO is a subset). However, the relationship of Forum to full linear logic is more immediate, since no additional symbols are required [Mil96].

The language Lygon [HPW96, Win97] chooses the other possibility to extend the notions of uniform proof and goal directed search to multiple conclusions, namely by requiring that *some* of the goal can be reduced. This approach ensures that goal-directed search is complete for many subsets of linear logic, such that Lygon encompasses the broadest subset of linear logic as a logic programming language, where a logic programming language is defined as subset of a logic such that a proof search strategy respecting some criteria[14], is complete [Win97]. Similar to the other linear logic programming languages mentioned so far, Lygon allows to encode state changes and concurrency.

Another approach linking linear logic and computation is described in [Abr93]. This work is inspired by the Curry-Howard isomorphism [dG95] which links (classical) intuitionistic logic and functional programming in form of the typed λ-calculus by interpreting formulæ as types and proofs as programs. The main difference between this approach and the linear logic programming languages mentioned so far, is that the latter consider computation as *proof search*, whereas the former consider computation as the *normalisation* of proofs (or expressions). Since intuitionistic linear logic can be seen as a refinement of ordinary intuitionistic logic, its computational interpretation can be seen as an refinement of the λ-calculus. However, the computational interpretation of full linear logic requires a more radical departure from the functional framework, namely multiset rewriting as represented by the Chemical Abstract Machine (CHAM) [BB90, BB92]. However, the logic(s) considered in [Abr93] are (second order) *propositional* logics.

2.2 Concurrent Programming

A widely used abstraction for the modeling of concurrent systems are *processes*. Unfortunately, there are as many ways to define processes as there are different programming styles (including, among others, temporal logic programming or Petri Nets). Nevertheless, *process calculi* have been well investigated and provide a clean framework for the description of concurrent processes. In this section we present some of the process calculi we are aware of. Common to most of these process calculi is the so-called *interleaving semantics* for concurrency, that is to say, that actions are *atomic*, and no two actions occur at the same time, except for synchronisation and communication. Consequently a particular execution of a concurrent system can be characterised by the *sequence* of actions that have been executed.

These calculi are nice theoretically models for concurrency, but they are not designed to used for actual programming. Hence we present along with the calculi some programming languages using them as their theoretical foundation. But similar to concurrent extensions of declarative languages, where processes have to be *encoded* as, *e.g.*, functions or predicates, programming languages uniquely based on process calculi *encode* the notions of functions and predicates via processes.

[14]Forum does not qualify as a logic programming language according to most of the criteria of [Win97].

2.2.1 Process Calculi and Process Algebras

One of the first process calculi is the *Calculus of Communicating Systems* (CCS) [Mil80, Mil89]. In CCS, every atomic action a has precisely one other atomic action $\bar{a}$ with which it communicates, and the result of a communication is not some atomic action, but the *silent step* τ. As operators on processes, CCS offers parallel composition and nondeterministic choice ((infinite) summation). Basic CCS does not provide a general operator for sequential composition, but only action prefixing. However, such an operator can be easily defined [Mil89, section 8.2]. The semantics of CCS is defined by a transition system and a relation of observational congruence (which abstracts from silent steps). A translation of a calculus with value passing in to the basic calculus of CCS is presented in [Mil89, section 2.8]. The idea is to introduce a specific action for each transmittable value[15]. Another, more direct treatment of value passing (in a slightly different calculus) can be found in [HI93]. Both calculi do not provide parameters for processes, but could be extended (easily) to do so.

The concurrency model of *(Theoretical) Communicating Sequential Processes* (CSP) [Hoa78, BHR84, Hoa87] is based on failure semantics[16]. CSP provides two different choice operators, namely internal $\sqcap$ and external $\Box$ choice. Internal choice is nondeterministic and cannot be influenced by the environment, whereas external choice depends on the environment. Other operators on processes of CSP include two parallel compositions (using intersection, $\|$, and interleaving, $|||$), sequential ; composition and iteration $\underline{*}P$. The latter can be defined recursively as $\underline{*}P = \mu q.(P; q)$ where μ denotes the least fixpoint operator. Further operators concern the handling of interruptions. As in CCS, passing of values along communication channels is encoded by providing a specific action for each transmittable value. The action executed by the process $c!v \rightarrow P \parallel c?x \rightarrow Q$ is $c.v$ (and the resulting process is $P \parallel Q[v/x]$, where $Q[v/x]$ denotes the process obtained by replacing in Q every action $c.x$ by $c.v$). An example of a programming language based on CSP is Occam [Bar92].

Another approach for modeling concurrency are *process algebras*, as for instance the *Algebra of Communicating Processes* (ACP) [BK84, BW90]. In contrary to CCS and CSP, processes in ACP are characterised as models of algebraic theories, describing the actions processes may execute by means of axioms or equations[17]. As for any axiomatic theory, different models are possible for a same theory description, but they may be distinguished by additional axioms. Communication between processes is synchronous and uses a communication function γ, where $\gamma(\mathsf{a}, \mathsf{b})$ describes the action modeling the communication between the actions a and b. As an example, the communication function for CCS could be defined as follows: $\gamma(a, \bar{a}) = \tau$, for every action a.

LOTOS[18] [LOT00, ELO01] is a (specification) language combining algebraic specification and process calculi, where processes are defined via process algebra operators (similar to those of CSP and CCS) whereas functions are described in equational logic in the style of ACT ONE. LOTOS is standardised by the ISO and has been applied

[15] Hence the need for *infinite* summation, since an infinite number of arguments might be possible.

[16] [Hoa78] is based on *trace semantics*, but is superseded by TCSP [BHR84] which is based on *failure semantics* and (later [Hoa87]) also called CSP (instead of TCSP). By CSP we refer to this later model.

[17] More recently, a presentation of ACP via a system of transition rules has been given [Fok00].

[18] LOTOS is the acronym for the (extended) Language Of Temporal Ordering Specifications.

extensively to the specification of the ISO/[19] (communication) protocols.

2.2.2 Calculi for Mobile Processes

Mobility in concurrent systems was probably first studied in the actor model [Hew77, Agh86]. Actors had considerable success as a model for research in system architecture and design. However, the only formally defined models of actors we are aware of do not provide a general theory or calculus for the general actor model. For instance, [Cli81] developed a semantics of actor systems, but without defining a notion of equivalence of actor systems, and [AMST92, AMST97] define a semantics, based on a transition system presented in [Agh86], which is restricted to a *particular* actor language, namely a simple functional language with concurrency primitives based on the actor model.

Extended CCS (ECCS) [EN86] adds to CCS the possibility to pass the names of communication links, in addition to the other values which can be communicated. Since the names of communication links are distinguished syntactically from the other values, the definition of ECCS is rather complicated. Nevertheless, ECCS can be considered the first proposal of an algebraic calculus for *mobile* processes, where mobility is to be understood in the sense of a varying communication structure. Since this form of mobility is achieved by passing the communication links between processes, the model of mobility introduced by ECCS is sometimes called a "link-passing" model.

The π-calculus [MPW92, Mil99] simplifies ECCS by dropping the difference between values and channels names, putting everything into one single syntactic category, namely *names*. In this calculus, processes are defined according to the following grammar[20]:

$$P \quad ::= \quad \sum_{i \in I} \pi_i.P_i \quad \mid \quad P \mid Q \quad \mid \quad !P \quad \mid \quad (\nu x)P$$

where $x, y, \ldots \in \mathcal{X}$ are *names* and I is a finite indexing set; in the case $I = \varnothing$, the sum is written as $\mathbf{0}$. The two possible action prefixes π are:

- $x(y)$, the *reception* of a value (which is bound to the name y) on the link (or channel) named x, and
- $\bar{x}y$, the *sending* of the name y on the link x.

While the original definition of the π-calculus is *monadic*, *i.e.*, only one value can be transmitted on a channel at a time as implied by the action prefixes above, an extension to a *polyadic* calculus has been proposed in [Mil93b]. The choice operator + is external choice as in CSP, | denotes parallel composition and $!P$ – pronounced "bang P" – is similar to the iteration operator of CSP and means $P \mid P \mid \ldots$, *i.e.*, as many copies of P in parallel as you wish. As in CCS, communication is synchronous.

This rather simple calculus is quite expressive: Several encodings of the λ-calculus into the π-calculus have been defined (consider for instance [Mil92b, San98]), and a simulation of the behaviour of higher-order processes is presented in [Mil93b]. However, we are not aware of an actual implementation of the full synchronous π-calculus. In

[19] is the acronym for Open Systems Interconnection, and ISO is the short name for the International Organization for Standardization

[20] This grammar is taken from [Mil93b]

fact, due to its synchronous communication scheme using non-located channels, global consensus problems [FLP85] arise [FGL+96].

Several extensions of the π-calculus exist. The asynchronous π-calculus, independently introduced by [HT91] and [Bou92] allows only reception as a prefix, and introduces sending as a basic process. Hence, communication is asynchronous, as it is impossible (since syntactically forbidden) for the sender of a message to wait until it is received. The asynchronous π-calculus has been shown to be strictly less expressive than the π-calculus [Pal97]. The main reason for the difference between the two calculi is the (synchronous) combination of input and output prefixes in a single summation which is needed in order to implement a *symmetric* leader election protocol [Bou88]. Programming languages based on the asynchronous π-calculus are for instance Pict [PT97, PT98] and TyCO [Vas94, LSV99].

The ρ-calculus [NS94, NM95][21] adds general constraints[22] to the γ-calculus [Smo94], a calculus for modeling higher-order concurrent (constraint) programming. In the ρ-calculus, the monolithic constraint store of ccp is considered as transparently distributed over the processes, since the ρ-calculus does not distinguish between a constraint c and a process imposing the constraint c. The global constraint corresponding to the constraint store of ccp can be obtained by combination of the constraints of the processes by means of the reduction rule [NM95]. An embedding of the ρ-calculus over the trivial constraint system $\rho(\emptyset)$ into the asynchronous, polyadic π-calculus is straightforward, since fact the process part of the ρ-calculus is a subcalculus of the π-calculus, as is shown in [NM95]. The ρ-calculus is a foundation of the concurrent constraint programming language Oz already mentioned in section 2.1.1 [Smo95a].

A more direct integration of the λ-calculus in the (asynchronous) π-calculus (where replication is restricted to input prefix) by directly extending both calculi is the blue calculus or $\pi^\star$-calculus [Bou97]. In this calculus, functions, *i.e.*, λ-abstractions, are also considered as processes, and any process can be applied to any name. In fact, a message m send on a link l is considered as an application of l to m, and the name of (the link in) an input prefix is considered the location of a resource that can be "fetched" (via a dedicated transition). This allows for example to express functions as processes without the need for an explicit coding of reply-channels and thus to encode the λ-calculus in a more "direct style".

A type system for an extension of the $\pi^\star$-calculus allowing a static analysis of non-interference for mobile processes has been proposed in [Pro01].

The fusion calculus [PV98] (an improved polyadic version of the monadic update calculus [PV97]) is both, a simplification and an extension of the π-calculus. It is an extension, since the π-calculus is a proper subcalculus of the fusion calculus, but is also a simplification, because one of the two binding constructs is removed by separating the binding of variables from reception. Communication of a name simply amounts to state the equality of the names which are "sent" and the names which are "received", *i.e.*, execution of the process $\bar{x}y \mid xz$ enforces the equality of the names y and z. Thus

[21] The main difference between the two presentations of the ρ-calculus is that [NM95] distinguishes between logical conjunction on constraints, $\dot{\wedge}$, and (parallel) composition (of processes or "terms"), $\wedge$, whereas [NS94] does not.

[22] The γ-calculus (restricted to closed expressions) can be identified with the ρ-calculus over name equations and conjunctions $\rho(x{=}y)$ [NM95].

communication in the fusion calculus is symmetric and effects *all* processes in the scope of the names communicated. Besides the lazy λ-calculus, the fusion calculus allows to encode some basic instances[23] of the ρ-calculus [VP98] so that the fusion calculus can also be used as a calculus for ccp.

Another calculus containing the π-calculus as a subcalculus is the χ-calculus [Fu97]. This calculus is inspired from the view of communication (between processes) as cut elimination (in a proof). When removing type information from proof nets of linear logic [Gir87], the resulting graphs are considered as representations of process terms where communication corresponds to cut elimination. This view leads to the definition of a calculus where, as in the fusion calculus, a special scope operator controls the effect of a communication.

In the π-calculus (as well as in the related calculi presented so far), reception is limited to the use of a single channel. Extensions of the (asynchronous) π-calculus without this restriction are the join-calculus [FG96] and $\mathcal{L}_\pi$ [CM98].

Processes in the join-calculus [FG96] can be seen as communicating via a multiset of messages: sending a message corresponds to place it in the multiset, and the (blocking) "joint reception" of several messages waits until all messages are present in the multiset and then removes them in a single atomic step from the multiset. The distributed join-calculus [FGL$^+$96] extends the join-calculus in order to model mobile processes with *locations*. Intuitively, the locations of a system form a tree-like structure, where the leafs are groups of processes. Every location resides on a physical site and be moved (together with all its sublocations) to another site.

Several programming languages based on the join-calculus have been proposed, as for instance jocaml [FFMS01] or Funnel[24]. While the former is an integration of the join-calculus with the functional programming language `ocaml` [LDG$^+$01], the latter is a prototype implementation of a programming language based on Functional Nets [Ode00], a programming framework based on the join-calculus.

In $\mathcal{L}_\pi$ [CM98] processes communicate, as in the join-calculus, via a multiset. Additionally, a process may have a guard, *i.e.*, a $\mathcal{L}_\pi$-process that, when executed in an encapsulated environment, has to terminate successfully before the process can proceed. Since executions of $\mathcal{L}_\pi$-processes can be seen as deductions in linear logic, $\mathcal{L}_\pi$ aims at providing a uniform declarative formalism capturing transformational and concurrent programming paradigms that is based on linear logic.

As we have already pointed out, some linear logic programming languages, as for example Forum, can also simulate the execution of processes as in the π-calculus (see section 2.1.4), since the proof theoretic behaviour of the connectives in linear logic is rather similar to the operators of process calculi.

2.3 Coordination

When modeling complex systems, different (asynchronous, communicating, heterogenous) activities require to be *coordinated*. Coordination languages and models aim at

[23] Namely $\rho(x = y)$, which uses constraints over name equation and conjunction, $\rho(x = y, \mathcal{C})$, which adds constants and $\rho(x = y, x \neq y)$, which adds inequalities.

[24] The current release of Funnel can be obtained from the URL `http://lampwww.epfl.ch/funnel`.

"providing a means of integrating a number of possibly heterogenous components together by interfacing with each component in such a way that the collective set forms a single application that can execute and take advantage of parallel and distributed systems" [PA98a]. Hence a *complete* programming language is composed of a computation language and a coordination language [GC92].

Among the different models of coordination, coordination languages based on a shared dataspace model are the most related to our computation model. This model of coordination was to our knowledge first introduced in the HEARSAY-II speech understanding system [EHRLR80], where several processes (called knowledge sources) communicate by means of a *blackboard* which records the different hypotheses generated by the knowledge sources. Thus all hypotheses can be seen and modified by all knowledge sources in the system. Coordination languages based on this principle are well suited for combination with imperative languages [GC92, PA98b]. In the following, we present some of these models and languages we are aware of and then present some combinations such coordination models with declarative programming.

2.3.1 Coordination Languages

The first genuine coordination language, Linda, is based on a shared dataspace model [Gel85, ACG86, CG89]. Processes communicate via a common tuple space by means of essentially three operations, namely out, in and read. $\mathsf{out}(t)$ puts the tuple t into the tuple space. $\mathsf{in}(t)$ waits until the tuple space contains a tuple t' that matches t, binds the free variables in t to the matching values of t' and removes t' from the tuple space. $\mathsf{read}(t)$ behaves identical to $\mathsf{in}(t)$ except that the matching tuple t' remains in the tuple space. This communication model is also called "generative communication" [Gel85], since communication involves the generation of a tuple which has an existence independent of the process which created it (until it is explicitly withdrawn from the tuple space).

Some presentations of Linda, *e.g.*, [ACG86, CG89], also include an additional fourth operation, namely $\mathsf{eval}(t)$, which – similar to out – puts the tuple t into the tuple space, but additionally starts a process evaluating the tuple t. Therefore such a tuple t is also called an *active* (or "live") tuple. When the evaluation of an active tuple has finished, it turns into an ordinary ("dead") tuple. In this extended model, since the distinction between processes and tuples vanishes, new semantical problems arise, as for instance the removal of an active tuple, *i.e.*, the removal of a running process. A common remedy is to (implicitly) allow only the removal of dead tuples [ACG86, CG89, NFP98].

Several extensions of the Linda-coordination communication model have been proposed. In this section we present some propositions for multiple tuple spaces, as well as other models for coordination. In the subsequent section we present models with more "expressive" tuple spaces, in the sense that the latter are considered as logical theories, leading to a combination of declarative programming and coordination.

The Kernel Language of Agents Interaction and Mobility (KLAIM) [NFP98] allows the description of mobile processes that coordinate themselves via multiple tuple spaces. In KLAIM, a system (also called a *net*) is modeled as a set of *nodes* which are *located* on different (physical) sites or localities. Roughly speaking, a node or location corresponds to a set of processes together with a mapping associating location variables to concrete

(physical) sites. Processes are defined in the style of a process algebra with *located* Linda-operations as basic actions. For instance, the operation $\mathbf{out}(t)@\ell$ puts the tuple t in the tuple space at the location ℓ. An additional action, namely $\mathbf{newloc}(u)$ allows the creation of new (virtual) locations which are attributed to a physical location by the runtime environment. Parameterised process identifiers allow the definition of guarded recursive processes, where the set of legal parameters include processes, locations and tuples (which on their turn are allowed to contain processes and locations).

Another extension of Linda with multiple, named tuple spaces is PoliS [Cia94]. In PoliS agents can only read the space they are contained in, but they can write into any space they know of. While agents cannot move from one space to another, they are able to create both, tuples and spaces. When a tuple is put into a space before this space has been created, the tuple is kept in the "meta tuple space" until the destination space comes into existence.

Linda 3 [Gel89] extends Linda with hierarchically ordered multiple tuple spaces. In Linda 3 tuple spaces are considered as elements of a new type `ts` together with a special operation `tsc` (which has to be called inside an `eval` operation) for the creation of new tuple spaces. Tuples containing entire tuple spaces, as well as active tuples, are considered in the same way as standard tuples. Thus, contrary to standard Linda, the removal of processes is possible in Linda 3. Bauhaus [CGZ94] adds a special operation or command, namely `move`, which allows a process to move tuples from one tuple space to another. Since processes are seen as "active" tuples, mobile processes can be expressed in Bauhaus by moving the tuple corresponding to the process executing the operation (which is denoted by a special symbol in Bauhaus).

In Law-Governed Linda (LGL) [ML95] the programmer specifies a *law*. When a process executes one of the Linda coordination primitives, a corresponding event is generated. For each of these events, the law specifies a sequence of so-called primitive operations to be executed. Thus, a law allows to enforce the use of specific coordination protocol.

Another approach to coordination is Gamma [BM96], where a program is described in terms of multiset transformations. Since there is no artificial sequential ordering of these transformations, they can be executed sequentially or in parallel, whenever their parallel application does not make multiple use of a same element, *i.e.*, if the different transformation modify different parts of the multiset. As in Linda, communication (and coordination) between processes is based on a shared data-space: in Gamma all processes transform the same multiset. Nevertheless, the notion of processes is implicit in Gamma, *i.e.*, a programmer specifies the tasks without the need for sequentialising them, but concurrent activities are not explicitly specified as such.

The Linda coordination model has also been formalised in form of a process algebra [NP96, BGZ97, BGZ98] and the Turing-completeness of a language based on the Linda-coordination primitives has been proved [BGZ97]. [BJ98] establishes a hierarchy of Linda-like languages[25] with respect to the set of available operations, in order to analyse which of the Linda-operations really add expressiveness, *i.e.*, which could not be encoded by means of the others. This hierarchy is extended in [BJ99] to a comparison of three different coordination models, namely languages based on the Linda

[25] In [BJ98] the primitives are called similar to the actions of ccp, for instance *ask* (instead of read), *tell* (instead of out), *etc.*

primitives, languages based on multiset rewriting (as for example Gamma or Bauhaus) and languages using communication transactions, *i.e.*, languages which allow to group several modifications (of the tuple space) into a single atomic transaction (as for example SP). Roughly summarised the results of the comparison imply that both, multiset rewriting and transactions, are more expressive than simple Linda-operations, and that transactions are more expressive for the full set of Linda-operations, *i.e.*, *ask*, *nask*, *get* and *tell*.

MANIFOLD [BAdB+00] encourages a discipline of programming where computation is clearly separated from coordination (or communication). Computation in **MANIFOLD** is expressed by means of atomic or computation processes[26] which read values from input streams (the process is blocked until a value is available) and write their results to output streams. Coordinator processes control the overall structure of the system by means of special actions, as for instance the creation and destruction of streams and processes. As Linda, **MANIFOLD** allows the computation processes to be written in different languages, under the assumption that the set of atomic values that are passed through the streams are understood by all languages used in the description of the system. Besides message passing using streams, processes can also communicate by broadcasting events to all (coordinator) processes. The semantics of a system is defined by means of two levels. In a first level, the behaviour of every (coordinator or computation) process is defined by means of a transition system. In a second level, the transition system corresponding to the processes of a system are combined into a second, global transition system defining the semantics of a system.

2.3.2 Coordination and Declarative Programming

Several combinations of logic programming with a coordination model based on a shared dataspace have been suggested [BC91, Cia94, ODN95, dBJ96, DNO97, DO99]. These proposals extend Linda by interpreting the tuple space as a logical theory, and by interpreting the operations on the tuple space in a similar way to, for instance, the "extra-logical operators" `assert` and `retract` of Prolog [DEDC96] (see section 2.1.1). However, the shared dataspace is still restricted to (logic) tuples, *i.e.*, unitary clauses or atoms.

(Extended) Shared Prolog (SP, ESP) [BC91, Cia94] uses the notion of a *logic tuple space*, *i.e.*, the tuples are seen as atoms, and unification replaces the pattern matching when accessing information in the shared dataspace. The logic tuple space is used to coordinate several processes, called *theories*. Each process (or theory) is composed of several rules. These rules have mainly three parts: a precondition testing for the presence of some tuples in the tuple space (possibly deleting some tuples), calls to predicates (locally) defined in the theory and a multiset of tuples to be added to the tuple space whenever the call to the predicates, *i.e.*, the solving of the goal, terminates successfully. Since ESP is based on PoliS (see section 2.3.1), it provides also for multiple, named logic tuple spaces.

A similar approach is followed by $\mathcal{A}$gents $\mathcal{C}$ommunicating through $\mathcal{L}$ogic $\mathcal{T}$heories ($\mathcal{ACLT}$) [ODN95], where the tuple space of Linda is replaced by a theory description

[26] In analogy to object oriented programming, definitions of processes are also called *classes* in [BAdB+00].

restricted to atoms (*i.e.*, unitary clauses), which is also called a knowledge base and which can be used for proving logical consequence. These advantages can only be exploited by agents written in a logic language (also called "logic agents" [DNOV96]), whereas agents written in *e.g.*, C-Linda just see a classical tuple space.

An extension of $\mathcal{ACLT}$ with atomic reactions triggered by communication events are Tuple Centres [DNO97, DO99]. These reactions are defined using a new "simple specification language" for the reaction rules which is shown to be Turing complete [DNO98]. A key property is that a reaction can trigger itself other reactions, which are executed before any new communication event (from another agent) is taken into account. Therefore, reactions allow to transform the tuple space into a higher level communication medium, which by itself can incorporate evolved communication protocols. Since the reactions of Tuple Centres are encoded via *special tuples* in the tuple space, agents may modify the behaviour of the communication medium.

Another interesting extension of logic programming with (several) blackboards is μLog [JdB94, dBJ96][27], where Linda-like primitives are added to a logic programming language. These primitives allow the addition (respectively, lecture or removal) of terms, blackboards and processes. In μLog the definitions of, on the one hand, predicates (conditions) and, on the other hand, processes are distinguished: the clauses defining processes are allowed to contain blackboard primitives, whereas predicates have to be defined by pure Horn clauses. However, μLog makes no distinction between goal solving and execution of a process. Additionally, μLog distinguishes between foreground and background processes. Foreground processes have to terminate (for a successful computation), whereas the background processes are (implicitly) killed on termination of the foreground processes. Also, the initial goals (or processes) in μLog are not allowed to share variables, to ensure that the communication between processes (or goals) uses only the blackboards.

The Coordination Language Facility (CLF) [AFP96, APPPr98] combines message oriented coordination using Linda-like primitives with transactions. In CLF, coordination of several *participants* uses special processes called *coordinators*, the behaviour of which is defined by means of *scripts*, *i.e.*, collections of rules describing the removal and insertion of a number of resources. For this reason the programming paradigm underlying CLF has also been called resource-based programming [And01]. Informally, a rule $l \diamond\!\!- r$ of CLF defines two sets of resources: the resources in r are to be inserted to the participants after the successful atomic removal of all resources in l from the participants. Thus the rules of CLF are a fragment of the linear logic programming language LinLog (see section 2.1.4), interpreting $\diamond\!\!-$ as linear implication $\circ\!\!-$.

2.4 (Executable) Specifications Techniques

Formal specifications provide precise descriptions of a system, allowing to check properties early in the development process and thus to reduce the cost of modifications. Whenever a formal specification is executable, the specification is even a first prototype of the system which facilitates communication between the developer(s) and the user(s) of the system.

[27] μLog is the successor of Multi-Prolog [dBJ93].

2.4.1 Algebraic Specifications

The notion of *abstract data types* plays a central role in classical algebraic[28] specifications, see for instance [EM85, EM90, Wir90, AKKB99] or the common algebraic specification language (CASL) [BB01, CAS01] as an example of a language. These types are called abstract, since the actual representation of the data is hidden, and in consequence they can only be manipulated using, in an applicative style, the operations provided with the abstract data type. This implies that there is, as in pure functional programming, no notion of state or side-effects. Consequently, when modeling dynamic systems and their behaviour, the state has to be explicitly coded, and will appear as a parameter and a result everywhere in the specification, rendering the specification more complicated and less "natural".

To avoid these problems, [DG94, Kho96] suggest the introduction of the notion of *implicit* state, leading to the framework called AS-IS, the acronym of Algebraic Specification with Implicit State. States are represented in AS-IS implicitly by means of an additional algebraic specification of *(elementary) access functions*. Every access function f (of profile $s_1 \times \cdots \times s_n \to s$) can be modified by means of an associated (elementary) *modifier* μ-f, *i.e.*, a function of domain $s_1 \times \cdots \times s_n \times s$. The intended semantics of an elementary update μ-$f(t_1, \ldots, t_n, t)$ is to change the definition of the access function f at all points of the domain matching the pattern $t_1, \ldots, t_n$ to the term t which may contain variables (that are used in the pattern). More complex modifiers can be obtained by the addition of guards, and sequential, and two forms of parallel composition [Kho96].

[AZ92, AZ95, Zuc99] suggest d-oids as structures describing the evolution of dynamic systems at the level of the models, *i.e.*, by transition systems the states of which are (instant) algebras. All the instant algebras (or states) of a system are required to share is the set of sorts. Thus the signature, *i.e.*, the set of operations defined in an instant algebra, is allowed to vary from one state to another. Associated with each transformation is a (partial) function relating elements of the domains before and after the transformation. These functions are called *tracking maps*, since they allow to keep track of object identities. While d-oids provide a theoretical foundation and model for dynamic systems, they currently do not include concurrent modifications and are thus not well suited as an actual programming language for concurrent systems.

Dynamic abstract data types (DADT) [EO94] are an (informal) proposal aiming at the generalisation of the already presented extensions of algebraic specifications for dynamic systems. Roughly speaking, a DADT consists of an ADT together with a collection of dynamic operations defining transformations between instances of the ADT. The specification of a DADT is separated into four different levels. The lower two levels correspond are similar to AS-IS : the first level (Value Type) defines the algebraic data types, and the second (Instant Structure) generalises the definitions of access functions of AS-IS. The two higher levels describe the dynamic evolution of the ADTs defined by the first two levels, *i.e.*, the third level corresponds, roughly speaking, to the elementary modifiers of AS-IS, and the fourth level is used for higher order combination of DADTs. The basis of the semantics of a DADT is a transition system the states of which are instant structures. As d-oids and AS-IS, DADTs do not

[28] In mathematics, "algebra" denotes a structure of a set together with operations on the set.

provide the notion of processes directly.

2.4.2 Abstract State Machines

(Gurevich) Abstract State Machines (ASMs) [Gur85, Gur91, Gur95, Gur97, Gur00] aim at providing a theoretical model that allows the expression of algorithms at the level of abstraction that is "natural" for the algorithm, avoiding the need to encode the algorithm in the language of a lower level machine. Thus ASMs are thought to be a general model for algorithms, in the sense that *"for every (sequential) algorithm A, there exists an equivalent (sequential) abstract state machine B"* [Gur00, theorem 6.13], in the same way as Turing Machines [Tur36] are a general model for computable functions.

Supporting this thesis of ASMs [Gur85, Gur00], the formalism of ASMs has been used successfully for the description of the semantics of different programming languages, such as for instance BABEL, C, C++, , COBOL, JAVA, Occam, PARLOG, Prolog, SDL, SML and VHDL[29]. Montages [KP97] are an extension of ASM dedicated to the specification of dynamic and static semantics of programming languages.

An ASM is characterised by its (finite) *vocabulary*, *i.e.*, a set of function symbols, Υ and a set of update rules $\mathcal{R}$. The functions in the vocabulary Υ are partitioned into *static*, *i.e.*, immutable, and *dynamic*, *i.e.*, mutable, functions. The states of an ASM are *algebras*[30], *i.e.*, models of first-order theories (where predicates are represented as boolean functions). An algebra (or state) A consists of a pair $A = \langle X, I \rangle$ of a *base set* (or *super-universe*) X and an interpretation I of the functions of the vocabulary Υ (as functions on X). Different types or sorts of elements can be accommodated by means of a partition of X into universes corresponding to the different types, or by extension of the formalism of ASMs to many-sorted algebraic specifications [Zam98]. A basic update rule $f(s_1, \ldots, s_n) := t$ modifies the value of the (dynamic) function f at the element $(s_1, \ldots, s_n)$ (in the domain of f) to the value of the term t[31] More complex updates can be obtained by a (finite) parallel composition of basic updates, the meaning of which is the atomic update of a several functions at the same time. An extension to an update of possibly infinite locations in a single step is provided by means of so-called parallel ASMs [Gur95, section 5]. Roughly speaking, parallel ASMs allow rules to be parameterised by variables the range of which are universes ; application (or firing) of a rule corresponds to fire the rule for all possible instances for the variable. Update rules can also be *guarded*, *i.e.*, the execution of the update depends on the validity of a boolean expression. The import (or creation) of new elements, *i.e.*, members, of a given universe, is achieved by means of a special universe, the so called *reserve*. Notice, that the vocabulary, as well as the base set, is not modified by the import of an element [Gur00, sections 4.4 and 4.5].

To cope with interactive algorithms, [Gur00, section 8] introduces the notion of an *environment* which can modify the state of the ASM (after each step of the algorithm). These modifications of the state by the environment can thus account for inputs to

[29] Refer to the URL `http://www.eecs.umich.edu/gasm/proglang.html` for a complete list of available ASM-specifications of programming languages.

[30] This is the reason why ASMs were initially called *evolving algebras* [Gur95].

[31] The difference of these updates with the modifiers of [DG94, Kho96] (see section 2.4.1) is that the former modify the *model* whereas the latter modify the specification.

the algorithm. On the other hand, output is "taken miraculously away" (by the environment) [Gur00] so that output does not need any further consideration. In order to restrict the possible modifications of the environment, [Gur00, section 8.4] suggests to annotate the (dynamic) functions (*i.e.*, those that can be changed during a computation) in a vocabulary as either *internal* (*i.e.*, modifiable only by the algorithm), *external* (*i.e.*, modifiable only by the environment) or *shared* (*i.e.*, modifiable by both, the algorithm and the environment).

Communicating Evolving Algebras [GR93b] aim to provide a theory of (true) concurrent computation in the framework of ASMs. A central notion is the *independence* of update rules. Two rules are considered independent when it is impossible to derive the sequencing in time from their joint effect[32]. Using the notion of independence, the definition of the parallel composition of ASMs can be defined. Communication between concurrent ASMs is based on the modification of the common part of the state. This model has been used for an encoding of the CHAM and the π-calculus [GR93b, section 4].

Distributed computation is simulated by considering several agents that share a common state of which every agent has a "personalised" view [Gur95, section 6]. Agents can be grouped together into teams which themselves are agents. Thus synchronisation of agents is achieved by specifying an action for the team as a whole, since the action cannot be performed by one of the agents separately. For instance, consider an agent A that wants to send a value v to agent B, *i.e.*, it has to modify the definition of a function f at a location t. Since v is known only by A and $f(t)$ only by B, the communication has to be described by another agent, namely the team formed from A and B. A more explicit specification for synchronisation between different agents is suggested in [Sch98], where rules can be labeled. Informally, all rules that have the same label are expected to be executed jointly, *i.e.*, in a single atomic step.

Interactive Abstract State Machines (IASMs) [dAMdIdSB98] are an extension of (distributed) ASMs with message passing as mechanism for communication between so-called *units*, *i.e.*, processes or components of a system. Besides the update rules as in classical ASMs, IASMs provide interactions which specify the communication behaviour of the unit. Informally, interactions are an extension of the external functions of ASMs [Gur95, section 3.3.2], which correspond to functions that are partially specified inside the system and controlled or changed from the outside. Using external functions, the reception of values can be modeled easily, and IASMs extend the concept to more complex interactions, keeping the modeling of the environment by partial specifications. Furthermore, the communication structure between the different units of an IASM can vary, and units can be created dynamically. The semantics of an IASM is defined by means of a translation into an ASM [dAMdSB98].

A formalism combining ASMs with algebraic specification with implicit state (AS-IS, see section 2.4.1) has been suggested in [GKZ99]. With respect to ASMs, the combined formalism allows the use of algebraic data types for the formal specification of the semantics of static functions, *i.e.*, functions which are never modified during the execution of the system. On the other hand, the combined formalism adds to AS-IS an additional layer which is used for the formal specification of the dynamic evolution

[32]Notice that *independence* is a coarser relation than *consistency*. For instance, while the updates $a := b$ and $b := a$ are consistent, they are not independent.

of systems. Thus, [GKZ99] suggests the specification of a dynamic system in three levels, where the first two correspond to AS-IS, that is to say the first level defines the data types which are static, *i.e.*, unmodified, during the execution of the system, and the functions which may change during the execution of the system (*i.e.*, the so-called elementary access functions) are defined in the second level. Finally, the third level defines update expressions, which correspond roughly speaking to an extension of the combinators of the elementary updates which we have already presented along with AS-IS [Kho96].

Algebraic state machines [BW00] are state machines or interactive state transition systems which are completely defined by algebraic and logic means. As in AS-IS, the states of algebraic state machines are algebras, specified by an algebraic specification. In algebraic state machines, transitions between states are described by particular *transition axioms*, also called *transition rules*. Informally, a transition rule is defined as a four-tuple. Two parts of a transition rule are logical formulæ defining pre- and post-conditions on the states between which the transition takes place. The other two parts of a transition rule define the inputs consumed and the outputs produced by the transition. Thus algebraic state machines can be considered as "black boxes", data-flow nodes or *components* which communicate with their environment by means of input and output channels. Using techniques similar to those of [Bro98], these components can be composed to more complex systems.

2.5 Multiparadigm Programming

> "Have you ever gotten into a do-it-yourself construction project, and become stuck because you were missing a particular tool? You had three choices: (1) go out and buy (or borrow) the right tool, (2) use the wrong tool as best as you could, or (3) give up. [...] When confronted by this problem in programming, you are limited to choices 2 and 3: program around the problem using the constructs provided by your language or give up." [Hai86]

To solve the problem mentioned above, multiparadigm programming has been suggested. The goal of multiparadigm programming is to allow a programmer to write the implementation of a system in different paradigms, such that for every part of the system the most appropriate paradigm (or language) can be used. Thus a program written in a multiparadigm language should be easier to write, understand and maintain. Etymologically, the word "paradigm" (from the word $\pi\alpha\rho\acute{\alpha}\delta\varepsilon\iota\gamma\mu\alpha$) means example[33], and is commonly used to refer to a category of entities sharing a common characteristic. Further definitions of the word (and its meaning in computer science) can be found for example in [Hai86], [Spi94, section 2.1], [Bud95] or [Cop98].

[33] More precisely: "**1.** One that serves as a pattern or model. **2.** A set or list of all the inflectional forms of a word or of one of its grammatical categories: *the paradigm of an irregular verb.* **3.** A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline." [The American Heritage Dictionary of the English Language, Fourth Edition, 2000, `http://www.bartleby.com/61/73/P0057300.html`]

The idea of multiparadigm methods is obviously not restricted to software engineering, and the benefits of similar approaches are also exploited in other domains. For instance, in the design of electronic systems the use of several languages in a single specification allows to (drastically) reduce the "time to market", since every part of a system can be expressed in the most appropriate language [Cos01].

In the following, we present some of the numerous proposals for multiparadigm frameworks or languages. More exhaustive surveys or enumerations of different approaches and research directions to multiparadigm programming languages and environments can be found in [Mul86] and [Spi94]. Most of them are oriented towards an implementation and focus less on the investigation of the semantic foundations of the suggested framework. We conclude the section with a presentation of some component based approaches we are aware of.

The approach to multiparadigm programming suggested in [SDE95, Spi94] models programming paradigms as object classes: a paradigm corresponds to a class and a module (of the system) is an object (or instance) of the class. This allows to organise the different paradigms in a tree-structured hierarchy using inheritance, where the "toplevel class" corresponds to the target architecture. Subclasses (*i.e.*, "subparadigms") are implemented using (as primitives) only the methods of its direct superclass(es). This layering allows to reconcile the different operational semantics of the paradigms, since services (this could be, dependent on the paradigm, procedures, clauses, functions, rules, ports, *etc.*) of another paradigm are accessible through so called "call gates" following the hierarchical ordering of the paradigms. An advantage of this approach is its extensibility: to add a new programming paradigm, it is sufficient to integrate a corresponding class in the existing hierarchy such that only the interface to its immediate neighbours (parents, children and siblings) need to be implemented. Furthermore, the implementation of *generators* of multiparadigm environments is possible [SDE95].

Nondeterminism is at the root of the approach to system composition using programs written in different paradigms presented [Zav89]. These programs are necessarily incomplete, since they lack those features of the system that are implemented in the other parts. Representing this incompleteness as nondeterminism, composition allows external programs to influence the behaviour without disturbing the semantics of the programming language, so that the part can still be analysed and validated as usual (for a program in that particular language). Hence, in the composed system, the different parts influence each other by controlling their nondeterministic choices mutually. The example of an prototype telephone network presented in [Zav89] the central part is written in the multiparadigm programming language PAISLey [Zav91]. Informally, a PAISLey-program is a set of function definitions that can be described by a hierarchical dataflow diagram. Synchronisation of different concurrent computations is achieved by structuring the dataflow diagram into looping processes which enforce the synchronous update of their state, *i.e.*, the synchronous evaluation of so-called exchange functions, which are similar to the communication gates of LOTOS or the channels of process calculi as CSP.

[ZJ96] tackles the problem of multiparadigm specification by augmenting the specification language Z with automata and grammars. In this approach a system is specified by means of several partial specifications that are all translated into first-order logic and then linked together. However, the method is limited to the specification of a

rather restricted class of systems, namely those that react to a sequence of atomic input events by the execution of a state changing operation (before the next input event is handled).

LIFE[34] [AKP93] uses ψ-terms for the representation of data-structures. Roughly speaking, a ψ-term corresponds to a (labeled) directed graph the nodes of which are sorted, and the labels of the arcs are called *features*. Thus ψ-terms conveniently represent record-like data structures. Furthermore, the subtyping and type intersection rules for ψ-terms account for some of the (multiple) inheritance convenience known from object-oriented languages. Higher-order functions are defined in LIFE as usual by means of rewrite rules. By extending unification to ψ-terms, LIFE encompasses also the paradigm of logic programming.

Considering objects as constants, $\mathcal{OLI}$ [LP96, LP97] integrates logic and object-oriented programming conservatively so that pure logic or object-oriented programming remains possible. $\mathcal{OLI}$ introduces the notion of "o-terms", *i.e.*, (calls to) objects without state-changing methods. This ensures that the evaluation of non-primitive o-terms, *i.e.*, o-terms containing unevaluated calls to other objects, does not modify the static theory described by the program. Thus the logic part of $\mathcal{OLI}$ integrates the evaluation of calls to objects similar to the reduction of expressions in functional-logic programs. The logic part is introduced into an object oriented language similar to SMALLTALK by means of a special built in object, namely `$logicBase`, which acts like an interpreter for a logic programming language, *i.e.*, goals can be solved using a method `query:` and the theory can be modified as in Prolog [DEDC96] by means of the methods `assert:` and `retract:`. While a semantics for the extension of logic programming with o-terms is given in [LP96, LP97], the overall (object-oriented) framework of $\mathcal{OLI}$ lacks a clearly defined formal semantics.

The integration of object oriented and logic programming proposed in [Con88] distinguishes between two kinds of predicates, namely *object* and *procedure* names. The procedures correspond to the classical predicates as in Prolog and they are defined by classical Horn clauses. Objects are defined by *object clauses*, *i.e.*, an extension of Horn clauses with *two* (positive) literals in the head, such that one is a procedure and the other is an object. The operational semantics is similar to the one of Prolog (see [DEDC96, chapter 4]), with the additional conditions that all procedure-literals are on the left of all object literals, and that the selection of literals proceeds from left to right. In the case of a procedure literal matching (a part of the head of) an object clause, a matching object literal has to be searched in the remainder of the goal.

Distributed Logic Objects (DLO) [CLSM96] extends the framework of [Con88] by using arbitrary number of positive literals in the heads of clauses. Furthermore, DLO is a committed choice language, *i.e.*, once a rule is selected, this choice is not undone by backtrack in the future. Computation in DLO is inherently concurrent, since the literals in the body of a clause are solved in parallel. A distributed implementation of DLO is described in [CLSM96].

I^+ [NL95] integrates the paradigms of object-oriented, functional, logic and parallel programming. Informally, I^+ distinguishes between two kinds of objects, namely functional and logic objects, where the methods of the former are interpreted as func-

[34] LIFE is the acronym of Logic, Inheritance, Functions and Equations.

tions, and those of the latter as predicates. Functions are defined by equations using pattern matching, and predicates by (augmented[35]) Horn clauses. Parallelism in I^+ corresponds to the concurrent execution of the different "active" objects of a program. In order to allow a programmer a better control of the parallel execution of the program, I^+ provides an asynchronous mode for method invocation besides the classical synchronous message passing scheme corresponding to a remote procedure call, where the caller is suspended until the callee returns an answer. In the asynchronous mode, the result of the call can be collected later on by means of a special, blocking operation. We are not aware of a semantic foundation of I^+.

Alma-0 [AS97, ABPS98] integrates the paradigms of logic and imperative programming, by extending a subset of Modula-2 [Mod96] in several aspects. First, boolean expressions can be used as statements and vice versa, and the evaluation of a boolean expression to `FALSE` corresponds to a failure, when used as a statement. Second, choice points can be set, such that backtracking upon failure is possible. Third, simulating unification, the equality predicate is generalised to assignment when one of the terms is an uninstantiated variable and the other a term the value of which is known. Finally, a new parameter passing scheme for procedures allows to pass variables by name, and expressions by value. An operational (respectively, denotational) semantics for the assignment-free subset of Alma-0 has been presented in [AB99] (respectively, [Apt00]), based on the interpretation of conjunctions in the clauses of a logic program as sequential composition[36]. In this view, formulas are seen as programs the execution of which corresponds to a proof search for the formula. However, we are not aware of a concurrent extension of Alma-0.

The multiparadigm language Leda [Bud95] provides, compared to Alma-0, the possibility to use the paradigms of functional and object-oriented programming, but without investigating the semantic foundations of this extension. As in I^+, the overall structure of a Leda-program is object-oriented, and the other paradigms are integrated for the implementation of the methods. However, Leda does not provide concurrency.

The multiparadigm language G [Pla91] claims to include the logic, functional, imperative, relational and object-oriented programming paradigms. As fundamental data structure, G provides *streams*, *i.e.*, possibly infinite sequences, which can be composed in an applicative, functional style. Additional operators allow to repeat and filter streams. In G, the different programming paradigms are not considered as a whole but "unbundled" into different characteristics that are then integrated into G. This has the drawback that some paradigms are included only partially. For example, the integration of the logic paradigm uses a technique based on filters instead of unification in the selection of the rules to execute.

Maude [CDE$^+$99] is a multiparadigm programming language based on rewriting logic [MOM01], the basic axioms of which are rewrite rules. There are two complementary views of a rewrite rule $t \to t'$ in rewriting logic. On the one hand, $t \to t'$ can be seen as a description of a local transition from a state a fragment of which matches the pattern t to a new state where an instance of t has been replaced by an instance of t'. On the other hand, $t \to t'$ can be interpreted as a logical inference rule allowing the inference of formulæ of the form t' from formulæ of the form t. A program

[35] They are called augmented, since terms are allowed to contain calls to methods of objects.

[36] Notice that conjunction is commutative, while sequential execution is not.

in Maude is a collection of modules, each of which corresponds to a rewriting theory. The implementation of Maude makes extensive use of the fact that rewriting logic is *reflective*, *i.e.*, that there exists a universal theory U in which any finite rewrite theory can be represented (including U itself). Reflection is supported in Maude by means of the system module `META-LEVEL` [CDE+98, CDE+99]. For instance, Maude allows the specification of different rewrite strategies on meta-level in Maude itself. Thus, different programming paradigms, as functional, logic, object-oriented and concurrent programming, can be specified and used in Maude [Mes92].

The programming language C++ [Str85] is also used for multiparadigm programming and design [Cop98], where the notion of paradigm is slightly different from the work mentioned up to now, since the considered paradigms are for instance the following: classes, overloaded functions, templates, modules and ordinary procedural programming [Cop98, page xiii]. That the declarative paradigms, *e.g.*, functional and/or logic programming, are missing, emphasises that, as for other imperative programming languages, C++ is a rather low-level language, lacking a well-defined semantics and the associated advantages.

Last, but not least, the UNIX operating system [RT78] is also a multiparadigm environment, since an application in UNIX can be constructed from different programs, written in any language for which an interpreter or compiler exists. These programs are supposed to read a sequence of characters from "standard input" and to write a sequence of characters to "standard output", so that they can be composed using buffered one-way communication channels, so-called "pipes". This model is widely used, although it is rather low-level, restricted and not provided with a clear semantics.

The composition of a system from a set of existing components is a particular kind of multiparadigm framework, since the different components might be written in different languages. Besides the possibility to use always the most appropriate language, so-called component based approaches, such as for instance the (Distributed) Component Object Model ((D)COM) [COM95] or JavaBeans [Eng97] are motivated by the possibility of increasing software reuse. In a component based approach, a (software) system is constructed by assembling several ready-to-use components which are considered as black boxes, similar to the construction of system in other engineering disciplines. Thus, research on components focuses mainly on the assembly of components, *i.e.*, their interfaces, leaving unspecified their internal structure, since it is not important for a *user* of a component [Szy98]. This implies that, in order to build any system, a set of predefined or built-in components has to be provided, for instance in form of a component library.

In the current component models mentioned above, components are provided in binary form, *i.e.*, ready for execution. Since the languages which have been used for the production of these components do not matter, such component approaches are inherently multiparadigm, although with a rather low-level and restrictive integration, similar to operating systems as UNIX.

The only formal *definition* of a component we are aware of is given in [Bro98], where a component is characterised by a function mapping input streams to output streams. Hence this definition also considers merely the input/output behaviour and neglects the internal structure of a component.

The *mixed language program* system (MLP system) [HS87] aims at facilitating the

development of distributed systems by allowing different components to be written in different programming languages[37]. The different components of such a system communicate using a common language, namely the *Universal Type System* (UTS) language which allows the definition of data types. Thus every language that is to be used in the MLP system has to provide the translations of its data-structures into the corresponding types of the UTS. While such translation have been defined for imperative programming languages as C or Pascal, we are not aware of an integration of declarative languages in the MLP system.

——— ⋆ ——— ⋆ ——— ⋆ ———

We have seen that among the many existing proposals for a convenient programming computation model combining different programming paradigms, none exactly meets our requirements, namely a clear separation of concerns allowing the use of the most appropriate programming style for every part, while keeping the well-defined semantics of a declarative language.

In the remaining chapters of this thesis we present our proposal for a combination of (mobile) processes and declarative languages in a component based approach. A main design pricniple of our computation model is to distinguish clearly between, on the one hand, concepts which are definable in classical declarative languages, such as functions, predicates or constraints and action or (mobile) processes on the other hand. Furthermore, we present the definition and semantics of a component.

[37] "A *mixed language program* is a program written in two or more programming languages. Such programs consist of several *program components*, where each program component is composed of one or more procedure written in the same language." [HS87, beginning of section II]

Chapter 3

A Computation Model for Concurrent Declarative Programming

Contents of the Chapter

In this chapter we present an component-based approach to concurrent declarative programming [ES00, ES01a]. We suggest to model a system as a set of *components*, where each component is internally composed of a *store* F, *i.e.*, a declarative program, and a set of *processes* p_i, interacting via the modification of the store by means of the execution of *actions* that can be defined by the programmer [ES01b]. Interaction between components is based on the same scheme as interaction between processes of

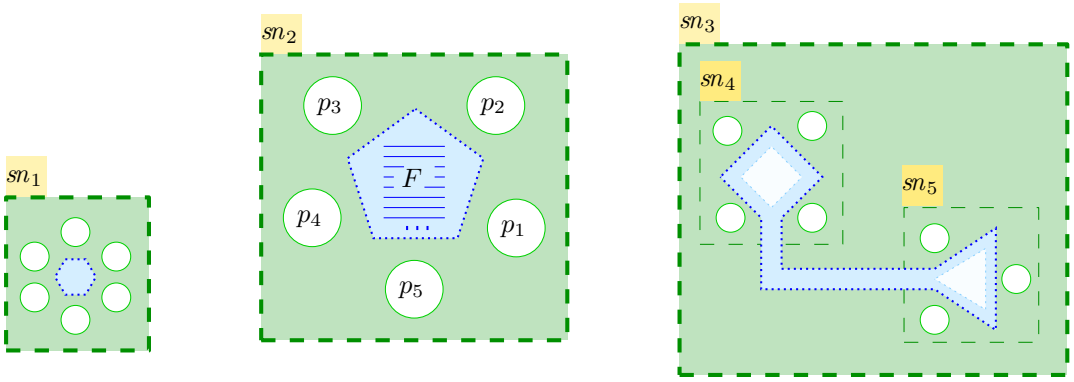

Figure 3.1: Execution Model of a System

the same component, namely the modification of the stores, *i.e.*, declarative programs. That is to say, processes are allowed to execute actions on all stores in the system. To take into account the difference between local and remote computation, we suggest to model interaction between components by a message-passing scheme, where each message corresponds to a sequence of elementary actions that should be executed on the receiving store.

Figure 3.1 shows a system of three components, named sn_1, sn_2 and sn_3, in execution. In component sn_2, five processes p_i ($i \in \{1; \ldots; 5\}$) execute on the store F. Notice that component sn_3 is the composition of two components, namely sn_4 and sn_5. By joining the stores of the subcomponents sn_4 and sn_5 in figure 3.1, we want to express that the separation in two subcomponents is invisible from the outside of component sn_3.

As we have already mentioned in the introduction, the description of a component can be stratified in several levels, a simplified view of which is shown in figure 3.2 (see definition 3.36 and figure 3.4 for the complete definition). The lowest level corresponds to the level of store, *i.e.*, classical declarative programs. At this level, a programmer declares the symbols of functions and predicates and specifies their meaning, for instance by means of conditional rewrite rules. Since the actions executed by processes modify the store, *i.e.*, the declarative programs, it seems natural to specify the actions on the meta-level with respect to the level of the stores. Finally, the description of processes requires both levels. On the one hand, processes execute actions, and need therefore the meta-level. On the other hand, processes depend on values of the store, namely in the guards of the guarded actions (see section 1.1.3.1). Notice that the implementation of our computation model manipulates all these different notions, and needs therefore a further level, which is not shown in figure 3.2.

To distinguish between the different components of a system, we attribute to each component of the system an identifier or name, called *component name* or *storename*. In the system of figure 3.1 the storenames of the three components shown are sn_1, sn_2 and sn_3[1]. If a process wants to execute an action on the store of another component, it necessarily needs to know the actions executable on the other store, as well as some

[1]We do not mention sn_4 and sn_5 since they are not visible from the outside of the component sn_3.

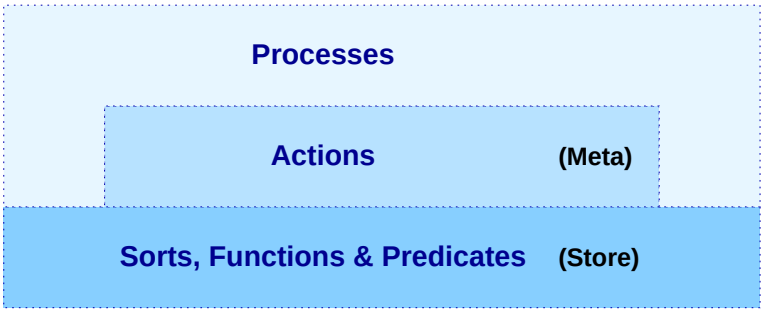

Figure 3.2: Basic Levels of a Component-Description

information about the signature of the store of the other component, in order to construct the parameters of the action in a meaningful way. Thus, a component needs to *import* (and conversely *export*) a subset of its actions as well as a subsignature of (the signature of) its store. In the case of components the stores of which are written using different declarative languages, interaction requires the *translation* of the values of one store into a representation meaningful for the other store.

⋆ ⋆ ⋆

The rest of this chapter is organised as follows. First we consider the level of the stores separately, that is to say we present the declarative language which we use for the description of the stores. Our computation model is not restricted to a particular declarative language, but can be applied to extend several declarative languages with concurrency. Thus, we present the general requirements on declarative languages which are to be used for the description of stores in our model, and illustrate these requirements with an *example* of a simple declarative language which we is used in the remainder of this thesis.

As already mentioned, the processes of a component modify the stores by the execution of *actions*. In a second step, we tackle thus in section 3.2 the definition of actions, which are defined as functions from stores to stores. We present the notion of a meta-signature and give some examples of definitions of actions for the simple declarative language presented in section 3.1.

The remaining sections of this chapter present the remaining, highest level of the description of a component as shown in figure 3.2, as well as the interaction between components. We give first in section 3.3 the definition of a component signature which defines all the symbols necessary for the description of a component, refining figure 3.2 (see figure 3.4).

Then we present in section 3.4 the necessary definitions for the description of the interaction between components. Actually, most of the symbols declared in the component signature that are used for the interaction between components, namely the *imported symbols*, are not defined in the component itself, but they supposed to be defined in another component. Furthermore, the *exported symbols* are defined in the component, but their definitions are covered by the sections corresponding to the stores

(section 3.1) and actions (section 3.2). However, we present the definition of *translations* which are needed for the interaction between components the stores of which are written in different declarative languages.

In section 3.5 we give the definition of *processes*. The basic processes in our model are, on the one hand, *guarded actions*, *i.e.*, pairs of a guard and a sequence of elementary actions, that are to be executed atomically, and, on the other hand, process calls. Besides the definition of the classical operators on process terms known from process algebras, we introduce expressions on processes and actions, in order to the description of processes in a style similar to the abstractions used in functional programming.

Finally, we present the definition of components and a brief presentation of the composition of components to both, components and systems in section 3.6.

3.1 Stores

The store of a component corresponds to a classical declarative program. Roughly speaking, a program can be seen as a theory description by means of a set of "formulæ". Thanks to this general view of stores, our approach to combine declarative programming and concurrency is generic in the sense that it is independent from the actual declarative language. Thus, (almost) any (pure) declarative language can be used for the description of a store. Additionally, this property allows us to build systems of several components with stores in different languages, where the store of each component might be written in the most appropriate language.

3.1.1 General Properties of Stores

A declarative program can be seen as a description of a theory, modeling a part of the "real world". In our computation model, this theory description is called a *store* and is shared by the processes of the component. The store of a component can be considered as a "knowledge base", modeling information about the state of the component and its environment, that exploited by the processes. Hence, the processes need to access the information contained in the store, as well as to modify the store. Access to the information uses the operational semantics of the declarative language the store is written in. The modification of the store is performed by the execution of actions and is presented in the following section dedicated to actions. In order to define the processes in a language independent way, we make the simplifying assumption that all stores are constructed from a signature and a set of rules. Thus we get the following definition.

3.1 Definition (store). *A* store *is a classical (declarative)* program $F = \langle \Sigma, \mathcal{R} \rangle$ *(written in the language $\mathcal{L}$), composed of a* signature Σ *and a set of* rules *(also called* phrases *or* formulæ*)* $\mathcal{R}$*. A* signature $\Sigma = \langle S, \Omega \rangle$ *is a pair of a set of* sorts S *and a ($\overline{S}$-indexed) family of* operator*,* function *or* predicate *symbols, such that* Σ *contains at least the sort* `Truth` *with its constructor* TRUE[2]*.* $\overline{S}$ *denotes the set of all sorts that*

[2]In Curry [HAK+00b] the corresponding sort and constructor are called `Success` and `success`. We have chosen `Truth`, since `success` is already used for the successfully terminating *process*.

can be constructed from the set of basic sorts specified by the set S[3]. *We note the* ($\overline{S}$*-indexed) family of sets of* terms *for a signature* Σ *and* variables X *as* $T^{\mathcal{L}}(\Sigma, X)$. *Furthermore, we have a decidable predicate* $eval_{\mathcal{L}}(F, t)$ *(also written as* $F \vdash_{\mathcal{L}} t$*), which holds if the term* t *of sort* `Truth`*, i.e.,* $t \in T^{\mathcal{L}}_{\texttt{Truth}}(\Sigma, X)$*, can be reduced to* TRUE *using the rules (or formulæ) of the store* $F = \langle \Sigma, \mathcal{R} \rangle$.

The predicate $eval_{\mathcal{L}}$ (or relation $\vdash_{\mathcal{L}}$) corresponds to a evaluation of a boolean expression in classical functional languages, or to a test for validity in logic programming. Whenever the (declarative) language $\mathcal{L}$ is clear from the context, we omit the corresponding index. Similarly, we write, by abuse of notation, S instead $\overline{S}$, if there is no risk of confusion.

3.2 Example. *The predicate* $eval_{\mathcal{TOY}}$ *(respectively,* $eval_{Curry}$*) is defined by (conditional) narrowing, the standard operational semantics of* $\mathcal{TOY}$ [LFSH99] *(respectively, Curry* [HAK+00b]*). Notice that a* $\mathcal{TOY}$ *(respectively, Curry) program is a set of rules and corresponds thus obviously to definition 3.1.*

3.3 Example. *In the logic programming language Prolog* [DEDC96]*, the operation* $eval_{Prolog}$ *is defined by the standard operational semantics with the additional condition that the answer substitution should be the identity substitution. Notice that we model* atoms *(i.e., applications of* predicates*) as terms of sort* `Truth`.

3.4 Example. *Besides declarative languages, classical* imperative *programming languages, such as* ADA [Ada95] *or* C [KR88]*, can, in general, be used for the description of stores. Intuitively, the rules of an imperative store define the values stored in each of the cells of the memory, and terms are straightforwardly defined as expressions. However, we have to require that expressions use only* functions[4] *which do not have side-effects, i.e., which do not modify the memory. The check for validity is then the (side-effect free) evaluation of a boolean-valued expression.*

Notice that the definition of stores is similar to the notion of a *logical system* in the framework of institutions [GB92]. Informally, a logical system is characterised by a signature Σ, a collection of Σ-sentences, a collection of Σ-models and a Σ-satisfaction relation (of Σ-sentences by Σ-models) [GB92, page 96].

3.1.2 Example of a declarative language

In this section, we define a simple functional logic language which we use in the remainder of the thesis to illustrate our proposal of a computation model for concurrent declarative programming languages, namely for the description of stores. Instead of this simple language, we might also have used another declarative language, as for instance Curry [Han97, HAK+00b], $\mathcal{TOY}$ [GHLR96, GHLR99] or Haskell [PJHA+99]. The main motivation for choosing our own simple declarative language is the need of an abstract data type of programs in order to define actions (see section 3.2).

In the following, we recall briefly the basic notions about (conditional) term rewriting and narrowing [DJ90, Klo92, BK86].

[3] For instance, functional types can be constructed using the type constructor $\rightarrow$, *e.g.*, the sort $s_1 \rightarrow s_2$ denotes the sort of functions of *domain* s_1 and *range* s_2.

[4] *Procedures* are captured in our model by the notion of processes. In fact, both execute *actions*.

3.1.2.1 Syntax

Roughly speaking, a program in our simple declarative language is defined by a *signature* Σ and a set of *rules* $\mathcal{R}$ defining the semantics of the functions or operations of the signature. A signature defines the data types of the program, as well as the operations on these data types. We follow a constructor-discipline [O'D85], *i.e.*, we distinguish between the *constructors* of the data types and *defined functions*.

3.5 Definition (signature). *A (constructor-based)* signature Σ *is a pair* $\Sigma = \langle S, \Omega \rangle$ *of a set of* sorts S *and a* (S^+*-indexed*[5]) *family of* function *symbols* Ω.

The family of function symbols is partitioned into constructor *symbols* C *and* defined function *(or* operator*) symbols* D*, i.e.,* $\Omega = C \uplus D$.

For a function symbol $f \in \Omega_{s_1 \cdot \ldots \cdot s_{n+1}}$, we call $s_1 \times \ldots \times s_n \to s_{n+1}$ the *profile* of the function symbol f. The *arity* of a function symbol of profile $s_1 \times \ldots \times s_n \to s_{n+1}$ is defined as the natural number n, corresponding to the number of arguments the function accepts.

We require that a signature Σ should contain at least the sort `Truth` ($\in S$), representing the *predicates* or *constraints*, which are used in the conditions of the rewrite rules. We also require that the signature contains at least two constructors of the sort `Truth`, namely "TRUE" ($\in C_{\texttt{Truth}}$) and the (polymorphic) equality predicate $=$, *i.e.*, for all sorts s different from `Truth`, *i.e.*, $s \in \big(S \setminus \{\texttt{Truth}\}\big)$ we have that $=_{s \cdot s} \in D_{s \cdot s \cdot \texttt{Truth}}$. In the following, we may omit these obligatory parts of a signature to shorten the notation.

3.6 Example. *The following is a signature of natural numbers,* Σ_{nat}*:*

$$
\begin{aligned}
S_{nat} &= \{\texttt{Truth}, Nat\} \\
C_{nat} &= \{zero : Nat;\ succ : Nat \to Nat;\ \text{TRUE} : \texttt{Truth};\ = : Nat \times Nat \to \texttt{Truth}\} \\
D_{nat} &= \{+, -, \text{mod} : Nat \times Nat \to Nat;\ <, \geq, : Nat \times Nat \to \texttt{Truth}\}
\end{aligned}
$$

In addition to a signature, we suppose that we are given a infinite ($\overline{S}$-indexed) family of (sets of) variables X.

3.7 Definition (term). *For a (constructor-based) signature* Σ *and a (*S*-indexed) family of (sets of) variables* X*, we define the (*S*-indexed) family of well formed* terms *$T(\Sigma, X)$ inductively as the smallest set such that the following conditions hold:*

- *all variables are terms:* $\forall x \in X_s\ (s \in S) : x \in T_s(\Sigma, X)$ *and*
- *all* well sorted applications *are terms:*
  $\forall f \in \Omega_{s_1 \cdot \ldots \cdot s_n},\ \forall t_i \in T_{s_i}(\Sigma, X)\ (i \in \{1; \ldots; n-1\},\ s_i \in S) :$
  $f(t_1, \ldots, t_{n-1}) \in T_{s_n}(\Sigma, X)$.

3.8 Notation. *We note the set of* free variables *of a term* t *as* $\mathcal{V}(t)$*. By abuse of notation we write also* $\mathcal{V}(t_1, \ldots, t_n) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \mathcal{V}(t_i)$ *for the set of free variables of a collection of terms* $t_1, \ldots, t_n$.

[5] In the language we consider here, the $\overline{S}$ is the set of sequences over S, where the concatenation of sorts is written as $\cdot$.

$$
\begin{array}{rll}
zero + x \to x & & (R_{+1}) \\
succ(x) + y \to succ(x + y) & & (R_{+2}) \\
x - zero \to x & & (R_{-1}) \\
succ(x) - succ(y) \to x - y & & (R_{-2}) \\
x \bmod y \to x & \mid x < y & (R_{\bmod 1}) \\
x \bmod y \to (x - y) \bmod y & \mid x \geq y & (R_{\bmod 2}) \\
zero < succ(x) \to \text{TRUE} & & (R_{<1}) \\
succ(x) < succ(y) \to x < y & & (R_{<2}) \\
x \geq zero \to \text{TRUE} & & (R_{\geq 1}) \\
succ(x) \geq succ(y) \to x \geq y & & (R_{\geq 2})
\end{array}
$$

Table 3.1: Rules for the Signature Σ_{nat}

When a term does not contain an application of a defined function, we call it a *constructor term*. A *pattern* is a well sorted term of the form $f(t_1, \ldots, t_n)$, where f is a defined function and the t_i's are constructor terms ($\forall i \in \{1; \ldots; n\}$). We call a term *operation-rooted* when it has an operation symbol at the root, otherwise it is said to be in *head normal form*. A (constructor) term t is called *ground*, if it contains no free variables, *i.e.*, using notation 3.8, if $\mathcal{V}(t) = \varnothing$.

The semantics of the operators is defined by means of (conditional) rewrite rules.

3.9 Definition (rule). *A (conditional rewrite)* rule *R is a triple $\langle \mathsf{lhs}, \mathsf{rhs}, \mathsf{condition} \rangle$, noted $\mathsf{lhs} \to \mathsf{rhs} \mid \mathsf{condition}$, such that*

- lhs[6] *is a linear pattern, i.e., each free variable of* lhs *occurs at most once in* lhs*, and*
- $\mathsf{condition} = \bigwedge_{i=1}^{n} t_i$ *is a* condition*, i.e., a possibly empty ($n \geq 0$) conjunction of terms (of sort* `Truth`*) $t_i \in T_{\mathtt{Truth}}(\Sigma, \mathcal{V}(\mathsf{lhs}))$.*

If the condition is the empty conjunction, *i.e.*, the constant TRUE, it is often omitted: we write an unconditional rewrite rule simply as $\mathsf{lhs} \to \mathsf{rhs}$. Notice that definition 3.9 could be refined by additional conditions on the use of free variables in the rhs and $\mathsf{condition}$. For instance, one might require that for all rules we have that the rule does not introduce new free variables, *i.e.*, $\mathcal{V}(\mathsf{rhs}) \cup \mathcal{V}(\mathsf{condition}) \subseteq \mathcal{V}(\mathsf{lhs})$.

3.10 Example. *The rules shown in table 3.1 complete example 3.6 by defining the classical semantics of the operators of the signature Σ_{nat}.*

The following notation extends the notion of free variables to rules, by taking the union of the free variables of the terms occurring in the rule.

3.11 Notation. *We extend the notation of free variables to rules by noting the set of* free variables *of a rule $R = \mathsf{l} \to \mathsf{r} \mid \mathsf{c}$ as $\mathcal{V}(R) = \mathcal{V}(\mathsf{l} \to \mathsf{r} \mid \mathsf{c}) = \mathcal{V}(\mathsf{l}) \cup \mathcal{V}(\mathsf{r}) \cup \mathcal{V}(\mathsf{c})$.*

[6] lhs (respectively, rhs) stands for left (respectively, right) hand side.

A set of (conditional) rewrite rules, as for instance in example 3.10, defines a (conditional) *Term Rewriting System* (TRS) which we also call a declarative *program*.

3.12 Definition (program). *A* program *or constructor-based term rewriting system F is a pair $F = \langle \Sigma, \mathcal{R} \rangle$ of a (constructor-based) signature Σ and a set of rules $\mathcal{R}$.*

Informally, a (declarative) program can be seen as a description of a theory defining the static relationships between the symbols of the signature. The use of such a theory description is defined by the operational semantics. One possible operational semantics is presented in the following subsection.

3.1.2.2 Operational Semantics

The operational semantics of a program is defined by rewriting [DJ90, Klo92] and narrowing [Sla74]. In our setting, rewriting allows to compute normal forms of an expression, whereas narrowing allows to compute a substitution such that an expression can be reduced to normal form. Before defining rewriting and narrowing formally, we introduce two further definitions, namely substitutions and positions.

Throughout this section, we suppose that we are given a program $F = \langle \Sigma, \mathcal{R} \rangle$.

3.13 Definition (substitution). *A* substitution σ *is a mapping $\sigma : X \to T(\Sigma, X)$ from variables to terms such that $\sigma(x)$ is of sort s for all variables $x \in X_s$ and such that the* s-domain *of σ, i.e., $SDom(\sigma) \stackrel{\text{def}}{=} \{x \in X \mid \sigma(x) \neq x\}$, is finite.*

By abuse of notation, we also note σ the unique extension of a substitution σ to terms, i.e., $\sigma : T(\Sigma, X) \to T(\Sigma, X)$. A unifier *of two terms t and t' is a substitution σ such that $\sigma(t) = \sigma(t')$.*

We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in SDom(\sigma)\}$. The set of all substitutions is denoted by $\mathcal{S}ub$ and Id denotes the identity-substitution, *i.e.*, $Id \stackrel{\text{def}}{=} \{x \mapsto x \mid x \in X\}$[7].

We call a term t' an *instance* of a term t if there exists a substitution σ with $\sigma(t) = t'$, in which case we write $t \leq t'$. A substitution σ is called more general than a substitution σ' if there exists a substitution ϑ such that for all variables x, $\vartheta\big(\sigma(x)\big) = \sigma'(x)$. We denote a most general unifier of two terms t and t' by $mgu(t, t')$.

3.14 Example. *We have $mgu\big(a + b,\ succ(x) + y\big) = \big\{\, a \mapsto succ(x);\ b \mapsto y \,\big\}$.*

Using substitutions, we can define the notion of a *variant* of a rewrite rule, which can be obtained by consistently renaming all free variables of the rule (see notation 3.11) by *fresh variables*, where a fresh variable is a variables that does not occur in the current environment (or computation).

3.15 Definition (variant). *Let W be a set of variables, and R a (conditional) rewrite rule, i.e., $R = \mathsf{l} \to \mathsf{r} \mid \mathsf{c}$. We call a* variant *of R with respect to W as a (conditional) rewrite rule $R' = \mathsf{l}' \to \mathsf{r}' \mid \mathsf{c}'$ such that there exists an injective substitution $\sigma \stackrel{\text{def}}{=} \{x \mapsto y \mid x \in \mathcal{V}(R)$ and $y \notin W\}$ and we have $\sigma(\mathsf{l}) = \mathsf{l}'$, $\sigma(\mathsf{r}) = \mathsf{r}'$ and $\sigma(\mathsf{c}) = \mathsf{c}'$.*

[7] Notice that we have $SDom(Id) = \varnothing$.

In the sequel, we consider most of the time variants of rules with respect to the set of all variables occurring in the current environment or computation. Therefore, whenever we omit the set W, it should be understood as the set of all variables occurring in the current environment or computation.

In order to differentiate the subterms of a term, we recall the notion of *positions*.

3.16 Definition (position). *A* position *is a sequence of positive integers identifying a subterm in a term. The empty sequence, denoted by Λ, denotes the term t itself for every term t. For every term of the form $f(t_1, \ldots, t_n)$, position p and positive number $\mathsf{i} \in \{1; \ldots; n\}$ the sequence $\mathsf{i} \cdot \mathsf{p}$ identifies the subterm at position p of t_i.*

We denote by $\mathcal{P}os$ the set of all positions.

Using positions, replacements of subterms can be easily described.

3.17 Notation. *We write $t|_\mathsf{p}$ for the subterm at the position p of the term t. The result of replacing $t|_\mathsf{p}$ by t' in t is denoted by $t[t']_\mathsf{p}$.*

We are now ready to define rewriting. Since our rules are allowed to contain variables in the condition and the right hand side which do not occur in the left hand side, we use variants of rules for rewriting.

3.18 Definition (unconditional reduction step). *A term t can be rewritten by an* unconditional reduction step *to a term t', i.e., $t \to_{\mathsf{p}, R} t'$ if there exist a position p, a variant $R' = l \to r$ of an unconditional rewrite rule $R \in \mathcal{R}$ and a substitution σ with $t|_\mathsf{p} = \sigma(l)$ and $t' = t[\sigma(r)]_\mathsf{p}$.*

We let $\overset{*}{\to}$ denote the transitive and reflexive closure of $\to$. A term t is *reducible* to a term t' if $t \overset{*}{\to} t'$, and *irreducible* if there is no term t' such that $t \to t'$. A *normal form* of a term t is an irreducible term t' such that t is reducible to t'. In the sequel, we suppose that the considered term rewriting systems are *confluent*, *i.e.*, that if there exists a normal of a term t, than this normal form is unique determined. We note the normal form of a term t as $t{\downarrow}$.

3.19 Example. *Considering the signature and rules of examples 3.6 and 3.10, we have that the term $succ(zero) + succ(zero)$ is reducible to $succ(succ(zero))$, since we have the following rewriting derivation:*

$$succ(zero)+succ(zero) \;\to_{\Lambda,\,(R_{+2})}\; succ(zero+succ(zero)) \;\to_{1\cdot\Lambda,\,(R_{+1})}\; succ(succ(zero)) \tag{3.1}$$

The application of a conditional rewrite rule $R = (l \to r \mid c)$ adds the condition that *all* terms of the condition have to be reducible to the constructor term TRUE in order to apply the rule R. Hence, the definition of a conditional reduction step generalises definition 3.18.

3.20 Definition (reduction step). *A term t can be rewritten to a term t' by means of a* reduction step*, i.e., $t \to_{\mathsf{p}, R} t'$ if there exist a position p, a variant $R' = l \to r \mid c$ of a (conditional) rewrite rule $R \in \mathcal{R}$ and a substitution σ with $t|_\mathsf{p} = \sigma(l)$, $t' = t[\sigma(r)]_\mathsf{p}$ and $\sigma(t_i){\downarrow} = \text{TRUE}$ for all terms t_i in c, i.e., $c = \bigwedge_{i=1}^{n} t_i$.*

3.21 Example. *For the program consisting of the signature and rules of examples 3.6 and 3.10 we have the following reduction step:*

$$zero \bmod succ(zero) \to_{\Lambda,\,(R_{\bmod 1})} zero \tag{3.2a}$$

since the substituted condition of rule $(R_{\bmod 1})$ *is reducible to* TRUE*, i.e.,*

$$\big(\sigma(x < y)\big) = \big(zero < succ(zero)\big) \to_{\Lambda,\,(R_{<1})} \text{TRUE} \tag{3.2b}$$

where σ *is defined as* $\sigma \stackrel{\text{def}}{=} \{x \mapsto zero; y \mapsto succ(zero)\}$.

The difference between narrowing and rewriting is that a narrowing step instantiates free variables of the considered term in order to apply a rewrite rule (whereas rewriting would simply fail). Stated otherwise, narrowing uses *unification* instead of *pattern matching* when selecting the rewrite rule to apply.

3.22 Definition (unconditional narrowing step). *An* unconditional narrowing step *is a transformation of a term* t *to a term* t'*, i.e.,* $t \rightsquigarrow_{\mathsf{p},\,R,\,\sigma} t'$ *if there exist a non-variable position* p *in* t *(i.e.,* $t|_{\mathsf{p}} \notin X$*, a variant* $R' = l \to r$ *of an unconditional rewrite rule* $R \in \mathcal{R}$ *and a unifier* σ *of* $t|_{\mathsf{p}}$ *and* l *such that* $t' = \sigma(t[r]_{\mathsf{p}})$*. In this case the term* t *is called* narrowable *at position* p *using the rule* R *and substitution* σ*.*

We let $\overset{*}{\rightsquigarrow}$ denote the transitive and reflexive closure of $\rightsquigarrow$. For a finite narrowing derivation, *i.e.*, a sequence of (unconditional) narrowing steps,

$$t_1 \;\rightsquigarrow_{\mathsf{p}_1,\,R_1,\,\sigma_1}\; t_2 \;\rightsquigarrow_{\mathsf{p}_2,\,R_2,\,\sigma_2}\; \cdots \;\rightsquigarrow_{\mathsf{p}_n,\,R_n,\,\sigma_n}\; t_{n+1} \tag{3.3}$$

(with $n \geq 0$) such that the term t_{n+1} is a constructor term, *i.e.*, not narrowable, we call the substitution $\vartheta \stackrel{\text{def}}{=} \sigma_n \circ \cdots \circ \sigma_1$[8] the *answer substitution* for the term t_1. Since the instantiations of the variables in the rules R_i by the substitution σ_i are not relevant for the computed result (*i.e.*, the term t_{n+1} and the answer substitution ϑ) of a narrowing derivation, we omit the corresponding parts of ϑ in the sequel.

3.23 Example. *Referring to example 3.10, the following are examples of narrowing steps:*

$$a \geq b \qquad \rightsquigarrow_{\Lambda,\,(R_{\geq 1}),\,\{b \mapsto zero\}} \qquad \text{TRUE} \tag{3.4a}$$

$$a \geq b \quad \rightsquigarrow_{\Lambda,\,(R_{\geq 2}),\,\{a \mapsto succ(x);\; b \mapsto succ(y)\}} \quad x \geq y \tag{3.4b}$$

Thus the unconditional narrowing step (3.4a) *yields the answer substitution* $\{b \mapsto zero\}$*, whereas the step* (3.4b) *leads to a term which can be further narrowed.*

In our setting, we use narrowing mainly for goal solving, *i.e.*, for searching an answer substitution such that the goal can be reduced to TRUE. We define a goal as a conjunction of terms (of sort `Truth`).

3.24 Definition (goal). *Let* $\Sigma = \langle S, \Omega \rangle$ *be a signature and* X *a (*S*-indexed family of) sets of variables. A* goal g *is defined as a (finite) conjunction of terms of sort* `Truth`*, i.e.,* $\mathsf{g} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} t_i$ *where we have for all* $i \in \{1; \ldots; n\}$ *that* $t_i \in T_{\mathtt{Truth}}(\Sigma, X)$*.*

[8] We denote by $\circ$ the composition of functions, *i.e.*, $(f \circ g)(x) \stackrel{\text{def}}{=} f\big(g(x)\big)$ $(\forall x)$.

Roughly speaking, a (conditional) narrowing step consists in removing a term from a goal, performing an unconditional narrowing step on the term and adding the new term together with the substituted condition of the used rule to the remainder of the goal.

3.25 Definition (narrowing step). *A* narrowing step *is the transformation of a goal* g *into a new goal, i.e.,*

$$\big(\{t\} \wedge \mathsf{g}\big) \;\leadsto_{(l \to r \,|\, (\bigwedge_{i=1}^{n} t_i)),\, \sigma}\; \Big(\{t'\} \;\wedge\; \big(\bigwedge_{i=1}^{n} \{\sigma(t_i)\}\big) \;\wedge\; \sigma(\mathsf{g})\Big) \tag{3.5}$$

if there exists a position p *of* t *such that* $t \leadsto_{\mathsf{p},\, l \to r,\, \sigma} t'$.

We say that a goal g is solved, if $\mathsf{g} = \{\text{TRUE}\}$ and call a goal g *solvable* if there exists a narrowing derivation leading from g to $\{\text{TRUE}\}$. The reason why we define conditional narrowing on conjunctions of terms (of sort `Truth`) and not just on a single term as for conditional rewriting (see definition 3.20), is that we cannot evaluate the conditions at once due to the presence of uninstantiated variables. Notice that the commutativity and associativity of conjunction implies that in equation (3.5) any conjunct of the goal g could have been chosen (instead of t).

3.26 Example. *Consider the goal* $\mathsf{g} = \big\{\, (x \bmod succ(succ(zero))) < succ(zero) \,\big\}$. *A narrowing derivation yielding the answer substitution* $\{x \mapsto zero\}$ *is the following*

$$\begin{aligned} &\big\{\, (x \bmod succ(succ(zero))) < succ(zero) \,\big\} \\ &\qquad \leadsto_{\Lambda,\, (R_{\bmod 1}),\, Id} \quad \big\{\, x < succ(zero);\; x < succ(succ(zero)) \,\big\} \\ &\qquad \leadsto_{\Lambda,\, (R_{<1}),\, \{x \mapsto zero\}} \big\{\, \text{TRUE};\; x < succ(succ(zero)) \,\big\} \\ &\qquad \leadsto_{\Lambda,\, (R_{<1}),\, \{x \mapsto zero\}} \big\{\, \text{TRUE} \,\big\} \end{aligned} \tag{3.6}$$

To compute a reduction (respectively, narrowing) step one must compute the step's position. A *rewriting strategy* (respectively, *narrowing strategy*) is a (partial) function $\mathcal{S}t : T(\Sigma, X) \to \mathcal{P}os \times \mathcal{R}$ (respectively, $\mathcal{S}t : T(\Sigma, X) \to \mathcal{P}os \times \mathcal{R} \times \mathcal{S}ub$) such that whenever $\mathcal{S}t(t) = (\mathsf{p}, R)$ (respectively, $\mathcal{S}t(t) = (\mathsf{p}, R, \sigma)$), t is reducible (respectively, narrowable) at position p using the rule R (and the substitution σ). For the unconditional case, different rewriting and narrowing strategies have been described, see for instance [AEH94, AEH00, AEH97]. The soundness and completeness of (lazy) constrained (or conditional) narrowing has been shown in [LF92].

To conclude the presentation of the simple declarative language, we show that it meets the general requirements on languages that are to be used for the description of stores in our computation model (see before, *i.e.*, section 3.1.1 and definition 3.1).

3.27 Example. *Obviously, the definition of a program (see definition 3.12) corresponds to the definition of a store (see definition 3.1). The predicate of definition 3.1 can be defined as follows:*

$$F \vdash t \;\Leftrightarrow\; t \overset{*}{\leadsto}_{Id} \text{TRUE} \textit{ with the identity-substitution } Id \tag{3.7}$$

3.1.3 Names

We conclude this section with the description of an additional built-in which is necessary in order to model mobile processes. Recall from the introduction, that processes of a component modify the store or declarative program by the execution of actions. As we show in the following section, these actions are defined on a meta-level with respect to the store, since they manipulate stores or declarative programs as data (see also figure 3.2).

Notice that at the level of the actions, *i.e.*, at the meta-level with respect to the store or declarative program, we distinguish between the value denoted by a constant c (which is a meta-representation of a *term*) and the constant c itself (which is a function *symbol*). This distinction is present in all languages providing assignment. In imperative programming languages, references or pointers allow to distinguish between the pointer or reference and the value which is referenced. Notice that the notion of variables in imperative languages denotes both, the value stored in a given place in the memory (when occurring at the right of :=) and the address of the place in the memory where the value is stored (when occurring on the left of :=). Similarly, in SML [MTHM97], where a symbol of type (or sort) `ref t` is distinguished from a symbol of type `t`. The distinction between structural equality (*i.e.*, equality of values) and physical equality or object identity is another reflection of this difference.

As already mentioned in section 1.1.2, we need, besides actions modifying the rules of a store, also actions for the modification of the *signature*. For the creation of new (function) symbols we introduced in section 1.1.2 the elementary action `new`. This action necessitates the introduction of a new parameterised sort representing the *names* or references to (function) symbols into the store itself. The following example illustrates this necessity.

3.28 Example. *Consider a process, say A, which creates a new communication channel, say c, such that another, concurrently executing process, say B, should send messages to A using c. For simplicity, we suppose that channels are represented as lists of messages. Notice that since the channel c is freshly created by the process A, it is in the local scope of A and the process B has no knowledge of c. Since processes in our computation model use the common store for their interaction and communication, A has to use the store to inform B about the new channel c. Thus suppose that A can send messages to B using a channel d. But passing the channel c to the process B is to be distinguished from passing the current value of c (which is probably an empty list of messages). Therefore, the type of messages that can be passed through the channel d have to be* names *of channels.*

Therefore we introduce a new parameterised sort to denote the sort of the name of a symbol of sort s.

3.29 Definition (`Name`). *The sort* `Name`(s) *denotes the sort of* symbol-names *such that the sort of the symbols is s. If c is of sort* `Name`(s)*, we denote by $c\uparrow$ the associated symbol of sort s.*

For the ease of programming, we require that the names of the symbols declared by the programmer are added implicitly. For a signature $\Sigma = \langle S, \Omega \rangle$, we call *name-signature* $\Sigma^{\mathtt{n}}$ the signature of all the names of the symbols of Σ. Thus we require that

the signature $\widetilde{\Sigma}$ of a store is the (disjoint) union of the signature Σ as defined by the programmer and the associated name-signature:[9]

$$\widetilde{\Sigma} \stackrel{\text{def}}{=} \Sigma \uplus \underbrace{\left\langle \left\{ \mathtt{Name}(s) \,\middle|\, s \in \overline{S} \right\},\ \left\{ \hat{f} : \mathtt{Name}(s) \,\middle|\, f \in \Omega_s \right\} \right\rangle}_{\stackrel{\text{def}}{=} \Sigma^{\mathtt{n}}} \tag{3.8}$$

Accordingly, the execution of the (elementary) action $\mathsf{new}(x,\ s)$ introduces two new symbols in the (signature of the) store, namely x of sort $\mathtt{Name}(s)$ and $x\uparrow$ of sort s. x stands for the name of (or a reference to) the symbol $x\uparrow$. The elementary action new together with the parameterised sort $\mathtt{Name}(s)$ allows to model mobility in the same way as the π-calculus [Mil99], where mobility is modeled by a varying communication structure due to the possibility of passing the names of communication channels.

3.2 User Defined Actions

We have already mentioned in the introduction that actions are the principal constituents for the description of processes, since each run of a process corresponds to the performance of a possibly infinite sequence of actions. In this section, we discuss the integration of the definition of actions in a computation model for concurrent declarative programming [ES01b].

The actions executed by processes operate on stores, *i.e.*, declarative programs. The effect of executing an action on a store is the modification of the store. Notice that, whenever a process has to execute actions on some physical device, the latter is considered as a component, that is to say, modeled as a store for the data description, together with processes for the control part of the device. This led us in section 1.1.2 to characterise an action as a total recursive function which goes from stores to stores.

Using our computation model, the combination of processes with a declarative programming language $\mathcal{L}$ becomes straightforward as long as it is possible to define or to use actions which modify programs written in $\mathcal{L}$. Unfortunately, actions over programs are often supposed to be not a fundamental part of a language and are not specified for most familiar programming languages. Exceptions are *reflective* languages, as for instance Maude [CDE+99] or Common Lisp [Ste90] in combination with its Metaobject Protocol (MOP) [KdRB91]. In a reflective language, programs can be represented as data in the language itself, and these data objects representing programs can be executed by an interpreter.

Nevertheless, even if classical programming languages do not provide actions as a distinguished notion, *particular* actions on programs exist, but they are mixed with "predefined built-ins" of the syntax of the language. For instance, Prolog [DEDC96] provides the "predicates" `assert` and `retract` which allow one to add and to remove clauses to and from a program. In SML [MTHM97] (respectively, Scheme [ADH+98] or Common Lisp [Ste90]), the "function" := (respectively, `setq`) permits to update the value associated to a "mutable cell". In Erlang [AVWW96], the "built-in functions"

[9] The (disjoint) union of two signatures $\Sigma_1 = \langle S_1,\ \Omega_1 \rangle$ and $\Sigma_2 = \langle S_2,\ \Omega_2 \rangle$ is defined in the obvious way, *e.g.*, $\Sigma_1 \uplus \Sigma_2 \stackrel{\text{def}}{=} \langle S_1 \uplus S_2,\ \Omega_1 \uplus \Omega_2 \rangle$.

`load_module`, `delete_module` and `purge_module` allow to exchange a version of a module by a new, "corrected" one.

All these built-in actions are supposed to be used inside the sentences defining a program. Thus these actions have the side-effect of modifying the program in which they are used. Stated otherwise, these actions do not have a parameter representing the program to which the action has to be applied. Thus, it is implicit that they applied to the program in which they occur. There is a further limitation related to these built-in actions. In fact, there may be different possibilities for the semantics of a particular action. For instance, the family of languages for `ccp` [Sar93, chapter 3] distinguishes between numerous different ways to add a formula to a store or to check for the entailment of a formula.

For these reasons, we suggest to allow the *definition of actions*, so that a programmer is not restricted to the actions built-in into the programming language, but can rather use always the action most suited for the particular need at hand. We believe that this should lead to more readable and consequently more easily maintainable programs.

3.2.1 Meta-Signatures for the Definition of Actions

There are many ways to define actions on programs for a given language $\mathcal{L}$, for example by providing a set of built-in actions (together with operators for combining them) or by providing a dedicated action description language (ADL). An ADL associated to a declarative language programming $\mathcal{L}$ is defined as a language that allows the description of actions over programs written in $\mathcal{L}$. Therefore, an ADL (for the language $\mathcal{L}$) is also a *meta-language* (for $\mathcal{L}$) since the entities manipulated by an ADL are $\mathcal{L}$-programs. This implies that an ADL needs at least to provide the data types corresponding to $\mathcal{L}$-programs.

A natural choice for an ADL of a declarative language $\mathcal{L}$ is the language in which a compiler or interpreter for $\mathcal{L}$-programs is implemented. In order to keep our computation model open to different implementation languages, we require only the definition of some abstract data type (ADT) for $\mathcal{L}$-programs. Then we define actions as functions over these ADT's, without imposing a particular way of their implementation. For instance, we use a syntax similar to SML [MTHM97] for the examples of specifications of actions in section 3.2.2.

We call signature of the ADT's of $\mathcal{L}$-programs a *meta-signature*, since this signature is on the meta-level with respect to the signatures of the programs or stores, as we have shown in figure 3.2.

3.30 Definition (meta-signature). *Consider a declarative programming language $\mathcal{L}$. A* meta-signature *for $\mathcal{L}$ is a pair $\mathrm{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$ of a set of* meta-sorts *$M_{\mathcal{L}}$ and a ($\overline{M_{\mathcal{L}}}$[10]-indexed) family of* meta-function *symbols $MO_{\mathcal{L}}$, such that $M_{\mathcal{L}}$ contains at all the sorts corresponding to the syntactic entities of the language $\mathcal{L}$.*

Obviously, the meta-sorts representing the syntactical entities of programs depend on the programming language. For functional language $\mathcal{L}$ for instance, the set of meta-sorts $M_{\mathcal{L}}$ contains at least the sorts $\texttt{symbol}_{\mathcal{L}}$ of *symbols*, $\texttt{variable}_{\mathcal{L}}$ of *variables*,

[10]Similar to definition 3.1, we note $\overline{M_{\mathcal{L}}}$ the set of sorts that can be constructed over the set of basic meta-sorts $M_{\mathcal{L}}$.

$\texttt{store}_{\mathcal{L}}$ of *stores* (or *programs*), $\texttt{term}_{\mathcal{L}}$ of *terms* and $\texttt{rule}_{\mathcal{L}}$ of *rules*. As an example of a meta-signature, we give a simplified presentation of the data type actually used in the implementation of the simple declarative language presented in section 3.1.2. This language is also used in the current prototype for the description of the stores.

3.31 Example. *In the case of the simple functional logic language presented in section 3.1.2, we might have the meta-signature shown in figure 3.3, where we use a syntax similar to* SML [MTHM97]. *The profile of a function* `name` *is written as (*`->` *is supposed to be right-associative):*

```
name : argument_sort_1 -> ... -> argument_sort_2 -> result_sort
```

The sort of lists with elements of sort `element_sort` *is denoted by* `element_sort list`*, and there are predefined sorts of boolean values* `bool` *and character strings* `string`*. The keyword* `type` *introduces the declaration of a new sort. Finally, comments are enclosed between* `(*` *and* `*)`*.*

This meta-signature or ADT is defined in a modular way. The signatures of basic data types (i.e., meta-sorts) such as strings and generic lists are missing. Data types of sorts, operations, variables, terms, rules and stores are described by their constructors (`make_?`*), testers (*`is_?`*) and accessors (*`get_?`*), the straightforward definitions of which are also omitted. Note that this ADT is not meant to be an (optimized) implementation of the considered declarative language, but just a support to ease the description of examples for the definition of actions in the subsequent section.*

Other examples of meta-signatures or ADT's of programs are `FlatCurry` [Han, HAK+00a], an intermediate representation of Curry (or other functional-logic) programs or the sort `Module` used in the `META-LEVEL` module of Maude [CDE+98, CDE+99] for the representation of Maude-modules.

Since different declarative programming languages can be implemented in very different languages, which are not necessarily declarative, not to speak of being based on constructor based term-rewriting systems, we restrict ourselves to examples of possible definitions of meta-operators, most prominently elementary actions. Therefore, we define actions (over a declarative language $\mathcal{L}$) as curryfied functions that may take some arguments and return total recursive functions from $\mathcal{L}$-programs to $\mathcal{L}$-programs.

3.32 Definition (elementary action). *Consider a declarative language $\mathcal{L}$ and its meta-signature $\mathrm{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$. An* elementary action $\mathtt{a}$ *over $\mathcal{L}$-programs is defined as a total recursive function the profile of which has the following form*

$$\mathtt{a} : \mathtt{s}_1 \to \ldots \to \mathtt{s}_n \to \mathtt{store}_{\mathcal{L}} \to \mathtt{store}_{\mathcal{L}} \tag{3.9}$$

where $\mathtt{s}_i \in M_{\mathcal{L}}$ $(\forall i \in \{1; \ldots; n\}, n \geq 0)$ and $\mathtt{store}_{\mathcal{L}}$ $(\in M_{\mathcal{L}})$ is the sort of $\mathcal{L}$-programs. We also require that the store returned by an elementary action be well-formed, i.e., a correct $\mathcal{L}$-program.

Obviously, whatever formalism is used for the definition of actions, we have to ensure that an action respects the conditions of definition 3.32, namely that the action is a *total* and *recursive* function and that the result of an action is a well-formed $\mathcal{L}$-program. A possibility to guarantee these properties is to (syntactically) restrict the

```
type store                (* STORES *)
make_store       : sort list -> operation list -> rule list -> store
get_sorts        : store -> sort list
get_operations   : store -> operation list
get_rules        : store -> rule list

type rule                 (* RULES *)
make_rule     : term -> term -> term -> rule
get_lhs       : rule -> term
get_rhs       : rule -> term
get_condition : rule -> term

type term                 (* TERMS *)
make_variable_term : variable               -> term
make_application   : operation -> term list -> term
is_variable        : term -> bool
is_application     : term -> bool
get_variable       : term -> variable
get_operation      : term -> operation
get_arguments      : term -> term list

type variable             (* VARIABLES *)
make_variable     : string -> sort -> variable
get_variable_name : variable -> string
get_variable_sort : variable -> sort

type operation            (* OPERATIONS *)
make_function        : string -> sort -> operation
make_constructor     : string -> sort -> operation
get_function_name    : operation -> string
get_constructor_name : operation -> sort
get_function_sort    : operation -> string
get_constructor_sort : operation -> sort
is_function          : operation -> bool
is_constructor       : operation -> bool

type sort                 (* SORTS *)
make_basic_sort      : string -> sort
make_functional_sort : sort -> sort -> sort
```

Figure 3.3: Sample of a Meta-Signature (or ADT) for the simple Declarative Language of section 3.1.2.

formalism used for their description, *i.e.*, the ADL, in order to either guarantee, or at least allow the definition of *effective* analyses.

Notice that a particular consequence of the requirement, that the resulting store of an elementary action has to be well formed, is that it has to be well-typed (since this is a necessary condition for a well-formed store). As we see in the sequel, for the set of actions introduced in section 1.1.2[11], namely tell, del and :=, the need for *dynamically* type-checking the store (*i.e.*,, at the moment of the execution of the action) can be avoided, and replaced by a *static* analysis of the program *before* its execution. Consider the assignment action $c := v$. For the resulting store to be well-typed, we have to require that the sorts of the constant c and the term (or value) v are the same. This condition can be checked during the analysis of a process, since we know the sorts of all symbols and terms in the store.

Notice that we suggest to define actions by a well mastered mathematical concept, namely total recursive functions, for the description of which appropriate (programming) languages have been defined and used for some time now. Notice that similarly, the semantics of the Haskell-type system is defined by a functional program (written in Haskell) [Jon99].

3.2.2 Examples of Definitions of Actions

In this section we give examples of definitions of actions on a declarative programming language similar to the language which was introduced in section 3.1.2. For this purpose, we use as ADL a functional language, syntactically close to SML [MTHM97], where we denote the application of a function `f` to two arguments `x_1` and `x_2` by `(f x_1 x_2)`, that is to say with parentheses *around* the function symbol and its arguments.

Recall that, according to definition 3.12, a store or program in our simple declarative language (see section 3.1.2) is a pair $\langle \Sigma, \mathcal{R} \rangle$ where the signature Σ is a pair $\Sigma = \langle S, \Omega \rangle$ of a set of sorts S and a family of sorted functions Ω, and $\mathcal{R}$ is a set of conditional rewrite rules. A corresponding ADT or meta-signature has been presented given in example 3.31 (and figure 3.3).

3.2.2.1 Adding Rules

The actions which are most straightforward to define are those which just add something to a part of the store, for example the addition of a rule. In example 1.1.5, for instance, the elementary action tell(is_eating(x)) adds the rule "is_eating(x) $\rightarrow$ TRUE".

Obviously, since in our example the store contains a *list* of rules, we have at least two different possibilities to implement this action, depending on the position where the new rule is inserted into the list of rules. Two reasonable choices are for instance the beginning and the end of the list – the following is a (naive) specification of the former:

```
add_rule : rule -> store -> store
add_rule rule store =
```

[11] In fact, most actions that do modify only the rules of the store enjoy the same property.

```
make_store (get_sorts store)
           (get_operations store)
           (cons rule (get_rules store))
```

Prolog provides two built-in "predicates", namely `asserta` and `assertz`, which add a new clause at the beginning (`asserta`) or the end (`assertz`) of the clauses defining the corresponding predicate [DEDC96, pages 44 – 47][12]. Obviously, the possibilities are different, if the ADT of stores is more sophisticated in order to implement "optimal" evaluation strategies, where rules are stored, for instance, within definitional trees [Ant92].

There are still more possibilities for the addition of a rule to a store. Consider the case of adding several times the same rule. Depending on the operational semantics, the existence of duplicated rules may be important (notice that the standard of Prolog [DEDC96] is not very precise about how this is handled in Prolog). In classical `ccp` for instance, telling the constraint c several times (*i.e.*, adding the constraint c several times to the store) is equivalent to telling it just once [Sar93]. On the other hand, in linear logic programming (see section 2.1.4), the number of occurrences of a formula does have an influence on the theory represented by the store.

These possibilities could be implemented by combining the addition of a rule with a test, such as adding a rule only if it is not yet present in the store (modulo some equivalence relation).

3.2.2.2 Removing Rules

There are several possibilities to remove a rule from a store. A programmer might want to remove a precise rule, or all rules of a specified form. Thus, when removing a rule, we should test each rule separately if it should be removed or not. Different possibilities of removal correspond then to different tests, which might be a functional parameter of the action. Examples for such test are identity, identity up to renaming of variables, pattern matching, unification, equality (equivalence with respect to the store), *etc.*. A possible implementation of the removal action is the following:

```
remove_rules : (rule -> bool) -> store -> store
remove_rules test store =
   make_store (get_sorts store)
              (get_operations store)
              (find_all (fun x -> not (test x)) rules)
```

where the function `(find_all t list)` returns the list of all elements `e` of the list `list` for which the evaluation of `(t e)` returns true. We use an anonymous function (or λ-abstraction) to inverse the result of the test `test`, that is to say the expression `(find_all (fun x -> not (test x)) rules)` denotes the list of all rules `r` in the list of rules `rules` for which `(test r)` returns `false`.

[12]The clause of a Prolog-program are searched sequentially [DEDC96, pages 22 – 23], and different primitives allow to add at the beginning or the end of the clauses. However, the standard of Prolog does not specify if the clauses are stored in a *list* (or multi-set, allowing double occurrences) or *set* (in which case no double occurrences are possible). Thus one has to *guess* (from the examples) that the clauses are stored in a list (and their multiplicity matters).

In Prolog, the built-in "predicate" `abolish(predicate)` removes all clauses for a given predicate `predicate` (in a single step), whereas `retract(pattern)` removes all rules which can be unified with the rule-pattern `pattern` (one by one upon backtracking) [DEDC96, pages 37 – 38, 154 – 155]. While `abolish` is not very "precise", the successful use of `retract` requires to control the number of necessary backtracks, which is in our opinion not straightforward.

3.2.2.3 Assignment

Probably the most common action is assignment (`:=`) as it is ubiquitous in imperative programming languages. Assignment is also used in some declarative languages such as SML [MTHM97], Common Lisp [Ste90], Scheme [ADH+98] or Oz [Smo95b]. Using the actions defined above (*i.e.*, the functions `add_rule` and `remove_rules`), the action of assignment might be defined as follows:

```
(:=) : operation -> term -> store -> store
(:=) operation term store =
(add_rule
   (make_rule (make_application operation nil) term true)
   (remove_rules
      (rule_pattern_match
         (make_rule (make_application operation nil)
                    (make_variable_term x)
                    (make_variable_term y)))
      store))
```

where `x` and `y` are variables, `true` is the boolean constant true, and `nil` the empty list. `rule_pattern_match` is a test function which implements a pattern-matching-based removal[13]:

```
rule_pattern_match : rule -> rule -> bool
rule_pattern_match rule1 rule2 =
   (term_matches (get_lhs rule1) (get_lhs rule2)) and
   (term_matches (get_rhs rule1) (get_rhs rule2)) and
   (term_matches (get_condition rule1) (get_condition rule2))
```

where we use the standard pattern matching function `term_matches`:

```
term_matches : term -> term -> bool
```

`(term_matches term1 term2)` returns `true` if `term1` matches `term2`.

Thus the assignment `c := term` removes all rules defining the (constant) operation `c` and adds a single rule which redefines `c` to have the value `term`. This view of assignment is also adopted in the coordination language Linda [Gel85, page 98].

Notice, that in our computation model, a "variable" in the sense of standard imperative programming languages corresponds to a "changing constant": using assignment,

[13] We suppose that `and` is evaluated "lazily" in a sequential manner from left to right. To make the example more readable, we write `and` instead of `&&` which would be required by SML [MTHM97].

we may change the theory which defines the value of the constant. However, in each of these theories, the value (of the constant) does not change, *i.e.*, it is constant. Notice further, that as in imperative programming, assignment leads to a *state change*. But in contrary to imperative programming languages, assignment is not the only possible action.

As already mentioned in section 1.1.2, a further reasonable requirement on the assignment action (`c := term`) is to reduce the new value, *i.e.*, `term`, to normal-form, *i.e.*, the action might add a rule defining `c` to have the value `term'` where `term'` is the normal-form of `term` (under the assumption that the store is confluent).

3.2.2.4 Modifying the Signature

So far, we have only considered the modifications of the rules of a store. Modifications of the other part of the store, *i.e.*, its signature, might be interesting too. For instance, consider an implementation of a window system. Such a system needs to store information about all the different windows that are currently displayed on the screen. Roughly speaking, a theory describing the current state of the window system might model every window by a constant. Hence, when a request for the creation of a new window arrives, the theory has to be changed, and a *new* constant corresponding to the new window needs to be created. A similar example is the dynamic creation of new communication channels, which is mandatory in order to model mobility through link passing as in the π-calculus [Mil99].

These examples have in common that the enrichment of the signature is limited to new constants, which then may be further used and modified by assignment as in classical imperative programming languages. However, there are also situations where the addition of a new operation, or even a new sort might be necessary. For instance, if a program has to be modified, the new version of the program might use new operations over new data-structures, that is to say new data-types. This happens for example if we want to change the implementation of an algorithm using another, more efficient data structure, as for instance graphs instead of lists.

Actions modifying the signature are executed implicitly in some of the interactive interpreters for modern declarative languages, whenever the definition of new global symbols is permitted, as for example the `let`-construct in `ocaml` [LDG+01] or SML/NJ [SML98].

Removing declarations from the signature is more problematic. In particular, we need to ensure that the symbol removed is no longer in use. While the condition of the wellformedness of the resulting store ensures this for the rules of the store, it is more problematic to ensure that no other process is still using this symbol. Therefore, we consider in the rest of this thesis only actions that enrich signatures.

Notice further that an elementary action which adds symbols to the signature of a store should have them as parameters. Otherwise the action could be used only once, since there is no point in adding the *same* symbol twice[14]. In general, the introduction of a new symbol is motivated by the use of this symbol in the sequel of the execution. Thus we suggest to consider elementary actions that enrich the signature as *binding* the symbols they introduce. This is similar to for instance, the receive operation of the

[14]Recall from definition 3.1, that signatures are *sets* of symbols.

π-calculus [Mil99], where the names received on a channel are bound in the subsequent process. In the sequel, we use the following notation for the new symbols introduced by an elementary action. Since an action can introduce both, sorts as well as operators, the set of new symbols introduced by an action corresponds to a signature $\widehat{\Sigma} = \langle \widehat{S}, \widehat{\Omega} \rangle$.

3.33 Notation. *We denote by* $\mathcal{N}\big(\mathsf{a}(t_1, \ldots, t_n)\big)$ *the signature of* new symbols *introduced by the execution of the call to the elementary action* $\mathsf{a}(t_1, \ldots, t_n)$.

The following example defines the set of new symbols for some of the actions presented in the introduction and in this chapter.

3.34 Example. *The actions* tell *(or* `add_rule`*),* del *(or* `remove_rule`*) and assignment (i.e.,* $:=$*) do not modify the signature of the store, we have immediately:*

$$\mathcal{N}\big(\mathsf{tell}(R)\big) \;=\; \mathcal{N}\big(\mathsf{del}(R)\big) \;=\; \mathcal{N}(c := v) \;\stackrel{\mathsf{def}}{=}\; \langle \varnothing, \varnothing \rangle \tag{3.10a}$$

where R is a rule, c a function symbol and v a (meta-representation of a) term (of the same sort as c).

3.35 Example. *Consider the elementary action* new *mentioned in section 1.1.2. Informally,* $\mathsf{new}(c, s)$ *adds two symbols to the signature of the store, namely the new name of a function symbol c of sort* $\mathtt{Name}(s)$ *and the associated (function) symbol $c\uparrow$ of sort s. Thus we define:*

$$\mathcal{N}\big(\mathsf{new}(c, s)\big) \;\stackrel{\mathsf{def}}{=}\; \big\langle \varnothing,\; \{c : \mathtt{Name}(s);\; c\uparrow : s\} \big\rangle \tag{3.10b}$$

3.3 Component Signatures

A component is defined as a part of a system. Consequently, the description of a component that interacts with the rest of the system necessarily depends on the specification of the system. In our computation model, a system is modeled as a set of components, which are identified by means of storenames. Thus we can represent a system by the set $\mathcal{SN}$ of all the storenames of the components forming the system, together with a bijective mapping from storenames to components. In the following we therefore define a component with respect to a set of storenames $\mathcal{SN}$ which represents the system the component is designed for.

In the previous two sections of this chapter, we have already presented two parts of a component that can be studied separately, namely the definition of a store, *i.e.*, a declarative program, and the definition of actions, *i.e.*, functions over meta-representations of declarative programs. In this section we present the notion of a component signature, before we give the definitions of the different parts of a component in the following sections. We illustrate the definition of a component signature by the example of the multiple counters, and define the notion of component terms.

3.3.1 Component Signatures

A component signature defines all the symbols that occur in the description of a component. Notice that these symbols belong to all the different levels shown in figure 3.4

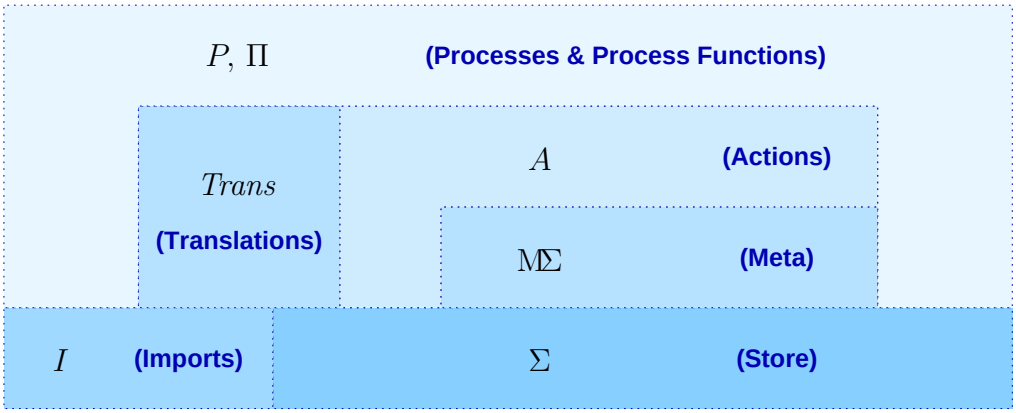

Figure 3.4: Levels of a System Description: Structure of a Component-Signature

(which refines figure 3.2), that is to say to the level of the store, to the meta-level of the store, as well as to the part describing the processes. Besides the symbols of the stores (and the associated meta-representations) of the components in the system, a component signature defines symbols for actions, translations, processes and functions on processes and actions. These symbols defined in a component signature allow therefore the construction of `process`es and `action`s, which are introduced as new sorts, along with the sort `storename` representing storenames. We comment on the different parts of a component signature after the definition.

3.36 Definition (component signature). *Let $\mathcal{SN}$ be a set of storenames. We define, for a storename $\widehat{sn} \in \mathcal{SN}$ and a declarative language $\mathcal{L}$, a* component signature $\mathbb{C}\Sigma$ *as an eight-tuple* $\mathbb{C}\Sigma = \langle \Sigma, \mathbb{M}\Sigma_{\mathcal{L}}, A, I, E, Trans, P, \Pi \rangle$ *where*

- $\Sigma = \langle S, \Omega \rangle$ *is a* signature *of a store, i.e., of a $\mathcal{L}$-program,*
- $\mathbb{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$ *is a* meta-signature *for $\mathcal{L}$,*
- *A is a ($\overline{(S \uplus M_{\mathcal{L}})}^{15}$-indexed) family of* action *symbols, the sorts of which are of the form $\mathtt{s}_1 \to \ldots \to \mathtt{s}_n \to \mathtt{store}_{\mathcal{L}} \to \mathtt{store}_{\mathcal{L}}$ $(n \geq 0)$,*
- *$I = \left\{ \mathbb{I}\Sigma_{sn} = \langle \Sigma_{sn}, \mathbb{M}\Sigma_{\mathcal{L}_{sn}}, A_{sn} \rangle \,\middle|\, sn \in (\mathcal{SN} \setminus \{\widehat{sn}\}) \right\}$ is a ($\mathcal{SN}$-indexed) family of* imported signatures *$\mathbb{I}\Sigma_{sn}$, i.e., triples of signatures $\Sigma_{sn} = \langle S_{sn}, \Omega_{sn} \rangle$, meta-signatures $\mathbb{M}\Sigma_{\mathcal{L}_{sn}} = \langle M_{\mathcal{L}_{sn}}, MO_{\mathcal{L}_{sn}} \rangle$ ($\mathcal{L}_{sn}$ is the (declarative) language used for the store of component sn) and ($\overline{(S \uplus M_{\mathcal{L}_{sn}})}^{15}$-indexed) families of actions symbols A_{sn} (the sorts of which are of the form $\mathtt{s}_1 \to \ldots \to \mathtt{s}_n \to \mathtt{store}_{\mathcal{L}_{sn}} \to \mathtt{store}_{\mathcal{L}_{sn}}$),*
- *$E = \langle E_\Sigma, E_{\mathbb{M}\Sigma}, E_A \rangle$ is a triple of a sub-signature[16] E_Σ, a sub-meta-signature $E_{\mathbb{M}\Sigma}$ and a subset of the action symbols E_A, i.e., $E_\Sigma \subseteq \Sigma$, $E_{\mathbb{M}\Sigma} \subseteq \mathbb{M}\Sigma$ and $E_A \subseteq A$,*

[15] As before (see definitions 3.1 (page 78) and 3.30 (page 88)), we note $\overline{S}$ the set of sorts that can be constructed over the set of basic sorts S.

[16] For two signatures $\Sigma_1 = \langle S_1, \Omega_1 \rangle$ and $\Sigma_2 = \langle S_2, \Omega_2 \rangle$ we say that Σ_1 is a *sub-signature* of Σ_2, written as $\Sigma_1 \subseteq \Sigma_2$, if $S_1 \subseteq S_2$ and $\Omega_1 \subseteq \Omega_2$. A similar relation is defined for all other kinds of signatures, in particular meta-signatures, in the obvious way.

- *Trans* $= \{ Tr_{sn} \mid sn \in SN \}$ *is a (*SN*-indexed) family of (*$(\overline{(S \uplus S_{sn})}^{15}$*-indexed) families of)* translation *symbols* Tr_{sn} *(the sorts of which are of the form* $s_1 \to s_2$ *with* $s_1 \in S$ *and* $s_2 \in S_{sn}$*),*
- P *is a (*$\overline{PS}^{15}$*-indexed) family of* process *symbols the range of which is* `process`*, containing at least the parameterless process* `success` $\in P_{\texttt{process}}$*, which always terminates successfully,*
- Π *is a (*$(\overline{PS}^{15}$*-indexed) family of* process function *symbols, the profile of which contains at least one of the three sorts* `action`*,* `process` *or* `storename`*,*

and where the set of sorts PS *is defined as*

$$PS \stackrel{\text{def}}{=} S \uplus M_{\mathcal{L}} \uplus \Big(\biguplus_{sn \in (SN \setminus \{\widehat{sn}\})} (S_{sn} \uplus M_{\mathcal{L}_{sn}}) \Big) \uplus \{\texttt{action}; \texttt{process}; \texttt{storename}\} \quad (3.11a)$$

The following paragraphs motivate and comment on the different parts of a component signature which are also shown in figure 3.4.

Σ: On the level of the store, a programmer has to specify the signature of the initial stores. Obviously, the component signature has to contain the signature of the store of the component itself, but in order to interact with other components, the component signature needs to incorporate also (parts of) the signatures of the initial stores of these other components (see the "imports" I). Notice that the signatures of the components of a system are not necessarily of the same kind, since they may be signatures of programs written in different declarative languages. For instance, some of these signatures may follow a constructor discipline, while others do not.

MΣ: Along with the signatures of the stores, a component signature contains the definitions of the meta-signatures or ADT's corresponding to meta-representations of the stores. As we have seen in section 3.2, these meta-signatures are necessary for the definition of actions and correspond to the meta-level in figure 3.4.

A: Definition 3.36 extends the definition of actions of the preceding section, *i.e.*, definition 3.32, to a richer set of admissible sorts for the parameters of an (elementary) action. Besides meta-terms, also terms of the store are allowed as parameters. Thus actions are different from meta-operators, since the sorts of the latter are constructed only from meta-sorts, whereas the former have sorts constructed from meta-sorts and sorts. The second kind of parameters is to be understood as parameters of the meta-sort corresponding to the syntactic entities of the corresponding sort. For instance, for the language presented in section 3.1.2, an example of such a syntactic entities are terms, and the corresponding meta-sort is `term`. As already mentioned in the introduction, the meta-terms are obtained from terms by means of an implicit application of the mapping *reify*. Roughly speaking, *reify* associates to a syntactic entity e of a declarative language $\mathcal{L}$ its representation as a meta-term (of the meta-sort `e` corresponding to this entity $\texttt{e} \in M_{\mathcal{L}}$). A particular case are terms of sort $\texttt{Name}(s)$ (see section 3.1.3). To such terms, the mapping *reify* associates the corresponding symbol (in the language of section 3.1.2, a meta-term of meta-sort $\texttt{symbol} \in M_{\mathcal{L}}$).

Besides offering a more convenient description of processes, the use of the sorts of the stores in the profiles of action symbols has the additional advantage of enabling static type-checking of the arguments of actions. For instance, considering the assignment action $c := v$ (see section 3.2.2.3), we can statically verify that the sorts of the function symbol c and the term v are the same. Thus it is no longer necessary to verify this condition at runtime[17].

I, E: The imported signatures allow the processes of the component to interact with other components of the system. Since processes modify stores, we need to import the signatures of the stores, the meta-signatures and the (elementary) actions defined on the remote component. The exported symbols of a component are those which other components are allowed to import. So it seems natural to require that these symbols are defined in the component signature and that not necessarily all symbols are exported.

$Trans$: Processes communicate by modification of the stores. If the stores are written in different (declarative) languages, the communication of a value from one store to another requires the *translation* of this value. These translations are described by means of translations functions $Trans$. Furthermore, since processes communicate by modifying the stores, the values received by a component are already translated (since they are already in the store). Hence, the burden of translation is on the sender of a message, as a part of the construction of a well-formed action that can be executed on the remote store. Thus it seems natural to index the translations of a component by functional sorts $s_1 \to s_2$, where s_1 is a sort of the store (*i.e.*, $s \in S$) of the component and s_2 is a sort imported from the component with storename sn (*i.e.*, $s_2 \in S_{sn}$).

PS: The set of sorts PS defines the (basic) sorts that can be used in the definition of processes and process functions. Besides the (disjoint) union of the sorts and meta-sorts, PS contains three additional sorts, namely `action`, `process` and `storename`. The sort `storename` represents storenames and allows to pass storenames as parameters to processes. The sort `action` represents actions, *i.e.*, pairs of a storename and a call to an (elementary) action. `action` is necessary for the definition of functions that yield actions. The sort `process` has the same rôle as `action`, but for processes.

P, Π: The last two kinds of symbols introduced by a component signature are *processes* and *process functions*. The difference between the processes P and the process functions Π is similar to the distinction between constructors and defined functions in constructor-based languages (as for instance the simple declarative language presented in section 3.1.2), *i.e.*, processes define the basic entities on which process functions operate. On the one hand, the meaning of process functions is defined by (rewrite) rules, and their operational semantics by rewriting. On the other hand, the processes are defined by process definitions, and their operational semantics is given by means of transition system in chapter 4. Thus, similar to the constructors in section 3.1.2,

[17]This remark holds only for actions that are part of the program which is analysed before the execution, and does not in general hold for actions that are received during execution of the program (because they are unknown when the program is analysed).

processes are undefined in the rewriting system defining the process functions. A further difference is that we require processes in P to have the result-sort `process`.

Notice that a process function cannot be defined as a classical function, meta-function or a translation function, since the profile of a process function is required to contain at least one occurrence of one of the sorts `action`, `process` or `storename`. This justifies also to consider process functions as a separate notion.

We do not distinguish between "process functions" and "action functions" (*i.e.*, process functions the result sort of which is `action`), as both are on the same level in figure 3.4. They are used for a high level description of processes, and are not modified during the execution of the system. Nevertheless, we use the term *action function* in the sequel to denote process functions the profile of which does not contain any occurrence of the sort `process`.

The following convention abbreviates the description of processes. In fact, by considering the definitions of the local store as imported from the component with storename $\widehat{sn}$, we can restrict ourselves to consider only imported symbols.

3.37 Notation. *To shorten the notation we integrate the signature Σ, meta-signature $\mathrm{M}\Sigma$ and family of actions A of the component into the family of imported symbols I (which becomes a SN-indexed family). We define thus:*

$$\Sigma_{\widehat{sn}} \stackrel{\text{def}}{=} \Sigma, \qquad \mathrm{M}\Sigma_{\widehat{sn}} \stackrel{\text{def}}{=} \mathrm{M}\Sigma \qquad \textit{and} \qquad A_{\widehat{sn}} \stackrel{\text{def}}{=} A \tag{3.12}$$

Furthermore, we renew the convention introduced in section 3.1.1 and confound, by abuse of notation, in the sequel, whenever there is no risk of confusion, the set of (basic) sorts S and the set of sorts $\overline{\mathrm{S}}$ that can be constructed from S (for all sets of sorts S).

We conclude this section with the introduction of the notion of *component terms*, *i.e.*, terms constructed using the symbols defined in a component signature. Informally, a component signature $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ can be seen as a signature, *i.e.*, a pair $\langle PS, PO \rangle$, where the set of sorts PS are defined as in equation (3.11a) (see on page 97 in definition 3.36) and the ($\overline{PS}$-indexed) family of operators combine all the symbols defined in a component signature, *i.e.*,[18]

$$PO \stackrel{\text{def}}{=} \Big(\biguplus_{sn \in SN} (\Omega_{sn} \uplus MO_{\mathcal{L}_{sn}} \uplus A_{sn}) \Big) \uplus \mathit{Trans} \uplus P \uplus \Pi \uplus SN \tag{3.11b}$$

Notice that, according to equation (3.11b), we consider the storenames as the constructors of the sort `storename`, *i.e.*, we have for all $sn \in SN$ that $sn \in PO_{\texttt{storename}}$. The family of operations PO is extended in the following section by, on the one hand, the additional constructors of the sort `process`, namely the symbols corresponding to operators of process algebras, as for instance $\|$ (parallel composition) or $\oplus$ (choice with priority), and, on the other hand, constructors of the sort `action`, in particular the composition of a storename and a call to an elementary action.

[18] Notice that equation (3.11b) uses notation 3.37.

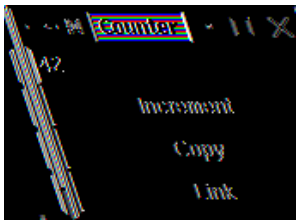

Figure 3.5: A Counter Window

Viewing a component signature as a signature according to equations (3.11a) and (3.11b), we define for a ($\overline{PS}$-indexed) family of variables X the ($\overline{PS}$-indexed) family of *component terms* $CT(\mathrm{C}\Sigma, X)$ as the terms (see definition 3.7) over the signature $\mathrm{C}\Sigma$[19] and the set of variables X, *i.e.*,

$$CT(\mathrm{C}\Sigma, X) \stackrel{\mathsf{def}}{=} T(\mathrm{C}\Sigma, X) \tag{3.13}$$

The extension of the notation of free variables of a term introduced in section 3.1.2.1 (see notation 3.8) from terms to component terms is straightforward.

3.3.2 Example of the Multiple Counters

Consider a (simplistic) application inspired from [Gui95] representing a system of multiple counters. The application starts by creating a window (as shown in figure 3.5) representing a counter which can be incremented manually by clicking on the button labeled Increment. The behaviour of the two other buttons in the counter window is as follows. The Copy-button creates an independent counter (with an associated new window) and initialises it with the current value of the counter being copied, whereas the Link-button creates a new view (*i.e.*, a new window) of the same counter. All links (or views) of a same counter should behave identically, *e.g.*, they increase the counter at the same time. Additionally we may want to use the current value of the counters for some calculations, in the same way as we would like to use any other constant in a classical declarative language. We use our solution for this problem as a running example for the illustration of the different definitions in the remainder of this chapter.

We suggest to separate the management of the windows from the counters. Hence we model the system using two components, a first one for the counters, say C, and a second one for the window system, say X. This is similar to real window systems, where applications may run on a machine connected via the network to the machine controlling the monitor. In this chapter, we focus on the component C. We start in this example with the presentation of the component signature for the component C. We specify each of the eight parts of the component signature separately, using the following set of storenames $\mathcal{SN} = \{\mathsf{C}; \mathsf{X}\}$.

The store of the component C describes a theory for counters. We model a *counter* c as a constant of type *Cnt*, *i.e.*, a pair $\langle val, wins \rangle$ of the current *value* *val* of the

[19] In fact, we consider $\mathrm{C}\Sigma$ enriched with constructors of the sorts `action` and `process` (see definitions 3.43 and 3.50).

$$S \stackrel{\text{def}}{=} \{Cnt;\ Evt;\ Wid;\ Evt\_List;\ Wid\_List\} \tag{3.14a}$$

$$C \stackrel{\text{def}}{=} \left\{ \begin{array}{l} cnt : Nat \times Wid\_List \to Cnt; \\ increment, copy, link : Evt; \\ nil_{Evt} : Evt\_List; \quad cons_{Evt} : Evt \times Evt\_List \to Evt\_List \\ nil_{Wid} : Wid\_List; \quad cons_{Wid} : Wid \times Wid\_List \to Wid\_List \end{array} \right\} \tag{3.14b}$$

$$D \stackrel{\text{def}}{=} \left\{ \begin{array}{l} get\_val : Cnt \to Nat;\ get\_wins : Cnt \to Wid\_List \\ head_{Evt} : Evt\_List \to Evt; \quad tail_{Evt} : Evt\_List \to Evt\_List \\ head_{Wid} : Wid\_List \to Wid; \quad tail_{Wid} : Wid\_List \to Wid\_List \\ append_{Evt} : Evt\_List \times Evt\_List \to Evt\_List \end{array} \right\} \tag{3.14c}$$

Table 3.2: Signature of the store for the Multiple Counters Example

counter c and a list of the *window identifiers wins* of the windows associated with c, *i.e.*, the windows displaying the value of c. The fields of a counter can be accessed by the functions *get_val* and *get_wins*. We represent the values of a counter by natural numbers which can be specified for instance as in the examples of section 3.1.2 (examples (3.6) and (3.10)). The window identifiers are represented by strings of characters, which we suppose to be a built-in sort. The processes of the counters have to react on *events* occurring in the windows. The only (high-level) events (occurring in a counter window) we consider are clicks on the different buttons. Thus we define the sort *Evt* by the set of the three constructors $\{increment, copy, link\}$. Obviously, we also need the sort of lists, which are classically represented by means of two constructors, namely *cons* which takes an element and a list and returns a list and *nil*, the empty list.

The signature Σ_{C} of the store of the component C is thus the enrichment of the signature Σ_{nat} (see example 3.6) with the following sorts, constructors and functions (we omit the declaration of the equality predicate = for the new sorts) as shown in table 3.2.

An example for the meta-signature of the simple declarative programming language has already been given in example 3.31, so we do not need to repeat it here. The actions we need for the modifications of the store C are assignment (:=) and the creation of new symbols new. Recall from section 3.1.3, that new takes two arguments, the first corresponding to the new (function) symbol to be introduced in the signature of the sort and the second to the sort of the new symbol. To simplify, we suppose that all these symbols are exported, *i.e.*, the component C does not hide any of its symbols.

Since there are only two components in the system, the family of imported symbols consists only of the imports from the component X which are shown in table 3.3. In order to display the counter windows, the component C imports two elementary actions from X, namely add-win and refresh-win. The action add-win(v, c, e) creates a new counter window for the counter c displaying the value v such that clicks on the buttons in this window have the effect of adding a corresponding *Evt* at the end of the list of *Evt*s e. The process in the component X that actually handles the creation of the window is also in charge of sending an action to the store C which adds the

$$\Sigma_{\mathsf{X}} \stackrel{\text{def}}{=} \big\langle \{\texttt{int};\ \texttt{string}\},\ \{\texttt{+} : \texttt{int} \times \texttt{int} \to \texttt{int}\} \big\rangle \tag{3.14d}$$

$$A_{\mathsf{X}} \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \mathsf{add\text{-}win} : & \texttt{int} \times \texttt{string} \times \texttt{string} \to (\texttt{store}_{\texttt{ocaml}} \to \texttt{store}_{\texttt{ocaml}}); \\ \mathsf{refresh\text{-}win} : & \texttt{int} \times \texttt{string} \to (\texttt{store}_{\texttt{ocaml}} \to \texttt{store}_{\texttt{ocaml}}) \end{array} \right\} \tag{3.14e}$$

Table 3.3: Imports from component the X by the component C

Wid of the new window to the counter c. Thus, the action add-win takes the names of the counter c and the event-list e as arguments, by means of a built-in encoding into character-`string`s (see section 3.4.2). The action refresh-win(v, w) refreshes the display of the window denoted by the *Wid* w such that the new value v is displayed.

Thus the imported signature from the component X needs to specify at least the sorts and operators necessary for the arguments of the actions add-win and refresh-win. Suppose that the component X uses `ocaml` [LDG+01] for the description of its store, and that the values of the counter are represented by `int`s and the *Wid*s by `string`s. Besides these two sorts, we import the addition `(+)` of `int`s.

We need two translations, namely *int-of-nat* and *string-of-wid*, in order to provide the parameters for the elementary actions executed on the component X. The former, *i.e.*, *int-of-nat*, translates terms of sort *Nat* (see example 3.6) to the built-in `int`egers of `ocaml`, whereas the latter translates *Wid*'s into `string`s of `ocaml` :

$$\mathit{Trans}_{\mathsf{X}} \stackrel{\text{def}}{=} \{\mathit{int\text{-}of\text{-}nat} : \mathit{Nat} \to \texttt{int};\ \mathit{string\text{-}of\text{-}wid} : \mathit{Wid} \to \texttt{string}\} \tag{3.14f}$$

We suggest to provide one control process per counter window, which we call cnt_ctrl. The process cnt_ctrl needs two parameters: on the one hand, the name of the constant representing the counter the value of which is displayed in the window and, on the other hand, the name of the list of events which occur in the window. A second process, called create_win, handles the creation of new counter windows. create_win takes the same arguments as cnt_ctrl. Thus, we the set of processes of the component C is defined as follows

$$P \stackrel{\text{def}}{=} \{\text{cnt\_ctrl}, \text{create\_win} : \texttt{Name}(\mathit{Cnt}) \times \texttt{Name}(\mathit{Evt\_List}) \to \texttt{process}\} \tag{3.14g}$$

Finally, we use a process function to express the update of all the windows of a same counter:

$$\Pi \stackrel{\text{def}}{=} \{\mathit{refresh\text{-}windows} : \texttt{int} \times \mathit{Wid\_List} \to \texttt{action}\} \tag{3.14h}$$

3.4 Interactions

In our computation model, processes of a same component use the common store for interaction. All processes of a component have access to all information in the store, *i.e.*, the signature and the rules. To communicate, processes modify the store by executing actions. In this section we present the interface between components.

Informally, interaction between components is based on the same scheme as interaction between processes on the same component: in fact, processes are allowed to modify the stores of other components, *i.e.*, they can execute actions on these stores. For instance, in the example of the multiple counters (see section 3.3.2), the component C needs to refresh the display of the windows associated to a counter whenever the value of the counter has changed. Since the display of the windows is handled by the component X, this action needs to be sent from C to X. In order to interact, components need to exchange the declarations or profiles of the symbols that are used for the interaction. We say that a component *exports* a declaration, when this declaration can be used by other components. The declarations of a remote component that a component uses for interaction are called *imported* declarations. We present the imports and exports in the subsequent section. Furthermore, to model the interaction between components in different languages, we introduce the notion of *translations* in section 3.4.2. In the example of the multiple counters, the value of the counter has to be translated such that it is understood on the component X.

3.4.1 Imports and Exports

We distinguish different levels of imports (respectively, exports). For instance, the declarative program describing a store may itself be a collection of files or modules, and the access to symbols defined in other modules might be restricted. This is to be distinguished from the import (respectively, export) of declarations of a store or declarations of actions from one component to another. The former is a facility to structure the program or store, whereas the latter is necessary for interaction between components. For instance, a component must be able to construct the parameters of an action to be executed on a remote store. In this section, we describe the imports and exports related to the interaction between components.

Since interaction between components in our computation model is based upon the execution of actions on the stores of remote components, a component needs to import the *actions* which can be executed on the stores of other components. Since these actions take parameters that are related to the store of the remote component, the *signature* of the store, *e.g.*, sorts, functions and predicates, as well as its *meta-signature* have to be imported. This led us in the preceding section to define the signature imported from a component with storename sn as a triple of a signature Σ_{sn}, meta-signature $\mathrm{M}\Sigma_{\mathcal{L}_{sn}}$ (where $\mathcal{L}_{sn}$ is the (declarative) language used for the store of component sn) and a family of actions symbols A_{sn}.

To avoid name-clashes, *i.e.*, two symbols with the same identifier defined in different components, the identifiers of the imported symbols could be prefixed with the name of the component they are defined in, similar to the prefixing of the module name as in, for example, `ocaml` [LDG+01] or Curry [HAK+00b].

The exported signature of a component is the part of the signature of the component which can be used, *i.e.*, imported, by other components. Thus, a component can export a sub-signature of its store, a sub-meta-signature of its stores and a subset of its action symbols.

3.38 Example. *As an example for imports, reconsider the example of the multiple counters (see section 3.3.2), in particular table 3.3 which gives signature imported by*

the component C *from the component* X.

The component C *exports its complete signature, meta-signature and set of actions.*

3.4.2 Translations

Consider a system of several components the stores of which are written in different declarative languages. When the processes of such a system are to interact, the values sent from one (store) to another (store) have to be *translated*, *i.e.*, values of one language have to be transformed into values of the other. Here we mean by values terms in some normal form, for instance ground constructor terms. It seems natural to require a translation to be a total recursive function, since to any value that is to be communicated has to correspond a unique translation which should be computable.

Note that these translation functions cannot be part of one of the stores involved, since they define relations between objects in both of the languages. In fact, they can be seen as functions in a "union-language" that combines both stores, *i.e.*, two programs written in different languages. Furthermore, a translation may need to use the operational semantics of the declarative languages, in order to reduce terms to normal-forms before and after the translation, if necessary.

We suggest to specify translations (from language $\mathcal{L}_1$ to language $\mathcal{L}_2$[20]) via a (general) term rewriting system, and to separate a translation into three steps. First, the term to be translated is reduced to normal form (using the operational semantics of language $\mathcal{L}_1$). Then a corresponding expression is generated, by application of a special *translation function*. Finally the expression is reduced to normal form (using the operational semantics of language $\mathcal{L}_2$) yielding the translation of the original term. The motivation of this separation is to let a programmer just specify the translation functions, and put the handling of the other phases into the implementation. Thus a translation function associates to a term t in normal form (in $\mathcal{L}_1$) a term t' (in $\mathcal{L}_2$) whose normal form corresponds to the translation of the term t.

Obviously, for declarative languages in which the notion of normal form is not defined, the first and/or the last step in the translation of a value are unnecessary and omitted. An example of such a language is Prolog [DEDC96], where all terms are distinguished elements of the least Herbrand model which defines the semantics of a Prolog-program. Thus, in Prolog, all terms are already in "normal form" and the normalisation steps before and after translation are not needed.

Since we translate *values* (and no meta-representations) we can define translation functions by a term rewriting system which translates from a signature Σ_1 into a signature Σ_2.

3.39 Definition (translation signature). *Let* $\Sigma_1 = \langle S_1, \Omega_1 \rangle$ *and* $\Sigma_2 = \langle S_2, \Omega_2 \rangle$ *be two signatures of stores. We define the* translation signature $\mathfrak{T}\Sigma_{\Sigma_1, \Sigma_2}$ *as the tuple*

$$\mathfrak{T}\Sigma_{\Sigma_1, \Sigma_2} \stackrel{\mathsf{def}}{=} \big\langle (S_1 \uplus S_2),\, (\Omega_1 \uplus \Omega_2 \uplus Tr) \big\rangle \tag{3.15}$$

where Tr *is a* $\overline{(S_1 \uplus S_2)}$*-indexed) family of* translation *symbols* Tr*, the sorts of which are of the form* $s_1 \to s_2$ *(where* $s_1 \in S_1$ *and* $s_2 \in S_2$*).*

[20] We do not require $\mathcal{L}_1$ and $\mathcal{L}_2$ to be different languages. In fact, translations can also be used for the transformation between different implementations of a data-type implemented using the same language.

We define the family of terms over a translation signature in the usual manner.

3.40 Definition (translation term). *For a translation signature* $\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}$ *and a* $\overline{(S_1 \uplus S_2)}$*-indexed) family of variables* X *we define the* $\overline{(S_1 \uplus S_2)}$*-indexed) family of* translation terms $\mathsf{Tr}(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}, X)$ *inductively as the smallest set such that the following conditions hold* $(\Sigma_j = \langle S_j, \Omega_j \rangle,\ j \in \{1;2\})$*:*

- *all well sorted terms of* Σ_j *are translation terms:*
  $\forall t \in T_s(\Sigma_j, X)(j \in \{1;2\},\ s \in S_j) : t \in \mathsf{Tr}_s(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}, X)$ *and*
- *all* well sorted applications *of translation functions are translation terms:*
  $\forall tr \in Tr_{s_1,s_2},\ \forall t \in \mathsf{Tr}_{s_1}(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}, X)\ (s_1 \in S_1,\ s_2 \in S_2) :$
  $tr(t) \in \mathsf{Tr}_{s_2}(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}, X)$.

We extend the notation of free variables of section 3.1.2 to translation terms, and note the set of free variables of a translation term t by $\mathcal{V}(t)$. The operational behaviour of translation functions is defined by t-rules, *i.e.*, (unconditional) rewrite rules.

3.41 Definition (t-rule). *Let* $\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}$ *be a translation signature and* X *a* $\overline{(S_1 \cup S_2)}$*-indexed) family of variables* $(\Sigma_j = \langle S_j, \Omega_j \rangle,\ j \in \{1;2\})$*. We define a* t-rule R *as a pair* $\mathsf{lhs} \to \mathsf{rhs}$ *such that*

- $\mathsf{lhs} = tr(t) \in \mathsf{Tr}_s(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2}, X)$ *is the application of a translation* tr *to a translation term* t*,*
- $\mathsf{rhs} \in \mathsf{Tr}_s\big(\mathfrak{T}\Sigma_{\Sigma_1,\Sigma_2},\ \mathcal{V}(\mathsf{lhs})\big)$ *is a translation term of the same sort as* lhs *and*
- $s \in S_2$ *is a sort of the signature* Σ_2*.*

In the sequel we write $\mathcal{T}r$ *for a set of t-rules.*

The first condition ensures that the t-rule defines a translation function, and the other two conditions ensure that the term is translated into a correct term over the signature Σ_2 into which we want to translate.

3.42 Example. *In the example of the multiple counters (see section 3.3.2), the translation int-of-nat of the sort of natural numbers Nat to the built-in* `int`*egers of* `ocaml` *can be defined by the following two t-rules:*

$$\textit{int-of-nat}(\textit{zero}) \to \mathtt{0} \tag{3.16a}$$

$$\textit{int-of-nat}\big(\textit{succ}(x)\big) \to \big(\mathtt{(+)}\ \textit{int-of-nat}(x)\ \mathtt{1}\big) \tag{3.16b}$$

Besides the translations defined by the programmer, we use in the sequel an additional translation *to-string* transforming `Name`s of symbols into character `string`s, which we suppose to be a predefined sort in languages used for the description of the stores. This translation is motivated by the same arguments as the introduction of the type `Name` in the stores (see example 3.28 in section 3.1.3), namely the necessity of passing communication links from one process to another, but here in the context of communications over the borders of components.

We therefore require that the translation *to-string* is a bijective mapping, that is to say, that there exists (for every language $\mathcal{L}$) a mapping *to-string*$^{-1}$ such that

to-string$^{-1}\big($*to-string*$(n)\big) = n$ for every name n[21]. Hence, in the example of the multiple counters (see equation (3.14e) in section 3.3.2), the name of the counter and the event-list can be encoded into character `string`s (on the component C before sending them as arguments of the action add-win to the component X) and be decoded to the corresponding `symbol`s (by the component X before modifying the store of the component C).

Another possibility to interaction between components written in different languages is to suppose that both components understand a common language, and to provide built-in translations for the elements of this third language. Examples for this approach are for instance our built-in translation *to-string* for `Name`s, the Universal Type System (UTS) language of the MLP system [HS87], the Interface Definition Language (IDL) of the Common Object Request Broker Architecture (CORBA) [COR01] or the interfaces of `ocaml` (respectively, ADA) to C [LDG$^+$01, chapter 17] (respectively, [Ada95, chapter B.3]). Besides the fact, that such a common third language does not always exists, a programmer has still to implement the translations for the higher-level (*i.e.*, not built-in) data-structures defined in his program. Thus it requires a disciplined programmer to avoid mixing translations with other "unrelated" functions[22].

Even when such a third language exists, its use is not necessarily straightforward. For instance, while it is possible to implement a program combining parts written in ADA and in `ocaml`, exploiting their interfaces to C is somewhat tricky, due to several specific compiler and linker options, additional libraries and required compilation order[23].

In Maude [CDE$^+$99], translations between two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ can be reified as functions from meta-representations of $\mathcal{L}_1$ to meta-representation of $\mathcal{L}_2$ [CDE$^+$98, section 4.3]. Thus in contrary to our proposal, the translations in Maude are described on the meta-level with respect to the languages (with the additional requirement that both meta-levels are specified in Maude). These translations are thus general in the sense that they specify how to translate a program written in $\mathcal{L}_1$ into a program written in $\mathcal{L}_2$. In our model, we need to be more specific, since we want to translate a value described in one program into a value with respect to another program. Therefore we define the translations separately using the symbols defined in the programs.

3.5 Processes

The processes of a component are specified in the style of a process algebra, see for instance [Fok00, BW90]. Basic process terms, *e.g.*, guarded actions or process calls,

[21] This can be obtained, for instance, by encoding both the name and the sort of the symbol (or name) in the `string` returned by *to-string*.

[22] The `ocaml`-Manual recommends to separate translations (when providing user-defined primitive C-functions) [LDG$^+$01, section 17.1.2]:

> "Implementing a user primitive is actually two separate tasks: on the one hand, decoding the arguments to extract C values from the given Caml values, and encoding the return value as a Caml value; on the other hand, actually computing the result from the arguments. Except for very simple primitives, it is often preferable to have two distinct C functions to implement these two tasks."

[23] With a colleague we needed recently about two hours for getting a simple "Hello-World"-style program running.

are combined by means of operators for constructing processes. Additionally, process functions allow the use of functional abstractions in the description of processes. In this section, we present the definition of processes. We start with the basic process terms and go on with the definition of the operators on processes. Finally we give the rules for defining processes.

3.5.1 Action Expressions and Guarded Actions

The basic process terms of our computation model are guarded actions. Informally, a guarded action is a pair of a *guard*, *i.e.*, a term of the store, and an *action expression*. Therefore, before giving the new definition of a guarded action, we define the notion of a (well-formed) action expression. Informally, a well-formed action expression is a component term of sort `action` (see equation (3.13)).

We have basically three kinds of action expressions. First of all, we have pairs $\langle sn,\ \mathtt{a}(t_1, \ldots, t_n)\rangle$ of a storename sn and a call to an elementary action $\mathtt{a}$. Then we have sequences of action expressions $a_1 ; a_2$. The last kind of action expression are applications of action functions.

3.43 Definition (action expression). *Let* $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi\rangle$ *be a component signature and* X *an* ($\overline{PS}$*-indexed) family of (sets of) variables, where* PS *is defined according to equation* (3.11a)*. Then we define the set of* action expressions $\mathcal{A}(\mathrm{C}\Sigma, X)$ *as the set of component terms of sort* `action`*, i.e.,*

$$\mathcal{A}(\mathrm{C}\Sigma, X) \stackrel{\mathsf{def}}{=} CT_{\mathtt{action}}(\widetilde{\mathrm{C}\Sigma}, X) \tag{3.17}$$

where the component signature $\widetilde{\mathrm{C}\Sigma}$ *is the component signature* $\mathrm{C}\Sigma$ *enriched with the following two* constructors *of the sort* `action`*:*

1. $\langle \bullet, \bullet\rangle$ *is a constructor of profile* $\mathtt{storename} \to (\mathtt{store}_{\mathcal{L}_{sn}} \to \mathtt{store}_{\mathcal{L}_{sn}}) \to \mathtt{action}$ *(for all storenames* $sn \in SN$*) and*
2. $\bullet ; \bullet$ *is a constructor of profile* $\mathtt{action} \to \mathtt{action} \to \mathtt{action}$*.*

where we use the symbol "$\bullet$*" as a placeholder for the parameters, since these operators are usually written in mixfix-syntax.*

Notice that definition 3.43 implies that any action expression is of one of the *three* following forms: $\langle sn, \mathtt{a}(t_1, \ldots, t_n)\rangle$ (using the constructor introduced by item *1.*), $a_1 ; a_2$ (introduced the constructor introduced by item *2.*) or $pf(t_1, \ldots, t_n)$ (where $pf \in \Pi$ is an action function with result-sort `action`), since a component signature does not contain any other operation with result sort `action`. We use this property for instance in the definition of new symbols in notation 3.46.

3.44 Example. *Considering the component signature for the component* C *presented in section 3.3.2, the following is an action expression describing the actions to execute upon a click on the* Increment*-button in a counter window:*

$$\begin{array}{l}\big\langle \mathsf{C},\ c := \mathit{cnt}\big(\mathit{succ}(\mathit{get\_val}(c\uparrow)),\ \mathit{get\_wins}(c\uparrow)\big)\big\rangle ; \\ \mathit{refresh\text{-}windows}\big(\mathit{int\text{-}of\text{-}nat}(\mathit{get\_val}(c\uparrow)),\ \mathit{get\_wins}(c\uparrow)\big)\end{array} \tag{3.18}$$

where the constant c *represents the name of the counter, i.e.,* c *is of the sort* $\mathtt{Name}(\mathit{Cnt})$*.*

In the sequel, we consider an extended definition of component terms (see equation (3.13)), where the constructors introduced in definition 3.43 are allowed in the construction of component terms.

The semantics of action functions is defined by means of (special) rewrite rules which we call p-rules. Below, we present the definition of p-rules along with the definition of process expressions (see definition 3.52). Action expressions which contain no application of action functions or translations, *i.e.*, action expressions that are sequences of pairs of storenames and (well-formed) calls to elementary actions, are said to be in *normal form*. Informally, the operational semantics of action expressions describes the execution of an action in two steps: before an action expression is executed, it is first reduced to normal form. This is also the reason why we do not introduce the sequential combination of action expressions as an action function, in order to emphasise its special operational behaviour.

3.45 Notation. *For a component signature* $\mathrm{C}\Sigma$ *and a family of variables* X *we denote by* $\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, X)$ *the set of action expressions in normal form, i.e., containing no calls to action functions (*$\in \Pi$*) or translations (*$\in Trans$*).*

Finally, to denote the set action expressions in normal form where all elementary actions are paired with the storename sn*, we write* $\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, X, sn)$*.*

The extension of the notation for new symbols introduced by an elementary action to action expressions needs to incorporate the storename of the store where the symbols are introduced. Thus, instead of a single signature, the new symbols introduced by an action expression are a (SN-indexed) family of signatures.

3.46 Notation. *The (*SN*-indexed) family of signatures of* new symbols *introduced by an action expression is defined inductively as follows:*

$$\mathcal{N}_{sn}\big(\langle sn,\ \mathsf{a}(t_1, \ldots, t_n)\rangle\big) \stackrel{\text{def}}{=} \mathcal{N}\big(\mathsf{a}(t_1, \ldots, t_n)\big) \tag{3.19a}$$

$$\mathcal{N}(a_1\,;a_2) \stackrel{\text{def}}{=} \mathcal{N}(a_1) \cup \mathcal{N}(a_2) \tag{3.19b}$$

$$\mathcal{N}\big(pf(t_1, \ldots, t_n)\big) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \mathcal{N}(t_i) \tag{3.19c}$$

Equations (3.19a) and (3.19b) express the intuition that the set of new symbols introduced by a sequence of elementary actions is the union[24] of the new symbols introduced by the actions separately. An informal motivation of equation (3.19c) is that the set of new symbols introduced by an action expression $pf(t_1, \ldots, t_n)$ (which is not in normal form) depends on the arguments of the action function pf, which in turn might not be known during the analysis of the process using this action expression. We give further motivation (and counter examples, see examples 3.53 and 3.54) for equation (3.19c) below, when we define the p-rules that specify the semantics of action functions in section 3.5.2.

The following notation gives the conditions governing the use of freshly introduced symbols in action expressions, namely that symbols have to be introduced *before* they are used.

[24] In fact, we are interested in the disjoint union, which is a consequence of the sensible use of new symbols (see notation 3.47).

3.47 Notation. *An action expression is said to* be sensible *if symbols are not used* before *they are introduced. We define the corresponding property by structural induction on action expressions:*

- *all calls to elementary actions as well as all applications of action functions are sensible and*
- *a sequential composition of action expressions $a_1 ; a_2 \in \mathcal{A}(\mathrm{C}\Sigma, X)$ is sensible if both $a_1 \in \mathcal{A}(\mathrm{C}\Sigma, X)$ and $a_2 \in \mathcal{A}(\widehat{\mathrm{C}\Sigma}, X)$ are sensible.*

  The component signature $\widehat{\mathrm{C}\Sigma}$ corresponds to the enrichment of the component signature $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ with the (SN-indexed family) of signatures of new symbols $\mathcal{N}(a_1) = \left\{ \widehat{\Sigma}_{sn} \,\middle|\, sn \in \mathit{SN} \right\}$ introduced by the action expression a_1:

$$\widehat{\mathrm{C}\Sigma} \stackrel{\mathsf{def}}{=} \left\langle (\Sigma \uplus \widehat{\Sigma}_{\widehat{sn}}), \mathrm{M}\Sigma, A, \widehat{I}, E, \mathit{Trans}, P, \Pi \right\rangle \tag{3.20a}$$

  where $\widehat{sn}$ denotes the storename of the local component, and where the enriched imported symbols $\widehat{I}$ are defined as follows:

$$\widehat{I} \stackrel{\mathsf{def}}{=} \left\{ \mathrm{I}\Sigma_{sn} = \left\langle (\Sigma_{sn} \uplus \widehat{\Sigma}_{sn}),\ \mathrm{M}\Sigma_{\mathcal{L}_{sn}},\ A_{sn} \right\rangle \,\middle|\, sn \in (\mathit{SN} \smallsetminus \{\widehat{sn}\}) \right\} \tag{3.20b}$$

Obviously, a single elementary action is sensible. The reason why an application of an action function is a sensible action expression is that the rules defining action functions are restricted in this sense, as we show along with the definition of action (and process) functions by p-rules (see definition 3.52). Notice that new symbols can be used in subsequent actions.

We conclude this section with the definition of a guarded actions, which are defined as pairs of a guard and an action expression.

3.48 Definition (guarded action). *Let $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ be a component signature (for a declarative lanagge $\mathcal{L}$ and a storename $\widehat{sn}$) and X a (PS-indexed) family of (sets of) variables, where PS is defined according to equation* (3.11a)*. Then we define a* guarded action *as a pair $[g \Rightarrow a]$, consisting of*

- *a* guard *g, i.e., a term of sort* `Truth` *of the local store, i.e., $g \in T^{\mathcal{L}}_{\mathtt{Truth}}(\Sigma, X)$, and*
- *a sensible action expression $a \in \mathcal{A}(\mathrm{C}\Sigma, X)$.*

We note the set of guarded actions *(for $\mathrm{C}\Sigma$ and X) as $\mathcal{G}(\mathrm{C}\Sigma, X)$.*

Notice that according to definition 3.48 the guard of a guarded action is required to be a term in the store of the *local* component $\widehat{sn}$. This implies that while processes are allowed to execute actions on all stores in a system, they can only read, *i.e.*, access information, from the local store (through the guards). The motivation for this restriction is that the check of the validity of the guard and the execution of the action expression are a single (locally) *atomic* operation. By locally atomic we mean that

the (sub-) sequences of actions for a given store[25] have to be executed atomically, and that on the local store, the check of the validity of the guard, the evaluation of the action expression to normal form, the execution of the action and the sending of the sequences of actions to the remote stores form a single, atomic operation. If we would allow a guard to contain parts of different stores, than we would have to ensure *global* atomic execution, which is in our opinion not a natural abstraction when the stores (*i.e.*, components) are explicitly distributed[26].

The following notation extends the notation of free variables to action expressions and guarded actions.

3.49 Notations. *The (SN-indexed) family of signatures of* new symbols *introduced by a guarded action is the set of new symbols introduced by its action expression, i.e.,*

$$\mathcal{N}\big([g \Rightarrow a]\big) \stackrel{\text{def}}{=} \mathcal{N}(a) \tag{3.21}$$

The set of free variables *of a guarded action is defined as follows:*

$$\mathcal{V}([g \Rightarrow a]) \stackrel{\text{def}}{=} \big(\mathcal{V}(g) \cup \mathcal{V}(a)\big) \tag{3.22}$$

The new symbols introduced by a guarded action are distinguished from free variables. Notice also that the guard may not use the new symbols introduced by the action expression.

3.5.2 Process Expressions and Process Terms

In this section we define the notion of process expressions, which generalises the notion of process terms introduced in section 1.1.3.2. Roughly speaking, process expressions are defined as component terms of sort `process`. Basic process expressions are *guarded actions* (see definition 3.48) and *process calls*, *i.e.*, applications of a process $\mathrm{q} \in P$. Process expressions can be composed using the operators of process algebras mentioned in section 1.1.3.2, namely parallel ($\|$) and sequential ($;$) composition, nondeterministic choice ($+$) and choice with priority ($\oplus$). Last, but not least, the application of a process function (with result sort `process`) also yields a valid process expression.

3.50 Definition (process expression). *Let $\mathrm{C}\Sigma$ be a component signature and X an ($\overline{PS}$-indexed) family of (sets of) variables, where PS is defined according to equation* (3.11a)*. We define the set of* process expressions *$\mathcal{P}(\mathrm{C}\Sigma, X)$ as the set of component terms of sort* `process`*, i.e.,*

$$\mathcal{P}(\mathrm{C}\Sigma, X) \stackrel{\text{def}}{=} CT_{\texttt{process}}(\widetilde{\widetilde{\mathrm{C}\Sigma}}, X) \tag{3.23}$$

where the component signature $\widetilde{\widetilde{\mathrm{C}\Sigma}}$ is an enrichment of the component signature $\widetilde{\mathrm{C}\Sigma}$ as defined in definition 3.43 (recall that $\widetilde{\mathrm{C}\Sigma}$ is itself an enrichment of the component signature $\mathrm{C}\Sigma$ with constructors of the sort `action`*) with the following five* constructors *of the sort* `process`*:*

[25] In the case of a component composed of several subcomponents (see section 3.6), we require the atomic execution of the subsequences on the stores of the subcomponents.

[26] Global atomic execution of actions would also require the implementation of a global consensus, the implementation of which is difficult or even impossible in case of a single faulty process [FLP85].

1. $[\bullet \Rightarrow \bullet]$, *the constructor of guarded actions (see definition 3.48), is a constructor of profile* `Truth` $\rightarrow$ `action` $\rightarrow$ `process` *and*

2. $\bullet\,;\bullet$, $\bullet \parallel \bullet$, $\bullet + \bullet$ *and* $\bullet \oplus \bullet$ *are constructors of profile* `process` $\rightarrow$ `process` $\rightarrow$ `process`.

where we use the symbol "$\bullet$" as a placeholder for the parameters, since these operators are usually written in mixfix-syntax.

In the sequel, we consider an extended definition of component terms (see equation (3.13)), where the constructors introduced in definitions 3.43 and 3.50 are allowed in the construction of component terms.

Notice that the operator ";" is polymorphic, since it denotes sequential composition of both, action *and* process expressions.

Process expressions without application of process functions and translations are called *process terms*. Similar to action expressions, we also say that process terms are process expressions in *normal form*.

3.51 Notation. *For a component signature* $\mathrm{C}\Sigma$ *and a family of variables* X *we denote the set of* process term*s or process expressions in normal form, i.e., containing no application of a process function (*$\in \Pi$*) or translation (*$\in$ *Trans), by* $\mathcal{P}^{\mathcal{N}}(\mathrm{C}\Sigma, X)$.

Notice that the notion of process terms according to notation 3.51 is a refinement of the informal notion of process terms as introduced in section 1.1.3.2.

There are several possibilities to define the meaning of process functions. We suggest to describe the meaning of process-functions by means of p-rules, *i.e.*, rewrite rules similar to those of the declarative programming language presented in section 3.1.2 (see definition 3.9) and to the t-rules used for the specification of translation functions in section 3.4.2 (see definition 3.41). The main differences with respect to the notion of rules as introduced in definition 3.9 are that, on the one hand, we use component terms instead of terms, and on the other hand, similar to t-rules, we do not require a constructor discipline and allow general component terms in the left hand sides.

3.52 Definition (p-rule). *Let* $\mathrm{C}\Sigma$ *be a component signature for a set of storenames* $\mathcal{SN}$ *and* X *a family of variables. We define a* p-rule $\Pi\mathrm{R}$ *as a pair* $\mathsf{lhs} \rightarrow \mathsf{rhs}$ *such that*

- $\mathsf{lhs} = pf(t_1, \ldots, t_n) \in CT_s(\mathrm{C}\Sigma, X)$ *is the application of a process function* pf *(of arity* n*) to* n *component terms* t_i $(i \in \{1; \ldots; n\})$ *and*

- $\mathsf{rhs} \in CT_s\big(\mathrm{C}\Sigma,\ \mathcal{V}(\mathsf{lhs})\big)$ *is a component term of the same sort as* lhs*, namely* s.

The first two conditions are similar to those of definition 3.41 and define a general rewrite rule. Since we need to know the exact set of new symbols in order to check the correctness of a process definition, we consider only p-rules satisfying additional conditions. The following examples motivates these conditions.

3.53 Example. *Consider create-i-symbols, an action function of profile Nat* $\rightarrow$ `action` *(where the sort Nat is defined as in example 3.6) which creates as many new symbols as*

is indicated by its parameter. We could define create-i-symbols by the following rewrite rules:

$$\textit{create-i-symbols}(\textit{zero}) \;\to\; \mathsf{skip} \tag{3.24a}$$
$$\textit{create-i-symbols}\big(\textit{succ}(i)\big) \;\to\; \big(\langle \mathit{sn},\, \mathsf{new}(x,\, \mathit{Nat})\rangle\,;\; \textit{create-i-symbols}(i)\big) \tag{3.24b}$$

It is impossible to detect (statically, i.e., without executing the program) the number of new symbols that will be introduced by the action expression create-i-symbols(c) where c is a constant (of sort Nat) defined in the store, since we do not know the value of c. However, the new symbols are used in the sequel of the execution of the process executing create-i-symbols(c) (otherwise there would be no point in introducing them), so that we need to verify the correct use of these symbols in the remaining process. Obviously, we need to know the exact set of symbols in order to accomplish this verification.

In order to avoid the problems mentioned in example 3.53 we require that in a p-rule $\mathrm{IR} = \mathsf{lhs} \to \mathsf{rhs}$ defining the process function *pf* (*i.e.*, the profile of the process function *pf* is of the form $s_1 \to \ldots \to s_n \to \mathtt{action}$), we have that for all applications of an elementary action $\mathsf{a}(t_1, \ldots, t_m)$ occurring in rhs that for all storenames $\mathit{sn} \in \mathcal{SN}$ that $\mathcal{N}_{\mathit{sn}}\big(\mathsf{a}(t_1, \ldots, t_m)\big) = \langle \varnothing, \varnothing\rangle$. Obviously, p-rule (3.24b) violates this condition, and we consider the definition of *create-i-symbols* as incorrect.

3.54 Example. *As a second example, consider the action function two-times (the profile of which is* $\mathtt{action} \to \mathtt{action}$*) which takes an action expression as argument and executes it twice:*

$$\textit{two-times}(x) \;\to\; x\,;\; x \tag{3.25}$$

The evaluation of the action expression $\textit{two-times}\big(\langle \mathit{sn},\ \mathsf{new}(c,\, s)\rangle\big)$ *yields an action expression which introduces the same symbol two times. Nevertheless, the action expression* $\textit{two-times}\big(\langle \mathsf{Bell},\, \mathsf{ring}\rangle\big)$ *should be consider a correct description of ringing an alarm bell two times.*

In contrary to the p-rules defining of *create-i-symbols*, the p-rule defining *two-times* should not be considered as illegal, since there are situations where it is useful. However, in order to avoid the potential problems related to p-rules similar to the p-rule (3.25) defining *two-times*, the use of an action which introduces new symbols as a parameter for a process function *pf* should be avoided if the corresponding formal parameter appears more than once in one of the p-rules defining *pf*.

The additional restrictions motivated by the examples 3.53 and 3.54 justify equation (3.19c), which defines the signature of new symbols introduced by an application of an action function as the union of the new signatures introduced by its arguments.

A process expression is said to be in *restricted form* if it contains neither a guarded action, nor an occurrence of $\oplus$ or $+$. If the rhs of a p-rule is a process expression in restricted form, the p-rule is said to be in restricted form as well. When all p-rules defining a process function *pf* are in restricted form, we call the process function *pf restricted* process function, otherwise we call *pf non-restricted*. We can deduce easily that when a process expression in restricted form does not contain calls to non-restricted process functions, its normal form is a restricted process term in the sense

of section 1.1.3.2. In the following, we note (for a component signature $C\Sigma$ and a set of variables X) the set of restricted process expressions that do not contain calls to non-restricted process functions as ${}^{r}\mathcal{P}(C\Sigma, X)$.

3.55 Example. *Considering the component signature of the example of the multiple counters presented in section 3.3.2, the following process expression (which is also a process term) is not* restricted*, since it uses guarded actions:*

$$\left[\text{TRUE} \Rightarrow \begin{array}{l} \langle \mathsf{C},\ \mathsf{new}(c,\ Cnt)\rangle\,;\,\langle \mathsf{C},\ c := zero\rangle; \\ \langle \mathsf{C},\ \mathsf{new}(e,\ Evt\_List)\rangle\,;\,\langle \mathsf{C},\ e := nil_{Evt}\rangle \end{array}\right];\ \text{create\_win}(c,\ e) \tag{3.26}$$

For technical reasons (*e.g.*, avoiding dead-locks) we have to ensure that process expressions always denote unique processes, that is to say, the evaluation of the process-functions has to terminate, and to return, deterministically, exactly one (restricted) process term. These constraints (on process functions) are similar to the constraints on the functions defining (elementary) actions, see section 3.2.1. As mentioned there, these conditions could either be enforced by syntactical restrictions, or by the use of effective analyses of the definitions.

The following examples show the use of process expressions and p-rules in order to simplify the description of processes by the use of functional abstractions.

3.56 Example. *Consider the sort of lists (of numbers), defined by the two constructors, namely cons which takes a number and a list and returns a list and nil, the empty list. We define a process function* $\hat{\mathrm{q}}$*, which takes a list of numbers* l *and a process expression* p *as arguments and evaluates to the parallel composition of the process expression* p *and a process* $\mathrm{q}(i)$ *for every* i *in the list* l*, by the following p-rules:*

$$\begin{array}{rcl} \hat{\mathrm{q}}(nil,\ p) & \rightarrow & p \\ \hat{\mathrm{q}}\big(cons(i,\ l),\ p\big) & \rightarrow & \mathrm{q}(i) \parallel \hat{\mathrm{q}}(l,\ p) \end{array}$$

Using the process function $\hat{\mathrm{q}}$*, we can abbreviate the process expression*

$$\mathrm{q}(1) \parallel \mathrm{q}(2) \parallel \mathrm{q}(3) \parallel \mathrm{q}(4) \parallel \mathrm{q}(5) \parallel p \qquad \textit{to} \qquad \hat{\mathrm{q}}([1;2;3;4;5],\ p)$$

where we use the (classical) shorthand $[1;2;3;4;5]$ *for the list*

$$cons(1,\ cons(2,\ cons(3,\ cons(4,\ cons(5,\ nil)))))$$

In the example of the multiple counters, the use of a process function allows to express the update of all windows of a counter with a single action expression.

3.57 Example. *The process function refresh-windows in the example of the multiple counter (see section 3.3.2) is defined as follows:*

$$refresh\text{-}windows(i,\ nil_{Wid}) \rightarrow \langle \mathsf{C},\ \mathsf{skip}\rangle \tag{3.27a}$$

$$\begin{array}{l} refresh\text{-}windows\big(i,\ cons_{Wid}(w,\ ws)\big) \\ \quad \rightarrow \big\langle \mathsf{X},\ \mathsf{refresh\text{-}win}\big(i,\ string\text{-}of\text{-}wid(w)\big)\big\rangle\,;\ refresh\text{-}windows(i,\ ws) \end{array} \tag{3.27b}$$

3.5.3 Process Definitions

As already mentioned in the introduction, a process is defined by a set (ordered by priority) of rules or clauses, each of which consists of a guarded action and a restricted process expression.

3.58 Definition (process definition). *Let* $\mathrm{C}\Sigma = \langle \Sigma, \mathrm{M}\Sigma_{\mathcal{L}}, A, I, E, \mathit{Trans}, P, \Pi \rangle$ *be a component signature. A* process definition *for the process* $\mathrm{q} \in P$ *is a sentence of the following form:*

$$\mathrm{q}(x_1, \ldots, x_m) \Leftarrow \bigoplus_{i=1}^{n} \big([g_i \Rightarrow a_i] \,;\, {}^{r}p_i \big) \tag{3.28}$$

where (for every $i \in \{1; \ldots; n\}$*, with* $n > 0$*):*

- $[g_i \Rightarrow a_i] \in \mathcal{G}\big(\mathrm{C}\Sigma, \{x_1; \ldots; x_m\}\big)$ *is a guarded action for the local component, i.e.,* $g_i \in T^{\mathcal{L}}_{\mathtt{Truth}}\big(\Sigma, \{x_1; \ldots; x_m\}\big)$ *and a sensible action expression* a_i*, i.e.,* $a_i \in \mathcal{A}\big(\mathrm{C}\Sigma, \{x_1; \ldots; x_m\}\big)$ *(see definition 3.48) and*

- ${}^{r}p_i$ *is a restricted process expression, i.e.,* ${}^{r}p_i \in {}^{r}\mathcal{P}\big(\widehat{\mathrm{C}\Sigma}, \{x_1; \ldots; x_m\}\big)$ *where the component signature* $\widehat{\mathrm{C}\Sigma}$ *corresponds to the enrichment of the component signature* $\mathrm{C}\Sigma$ *with the signatures of new symbols* $\mathcal{N}([g_i \Rightarrow a_i])$ *introduced by the guarded action* $[g_i \Rightarrow a_i]$ *(see notation 3.47 for a formal definition of* $\widehat{\mathrm{C}\Sigma}$*).*

Intuitively, the operational behaviour of a process call $\mathrm{q}(t_1, \ldots, t_m)$ is similar to the alternative construct of the guarded command language of [Dij75]. That is to say, we have to evaluate which of the guards of the rules of the process definition for q are valid, and then to choose among them the rule with the highest priority. Choosing a rule means to atomically execute the sequence of elementary actions obtained by evaluating the action expression associated with the guard and afterwards to behave like the normal form of the associated (restricted) process expression.

3.59 Example. *The process definitions of the store of the component* C *for the example of the multiple counters (see section 3.3.2) are shown in table 3.4 (the rewrite rules defining the functions get_val, get_wins, head and tail of the store of* C *are shown in table 3.5 on page 118).*

The process controlling a counter-window is cnt_ctrl*, which takes two parameters: the name c of the associated counter and the name e of the event-queue to which the window system sends all the events occurring in the window being controlled (an example of a counter window is shown in figure 3.5). Intuitively,* cnt_ctrl *handles the events in the list e one by one. For instance, an event corresponding to a click on the* Increment*-button removes the event from the list e, increments the counter c (i.e., assigns to c the pair of the successor of the old value and the old list of windows) and triggers the redrawing of all windows associated to c (using the process function refresh-windows). Then the process continues to execute* cnt_ctrl*. An event representing a click on the* Copy*-button removes the event from the list e and creates and initialises a new counter* c' *and a new event-list* e'*. Then the process continues in parallel with a new process which creates a new window for the new counter. The handling of clicks on the* Link*-button is similar to the handling of* Copy*.*

$$
\begin{array}{l}
\mathrm{cnt\_ctrl}(c,\ e) \Leftarrow \\
\quad \left[\mathit{head}(e\uparrow) = \mathit{increment} \Rightarrow \begin{array}{l} \langle \mathsf{C},\ e := \mathit{tail}(e\uparrow)\rangle; \\ \langle \mathsf{C},\ c := \mathit{cnt}\big(\mathit{succ}(\mathit{get\_val}(c\uparrow)),\ \mathit{get\_wins}(c\uparrow)\big)\rangle; \\ \mathit{refresh\text{-}windows}\big(\mathit{int\text{-}of\text{-}nat}(\mathit{succ}(\mathit{get\_val}(c\uparrow))),\ \mathit{get\_wins}(c\uparrow)\big) \end{array} \right]; \\
\quad \mathrm{cnt\_ctrl}(c,\ e) \\
\oplus \left[\mathit{head}(e\uparrow) = \mathit{copy} \Rightarrow \begin{array}{ll} \langle \mathsf{C},\ e := \mathit{tail}(e\uparrow)\rangle; & \\ \langle \mathsf{C},\ \mathsf{new}(c',\ \mathit{Cnt})\rangle; & \langle \mathsf{C},\ c' := \mathit{cnt}\big(\mathit{get\_val}(c\uparrow),\ \mathit{nil}_{\mathit{Wid}}\big)\rangle; \\ \langle \mathsf{C},\ \mathsf{new}(e',\ \mathit{Evt\_List})\rangle; & \langle \mathsf{C},\ e' := \mathit{nil}_{\mathit{Evt}}\rangle \end{array} \right]; \\
\quad \mathrm{cnt\_ctrl}(c,\ e) \parallel \mathrm{create\_win}(c',\ e') \\
\oplus \Big[\mathit{head}(e\uparrow) = \mathit{link} \ \Rightarrow\ \langle \mathsf{C},\ e := \mathit{tail}(e\uparrow)\rangle\,;\, \langle \mathsf{C},\ \mathsf{new}(e',\ \mathit{Evt\_List})\rangle\,;\, \langle \mathsf{C},\ e' := \mathit{nil}_{\mathit{Evt}}\rangle \Big]\,; \\
\quad \mathrm{cnt\_ctrl}(c,\ e) \parallel \mathrm{create\_win}(c,\ e') \\
\\
\mathrm{create\_win}(c,\ e) \Leftarrow \\
\quad \Big[\textsc{true} \ \Rightarrow\ \langle \mathsf{X},\ \mathsf{add\text{-}win}(\mathit{get\_val}(c\uparrow),\ \mathit{to\text{-}string}(c),\ \mathit{to\text{-}string}(e))\rangle \Big]\,;\ \mathrm{cnt\_ctrl}(c,\ e)
\end{array}
$$

Table 3.4: Process Definitions for the Component C

The process create_win *takes the names of a counter c and an event-list e as arguments and sends a call to the elementary action* add-win *to the component* X *and subsequently behaves as the process* cnt_ctrl *controlling the new window for the counter c.*

In analogy to logic programming, where the free variables of a clause or rule have to be *renamed* each time the clause is used (see section 3.1.2.2), we have to rename the new symbols introduced by elementary actions (see section 3.2.2.4) in the rules of a process definition. However, in contrary to the variant of a rule (see definition 3.15), we need to rename the *symbols*, *i.e.*, the names of constants and functions, instead of variables (which in turn need not to be renamed). Similar to the renamed rules in section 3.1.2, we call a renamed rule of a process definition a *variant*.

3.60 Definition (p-variant). *Let* $b = \big([g \Rightarrow a]\,;\ \vec{p}\big)$ *be a rule or clause of a process definition. A* p-variant *of b is defined as the rule obtained from b by replacing consistently all symbols in* $\mathcal{N}(a)$*, i.e., the signatures of new symbols introduced by the action expression a, by fresh symbols. In the sequel, we denote "rename" the operation which returns a variant for a given rule.*

The following example illustrates definition 3.60.

3.61 Example. *Consider the process definition of* cnt_ctrl *in the example of the multiple counters as shown in table 3.4. Every time the* Link*-button in a particular window is clicked, we have to create a* new *counter window. Therefore, the event-list associated to the new window has to get a* fresh *name, i.e., we have to rename* e'*. A possible renaming of the rule of* cnt_ctrl *handling clicks on* Link *is the following (where* e' *has*

been renamed to $\tilde{\boldsymbol{e}}$*):*

$$\Big[head(e\uparrow) = link \;\Rightarrow\; \langle \mathsf{C},\, e := tail(e\uparrow)\rangle\,;\, \big\langle \mathsf{C},\, \mathsf{new}(\tilde{\boldsymbol{e}},\, Evt\_List)\big\rangle\,;\, \big\langle \mathsf{C},\, \tilde{\boldsymbol{e}} := nil_{Evt}\big\rangle\Big]\,;\\ \mathrm{cnt\_ctrl}(c,\, e) \parallel \mathrm{create\_win}(c,\, \tilde{\boldsymbol{e}}) \tag{3.29}$$

3.6 Components and Systems

We conclude the presentation of our computation model with the definition of components and their composition in order to construct systems.

3.6.1 Components

In the preceding sections we have presented how the different symbols occurring in a component signature are defined. All these different definitions together are called a *component*. As already mentioned in section 3.3, a component is defined as part of a system. Thus the following definition depends on a set of storenames $\mathcal{SN}$ representing the other parts of the system.

3.62 Definition (component). *Let* $\mathcal{SN}$ *be a set of storenames. A* component *is defined as an eight-tuple*

$$\mathcal{C} \stackrel{\mathsf{def}}{=} \langle \widehat{sn},\, \mathrm{C}\Sigma,\, \mathcal{R},\, \mathfrak{A},\, \mathcal{T}r,\, \mathcal{R}^{\mathrm{p}},\, \Pi\!\!\mathcal{R},\, p^{\mathtt{i}}\rangle \tag{3.30}$$

where

- $\widehat{sn} \in \mathcal{SN}$ *is the* storename *(or* component-name*) of the component,*
- $\mathrm{C}\Sigma = \langle \Sigma,\, \mathrm{M}\Sigma_{\mathcal{L}},\, A,\, I,\, E,\, Trans,\, P,\, \Pi\rangle$ *is a* component signature *with respect to the set of storenames* $\mathcal{SN}$ *and the storename* $\widehat{sn}$ *(see definition 3.36),*
- $\mathcal{R}$ *is a set of* rules *or* formulæ *such that* $F \stackrel{\mathsf{def}}{=} \langle \Sigma,\, \mathcal{R}\rangle$ *is a* store*, also called the* initial store *(see definition 3.1),*
- $\mathfrak{A}$ *is a set of* actions definitions *for the elementary actions* A*,*
- $\mathcal{T}r$ *is a set of* translation definitions *for the translation symbols* $Trans$*, i.e., a set of t-rules (see definition 3.41),*
- $\mathcal{R}^{\mathrm{p}}$ *is a set of* process definitions *for the processes* P *(see definition 3.58),*
- $\Pi\!\!\mathcal{R}$ *is a set of* p-rules *defining the process functions* Π *(see definition 3.52) and*
- $p^{\mathtt{i}} \in {}^{r}\mathcal{P}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing)$ *is a closed restricted process expression (i.e., a process expression containing no free variables), called the* initial process expression *or* initial process term.

According to definition 3.62, a component is characterised by its component name or storename, its component signature with the corresponding definitions and its initial process term. The different symbols introduced in a component signature are defined by the store (defining the symbols of the signature for instance by (conditional) rewrite rules as in definition 3.9), the action definitions (which might be defined as in the examples of section 3.2.2), the translation definitions (defining the translation from the store of the component into other components by means of t-rules, see definition 3.41) and the definitions of processes (see definition 3.58) and process functions (by p-rules, see definition 3.52). The symbols defined in the imported signatures (in the component signature) are left without definition by the component, since they are imported from other components, which have to provide the necessary definitions. The evolution of the store of the component is defined by the execution of its initial process term on its initial store. We require the initial process term to be closed in order to have a concrete process to execute. Notice that it is not restrictive to require the initial process expression to be in restricted form since we can transform a general process expression into a restricted process expression by introducing some additional processes.

Different kinds of definitions can be distinguished among the definitions of a component, namely those, which are *static*, *i.e.*, which do not change during the execution, and those which are *dynamic*. On the one hand, the storename $\widehat{sn}$, the imported signatures I, exports E, as well as the actions A, translations *Trans*, processes P and process functions Π with their related definitions (*i.e.*, $\mathfrak{A}$, $\mathcal{T}r$, $\mathcal{R}^{\mathrm{p}}$ and $\Pi\mathcal{R}$) are static. On the other hand, *i.e.*, the signature Σ and the set of rules $\mathcal{R}$, are called dynamic, since they may change due to the execution of actions. The evolution of the dynamic part is described by the execution of the initial process term on the initial store. We also call the pair of the initial process term and the initial store the *initialisation* of a component .

3.63 Example. *Consider the example of the multiple counters which has been presented together with the component signature of the component* C *in section 3.3.2. Table 3.5 gives the rewrite rules defining the functions which are not part of the signature* Σ_{nat} *the rules for which are defined in table 3.1. The first two rules of table 3.5 are straightforward definitions of access functions for the sort Cnt, and the remaining rules are classical definitions for the partial functions head and tail. The only action used on the store of the component* C *is assignment, which has been defined in section 3.2.2.3. The translation int-of-nat has been defined in example 3.42*[27]*, the process definitions are shown in table 3.4 and the process function refresh-windows is specified in example 3.57.*

It remains thus the specification of the initial process term. Consider the process expression of example 3.55, equation 3.26, which is not in restricted form. In order to use this term as initial process term, we have to wrap it into a new (parameterless) process, say start ($\in P_{\mathtt{process}}$)*. Hence, if we define* start *by the process definition*

$$\mathrm{start} \Leftarrow \left[\mathrm{TRUE} \Rightarrow \begin{array}{l} \langle \mathsf{C},\ \mathsf{new}(c,\ Cnt)\rangle\,;\,\langle \mathsf{C},\ c := zero\rangle; \\ \langle \mathsf{C},\ \mathsf{new}(e,\ Evt\_List)\rangle\,;\,\langle \mathsf{C},\ e := nil_{Evt}\rangle \end{array}\right];\ \mathrm{create\_win}(c,\ e) \quad (3.31)$$

we can define the initial process term of the component C *as a call to the process* start*.*

[27] The translation of *Wid*s (which are represented as strings) to `string`s should be obvious.

$$get\_val\big(cnt(v,\ ws)\big) \to v \qquad (R_{get\_val})$$
$$get\_wins\big(cnt(v,\ ws)\big) \to ws \qquad (R_{get\_wins})$$
$$head_{Evt}\big(cons_{Evt}(e,\ es)\big) \to e \qquad head_{Wid}\big(cons_{Wid}(w,\ ws)\big) \to w \qquad (R_{head})$$
$$tail_{Evt}\big(cons_{Evt}(e,\ es)\big) \to es \qquad tail_{Wid}\big(cons_{Wid}(w,\ ws)\big) \to ws \qquad (R_{tail})$$

Table 3.5: Rules for the Store of the Component C

3.6.2 Composing Components: Systems

We model a system simply as a (finite) set of components such that the exported and imported signatures match. However, as shown in figure 3.1, a component itself might also be the combination of several components. In this section we present both, the construction of systems from a set of components, as well as the combination of components, which allows to consider a system as a component.

Intuitively, a system is constructed by putting together the components corresponding to the storenames with respect to which the components have been defined.

3.64 Definition (system). *Let $\mathcal{S}$ be a set of components*

$$\mathcal{S} \stackrel{\text{def}}{=} \{\, \mathcal{C}_{sn_1}; \ldots; \mathcal{C}_{sn_n} \,\} \tag{3.32a}$$

and consider the associated set of storenames $\mathbb{SN}$, i.e., $\mathbb{SN} \stackrel{\text{def}}{=} \{sn_1; \ldots; sn_n\}$. $\mathcal{S}$ is called a system *if for all storenames $sn \in \mathbb{SN}$, the component $\mathcal{C}_{sn}$ is defined (with respect to the set of storenames $\mathbb{SN}$) as*

$$\mathcal{C}_{sn} \stackrel{\text{def}}{=} \big\langle sn,\ \mathrm{C}\Sigma_{sn},\ \mathcal{R}_{sn},\ \mathfrak{A}_{sn},\ \mathcal{T}r_{sn},\ \mathcal{R}^{\mathrm{p}}_{sn},\ \Pi\!\mathcal{R}_{sn},\ p^{\mathtt{i}}_{sn} \big\rangle \tag{3.32b}$$

with component signatures

$$\mathrm{C}\Sigma_{sn} \stackrel{\text{def}}{=} \big\langle \Sigma_{sn},\ \mathrm{M}\Sigma_{sn},\ A_{sn},\ I_{sn},\ E_{sn},\ Trans_{sn},\ P_{sn},\ \Pi_{sn} \big\rangle \tag{3.32c}$$

such that for all pairs of storenames sn_1, $sn_2 \in \mathbb{SN}$ with $sn_1 \neq sn_2$

$$\big(I_{sn_1}\big)_{sn_2} \subseteq E_{sn_2} \tag{3.33}$$

Notice that a system is *complete* in the sense that all components of the system are specified, since all components are required to be defined with respect to the same set of storenames[28]. Thus, in principle, every component of a system knows about all other components. However, this does not imply that all components have necessarily to interact with each other, but only that such an interaction should be possible. The requirement of matching interfaces is expressed by condition (3.33) which implies that (for any pair of storenames sn_1 and sn_2) the component signature $\mathrm{C}\Sigma_{sn_1}$ (of the component $\mathcal{C}_{sn_1}$) is only allowed to import a subset from the exported symbols E_{sn_2} of the component signature $\mathrm{C}\Sigma_{sn_2}$ (of the component $\mathcal{C}_{sn_2}$).

[28] A straightforward generalisation might be to require that all components are defined with respect to a subset of a common set of storenames.

3.65 Example. *The complete system for the example of the multiple counters is the composition of two components, namely* C *which models the control of the counters and* X *which models the display of the windows. The component* C *is presented in example 3.63. We do not specify the component* X *completely, but consider it as a predefined component (like the actual physical screen used for the display of the windows), the interface of which is presented in section 3.3.2 in form of the imported signature (see equations* (3.14d) *and* (3.14e)*). Intuitively, the store of* X *describes a theory of counter windows, describing all relevant properties of the different windows, as for instance their location on the screen, their name, and the event queue where the messages corresponding to clicks have to be sent to. A process running on the component* X *ensures that the events occurring on the display lead to the execution of the appropriate actions, i.e., the adding of a message corresponding to the button in the event queue associated to the window.*

We conclude the presentation of our computation model with the introduction of a second kind for the combination of components, which allows to consider an incomplete system as a single component. By incomplete system, we mean a set of components which is a subset of the components of a system. Intuitively, by taking the disjoint union of the different parts of two components, we obtain the combination of the components.

3.66 Definition (Combination of Components: $|||$). *Let SN be a set of storenames, and consider two components $\mathcal{C}_1$ and $\mathcal{C}_2$ defined with respect to SN as*

$$\begin{aligned} \mathcal{C}_i &\stackrel{\text{def}}{=} \langle sn_i,\, \mathbb{C}\Sigma_i,\, \mathcal{R}_i,\, \mathfrak{A}_i,\, \mathcal{T}r_i,\, \mathcal{R}_i^{\mathrm{p}},\, \Pi\!\!\Pi\mathcal{R}_i,\, p_i^{\mathrm{i}} \rangle \\ &\quad \textit{with component signatures} \\ \mathbb{C}\Sigma_i &\stackrel{\text{def}}{=} \langle \Sigma_i,\, \mathbb{M}\Sigma_{\mathcal{L}_i},\, A_i,\, I_i,\, E_i,\, \mathit{Trans}_i,\, P_i,\, \Pi_i \rangle \end{aligned} \tag{3.34a}$$

(for $i \in \{1;2\}$) such that $\mathcal{C}_1$ and $\mathcal{C}_2$ are a part of a system, i.e., we have that

$$\big(I_{sn_1}\big)_{sn_2} \subseteq E_{sn_2} \qquad \textit{and} \qquad \big(I_{sn_2}\big)_{sn_1} \subseteq E_{sn_1} \tag{3.34b}$$

Then we define for a fresh storename $\widehat{sn}$ (i.e., $\widehat{sn} \notin SN$) the combination *of $\mathcal{C}_1$ and $\mathcal{C}_2$ as the following component with respect to the storenames $\big(SN \setminus \{sn_1;\, sn_2\}\big) \uplus \{\widehat{sn}\}$*

$$\begin{aligned} &\mathcal{C}_1 \,|||\, \mathcal{C}_2 \stackrel{\text{def}}{=} \\ &\quad \big\langle \widehat{sn},\, \mathbb{C}\Sigma_1 \,\widetilde{\uplus}\, \mathbb{C}\Sigma_2,\, \mathcal{R}_1 \uplus \mathcal{R}_2,\, \mathfrak{A}_1 \uplus \mathfrak{A}_2,\, \mathcal{T}r_1 \uplus \mathcal{T}r_2,\, \mathcal{R}_1^{\mathrm{p}} \uplus \mathcal{R}_2^{\mathrm{p}},\, \Pi\!\!\Pi\mathcal{R}_1 \uplus \Pi\!\!\Pi\mathcal{R}_2,\, p_1^{\mathrm{i}} \parallel p_2^{\mathrm{i}} \big\rangle \end{aligned} \tag{3.35}$$

where (using the auxiliary operators $\overset{\mathsf{m}}{\uplus}$, $\overset{\mathsf{i}}{\cup}$ and $\overset{\mathsf{t}}{\uplus}$) the combination of the component

signatures $\mathbb{C}\Sigma_1 \widetilde{\uplus}\, \mathbb{C}\Sigma_2$ *is defined as follows*

$$\mathbb{C}\Sigma_1 \widetilde{\uplus}\, \mathbb{C}\Sigma_2 \stackrel{\text{def}}{=} \left\langle \Sigma_1 \uplus \Sigma_2,\ \mathbb{M}\Sigma_{\mathcal{L}_1} \overset{\mathsf{m}}{\uplus} \mathbb{M}\Sigma_{\mathcal{L}_2},\ A_1 \uplus A_2,\ I_1 \overset{\mathsf{i}}{\cup} I_2,\ E_1 \uplus E_2,\ \mathit{Trans}_1 \overset{\mathsf{t}}{\uplus} \mathit{Trans}_2,\ P_1 \uplus P_2,\ \Pi_1 \uplus \Pi_2 \right\rangle \tag{3.36a}$$

$$\mathbb{M}\Sigma_{\mathcal{L}_1} \overset{\mathsf{m}}{\uplus} \mathbb{M}\Sigma_{\mathcal{L}_2} \stackrel{\text{def}}{=} \begin{cases} \mathbb{M}\Sigma_{\mathcal{L}_1} & \textit{if } \mathcal{L}_1 = \mathcal{L}_2 \\ \mathbb{M}\Sigma_{\mathcal{L}_1} \uplus \mathbb{M}\Sigma_{\mathcal{L}_2} & \textit{otherwise} \end{cases} \tag{3.36b}$$

$$I_1 \overset{\mathsf{i}}{\cup} I_2 \stackrel{\text{def}}{=} (I_1 \setminus \mathbb{E}\Sigma_{sn_2}) \cup (I_2 \setminus \mathbb{E}\Sigma_{sn_1}) \tag{3.36c}$$

$$\mathit{Trans}_1 \overset{\mathsf{t}}{\uplus} \mathit{Trans}_2 \stackrel{\text{def}}{=} \left\{\, \mathit{Tr}^1_{sn} \uplus \mathit{Tr}^2_{sn} \;\middle|\; sn \in (\mathcal{SN} \setminus \{sn_1; sn_2\}),\ \mathit{Tr}^1_{sn} \in \mathit{Trans}_1 \textit{ and } \mathit{Tr}^1_{sn} \in \mathit{Trans}_2 \right\} \tag{3.36d}$$

Definition 3.66 allows to consider incomplete systems as components. Indeed, the system of the two components $\mathcal{C}_1$ and $\mathcal{C}_2$ is, in general, incomplete, since both components might interact with other components the storenames of which are in the set $\mathcal{SN} \setminus \{sn_1; sn_2\}$. Thus we have obviously that condition (3.34b) is a particular case of condition (3.33). On the other hand, the combination $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ can be considered as a new component, which *hides* the internal separation in two components[29]. Notice that the component $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ has the same interface to its a environment as a component defined directly according to definition 3.62, since in both cases the exported symbols are a signature, a meta-signature and a family of actions.

Roughly speaking, the definitions of $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ are the disjoint union of the definitions of $\mathcal{C}_1$ and $\mathcal{C}_2$, together with the parallel composition of the initial process terms. The *disjoint* union ensures that we can always determine the component where a symbol was originally defined in[30]. However, since $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ hides the separation between $\mathcal{C}_1$ and $\mathcal{C}_2$, their mutual interfaces have to be omitted, which is expressed formally by the equations (3.36). In the case that the two components use the same declarative languages for the description of their stores, we do not need to duplicate the corresponding meta-signatures[31]. Notice that if the meta-signatures are different, the signatures are of different kinds. This is no problem, since the disjoint union ensures that the definitions of the stores do not interfere. Since both components are defined with respect to the same set of storenames $\mathcal{SN}$, the imported symbols (from a component sn_3) have to be parts of a common set (namely the set of symbols exported by sn_3), such that we use in equation (3.36c) the union instead of the disjoint union. Furthermore, as $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ is defined with respect to a set of storenames which does not contain sn_1 and sn_2, we remove the corresponding "internal" imports, *i.e.*, the imports from $\mathcal{C}_2$ by $\mathcal{C}_1$ (and vice versa, see condition (3.36c)). Similarly, the "internal" translations, *i.e.*, translations needed in the interaction between $\mathcal{C}_1$ and $\mathcal{C}_2$ are hidden from the outside by condition (3.36d).

Notice further, that definition 3.66 does not allow the *extension* of the component $\mathcal{C}_1 \,|||\, \mathcal{C}_2$ with additional processes or functions. This restriction reflects the idea that

[29] The extension of definition 3.66 to more than two components is straightforward.

[30] In the case that both components define symbols of the same name, we can easily ensure the disjointness by adding a prefix or suffix encoding the original storename.

[31] To simplify, we suppose that a given declarative language has an unique meta-signature. This requirement could be weakened, necessitating a more complicated definition of $\overset{\mathsf{m}}{\uplus}$.

the operator $|||$ represents a means to hide the internal structure of a part of a system from the outside.

The operational behaviour of the component $\mathcal{C}_1 ||| \mathcal{C}_2$ is defined by the operational behaviour of the components $\mathcal{C}_1$ and $\mathcal{C}_2$, extended with a simple "dispatcher" for the actions that are sent to the component. That is to say, from the point of view of $\mathcal{C}_1$ and $\mathcal{C}_2$, the combination is completely transparent: they interact directly with each other as well as with the other component of the system as usual, since they are not aware of the existence of the storename $\widehat{sn}$ of $\mathcal{C}_1 ||| \mathcal{C}_2$. On the other hand, the other components of the system interact with the component $\mathcal{C}_1 ||| \mathcal{C}_2$. Hence, the implementation of $\mathcal{C}_1 ||| \mathcal{C}_2$ has to be able to forward the messages to the either $\mathcal{C}_1$ or $\mathcal{C}_2$. Notice that the disjoint union of the actions allows to distinguish easily to which component the action should be forwarded. For instance, since we know for all actions[30] $\mathsf{a} \in A_1 \uplus A_2$ that either $\mathsf{a} \in A_1$ or $\mathsf{a} \in A_2$ (but not both), the store to which the a is to be applied can be easily determined: if $\mathsf{a} \in A_1$ then the apply a to the store of $\mathcal{C}_1$, otherwise to the store of $\mathcal{C}_2$.

——— ⋆ ——— ⋆ ——— ⋆ ———

In this chapter we have defined our computation model for concurrent declarative programming. Roughly speaking, we model a system as a set of component, where each component is composed of a store which is modified by the execution of actions. These actions, which can be defined by the programmer, are executed either by processes of the component, or are received as messages emanating from processes of other components in the system. The interaction of components written in different languages is possible due to the use of dedicated translation functions.

Our computation model distinguishes clearly between the different levels of a system. For instance, actions are defined at a meta-level with respect to the store since actions modify stores. Furthermore, our extension of declarative programming is conservative in the sense that the definition (and semantics) of functions and predicates has not been changed. In fact, since we can express the dynamics of a system by means of actions and processes, there is no need to encode these notions by using functions or predicates. Thus all techniques applying to declarative programs can still be used for the stores. Similarly, all state transforming interaction can be expressed at the level of actions and processes. Last, but not least, the process functions allow the use of the powerful abstraction mechanisms of functional programming to be applied to the description of processes.

As we have already mentioned at the beginning of this chapter, we have used a simple monomorphic type system, and did not use higher-order functions, in order to keep the presentation as simple as possible. However, we believe that an extension including to higher-order functions and polymorphic type systems [Mil78] is rather straightforward.

Finally, our computation may seem complicated and complex compared to classical process calculi or computation models. However, this complexity is unavoidable in order to distinguish clearly between the different concepts and notions. In a simpler model, all these different concepts have to encoded inside the notions provided by the model. While this is possible, we prefer to keep different notions clearly separated, in

order to ease the expression and understanding of the overall system.

Chapter 4

Operational Semantics

The motivation for the investigation of a semantics for a programming language or framework is the precise formal definition of a program. Clearly, such a precise definition is a necessary precondition for any reasoning about programs, as for example program analysis or verification. In this chapter, we present the operational semantics of our framework which we have presented in the preceding chapter. We give the operational semantics of a system in two steps. First we consider a single component, and extend the semantics in a second step to a system of several components.

4.1 Operational Semantics of a Component

The operational semantics of a component has to take into account two different aspects, namely the execution of processes (*i.e.*, the execution of actions causing the modification of the store), and the classical operational semantics of the store, *e.g.*, interactive goal-solving or evaluation of expressions. A further aspect (which can be seen as a part of the execution of processes) is the operational semantics of process functions, *i.e.*, the reduction of process (and action) expressions to normal form. Thus, we present the operational semantics of a component in four steps. First, we define the execution of (closed) guarded actions in normal form. The operational semantics of action and process expressions, *i.e.*, their reduction to normal form, is defined by a rewrite system for process functions. In a third step, we describe the execution of processes by a transition system $\mathsf{T}_{\mathcal{C}}$. Finally we combine the rules of $\mathsf{T}_{\mathcal{C}}$ with rules describing the interactive use of the store, leading to a second transition system $\mathcal{T}_{\mathcal{C}}$ which defines the operational semantics of a component.

Throughout this section, we consider the operational semantics of a component $\mathcal{C} = \langle \widehat{sn},\ \mathrm{C}\Sigma,\ \mathcal{R},\ \mathfrak{A},\ \mathcal{T}r,\ \mathcal{R}^{\mathrm{p}},\ \Pi\mathcal{R},\ p^{\mathtt{i}} \rangle$ with storename $\widehat{sn}$.

4.1.1 Execution of Closed Guarded Actions in Normal Form

To execute a closed guarded action in normal form, *i.e.*, a pair of a guard and a sequence of pairs of storenames and calls to elementary actions containing no free variables, as for instance

$$\Big[g \ \Rightarrow\ \langle sn_1,\ \mathsf{a}_1(t_{1,1},\ \ldots\ ,\ t_{1,l_1}) \rangle ;\ \ldots\ ;\ \langle sn_k,\ \mathsf{a}_k(t_{k,1},\ \ldots\ ,\ t_{k,l_k}) \rangle \Big]$$

in the store F (of a component $\mathcal{C}$ with storename sn), we have first to test the validity of the guard g in the store F. If g holds in the store F, *i.e.*, $F \vdash g$, we have to execute the sequence of (instantiated) storename/elementary action pairs

$$\langle sn_1,\ \mathsf{a}_1(t_{1,1},\ \ldots\ ,\ t_{1,l_1})\rangle;\ \ldots\ ;\ \langle sn_k,\ \mathsf{a}_k(t_{k,1},\ \ldots\ ,\ t_{k,l_k})\rangle$$

In these pairs, the storename determines the component where the elementary action is to be executed. As an elementary action (when supplied with arguments) is a function from stores to stores, the effect of executing an elementary action means to replace, *i.e.*, to "destructively update", the store F by the store the result or normal form of $\big(\mathsf{a}_i(t_{i,1},\ \ldots\ ,\ t_{i,m_i})\big)(F)$, *i.e.*, the application of the action to F.

However, to take into account the difference between local and distributed computations, we distinguish between elementary actions intended for the local component (*i.e.*, associated with the storename $\widehat{sn}$) and all the others. In the first case, we can directly update the local store, in the second we send a message containing the elementary action to the remote component. It is then up to the remote component to ensure that the elementary action is eventually correctly executed. In this section, we are only concerned with the actions to be executed on the local store. The execution of actions on other, remote stores, is presented in section 4.2.

To express the execution of an action expression in normal form formally, we define two functions, namely *exec* and *sel*. These functions take as a parameter an action expression in normal form, *i.e.*, a sequence of pairs of a storename sn_i and an elementary action a_i. We represent these sequences as lists. In the sequel, we use the data type of (polymorphic) lists $\mathcal{S}eq(E)$ with elements of sort E, which is defined by the constructors *nil* (without parameters) for the empty list and *cons* which takes an element e and a list l and constructs the list with the element e in front. The first element of a (non-empty) list l is denoted by $head(l)$, and the remaining list by $tail(l)$. Furthermore, we write $[e]$ for the (singleton) list $cons(e, nil)$ and $l_1 :: l_2$ for the concatenation of the lists l_1 and l_2[1].

The function *sel* takes two arguments, a storename sn and an action expression $l \in \mathcal{A}^{\mathcal{N}}(\mathbb{C}\Sigma,\ X)$ in normal-form, *i.e.*, a sequence of pairs of storenames and elementary actions, which we represent by a list l, and returns the sub-list of l consisting of those pairs of storenames and elementary actions of l, the storename of which is sn.

$$sel(sn, l) = \begin{cases} l & \text{if } l = nil, \\ cons\big(\langle sn',\ \mathsf{a}(t_1,\ \ldots\ ,\ t_m)\rangle,\ sel\big(sn,\ tail(l)\big)\big) & \text{if } sn = sn' \text{ and } head(l) = \langle sn',\ \mathsf{a}(t_1,\ \ldots\ ,\ t_m)\rangle, \\ sel\big(sn,\ tail(l)\big) & \text{otherwise.} \end{cases} \tag{4.1}$$

The function $exec_{sn}$ describes the execution on a store F (the storename of which is sn) of an action expression $l \in \mathcal{A}^{\mathcal{N}}(\mathbb{C}\Sigma,\ X,\ sn)$ in normal-form where all storenames

[1]The data type $\mathcal{S}eq(E)$ is a polymorphic extension of the sort of lists used in the example of the multiple counters: see section 3.3.2 for the signature and example 3.63 for the rewrite rules.

are the same, *i.e.*, a sequence of pairs of a storename sn and an elementary action a_i.

$$exec_{sn}(l,\, F) = \begin{cases} F & \text{if } l = nil, \\ exec_{sn}\Big(tail(l),\, \big(\mathsf{a}(t_1,\, \ldots\, ,\, t_m)\big)(F)\Big) & \text{if } head(l) = \langle sn,\, \mathsf{a}(t_1,\, \ldots\, ,\, t_m)\rangle \end{cases} \tag{4.2}$$

4.1.2 Evaluation of Actions and Process Expressions

The execution of (guarded) actions as defined above considers only (closed) guarded actions in normal form. Hence general action (and process) expressions need to be reduced to normal form before their execution. This reduction is similar to classical rewriting as presented in the context of the operational semantics of a simple declarative programming language in section 3.1.2.2.

Roughly speaking, the reduction of an action or process expression rewrites process functions according to their definitions (*i.e.*, p-rules), until the expression does no longer contain any process functions. That is to say, in every reduction step, the sub-term which is reduced is either rooted with a process functions or in a position which has to be reduced in order to eliminate an occurrence of a process function. Thus calls to elementary actions and processes as well as expressions constructed by means of the operators ;, $\|$, + and $\oplus$ play a similar rôle in the evaluation of action and process expressions as the constructors in classical rewriting.

In the following we present the essential notions related to the operational semantics of process functions. Recall from definition 3.36 that a component signature is an eight-tuple $\mathfrak{C}\Sigma = \langle \Sigma,\, \mathfrak{M}\Sigma_{\mathcal{L}},\, A,\, I,\, E,\, \mathit{Trans},\, P,\, \Pi\rangle$ where the signature of the store $\Sigma = \langle S,\, \Omega\rangle$ is a pair of a set of sorts and a family of operators or functions. Hence, process functions are defined by means of a rewriting system

$$\Big\langle \Big\langle PS,\, \big(\Pi \uplus \{\, ;;\, \|;\, +;\, \oplus;\, \langle\bullet,\, \bullet\rangle;\, [\bullet \Rightarrow \bullet]\,\}\big)\Big\rangle,\, \Pi\mathcal{R} \uplus \mathcal{R} \uplus \mathcal{T}r\Big\rangle \tag{4.3}$$

where $\mathcal{R}$ is the set of rules of the current store, $\mathcal{T}r$ is the set of t-rules defining the translation functions (in *Trans*) and $\langle\bullet,\, \bullet\rangle$ (respectively, $[\bullet \Rightarrow \bullet]$) is the constructors of action (respectively, process) expressions introduced in definition 3.43 (respectively, 3.50). Since action expressions are a special case of process expressions, we do not present the reduction of action expressions separately.

The reduction of a process expression is defined by means of unconditional reduction steps as in definition 3.18. We refer therefore the reader to section 3.1.2 for further details and recall here only the main definitions and notations. Recall that we note $t|_{\mathsf{p}}$ the sub-term of term t at the position p and write $t[t']_{\mathsf{p}}$ for the replacement of the sub-term at the position p in a term t by t'.

Similar to definition 3.20, a *reduction step* $p \dashrightarrow_{\mathsf{p},\, \Pi\mathrm{R}} p'$ is the application of the p-rule, t-rule or rule $\Pi\mathrm{R} = \mathsf{lhs} \rightarrow \mathsf{rhs}$ to the process expression p at the position p, yielding the process expression $p' = p[\sigma(\mathsf{rhs})]_{\mathsf{p}}$, where the substitution σ is defined by the equation $p|_{\mathsf{p}} = \sigma(\mathsf{lhs})$.

Recall from section 3.5 that a process expression without any call to a process or translation function is said to be in *normal form* (see notations 3.45 and 3.51). Notice

that a process expression in normal form is not necessarily *irreducible*[2] since one of its sub-terms might be an application of a defined *function* such that the process expression could be further reduced by the application of a *rule* R ($\in \mathcal{R}$).

Consequently, a process expression has, in general, not a uniquely determined normal form. However, by specifying an appropriate evaluation *strategy*, we can ensure an unique normal form. In this thesis, we opt for the simplest strategy, namely to completely evaluate a process expression to an irreducible process expression. A more general approach would allow the programmer to specify the strategy to be used, for instance by *marking* some sub-terms such that they are not evaluated. However, this marking demands further investigation, since it requires the use of an additional level (*above* the level of processes) in the description of a component, since it concerns the implementation of our computation model. Notice that the use of `Name`s (or meta-terms) allow to pass parameters which are not evaluated.

In the sequel, we call the irreducible process expression obtained by application of reduction steps to a process expression p the[3] normal form of p, noted $p{\Downarrow}$. Similarly, we note the normal form of an action expression a as $a{\Downarrow}$. Notice, that we have by definition that, for any process expression p, $p{\Downarrow}$ is a process term according to notation 3.51, *i.e.*, a process expression without any occurrence of a process function.

The separation of the evaluation or *normalisation* of action and process expressions from the *execution* of processes and actions reflects the fact that we consider these expressions as a means to facilitate the description of processes, for instance to abbreviate the notation as we have shown in examples 3.56 and 3.57. Furthermore, by clearly separating the different kinds of execution, we aim at making the operational behaviour of a program easier to understand. In fact, since the execution of actions and processes modifies the store, the moment when a process function is evaluated may influence the resulting action or process, if the process function depends on a value from the store.

4.1.3 Execution of Process Terms

After the description of the reduction or evaluation of process expressions to process terms, we present in this section the transition system $\mathsf{T}_{\mathcal{C}} = \langle \mathsf{Q},\ \longrightarrow,\ \langle F,\ p^{\mathtt{i}}{\Downarrow}{\Downarrow}\rangle\rangle$ describing the execution of process terms. The states of $\mathsf{T}_{\mathcal{C}}$, *i.e.*, Q, are pairs, *e.g.*, $\langle F,\ p\rangle$, consisting of a store F and a process term p. The initial state of $\mathsf{T}_{\mathcal{C}}$ is built from the initial store F and the normal form of the initial process expression $p^{\mathtt{i}}{\Downarrow}{\Downarrow}$, which are both specified by the programmer as parts of the specification of the component $\mathcal{C}$ (see definition 3.62). In the sequel, we will omit the normalisation of the initial process term.

As common in process calculi, we define the transition relation in the style of the Chemical Abstract Machine (CHAM) [BB92], *i.e.*, modulo a congruence relation, namely $\equiv$, on process terms. The CHAM uses the metaphor of a chemical solution in

[2]Recall from section 3.1.2.2 that a term t is called *irreducible* if no reduction step can be executed starting with t.

[3]Recall from section 3.5.2 that we require process and translation functions to be total recursive functions in order to guarantee that $p{\Downarrow}$ is uniquely determined. However, the reduction steps for process expressions may involve also reduction using the rules of the store. A programmer has therefore to ensure that the terms of the store which are used in process expressions are always reducible to an unique normal form, in this during the complete execution of the system.

$$\begin{aligned} \mathsf{success}\,;p &\equiv p \\ \mathsf{success} \parallel p &\equiv p \end{aligned} \qquad (Unit_{\equiv})$$

$$\begin{aligned} p_1 \parallel p_2 &\equiv p_2 \parallel p_1 \\ p_1 + p_2 &\equiv p_2 + p_1 \end{aligned} \qquad (Comm_{\equiv})$$

Table 4.1: Axiom Schemes Defining the Structural Congruence $\equiv$ on Process Terms

order to represent the states of a transition system as a multiset of floating molecules. Heating or cooling the chemical solution does only change the solution in a reversible manner: this is modeled by the congruence relation (which can be applied in both directions). The transitions correspond to the chemical reactions which transform the solution. The congruence relation $\equiv$ is defined by the axiom schemes shown in table 4.1; the extension to a congruence relation (reflexivity *etc.*) should be obvious. The inference rules defining the transition relation $\longrightarrow$ are given in table 4.2. These rules define exactly the set of legal (or correct) transitions (with respect to $\mathsf{T}_{\mathcal{C}}$), in the sense that all correct transitions can be inferred by these rules and all transitions that can be inferred by these rules are correct.

Informally, the congruence relation $\equiv$ states that the process term success is a unit element for sequential (;) and parallel ($\parallel$) composition (see rules ($Unit_{\equiv}$)), and that the operators $\parallel$ and $+$ are commutative (see rule ($Comm_{\equiv}$)). Notice that success is not a neutral element for $+$. In fact, $\mathsf{success} + p$ has the choice between (immediate) termination or the behaviour of p, whereas p cannot, in general, terminate immediately[4]. Obviously, *sequential* composition and choice with *priority* are not commutative by their very nature. Notice finally, that we do not need the axioms for associativity (of ;, $\parallel$, $+$ and $\oplus$), since the inference rules shown in table 4.2 imply that the corresponding processes have the same behaviour.

We comment the rules of table 4.2 one by one. The combination of the structural congruence and the transition relation is described by rule ($\mathsf{R}_{\equiv}$). In the language of the CHAM, rule ($\mathsf{R}_{\equiv}$) allows to include "heating" and "cooling" steps, *i.e.*, transformation according to $\equiv$ before and after a transition (or "reaction" in the metaphor of a chemical solution). Stated otherwise, rule ($\mathsf{R}_{\equiv}$) allows to define the transition relation "modulo the congruence $\equiv$".

Using the auxiliary functions *sel* and *exec* defined in section 4.1.1, we describe by rule (R_{action}) the effects of executing a closed (guarded) action, *i.e.*, an action expression in normal form without free variables. Under the premise of the validity of the guard in the current store ($F \vdash g$), the local store is replaced by the application of the action expression in normal form, *i.e.*, a sequence of elementary actions, to the store. Since this sequence may contain elementary actions on several stores, we have to distinguish between elementary actions meant for the local store (recall that we suppose that we are given a component $\mathcal{C} = \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \Pi\mathcal{R}, p^{\mathsf{i}} \rangle$ with storename $\widehat{sn}$) and all the others. Notice that rule (R_{action}) describes only the execution of the "local" actions.

[4]Consider the process term [TRUE $\Rightarrow$ skip] which executes an action before successful termination.

$$\frac{p \equiv p' \qquad \langle F,\, p'\rangle \longrightarrow \langle F',\, p''\rangle \qquad p'' \equiv p'''}{\langle F,\, p\rangle \longrightarrow \langle F',\, p'''\rangle} \tag{$\mathsf{R}_{\equiv}$}$$

$$\frac{F \vdash g}{\Big\langle F,\ \big[g \Rightarrow \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big]\Big\rangle \longrightarrow \Big\langle exec_{\widehat{sn}}\Big(sel\big(\widehat{sn},\ \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big),\ F\Big),\ \mathsf{success}\Big\rangle} \tag{R_{action}}$$

$$\frac{\big(\mathrm{q}(x_1,\, \ldots\, ,\, x_n) \Leftarrow \bigoplus_{i=1}^{m}([g_i \Rightarrow a_i]\,; p_i)\big) \in \mathcal{R}^{\mathrm{p}} \qquad \big\langle F,\ \big(\bigoplus_{i=1}^{m} rename([g_i \Rightarrow a_i\Downarrow]\,; p_i)\big)[v_j/x_j]\big\rangle \longrightarrow \langle F',\, p'\rangle}{\langle F,\, \mathrm{q}(v_1,\, \ldots\, ,\, v_n)\rangle \longrightarrow \langle F',\ p'\Downarrow\rangle} \tag{R_{call}}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1\,; p_2\rangle \longrightarrow \langle F',\, p_1'\,; p_2\rangle} \tag{$\mathsf{R}_{;}$}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 \parallel p_2\rangle \longrightarrow \langle F',\, p_1' \parallel p_2\rangle} \tag{$\mathsf{R}_{\parallel}$}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 + p_2\rangle \longrightarrow \langle F',\, p_1'\rangle} \tag{R_{+}}$$

$$\frac{\langle F,\, p_1\rangle \longrightarrow \langle F',\, p_1'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \longrightarrow \langle F',\, p_1'\rangle} \tag{$\mathsf{R}_{\oplus}$}$$

$$\frac{\langle F,\, p_2\rangle \longrightarrow \langle F',\, p_2'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \longrightarrow \langle F',\, p_2'\rangle} \quad \text{if } \nexists\, p_1',\, \nexists\, F'', \text{ such that } \langle F,\, p_1\rangle \longrightarrow \langle F'',\, p_1'\rangle \tag{$\mathsf{R}'_{\oplus}$}$$

Table 4.2: Inference Rules Defining the Transition Relation $\longrightarrow$ of $\mathsf{T}_{\mathcal{C}}$

We present the mechanisms for sending messages containing the sequence of elementary actions (together with their arguments) to remote components in section 4.2, where we define the operational semantics of system composed of several components.

Notice that the execution of an action is *locally atomic*, *i.e.*, all the elementary actions (for a same store) of an action are executed in a single transition step, so that actions executed by other processes cannot interfere. An example for the usefulness of the atomic execution of actions is the program for the dining philosophers (see example (1.1.5)) where a philosopher needs to take the two necessary chop sticks in a single atomic action.

According to Rule (R_{call}), a call to a process corresponds to the execution of an (instantiated) *variant* of the process definition. Recall from definition 3.60 that a variant of a process definition is obtained by renaming all new symbols introduced by new elementary actions, and that we note the renaming by the function *rename*. This is similar to the application of clauses in logic programming, where implicitly each variable is renamed by a fresh one, *i.e.*, a new and unused variable. However, in our computation model the programmer has explicitely to specify which symbols should be replaced by fresh ones via the elementary action new.

Notice that the action expression of the selected definition is evaluated in the same store in which the guard is checked, due to the locally atomic execution of guarded actions. Notice further, that the process expressions are evaluated in the store *after* the execution of the action. By the way, it can be seen easily, that there exists an $i \in \{1; \ldots; m\}$ such that $p' \equiv \mathit{rename}(p_i)[v_j/x_j]$ $(j \in \{1; \ldots; n\})$, *i.e.*, the process expression p' is (equivalent to) one of the process expressions of the guarded commands defining the process q. The reason is that the transition in the premise of rule (R_{call}) is necessarily an execution of an action, *i.e.*, a transition according to rule (R_{action}), due to the syntactical form of process definitions.

Rules $(\mathsf{R}_;)$ to (R_+) describe the standard semantics of the operators ;, $\|$ and +. According to rule $(\mathsf{R}_;)$, executing the process term $p_1 ; p_2$ means to execute first p_1. When p_1 has terminated its execution, that is to say when it has become the process term $\mathsf{success}$, *i.e.*, $p_1' \equiv \mathsf{success}$, rule $(\mathsf{R}_\equiv)$ together with the first axiom of $(\mathit{Unit}_\equiv)$ ensure that p_2 can start its execution. Rule $(\mathsf{R}_\|)$ specifies an *interleaving* semantics for the parallel composition $p_1 \| p_2$, *i.e.*, there is only one execution step (or transition) at a time such that the steps executed by concurrent processes are "interleaved" in a nondeterministic way. This semantics of the parallel composition is to be distinguished from so-called *true* concurrency models, where several execution steps are allowed to happen at the same time. However, an interleaving semantics is conceptually simpler, since the problem of two "contradicting" steps (or actions in our case) cannot arise. Hence this approach has been adopted by most process calculi (see section 2.2). The execution of the process term $p_1 + p_2$ as described by rule (R_+) consists of the execution of p_1, discarding p_2. Since the operator + is commutative, see the axiom schemes $(\mathit{Comm}_\equiv)$, the execution of $p_1 + p_2$ can choose nondeterministically between p_1 and p_2.

Since the operator $\oplus$ is not commutative, we need two inference rules to define its operational behaviour[5]. When executing the process-term $p_1 \oplus p_2$, the process p_2 will only be executed if (in the current store) an execution of p_1 is impossible (see the side-condition of Rule $(\mathsf{R}'_\oplus)$). In contrary, p_1 can be executed independently from the executability of p_2 (see Rule $(\mathsf{R}_\oplus)$). Notice that thanks to our syntactical restriction to restricted process expressions in processes definitions as well as for the initial process term, the operator of choice with priority can only appear inside a process call, *i.e.*, in rule (R_{call}). Thus, when we check the guards of the process in a sequential manner the operator $\oplus$ can be easily implemented. In fact, this is the motivation of the syntactical restriction.

We call *inference tree* for a transition $\langle F, p\rangle \longrightarrow \langle F', p'\rangle$ a tree the nodes of which correspond to transitions (or other statements occurring in the inference rules). The nodes of an inference tree are labeled with inference rules such that the children correspond to the premises allowing to infer the parent using the inference rule with which the parent is labeled. Suppose that we can prove the validity of the transition $\langle F, p_2\rangle \longrightarrow \langle F', p_2'\rangle$. Under this hypothesis, figure 4.1 shows the remainder of an inference tree for the transition $\langle F,\ p_1 \| (p_2 + p_3)\rangle \longrightarrow \langle F',\ p_1 \| p_2'\rangle$. We draw inference trees with its root at the bottom of the figure, such that the applications of the inference rules are in the same direction as in their definitions. Notice that the notion of inference trees has similarities to the proof figures which are used in linear logic

[5]Notice that we need only *one* rule for the equally non commutative operator ";" since in the process term $p_1 ; p_2$ only p_1 can be executed.

$$\dfrac{p_1 \parallel (p_2 + p_3) \equiv (p_2 + p_3) \parallel p_1 \quad \dfrac{\dfrac{\vdots}{\langle F,\, p_2\rangle \longrightarrow \langle F',\, p_2'\rangle}}{\langle F,\, p_2 + p_3\rangle \longrightarrow \langle F',\, p_2'\rangle}(\mathsf{R}_+)}{\langle F,\, (p_2 + p_3) \parallel p_1\rangle \longrightarrow \langle F',\, p_2' \parallel p_1\rangle}(\mathsf{R}_\parallel) \quad p_2' \parallel p_1 \equiv p_1 \parallel p_2'}{\langle F,\ p_1 \parallel (p_2 + p_3)\rangle \longrightarrow \langle F',\ p_1 \parallel p_2'\rangle}(\mathsf{R}_\equiv)$$

Figure 4.1: Example of an Inference Tree

programming to represent proofs, *i.e.*, executions.

4.1.4 Combined Operational Semantics of a Component

Besides the execution of the processes modifying the store, described by the transition system $\mathsf{T}_\mathcal{C}$, the operational semantics of a component $\mathcal{C}$ has another, orthogonal aspect, namely the classical operational semantics of the declarative programming language $\mathcal{L}$ used for the description of the store, as for instance goal solving or evaluation of expressions. We suppose that the operational semantics of $\mathcal{L}$ is described by a relation $\curvearrowright$, which we do not precise further. Intuitively, $\curvearrowright$ describes the transformation steps of a *goal*, *i.e.*, $\curvearrowright$ is a relation between goals. Examples for $\curvearrowright$ are for instance rewriting $\rightarrow$ or narrowing $\rightsquigarrow$ presented in section 3.1.2.

We describe the operational semantics of a component $\mathcal{C}$ via a new transition system, namely $\mathcal{T}_\mathcal{C} = \langle \mathcal{Q},\, \longmapsto,\, \langle F,\, p^{\mathtt{i}},\, \mathtt{g}^{\mathtt{i}},\, \mathtt{g}^{\mathtt{i}}\rangle\rangle$, where $\mathtt{g}^{\mathtt{i}}$ denotes a (possibly empty) initial goal the user wants to solve, or the expression which is to be reduced. The states $\mathcal{Q}$ of the transition system $\mathcal{T}_\mathcal{C}$ are *configurations*, *i.e.*, four-tuples $\langle F,\, p,\, \mathtt{g}^{\mathtt{i}},\, \mathtt{g}\rangle$, where F is the current store, p is the current process term, $\mathtt{g}^{\mathtt{i}}$ is the initial goal to solve and $\mathtt{g}$ is the expression representing the current state of the evaluation of $\mathtt{g}^{\mathtt{i}}$ (according to the operational semantics of the declarative language used for F). We need the initial goal in a configuration, since we need to know the solving of which goal we should restart if a modification of the store invalidates the evaluation effectuated so far.

Classically, configurations of concurrent languages and process calculi, as *e.g.*, CSP [Hoa85], the π-calculus [Mil99] or also our operational semantics for the execution of processes $\mathsf{T}_\mathcal{C}$, are described only by the first two parts of our configurations, namely $\langle F,\, p\rangle$, which are enough to express the execution of processes. As for declarative languages, a configuration classically uses the first and the fourth parts $\langle F,\, \mathtt{g}\rangle$ which allow to express the rules of the operational semantics $\curvearrowright$ of the declarative language, as for instance the definition of the relation $\rightsquigarrow$ in section 3.1. Combining these two operational semantics adds the possibility to execute concurrent processes without loosing the characteristics of declarative languages. For instance, goals can be solved while the processes are running. This feature is useful to allow to query, at any moment, the current store or state of an evolving system – without being limited to a fixed, predefined set of possible queries that has been established during the specification of the system. In particular, in a programming framework for rapid prototyping, where the prototype is a means to explore the set of interesting queries, it is interesting to dispose of a rich query language.

$$\frac{\langle F,\, \mathsf{g}\rangle \curvearrowright \langle F,\, \mathsf{g}'\rangle}{\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}\rangle \longmapsto \langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}'\rangle} \tag{G}$$

$$\frac{\langle F,\, p\rangle \longrightarrow \langle F',\, p'\rangle}{\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}\rangle \longmapsto \langle F',\, p',\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}'\rangle} \quad \text{where } \mathsf{g}' \stackrel{\text{def}}{=} \begin{cases} \mathsf{g} & \text{if } F = F' \\ \mathsf{g}^{\mathsf{i}} & \text{otherwise} \end{cases} \tag{P}$$

Table 4.3: Inference Rules for the transition system $\mathcal{T}$

The transition relation of $\mathcal{T}_{\mathcal{C}}$, *i.e.*, $\longmapsto$, is defined by the two inference rules shown in table 4.3. Rule (G) concerns interactive goal-solving, *i.e.*, the use of the operational semantics of the declarative language, as for instance goal solving (in logic languages) or evaluation of expressions (in functional languages). In the example of the Dining Philosophers, rule (G) allows us to ask for the currently eating philosophers by solving the goal is_eating(x) (see section 1.1.5). Rule (P) describes the modifications of the store by the processes via the transition system T. When a process modifies the store, we have to restart the goal-solving at the initial goal, *i.e.*, g^{i}, as the modification may invalidate the already achieved derivation.

Obviously, rule (P) is only one of the many possibilities. A rather simple refinement would be to restart the goal-solving only if the execution of a process has altered some of the definitions used so far in the evaluation of the goal. Another, completely different option for rule (P) would be to solve the goal in an unmodified "private copy" of the store. More sophisticated techniques, namely rearranging of the search tree, *i.e.*, the order of the application of rules, have been investigated in [FFS95, FFS98]. These methods allow the reuse of as much as possible of the search tree after the theory has been modified.

Finally, we use a last rule to model the interactive interpreter for the store. At every moment, a user of the interpreter can decide to change the initial goal $\mathsf{g}_1^{\mathsf{i}}$ that is currently solved, and replace it by another goal $\mathsf{g}_2^{\mathsf{i}}$. Obviously, we have to restart goal solving from the new initial goal $\mathsf{g}_2^{\mathsf{i}}$. Notice that due to rule (I), a programmer does not need to specify the initial goal for a component, since all components can start with a default empty goal, *i.e.*, TRUE, and the user of the system can interactively change to another goal.

$$\frac{\langle F,\, p,\, \mathsf{g}_1^{\mathsf{i}},\, \mathsf{g}\rangle}{\langle F,\, p,\, \mathsf{g}_2^{\mathsf{i}},\, \mathsf{g}_2^{\mathsf{i}}\rangle} \tag{I}$$

Notice that the operational semantics for a component does not consider explicitly composite components, *i.e.*, components of the form $\mathcal{C}_1 \,|||\, \mathcal{C}_2$. In fact, a composite component can be considered as a particular case of a system the operational semantics of which is defined in the following section.

4.2 Semantics of a System

In this section we consider the operational semantics of a system of several components. We do not want to model the semantics of such a system by a *single* transition system,

since we find it unrealistic to assume that we might have total knowledge about the states of *all* the components of a distributed system at the same time. Indeed, if we were up to defining the states of the global transition system, we would be forced to the simplifying assumption that there exists a point during the execution of the system, such that all components are in a fixed state. Therefore we do not believe that a single transition system as in the preceding sections is an appropriate model for a distributed system, since it gives a centralised view of a system. An example of such a semantics is the operational semantics of KLAIM [NFP98], where a global transition system is used in order to specify the synchronisation between the different components (or nets) in a system. We prefer to model a distributed system as a collection of concurrently executing transition systems.

In the preceding section, we have presented the transition system $\mathcal{T}_{\mathcal{C}}$ defining the operational semantics of a single component $\mathcal{C}$. In this section, we give some meta-rules and conditions describing the interactions between several instances of $\mathcal{T}$ representing the components of a system. Recall that components in our framework communicate via message-passing, where a message corresponds to a sequence of elementary actions to be executed. Thus we need to specify how the messages are sent and received.

In order to model the delay due to the transmission of messages over the communication network, we introduce the notion of a *mailbox*, acting as a (fifo[6]-) channel on which a component, *e.g.*, $\mathcal{C}$, receives action expressions in normal form (*i.e.*, sequences of elementary actions) emanating from other components that want them to be executed on the component $\mathcal{C}$. Thus we extend the configuration of a component by a fifth part, namely the mailbox m, leading to a new transition system the transition relation of which we note by $\rightarrowtail$. Recall from notation 3.45 that we note $\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing, sn)$ the set of action expressions in normal form where all calls to elementary actions are paired with the storename sn. In the sequel, we represent the mail-boxes as lists, *i.e.*, mailboxes are of sort $\mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing, sn)\big)$, using the sort $\mathcal{S}eq(E)$ of lists (of elements of sort E) introduced in section 4.1.1.

As for the transition systems in the preceding section, the translation relation $\rightarrowtail$ of the extended transition system for a component is defined by a set of inference rules. The first inference rule for $\rightarrowtail$ allows the reception of messages in the mailbox.

$$\frac{\mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing, \widehat{sn})\big)}{\big\langle F, p, \mathsf{g}^{\mathtt{i}}, \mathsf{g}, \mathsf{m}\big\rangle \rightarrowtail \big\langle F, p, \mathsf{g}^{\mathtt{i}}, \mathsf{g}, (\mathsf{m} :: \mathsf{m}')\big\rangle} \qquad (\mathsf{M})$$

The premise of rule (M), namely $\mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing, \widehat{sn})\big)$ models the reception of a list of messages, where a message is an action expression in normal-form such that all elementary actions are paired with the storename of the receiving component, *i.e.*, $\widehat{sn}$. We consider the reception of a list of messages in order to model the independence of the execution of different components. Hence, we can model the reception of an arbitrary number of messages (including zero), allowing to model the fact that a "slower" component will receive, in general, more messages per transition than "faster" components.

The second inference rule for $\rightarrowtail$ tells us that all transitions with respect to $\longmapsto$ are

[6]fifo is the acronym of first-in-first-out.

also transitions with respect to $\rightarrowtail$.

$$\frac{\langle F,\, p,\, \mathsf{g^i},\, \mathsf{g}\rangle \longmapsto \langle F',\, p',\, (\mathsf{g^i})',\, \mathsf{g}'\rangle \qquad \mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, \widehat{sn})\big)}{\langle F,\, p,\, \mathsf{g^i},\, \mathsf{g},\, \mathsf{m}\rangle \rightarrowtail \langle F',\, p',\, (\mathsf{g^i})',\, \mathsf{g}',\, (\mathsf{m} :: \mathsf{m}')\rangle} \qquad (\mathsf{C})$$

The second premise of rule (C) allows the reception of a sequence of messages in the mailbox during the transition.

The execution of (sequences of) actions that are already in the mailbox has to be interleaved with the execution of actions by the processes of the component. Therefore, we introduce a rule similar to rule (R_{action}) (see table 4.2), which describes the execution of actions by processes of the component.

$$\frac{\mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, \widehat{sn})\big)}{\langle F,\, p,\, \mathsf{g^i},\, \mathsf{g},\, [a] :: \mathsf{m}\rangle \rightarrowtail \langle exec_{\widehat{sn}}(a,\, F),\, p,\, \mathsf{g^i},\, \mathsf{g}',\, \mathsf{m} :: \mathsf{m}'\rangle} \quad \mathsf{g}' \stackrel{\text{def}}{=} \begin{cases} \mathsf{g} & \text{if } F = exec_{\widehat{sn}}(a,\, F) \\ \mathsf{g^i} & \text{otherwise} \end{cases} \qquad (\mathsf{E})$$

The main difference between rules (E) and (R_{action}) is that (E) considers action expressions and (R_{action}) guarded actions. Hence, rule (E) has no premise corresponding to the check of the guard. Similarly, the selection of the actions addressed to the store can be omitted in rule (E) since all elementary actions received in the mailbox are (by definition) to be executed on the current store. Clearly, the same remarks as for rule (P) apply to rule (E), concerning the fact of restarting the goal solving after the execution of the actions. Notice also, that rule (E) (as rule (C)) allows the reception of new messages, *i.e.*, action expressions in normal-form, during the execution of actions from the mailbox.

As mentioned at the beginning of this section, we specify the interaction between different components not by a global transition system, but give meta-rules which relate *event*s occurring in different components. We distinguish two kinds of events, namely SEND, the sending of a sequence of actions, and RECEIVE, their reception. These events have two arguments, namely the component name or storename sn of their destination and the contents of the message, *i.e.*, an action expression a in normal form (where all calls to elementary actions are paired with the storename sn), *i.e.*, $a \in \mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, sn)$.

Recall that rule (R_{action}) describes the execution of a guarded action $[g \Rightarrow a]$ only partially, in the sense that only the modification of the local store is expressed. However, the execution of the guarded action has also the effect of sending messages to remote components. Thus, we associate to each execution of a guarded action the set of SEND-events corresponding to the messages sent to other components. This is tantamount to label the transitions of a component with the set of events associated to the transition. The inference rules for the labeled transition relation are shown in table 4.4. According to rule ($\hat{\mathsf{R}}_{action}$), we label a transition $\langle F,\, [g \Rightarrow a]\rangle \longrightarrow \langle F',\, \mathsf{success}\rangle$ with the set of events $\mathcal{E}v^{\mathsf{s}}([g \Rightarrow a])$ which is defined as (recall that the storename of the local component is $\widehat{sn}$)

$$\mathcal{E}v^{\mathsf{s}}([g \Rightarrow a]) \stackrel{\text{def}}{=} \big\{\, \mathsf{SEND}\big(sn',\ sel(sn',\, a)\big) \;\big|\; sn' \in \mathcal{SN} \setminus \{\widehat{sn}\} \text{ and } sel(sn',\, a) \neq nil \,\big\} \qquad (4.4)$$

We omit the formal definition of the straightforward[7] extension of the labeling to the transition system $\mathcal{T}_{\mathcal{C}}$, *i.e.*, to the transition relation $\longmapsto$.

[7] Labeling rule (P) amounts to pass on the set of events, whereas rules (G) and (I) are labeled with the empty set.

$$\frac{p \equiv p' \qquad \langle F,\, p'\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p''\rangle \qquad p'' \equiv p'''}{\langle F,\, p\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p'''\rangle} \tag{$\hat{\mathsf{R}}_{\equiv}$}$$

$$\frac{F \vdash g}{\big\langle F,\ [g \Rightarrow a]\big\rangle \xrightarrow{\mathcal{E}v^{\mathsf{s}}([g \Rightarrow a])} \big\langle exec\big(sel(\widehat{sn},\, a),\ F\big),\ \mathsf{success}\big\rangle} \tag{$\hat{\mathsf{R}}_{action}$}$$

$$\frac{\begin{array}{l}\big(\mathrm{q}(x_1,\, \ldots,\, x_n) \Leftarrow \bigoplus_{i=1}^{m}([g_i \Rightarrow a_i]\,;p_i)\big) \in \mathcal{R}^{\mathrm{p}} \\ \big\langle F,\ \big(\bigoplus_{i=1}^{m} rename([g_i \Rightarrow a_i\Downarrow]\,;p_i)\big)[v_j/x_j]\big\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p'\rangle\end{array}}{\langle F,\, \mathrm{q}(v_1,\, \ldots,\, v_n)\rangle \xrightarrow{\mathcal{E}v} \langle F',\ p'\Downarrow\rangle} \tag{$\hat{\mathsf{R}}_{call}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle}{\langle F,\, p_1\,;p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\,;p_2\rangle} \tag{$\hat{\mathsf{R}}_{;}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \parallel p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1' \parallel p_2\rangle} \tag{$\hat{\mathsf{R}}_{\parallel}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle}{\langle F,\, p_1 + p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle} \tag{$\hat{\mathsf{R}}_{+}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_1'\rangle} \tag{$\hat{\mathsf{R}}_{\oplus}$}$$

$$\frac{\langle F,\, p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_2'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{\mathcal{E}v} \langle F',\, p_2'\rangle} \qquad \begin{array}{l}\text{if } \nexists p_1',\, \nexists F'',\, \nexists a', \text{ such that} \\ \langle F,\, p_1\rangle \xrightarrow{\mathcal{E}v'} \langle F'',\, p_1'\rangle\end{array} \tag{$\hat{\mathsf{R}}'_{\oplus}$}$$

Table 4.4: Inference Rules for $\longrightarrow$ Labeled with the Associated Events

The extension of the labeling to the transition relation $\rightarrowtail$ has to consider besides SEND-events also RECEIVE-events. Intuitively, the reception of a message in the mailbox as modeled in the rules (M), (C) and (E) corresponds to the occurrence of a RECEIVE event. The labeled versions of theses inference rules are as follows:

$$\frac{\mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, \widehat{sn})\big)}{\big\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g},\, \mathsf{m}\big\rangle \overset{\mathcal{E}v^{\mathsf{r}}(\mathsf{m}')}{\rightarrowtail} \big\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g},\, (\mathsf{m} :: \mathsf{m}')\big\rangle} \tag{$\hat{\mathsf{M}}$}$$

$$\frac{\big\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}\big\rangle \overset{\mathcal{E}v^{\mathsf{s}}}{\longmapsto} \big\langle F',\, p',\, (\mathsf{g}^{\mathsf{i}})',\, \mathsf{g}'\big\rangle \qquad \mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, \widehat{sn})\big)}{\big\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g},\, \mathsf{m}\big\rangle \overset{\mathcal{E}v^{\mathsf{s}} \uplus \mathcal{E}v^{\mathsf{r}}(\mathsf{m}')}{\rightarrowtail} \big\langle F',\, p',\, (\mathsf{g}^{\mathsf{i}})',\, \mathsf{g}',\, (\mathsf{m} :: \mathsf{m}')\big\rangle} \tag{$\hat{\mathsf{C}}$}$$

$$\frac{\mathsf{m}' \in \mathcal{S}eq\big(\mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, \widehat{sn})\big)}{\big\langle F,\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g},\, [a] :: \mathsf{m}\big\rangle \overset{\mathcal{E}v^{\mathsf{r}}(\mathsf{m}')}{\rightarrowtail} \big\langle exec_{\widehat{sn}}(a,\, F),\, p,\, \mathsf{g}^{\mathsf{i}},\, \mathsf{g}',\, \mathsf{m} :: \mathsf{m}'\big\rangle} \quad \mathsf{g}' \overset{\mathsf{def}}{=} \begin{cases} \mathsf{g} & \text{if } F = exec_{\widehat{sn}}(a,\, F) \\ \mathsf{g}^{\mathsf{i}} & \text{otherwise} \end{cases} \tag{$\hat{\mathsf{E}}$}$$

where the set of RECEIVE-events associated to a sequence of action expressions in normal form is defined as

$$\mathcal{E}v^{\mathsf{r}}\big([a_1; \ldots ; a_n]\big) \overset{\mathsf{def}}{=} \big\{\ \mathsf{RECEIVE}(sn,\, a_i) \ \big|\ i \in \{1; \ldots ; n\}\ \big\} \tag{4.5}$$

Using the labeling of the transition relation $\rightarrowtail$ as described by the rules ($\hat{\mathsf{M}}$), ($\hat{\mathsf{C}}$) and ($\hat{\mathsf{E}}$), we can now define the meta-rules which relate the events occurring at different components. We may imagine the multiset $\mathcal{E}$ of all events occurred during a particular execution (or *run*) of a system. A run of a system consists in our model of a set of sequences of transitions (for the transition relation $\rightarrowtail$), such that there is exactly one sequence of transitions for each component of the system. Thus we require that for each sequence of transitions $\langle F_0, p_0, \mathsf{g}_0^{\mathsf{i}}, \mathsf{g}_0, \mathsf{m}_0\rangle \overset{\mathcal{E}v_1}{\rightarrowtail} \langle F_1, p_1, \mathsf{g}_1^{\mathsf{i}}, \mathsf{g}_1, \mathsf{m}_1\rangle \overset{\mathcal{E}v_1}{\rightarrowtail} \cdots \overset{\mathcal{E}v_i}{\rightarrowtail} \langle F_i, p_i, \mathsf{g}_i^{\mathsf{i}}, \mathsf{g}_i, \mathsf{m}_i\rangle \overset{\mathcal{E}v_{i+1}}{\rightarrowtail} \cdots$ in the run, we have that $\mathcal{E} \supseteq \bigcup_{i>0} \mathcal{E}v_i$. Furthermore, we have to require that the multiset of occurred events $\mathcal{E}$ contains only events that have actually occurred, *i.e.*, that have been added by a transition. In other words, we require that for all events $e \in \mathcal{E}$, we have a sequence of transitions in the run such that there exists $i > 0$ such that $e \in \mathcal{E}v_i$.

To express the interaction between components, we need to link the SEND and RECEIVE events in $\mathcal{E}$. We say that two events SEND and RECEIVE *correspond* (to each other) when they have exactly the same arguments. Consequently, a RECEIVE-event e ($\in \mathcal{E}$) that corresponds to a SEND-event e' ($\in \mathcal{E}$) might model the reception of a message the sending of which is modeled by e'. To specify the relation between events more precisely, we use further conditions, refining the notion of corresponding events. Notice first, that $\mathcal{E}$ can be considered as a *set*, *i.e.*, distinguishing events with the same parameters, if we define an appropriate *order* on the occurred events. To define this order, we extend the sequential order of the transitions of a component $\mathcal{C}$ to a partial[8] order on the events occurring at the component $\mathcal{C}$, which we note $\prec_{\mathcal{C}}$. Hence, the distinction between SEND-events can be based on the *sender* of the message together with the partial orders $\prec$ (on the sender-side), since in one transition at most one message is sent from one component to another. Similarly, we can distinguish RECEIVE-events by the sender together with the partial order on the receiver-side. However, since the reception of several messages is possible in a single transition, we have to take into account the order of the messages in the *list*[9] (which is added to the mailbox) to extend the partial order such that two receptions of the same action expression from the same component an be distinguished.

Our requirements on the interaction between components of a system can now be expressed as properties of a *correspondence function* over $\mathcal{E}$ considered as a *set*. A reasonable requirement seems to be that the We require that the correspondence function is bijective and idempotent. Hence, for every event e ($\in \mathcal{E}$) we denote its corresponding event, *i.e.*, the event associated to e by the correspondence function, by $\bar{e}$. The idempotency of the correspondence function ensure that we have $\bar{\bar{e}} = e$. The requirement that the correspondence function be bijective expresses already two properties of the communication medium, namely that no message can be lost (*i.e.*, all messages that are sent arrive eventually) and that no phantom (*i.e.*, unsent) messages are received. The former would mean that the restriction of the correspondence function to the domain of RECEIVE-events is not surjective (there are SEND-events which do not correspond to a RECEIVE-event). The latter (no phantom messages) is just the

[8]The order is necessarily partial since several events may occur at a single transition.

[9]The order induced by a list is the reason why we used lists instead of sets. Furthermore, lists reflect more closely the idea of a fifo-channel.

dual.

A further meta-rule or condition relating SEND and RECEIVE events could be used to specify that the underlying communication medium preserves the order of messages along a communication link[10], *i.e.*, for two SEND events $e_1 \stackrel{\text{def}}{=} \mathsf{SEND}(sn', a_1)$ and $e_2 \stackrel{\text{def}}{=} \mathsf{SEND}(sn', a_2)$ occurring at the component $\mathcal{C}$ such that $e_1 \prec_{\mathcal{C}} e_2$, we have also that the corresponding events are in the same order. $\bar{e}_1 \prec_{\mathcal{C}'} \bar{e}_2$ where $\mathcal{C}'$ is the component with storename sn'.

——— ⋆ ——— ⋆ ——— ⋆ ———

In this chapter, we have presented the operational semantics of a system as a set of several concurrently executing transition systems, each of them defining the operational semantics of a single component. Interaction between these transition systems is expressed by establishing a correspondence between the communication events occurring at the components. The operational semantics of a component itself has essentially two parts, namely the execution of processes (modifying the store) and interactive goal solving. The operational semantics we presented here reflects this separation in the way that it is defined as a combination of two distinct transition relations. With respect to previous presentations of the operational semantics of our framework [Ser98, ES99, ES00], we use here an approach in the style of the CHAM [BB92] and give a description of the semantics of a system of several components.

[10]We do not suppose a global ordering of the messages, we demand only that the messages sent from component $\mathcal{C}_1$ to component $\mathcal{C}_2$ arrive at the component $\mathcal{C}_2$ in the order they were sent from the component $\mathcal{C}_1$.

Chapter 5

Compositional Semantics of a Component

In this chapter we present a compositional semantics for processes of a component. Intuitively, a semantics is called *compositional*, if the semantics of a composed entity (in our case, a composed process term) can be obtained by composition of the semantics of its components. Compositionality allows to considerably reduce the complexity of (program) analyses, since it allows to analyse (small) parts of a program separately, and to compose the obtained results to obtain a global result. Hence a compositional semantics facilitates reasoning about programs, which is the main motivation for the definition of a precise formal semantics.

We show first that the semantics of execution traces as generated directly by the operational semantics of processes as presented in chapter 4 is not compositional. Then we present a second semantics based on *labeled* execution traces (originally inspired from the compositional semantics for ccp defined in [dBP91]) which we show to be compositional.

Throughout this chapter we suppose we are given a single component $\mathcal{C}$ defined as $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn}, \mathbb{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{I}r, \mathcal{R}^{\mathrm{p}}, \Pi\!\!I\mathcal{R}, p^{\mathtt{i}} \rangle$, and we consider only the part of the semantics of a component dealing with the execution of processes.

5.1 Semantics of Execution Traces

The semantics of a process we consider in this chapter is based on *execution traces* or *trace* for short. Informally, a trace is the sequence of (guarded) actions a process has executed. Similar to traces in a snowy winter landscape, the trace of a process allows to reconstruct the (execution) path the process has taken.

To define the traces of a process according to the transition system presented in section 4.1, we label all transitions with the action that is executed by the transition. Thus we obtain the transition system $\tilde{\mathsf{T}}_{\mathcal{C}} = \langle \mathsf{Q}, \rightarrow, \langle F, p^{\mathtt{i}} \rangle \rangle$. The states Q as well as the initial state $\langle F, p^{\mathtt{i}} \rangle$ of $\tilde{\mathsf{T}}_{\mathcal{C}}$ are the same as for the transition system $\mathsf{T}_{\mathcal{C}}$, namely pairs of a store F and a process term p. The inference rules of $\tilde{\mathsf{T}}_{\mathcal{C}}$ are shown in table 5.2. These rules are essentially the same as those for $\mathsf{T}_{\mathcal{C}}$ (see table 4.2) and just introduce the labels of the transitions. Thus the transition relation $\rightarrow$ is a *ternary* relation between

states, guarded actions and states (instead of being binary relation between states). We recall in table 5.1 the axiom schemes defining the congruence relation $\equiv$ to make this chapter "self-contained", since they are exactly the same as those shown in table 4.1. As already mentioned in section 4.1.3, the extension of $\equiv$ to a congruence relation is obvious. As for the inference rules of table 4.2, the inference rules in table 5.2 define exactly the set of correct (or valid) transitions (with respect to $\tilde{\mathsf{T}}_{\mathcal{C}}$).

Informally, a trace is a sequence of labels of the transition system $\tilde{\mathsf{T}}_{\mathcal{C}}$, *i.e.*, a sequence of guarded actions (in normal form).

5.1 Definition (trace). *Let* $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn},\ \mathrm{C}\Sigma,\ \mathcal{R},\ \mathfrak{A},\ \mathcal{T}r,\ \mathcal{R}^{\mathrm{p}},\ \mathrm{I}\!\mathrm{I}\mathcal{R},\ p^{\mathtt{i}} \rangle$ *be a component. A* trace $\mathtt{t}$ *is a (possibly infinite) sequence of (closed) guarded actions* $a_i \in \mathcal{G}(\mathrm{C}\Sigma, \varnothing)$ $(\forall i > 0)$

$$\mathtt{t} \ = \ a_1 \,;\, a_2 \,;\, \ldots \,;\, a_i \,;\, \ldots \ = \ (a_i)_{i \geq 0} \tag{5.1}$$

In the sequel, we note sets of traces as T.

For a finite trace $\mathtt{t} = a_1; \ldots ; a_n$, we call n the *length* of the trace $\mathtt{t}$. Notice that definition 5.1 allows a trace to have the length 0. In the sequel we call traces of length 0 *empty* and let ε denote an empty trace.

In order to define the semantics of a process, we have to relate traces to executions with respect to the labeled transition system $\tilde{\mathsf{T}}$. For this purpose, we use the notion of a *transition sequence* which describes the executions or derivations of a transition system. We define transition sequences with respect to a general transition system, since we reuse the same notion later on for the transition system providing the compositional semantics.

5.2 Definition (transition sequence). *A* transition sequence d *of a labeled transition system* $T = \langle Q, \rightarrow, q \rangle$ *is a sequence*

$$d \ = \ q_1 \stackrel{\ell_1}{\longrightarrow} q_2 \stackrel{\ell_2}{\longrightarrow} q_3 \stackrel{\ell_3}{\longrightarrow} \ldots \tag{5.2}$$

where the q_i *are states of the transition system (i.e.,* $\forall i \geq 1, q_i \in Q$*), and the transitions between the states are correct with respect to the transition relation* $\rightarrow$ *of the transition system (i.e.,* $\forall i, (q_i,\ \ell_i,\ q_{i+1}) \in \rightarrow$*).*

In the sequel, we denote the set of transition sequences for a given initial state q and a transition system T by $TS_T(q)$. For instance, we say that the transition sequence

$$d \ = \ \big(\langle F_1,\, p_1\rangle \stackrel{a_1}{\longrightarrow} \langle F_2,\, p_2\rangle \stackrel{a_2}{\longrightarrow} \langle F_3,\, p_3\rangle \stackrel{a_3}{\longrightarrow} \ldots\big) \in TS_{\tilde{\mathsf{T}}}\big(\langle F,\, p\rangle\big) \tag{5.3a}$$

(with $F_1 = F$ and $p_1 = p$) *produces* (written $d \hookrightarrow \mathtt{t}$) the trace

$$\mathtt{t} \ = \ a_1 \,;\, a_2 \,;\, a_3 \,;\, \ldots \tag{5.3b}$$

Furthermore, we write $d \cdot d'$ for the *concatenation* of a finite transition sequences $d = q_1 \stackrel{\ell_1}{\longrightarrow} q_2 \stackrel{\ell_2}{\longrightarrow} q_3 \stackrel{\ell_3}{\longrightarrow} \ldots \stackrel{\ell_{n-1}}{\longrightarrow} q_n$ and a (possibly infinite) transition sequence $d' = q'_1 \stackrel{\ell'_1}{\longrightarrow} q'_2 \stackrel{\ell'_2}{\longrightarrow} q'_3 \stackrel{\ell'_3}{\longrightarrow} \ldots$ such that the states q'_1 and q_n are equivalent, *i.e.*, for all states q and for all transition labels ℓ we have that $(q'_1 \stackrel{\ell}{\rightarrow} q) \Leftrightarrow (q_n \stackrel{\ell}{\rightarrow} q)$.

$$\begin{aligned} \mathsf{success}\,;p \;&\equiv\; p \\ \mathsf{success} \parallel p \;&\equiv\; p \end{aligned} \qquad (Unit_{\equiv})$$

$$\begin{aligned} p_1 \parallel p_2 &\equiv p_2 \parallel p_1 \\ p_1 + p_2 &\equiv p_2 + p_1 \end{aligned} \qquad (Comm_{\equiv})$$

Table 5.1: Axiom Schemes Defining the Structural Congruence $\equiv$ on Process Terms

$$\frac{p \equiv p' \qquad \langle F,\, p'\rangle \xrightarrow{a} \langle F',\, p''\rangle \qquad p'' \equiv p'''}{\langle F,\, p\rangle \xrightarrow{a} \langle F',\, p'''\rangle} \qquad (\tilde{\mathsf{R}}_{\equiv})$$

$$\frac{F \vdash g}{\big\langle F,\ \big[g \;\Rightarrow\; \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle ; \ldots ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big]\big\rangle \xrightarrow{[g \Rightarrow \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle ; \ldots ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle]} \Big\langle exec\Big(sel\big(\widehat{sn},\ \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle ; \ldots ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big),\ F\Big),\ \mathsf{success}\Big\rangle} \qquad (\tilde{\mathsf{R}}_{action})$$

$$\frac{\big(\mathrm{q}(x_1,\, \ldots\, ,\, x_n) \Leftarrow \bigoplus_{i=1}^{m}([g_i \Rightarrow a_i]\,;p_i)\big) \in \mathcal{R}^{\mathrm{p}} \qquad \big\langle F,\ \big(\bigoplus_{i=1}^{m} rename([g_i \Rightarrow a_i\Downarrow]\,;p_i)\big)[v_j/x_j]\big\rangle \xrightarrow{a} \langle F',\, p'\rangle}{\langle F,\, \mathrm{q}(v_1,\, \ldots\, ,\, v_n)\rangle \xrightarrow{a} \langle F',\ p'\Downarrow\rangle} \qquad (\tilde{\mathsf{R}}_{call})$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a} \langle F',\, p_1'\rangle}{\langle F,\, p_1\,;p_2\rangle \xrightarrow{a} \langle F',\, p_1'\,;p_2\rangle} \qquad (\tilde{\mathsf{R}}_{;})$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \parallel p_2\rangle \xrightarrow{a} \langle F',\, p_1' \parallel p_2\rangle} \qquad (\tilde{\mathsf{R}}_{\parallel})$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a} \langle F',\, p_1'\rangle}{\langle F,\, p_1 + p_2\rangle \xrightarrow{a} \langle F',\, p_1'\rangle} \qquad (\tilde{\mathsf{R}}_{+})$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{a} \langle F',\, p_1'\rangle} \qquad (\tilde{\mathsf{R}}_{\oplus})$$

$$\frac{\langle F,\, p_2\rangle \xrightarrow{a} \langle F',\, p_2'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{a} \langle F',\, p_2'\rangle} \qquad \begin{array}{l}\text{if } \nexists\, p_1',\, \nexists\, F'',\, \nexists\, a', \text{ such that} \\ \langle F,\, p_1\rangle \xrightarrow{a'} \langle F'',\, p_1'\rangle\end{array} \qquad (\tilde{\mathsf{R}}'_{\oplus})$$

Table 5.2: Labeled Inference Rules Defining the Transition Relation $\xrightarrow{a}$ of $\tilde{\mathsf{T}}_{\mathcal{C}}$

We call a transition sequence d *maximal* if d cannot be prolonged, *i.e.*, if d is finite there is no further transition possible from the final state of d. A trace $\mathtt{t}$ is called *maximal* if there exists a maximal transition sequence d such that $d \hookrightarrow \mathtt{t}$. Notice that not all traces are produced by a transition sequence, and that there might be several transition sequences producing the same trace, even if we consider only transition sequences for the same store and process term, due to the operator of *non-deterministic* choice $+$[1].

In a trace based semantics, a process is characterised by the *set* of all its possible traces, that is to say, those traces that can possibly be produced by a transition sequence corresponding to an execution of the process. Using definitions 5.1 and 5.2 we can define a first semantics of processes based on the transition system $\tilde{\mathsf{T}}_{\mathcal{C}}$, *i.e.*, based on the operational semantics of chapter 4 where every transition is labeled by the executed action.

5.3 Definition (operational semantics: $\mathcal{O}$). *Let $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \Pi\mathcal{R}, p^{\mathtt{i}} \rangle$ be a component. The* operational semantics *$\mathcal{O}$ associates to a process term p and a store F the set of all maximal traces produced by the transition sequences (with respect to the transition system $\tilde{\mathsf{T}}_{\mathcal{C}}$) of the process term p when started on the store F:*

$$\mathcal{O}(p,\, F) \;\stackrel{\text{def}}{=}\; \left\{ \mathtt{t} \;\middle|\; \exists d \in TS_{\tilde{\mathsf{T}}_{\mathcal{C}}}\big(\langle F,\, p\rangle\big) : d \hookrightarrow \mathtt{t} \right\} \tag{5.4}$$

Unfortunately, this first semantics is not compositional as can been seen from the following example.

5.4 Example. *Consider a component signature $\hat{\mathrm{C}\Sigma}$ and a component $\hat{\mathcal{C}}$ defined as*

$$\hat{\mathcal{C}} \;\stackrel{\text{def}}{=}\; \big\langle sn,\, \hat{\mathrm{C}\Sigma},\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing,\, \mathsf{success} \big\rangle \tag{5.5a}$$

$$\hat{\mathrm{C}\Sigma} \;\stackrel{\text{def}}{=}\; \big\langle \langle \{\mathtt{Truth}\},\, \{P, Q : \mathtt{Truth}\} \rangle,\, \hat{\mathrm{A}\Sigma},\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing \big\rangle \tag{5.5b}$$

where the action signature $\hat{\mathrm{A}\Sigma}$ defines at least two elementary actions, namely tell *for adding predicates (i.e., functions of codomain* Truth*), and* skip *which does not do anything. The storename of $\hat{\mathcal{C}}$ is sn, its initial store is empty, i.e., there are no rules defining the two predicates P and Q, and its initial process term is* success*. All the other parts of $\hat{\mathcal{C}}$ are empty, i.e., $\hat{\mathcal{C}}$ does not interact with its environment at all.*

Consider the following two process terms:

$$\hat{p}_1 \;\stackrel{\text{def}}{=}\; [P \Rightarrow \langle sn,\, \mathsf{skip}\rangle]\,;\mathsf{success} \tag{5.5c}$$

$$\hat{p}_2 \;\stackrel{\text{def}}{=}\; [Q \Rightarrow \langle sn,\, \mathsf{skip}\rangle]\,;\mathsf{success} \tag{5.5d}$$

where P and Q are two different atomic predicates. When we start the execution of $\hat{p}_1$ (respectively, $\hat{p}_2$) in the empty initial theory F (which neither entails P nor Q), the set of traces associated to $\hat{p}_1$ (respectively, $\hat{p}_2$) contains only the empty trace, thus

$$\mathcal{O}(\hat{p}_1,\, F) \;=\; \mathcal{O}(\hat{p}_2,\, F) \;=\; \{\varepsilon\} \tag{5.5e}$$

[1] For instance, consider two processes q_1 and q_2 which are both defined by the process definition "$\mathrm{q}_i \Leftarrow [\textsc{true} \Rightarrow a]\,;$ success" ($i \in \{1;2\}$). Then we have two distinct transition sequences for the process term $\mathrm{q}_1 + \mathrm{q}_2$ which both produce the (maximal) trace a.

However, considering the parallel composition of $\hat{p}_1$ (respectively, $\hat{p}_2$) with following process term

$$\hat{p}_3 \stackrel{\text{def}}{=} [\text{TRUE} \Rightarrow \langle sn, \mathsf{tell}(P)\rangle] \,;\mathsf{success} \tag{5.5f}$$

we find that the semantics of $\hat{p}_1 \parallel \hat{p}_3$ and $\hat{p}_2 \parallel \hat{p}_3$ are different:

$$\mathcal{O}(\hat{p}_1 \parallel \hat{p}_3,\, F) = \left\{ [\text{TRUE} \Rightarrow \langle sn, \mathsf{tell}(P)\rangle] \,;\, [P \Rightarrow \langle sn, \mathsf{skip}\rangle] \right\} \tag{5.5g}$$

$$\mathcal{O}(\hat{p}_2 \parallel \hat{p}_3,\, F) = \{\varepsilon\} \tag{5.5h}$$

The source of the non-compositionality of the operational semantics $\mathcal{O}$ is that traces, *i.e.*, sequences of the actions executed by a single process, represent only the behaviour of this process without taking into account its context, that is to say possible actions of other parallel processes or the environment. Thus two processes, *e.g.*, $\hat{p}_1$ and $\hat{p}_2$, although having the same semantics, may behave differently in some contexts, *e.g.*, when executed in parallel with $\hat{p}_3$. Notice that the process $\hat{p}_3$ modifies the current theory, enabling the execution of $\hat{p}_1$. Thus one possible idea to obtain a compositional semantics is to incorporate assumptions or hypotheses about the possible behaviour of the environment.

5.2 Semantics of Labeled Execution Traces

As already mentioned, the main reason of the non-compositionality of the semantics $\mathcal{O}$ based on simple execution traces is that the semantics of a process term does not take into account the behaviour of the environment of the process. Recall that in a compositional semantics, the semantics of a composed process can be computed from the semantics of its components. Stated otherwise, two process terms that have the same compositional semantics, behave in the same way in *every* context. Therefore every compositional semantics needs to distinguish $\hat{p}_1$ and $\hat{p}_2$ of example 5.4, since they behave differently in a particular context, namely the parallel composition with $\hat{p}_3$. The semantics $\mathcal{O}$ cannot be compositional since it assigns the same set of traces to $\hat{p}_1$ and $\hat{p}_2$.

A possibility to obtain a compositional semantics starting from $\mathcal{O}$ is to incorporate in the semantics of $\hat{p}_1$ the fact that if (and only if) another (concurrent) process tells the predicate P, then $\hat{p}_1$ can be executed. This allows to distinguish $\hat{p}_1$ from $\hat{p}_2$ since $\hat{p}_2$ can only execute if the predicate Q is added to the store. We define a semantics containing this additional information by means of the sets of traces of a new labeled transition system. A special inference rule allows to integrate assumptions, or hypotheses, about actions that might be executed by other concurrent processes, into the semantics of a process. To distinguish between the actions that are really executed by the process and those which are supposed, or expected, to be executed by the environment of the process, the actions in the traces are *labeled*.

Extending the ideas of [dBP91] we distinguish *three* labels for actions. An action a^{p} labeled with p is an action that has been executed by the process itself. The other two labels represent "hypothetical actions" that might be executed by the environment of the processes. Reflecting the differences between local and distributed computation we distinguish further between actions that might be executed by concurrent processes of

the local component and actions that might have been received from other components via the mailbox (see section 4.2). We label the former with o and the latter with e[2].

5.5 Definition (labeled action). *Let $\mathrm{C}\Sigma$ be a component signature and X a family of variables. A* labeled action a^ℓ *is a guarded action $a \in \mathcal{G}(\mathrm{C}\Sigma, X)$ the label of which is taken from the set $\ell \in \{\mathsf{p}; \mathsf{o}; \mathsf{e}\}$.*

We denote the set of labeled guarded actions by ${}^\ell\mathcal{G}(\mathrm{C}\Sigma, X)$. We call a labeled guarded action $a^\ell \in {}^\ell\mathcal{G}(\mathrm{C}\Sigma, X)$ hypothetical*, if its label ℓ is either o or e.*

To define the compositional semantics, we introduce a further transition system, namely $\mathsf{CT}_\mathcal{C}$. The transition system $\mathsf{CT}_\mathcal{C}$ is a triple $\mathsf{CT}_\mathcal{C} = \langle \mathsf{Q}, \rightarrow, \langle F, p^{\mathtt{i}} \rangle \rangle$, where the states as well as the initial state are the same as already for the transition systems $\mathsf{T}_\mathcal{C}$ and $\tilde{\mathsf{T}}_\mathcal{C}$. However, the labels of the transitions are labeled actions, *i.e.*, the transition relation $\rightarrow$ is a ternary relation between states, labeled actions and states. The inference rules of $\mathsf{CT}_\mathcal{C}$ are shown in table 5.3, where the congruence relation $\equiv$ is defined in table 5.1. As for the rules of tables 4.2 and 5.2, the inference rules of table 5.3 define exactly the set of transitions.

The first eight rules (above the horizontal line), namely rules $(\mathsf{CR}_\equiv)$, (CR_{action}), (CR_{call}), $(\mathsf{CR}_;)$, $(\mathsf{CR}_\parallel)$, (CR_+), $(\mathsf{CR}_\oplus)$ and $(\mathsf{CR}'_\oplus)$, are essentially the same as the rules of $\tilde{\mathsf{T}}_\mathcal{C}$ shown in table 5.2. The difference is that the transitions are labeled with labeled (guarded) actions. Notice that all actions in these rules are labeled with p since these rules describe the execution of a process. In particular, in rule $(\mathsf{CR}'_\oplus)$, the requirement that the transition in the side-condition has to be labeled with p ensures that only transitions corresponding to executions of p_1 are forbidden.

The two other rules (below the horizontal line), namely rules (CR_o) and (CR_e), allow to incorporate hypothetical actions in the semantics, *i.e.*, the traces of a process. On the one hand, rule (CR_o) mimics rule (CR_{action}), with the only difference of the label of the action: since rule (CR_o) models a hypothetical action that might be executed by a concurrent process of the same component, the action is labeled with o. On the other hand, rule (CR_e) correspond to rule (E) which we already presented in section 4.2 along with the operational semantics of a system of several components. This rule models the execution of (sequences of elementary) actions received in the mailbox (see section 4.2). We do not mention the mailbox explicitly since we are considering only a single component. Since actions received in the mailbox are considered as executed by someone external to the component, they are labeled with the label e.

Labeled traces are the extension of traces to the transition system $\mathsf{CT}_\mathcal{C}$.

5.6 Definition (labeled trace). *Let $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \mathrm{I}\!\mathcal{R}, p^{\mathtt{i}} \rangle$ be a component. A* labeled trace t *is a possibly infinite sequence of (closed) labeled guarded actions $a_i^\ell \in {}^\ell\mathcal{G}(\mathrm{C}\Sigma, \varnothing)$ $(\forall i > 0)$*

$$\mathsf{t} \;=\; a_1^{\ell_1} ; a_2^{\ell_2} ; \ldots ; a_i^{\ell_i} ; \ldots \;=\; (a_i^{\ell_i})_{i \geq 0} \tag{5.6}$$

In the sequel, we note sets of labeled traces as T.

As for traces, we call a labeled trace *maximal* if there a exists a maximal transition sequence (of the transition system $\mathsf{CT}_\mathcal{C}$) producing it (see section 5.1). Similarly, we

[2]The labels are the initial letters of, respectively, process, other processes and environment.

$$\frac{p \equiv p' \qquad \langle F,\, p'\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p''\rangle \qquad p'' \equiv p'''}{\langle F,\, p\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p'''\rangle} \tag{$\mathsf{CR}_{\equiv}$}$$

$$\frac{F \vdash g}{\Big\langle F,\, \big[g \;\Rightarrow\; \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big]\Big\rangle \xrightarrow{[g \Rightarrow \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle]^{\mathsf{p}}} \Big\langle exec\Big(sel\big(\widehat{sn},\; \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big),\; F\Big),\; \mathsf{success}\Big\rangle} \tag{CR_{action}}$$

$$\frac{\big(\mathrm{q}(x_1,\, \ldots\, ,\, x_n) \Leftarrow \bigoplus_{i=1}^{m}([g_i \Rightarrow a_i]\,;p_i)\big) \in \mathcal{R}^{\mathrm{p}} \qquad \Big\langle F,\; \big(\bigoplus_{i=1}^{m} rename([g_i \Rightarrow a_i{\Downarrow}]\,;p_i)\big)[v_j/x_j]\Big\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p'\rangle}{\langle F,\, \mathrm{q}(v_1,\, \ldots\, ,\, v_n)\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\; p'{\Downarrow}\rangle} \tag{CR_{call}}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle}{\langle F,\, p_1\,;p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\,;p_2\rangle} \tag{$\mathsf{CR}_{;}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \parallel p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1' \parallel p_2\rangle} \tag{$\mathsf{CR}_{\parallel}$}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle}{\langle F,\, p_1 + p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle} \tag{CR_{+}}$$

$$\frac{\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_1'\rangle} \tag{$\mathsf{CR}_{\oplus}$}$$

$$\frac{\langle F,\, p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_2'\rangle}{\langle F,\, p_1 \oplus p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F',\, p_2'\rangle} \qquad \begin{array}{l}\text{if } \nexists\, p_1',\, \nexists\, F'',\, \nexists\, a', \text{ such that}\\ \langle F,\, p_1\rangle \xrightarrow{(a')^{\mathsf{p}}} \langle F'',\, p_1'\rangle\end{array} \tag{$\mathsf{CR}'_{\oplus}$}$$

$$\frac{F \vdash g}{\big\langle F,\; p\big\rangle \xrightarrow{[g \Rightarrow \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle]^{\circ}} \Big\langle exec\Big(sel\big(\widehat{sn},\; \langle sn_1,\, \mathsf{a}_1(t_{1,1},\, \ldots\, ,\, t_{1,k_1})\rangle; \ldots\, ; \langle sn_n,\, \mathsf{a}_n(t_{n,1},\, \ldots\, ,\, t_{n,k_n})\rangle\big),\; F\Big),\; p\Big\rangle} \tag{CR_{o}}$$

$$\frac{a \in \mathcal{A}^{\mathcal{N}}(\mathrm{C}\Sigma,\, \varnothing,\, sn)}{\langle F,\, p\rangle \xrightarrow{[\textsc{true}\,\Rightarrow\, a]^{\mathsf{e}}} \big\langle exec(a,\, F),\; p\big\rangle} \tag{CR_{e}}$$

Table 5.3: Labeled Inference Rules Defining the Transition Relation $\xrightarrow{a^{\ell}}$ of $\mathsf{CT}_{\mathcal{C}}$

have further that not all labeled traces are produced by a transition sequence, and that a labeled trace might be produced by several transition sequences.

An important property of a concurrent system is the *fairness* of its executions, see for instance [MP91, chapter 2]. Informally, an execution or labeled trace is called *fair* if every action or transition that is enabled is executed eventually. To define this notion formally, we introduce first the notion of a *p-index* which corresponds to the index (or position) of the *last* action a process has executed, *i.e.*, that is labeled with p.

5.7 Definition (p-index). *The* p-index $index_{\mathsf{p}}(\mathsf{t})$ *of a labeled trace* $\mathsf{t} = (a_i^{\ell_i})_{i>0}$ *is defined as the last (i.e., greatest) index* i *such that the* i*-th action is labeled with* p*. Formally:*

$$index_{\mathsf{p}}(\mathsf{t}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \forall i > 0 \text{ we have } \ell_i \neq \mathsf{p} \\ n & \text{if } \exists n > 0 \text{ such that } (\ell_n = \mathsf{p} \text{ and } \forall i > n \text{ we have } \ell_i \neq \mathsf{p}) \\ \infty & \text{otherwise} \end{cases} \quad (5.7)$$

According to definition 5.7, the p-index of a labeled trace t produced by an execution of $\mathsf{success}$ is 0, since t does not contain any action labeled with p. A p-index of ∞ signifies, informally, that the execution of the process does not terminate.

Using the notion of p-index, we can express the notion of *fairness* formally. Roughly speaking, we consider a labeled trace with a finite p-index, *e.g.*, n, as fair if it corresponds to a *terminating* execution of a process term, *i.e.*, where the final process term is $\mathsf{success}$. Additionally, all traces with an infinite p-index are considered as fair as well. Thus, a trace is fair, if a process executes eventually all actions it can execute.

5.8 Definition (fair labeled trace). *We call a labeled trace* $\mathsf{t} = (a_i^{\ell_i})_{i\geq 0}$ fair *(with respect to a process term* p *and a store* F*) if*

1. *either the number of actions of* t *labeled with* p *is infinite or null, i.e.,* $index_{\mathsf{p}}(p) \in \{0; \infty\}$*, or*

2. $index_{\mathsf{p}}(\mathsf{t}) = n$ *and there exists a transition sequence* d *producing* t *such that* p_{n+1}*, the process term of the* $(n+1)$*-th state of* d *represents the successful termination (of the execution of* p*), i.e.,* $p_{n+1} \equiv \mathsf{success}$*.*

Notice that definition 5.8 considers a labeled trace t execution which does not contain any execution of an action by the process (which would be labeled with p) as fair with respect to *any* process term p (case $index_{\mathsf{p}}(\mathsf{t}) = 0$). While these traces are classically not considered as fair, we need to include them in order to account for the sequential composition of a process *after* a non-terminating process. This should become clear along with the definition of the semantical operator $\tilde{;}$ in section 5.3.1.1.

We extend the notion of fairness to transition sequences as follows. For a fair labeled trace t, we call a transition sequence d producing t satisfying the condition *2* of definition 5.8 a *fair transition sequence* for t.

The semantics $\mathcal{M}$ is defined similar to $\mathcal{O}$, but using labeled traces and transitions with respect to the transition system $\mathsf{CT}_{\mathcal{C}}$. Recall that we note the set of transition sequences with respect to a transition system T and initial state q as $TS_T(q)$ (see definition 5.2).

5.9 Definition (compositional semantics: $\mathcal{M}$). *Let $\mathcal{C}$ be a component defined as $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn},\ \mathrm{C}\Sigma,\ \mathcal{R},\ \mathfrak{A},\ \mathcal{T}r,\ \mathcal{R}^{\mathrm{p}},\ \Pi\!\!I\mathcal{R},\ p^{\mathtt{i}} \rangle$. The* compositional semantics $\mathcal{M}$ *associates to a process term p and a store F the set of all maximal fair labeled traces produced by the transition sequences (with respect to the transition system $\mathsf{CT}_{\mathcal{C}}$) of the process term p when the execution is started on the store F:*

$$\mathcal{M}_{\mathcal{C}}(p, F) \stackrel{\text{def}}{=} \big\{ \mathsf{t} \;\big|\; \mathsf{t} \text{ fair } \textit{and } \exists d \in TS_{\mathsf{CT}_{\mathcal{C}}}\big(\langle F, p\rangle\big) : d \hookrightarrow \mathsf{t} \big\} \tag{5.8}$$

Notice that definitions 5.8 and 5.9 together imply that for any labeled trace in the range of $\mathcal{M}_{\mathcal{C}}$ we have the existence of a maximal fair transition sequence producing it.

We conclude this section with some examples illustrating the semantics $\mathcal{M}_{\mathcal{C}}$. However, since the sets of labeled traces are in general of infinite cardinality, we cannot describe them extensively, and restrict ourselves to more or less formal descriptions of them.

5.10 Example. *The semantics of the process term* $\mathsf{success}$ *is the (infinite) set of all (infinite) labeled traces which do not contain any action labeled with* p *such that there exists a transition sequence (using only the rules* $(\mathsf{CR_o})$ *and* $(\mathsf{CR_e})$*) producing the trace.*

The preceding example shows that for all components $\mathcal{C}$ (which contain at least one action a), we have the following two inclusions, for all stores F and process terms p:

$$\mathcal{M}_{\mathcal{C}}(\mathsf{success}, F) \subseteq \mathcal{M}_{\mathcal{C}}(p, F) \tag{5.9a}$$

$$\varepsilon \notin \mathcal{M}_{\mathcal{C}}(p, F) \tag{5.9b}$$

The reason of the validity of inclusion (5.9b) is that the labeled traces (in the compositional semantics) are required to be *maximal*, and rule $(\mathsf{CR_o})$ allows to make always the assumption of an execution of the (guarded) action $[\textsc{true} \Rightarrow \mathsf{a}]$. Using inclusion (5.9b), we consider in the sequel only labeled traces different from ε, *i.e.*, we write a labeled trace as $(a_i^{\ell_i})_{i>0}$.

5.11 Example. *Reconsider the two processes of example 5.4. We show that the sets of labeled traces associated to $\hat{p}_1$ and $\hat{p}_2$ by the compositional semantics $\mathcal{M}_{\hat{\mathcal{C}}}$ are different. Consider the labeled trace*[3]

$$\mathsf{t} \;=\; \big[\textsc{true} \Rightarrow \langle sn, \mathsf{tell}(P)\rangle\big]^{\mathsf{o}};\ \big[P \Rightarrow \langle sn, \mathsf{skip}\rangle\big]^{\mathsf{p}};\ \Big(\big[\textsc{true} \Rightarrow \langle sn, \mathsf{skip}\rangle\big]^{\mathsf{o}}\Big)^{\infty} \ldots \tag{5.10}$$

We have $\mathsf{t} \in \mathcal{M}_{\hat{\mathcal{C}}}(\hat{p}_1, F)$ but $\mathsf{t} \notin \mathcal{M}_{\hat{\mathcal{C}}}(\hat{p}_2, F)$. Thus the semantics $\mathcal{M}_{\hat{\mathcal{C}}}$ distinguishes between $\hat{p}_1$ and $\hat{p}_2$.

5.3 Compositionality of the Semantics $\mathcal{M}$

Basically, compositionality of a semantics means that the semantics of composed term, say $p_1 \parallel p_2$, can be obtained by composition of the semantics of its constituents, that is in our example the semantics of p_1 and p_2. Hence we need to define for each operator

[3] The tail of t consisting of an infinite number of labeled actions $\big([\textsc{true} \Rightarrow \langle sn, \mathsf{skip}\rangle]^{\mathsf{o}}\big)^{\infty}$ is necessary in order to respect the requirement of a *maximal* labeled trace.

on process terms an associated "semantical" operator. In this section, we define first the semantic operators and state the theorem of the compositionality afterwards.

Before defining the semantical operators, we announce a hypothesis under which we can prove the compositionality of the semantics $\mathcal{M}_{\mathcal{C}}$[4]. Informally, we require that all guards are valid for at least one store, and we can reach this store by means of the execution of actions.

5.12 Hypothesis. *For every guard g and every store F, we have a (closed) action expression in normal form, e.g., a, such that $exec(a, F) \vdash g$ where exec is the function describing the execution of elementary actions of section 4.1.1.*

Notice that hypothesis 5.12 does not imply a strong restriction concerning the components to which the results of this chapter apply. In fact, as we already mentioned in section 1.1.2, we suppose that every declarative language used for the description of stores has a set of predefined actions. Hypothesis 5.12 amounts to require that the set of predefined actions allows to construct *all* possible stores, a property that, in our opinion, a decent set of predefined elementary actions should have. Furthermore, hypothesis 5.12 rules out the possibility of guards which are always false. Notice that in a real program a guard which is known to be *never* valid makes no sense: why considering a situation which can provably never occur?

Another implication of hypothesis 5.12 is that success is the only process term the semantics of which contains only labeled traces that do not contain any action labeled with p. Since a process different form success can only be blocked on a guard, the process can make the assumption of another (concurrent) process executing an action expression such that the guard becomes true; and this action expression exists according hypothesis 5.12.

As a consequence, our semantics $\mathcal{M}$ does not deal with failures, *i.e.*, processes which neither can continue their execution nor have finished their execution successfully. However, we conjecture that no compositional semantics based on (infinite) traces can have both, a general sequential composition and a handling of failures. Indeed, one the one hand, the compositional handling of general sequential composition requires the possibility to deduce successful termination from the set of traces. Furthermore we need to distinguish success from failure, since the subsequent process can start its execution only in the case of success. On the other hand, both, successful termination and failure lead to traces which cannot execute any further actions. Since a trace does not contain any information about the process, it is thus impossible to distinguish between failure and successful termination by considering only the traces. In the metaphor of the beginning of the chapter, the traces in the snow do not tell us if the ski that made them was red or blue.

5.3.1 Semantical Operators

In this section we define the semantical operators associated to the combinators of process terms. These semantical operators are functions taking two sets of labeled traces as arguments and returning a new set of labeled traces. The goal is to define

[4]We sill omit the index $\mathcal{C}$ for the rest of this chapter to alleviate the notation.

these operators such that they mimic the operational behaviour of their counterparts on the process terms, such that we can prove the compositionality of the semantics $\mathcal{M}$.

5.3.1.1 Sequential Composition: $\tilde{;}$

The execution of the process term $p_1 ; p_2$ means to execute first p_1 and, after the successfully terminating execution of p_1, to proceed with the execution of p_2. The main idea of the semantics $\mathcal{M}$ is that a labeled trace for a process term may contain hypothetical transitions thanks to rules ($\mathsf{CR_o}$) and ($\mathsf{CR_e}$). To define the semantical operator $\tilde{;}$ for sequential composition (see definition 5.14), we exploit this feature to construct the labeled traces for $p_1 ; p_2$ by a combination of, on the one hand, a labeled trace t_1 for p_1 which contains a hypothetical execution of p_2 *after* the (real) execution of p_1, and, on the other hand, a labeled trace t_2 which contains a hypothetical execution of p_1 *before* the (real) execution of p_2. The definition of the combination of two labeled traces meeting this conditions is then merely a relabelling of the actions, since the two labeled traces differ only in the labels.

5.13 Example. *Consider the following three labeled traces* t_1, t_2 *and* t_3*:*

$$\mathsf{t}_1 \stackrel{\mathsf{def}}{=} a^{\mathsf{e}} ; b^{\mathsf{o}} ; c^{\mathsf{p}} ; d^{\mathsf{o}} ; e^{\mathsf{o}} ; \mathsf{t}' \tag{5.11a}$$

$$\mathsf{t}_2 \stackrel{\mathsf{def}}{=} a^{\mathsf{e}} ; b^{\mathsf{o}} ; c^{\mathsf{o}} ; d^{\mathsf{o}} ; e^{\mathsf{p}} ; \mathsf{t}' \tag{5.11b}$$

$$\mathsf{t}_3 \stackrel{\mathsf{def}}{=} a^{\mathsf{e}} ; b^{\mathsf{o}} ; c^{\mathsf{p}} ; d^{\mathsf{o}} ; e^{\mathsf{p}} ; \mathsf{t}' \tag{5.11c}$$

where t' *is a labeled trace which does not contain any action labeled with* p.

Suppose that t_1 *(respectively,* t_2*) is a (labeled) trace in the semantics of a process term* p_1 *(respectively,* p_2*), we have that* t_3 *is a (labeled) trace in the semantics of* $p_1 ; p_2$*. In fact, the actions* a *and* b *are hypothetical for both,* p_1 *and* p_2*, i.e., both* p_1 *and* p_2 *suppose that first the environment (i.e., another component) modifies the store by the execution of action* a *and afterwards a concurrent process modifies the store by executing* b*. In a labeled trace for* $p_1 ; p_2$*, we can make the same hypotheses. Since* c *is executed by* p_1 *(and its execution is supposed by* p_2*), it is also executed by* $p_1 ; p_2$*. After the execution of* c*,* p_1 *has terminated.* d *is once more supposed to executed by a concurrent process. Finally, process* p_2 *executes* e*, and therefore* e *is also labeled with* p *in* t_3*.*

As a first step of the definition of the semantic operator $\tilde{;}$, we define a partial operator which defines how to combine two labeled traces that respect appropriate conditions. The extension to sets of labeled traces, together with the selection of matching pairs is presented in a second step.

5.14 Definition ($\tilde{;}$). *Let* $\mathsf{t}_1 = (a_i^{x_i^1})_{i>0}$ *and* $\mathsf{t}_2 = (a_i^{x_i^2})_{i>0}$ *be two labeled sequences such that there are indexes* m*,* $n \in \mathbb{N} \cup \{\infty\}$ *satisfying the following conditions (table 5.4 summarises these conditions):*

- *the* m*-th labeled action is the last action labeled with* p *in* t_1*, i.e.,* $index_{\mathsf{p}}(\mathsf{t}_1) = m$ *(thus* $\forall j$ *such that* $j > m$ *we have that* $x_j^1 \in \{\mathsf{e}; \mathsf{o}\}$*),*
- *the* n*-th labeled action is the first action labeled with* p *in* t_2*, i.e.,* $\forall j$ *such that* $0 < j < n$ *we have that* $x_j^2 \in \{\mathsf{e}; \mathsf{o}\}$*,*

j	$1 \ldots m$	$m+1 \ldots n-1$	$n \ldots$
ℓ_j^1	$\{\mathsf{e};\mathsf{o};\mathsf{p}\}$	$\{\mathsf{e};\mathsf{o}\}$	$\{\mathsf{e};\mathsf{o}\}$
ℓ_j^2	$\{\mathsf{e};\mathsf{o}\}$	$\{\mathsf{e};\mathsf{o}\}$	$\{\mathsf{e};\mathsf{o};\mathsf{p}\}$

For all intervals of j, the label ℓ_j^1 (respectively, ℓ_j^2) of trace t_1 (respectively, t_2) may take a value from the set of the corresponding column.

Table 5.4: Conditions on the Labels of t_1 and t_2 for the operator $\tilde{;}$

- *p_1 has finished its execution before p_2 starts, i.e., ($m < n$ or $m = \infty$[5]) and*
- *in both labeled traces, the actions labeled* e *coincide, i.e., $\forall j$, $x_j^1 = \mathsf{e}$ if and only if $x_j^2 = \mathsf{e}$.*

Then we define the partial operator $\tilde{;}$ by

$$\mathsf{t}_1 \tilde{;} \mathsf{t}_2 \stackrel{\text{def}}{=} (a_i^{y_i})_{i>0} \qquad \text{where} \quad y_i \stackrel{\text{def}}{=} \begin{cases} x_i^1 & \text{if } i \le m \\ x_i^2 & \text{otherwise} \end{cases} \tag{5.12}$$

We leave $\tilde{;}$ undefined for all other pairs of labeled traces.

Notice that this definition of $\tilde{;}$ includes also the cases where (one of) the labeled traces t_1 and t_2 contain no action labeled with p at all. In fact, this may arise in realistic situations. As a first example, consider the case of $p_1 = \mathsf{success}$. Since p_1 has already successfully terminated, there cannot be any action executed by this process, and p_2 may start its execution at once. As a second example, consider the case of a non-terminating process p_1. Now the execution of p_2 *never* starts.

The extension of $\tilde{;}$ to sets of labeled traces is straightforward, we just have to take those labeled traces which can be obtained by combining traces of the sets by $\tilde{;}$ (on pairs of labeled traces).

5.15 Definition ($\tilde{;}$ on sets). *For two sets of labeled traces T_1 and T_2 we define:*

$$T_1 \tilde{;} T_2 \stackrel{\text{def}}{=} \left\{ \mathsf{t} \;\middle|\; \exists \mathsf{t}_1 \in T_1,\ \exists \mathsf{t}_2 \in T_2 \text{ such that: } \mathsf{t} = \mathsf{t}_1 \tilde{;} \mathsf{t}_2 \right\} \tag{5.13}$$

5.3.1.2 Parallelism: $\tilde{\|}$

To execute two processes in parallel, both processes have to make assumptions about the execution of the other. If this is the case, the actions executed by the parallel composition are the actions executed by the processes. Obviously, both processes cannot execute an action at the same time. Similar to the definition of $\tilde{;}$, we define the semantical operator $\tilde{\|}$ in two steps. The following definition is partial in the sense that it considers only labeled traces which agree on the sequences of actions.

[5]Notice that in this case, we necessarily also have $n = \infty$.

5.16 Definition ($\tilde{\|}$). *Let* $\mathsf{t}_1 = (a_i^{x_i^1})_{i>0}$ *and* $\mathsf{t}_2 = (a_i^{x_i^2})_{i>0}$ *be two labeled traces the sequences of actions of which are identical. We define the partial operator* $\tilde{\|}$ *by*

$$\mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2 \stackrel{\text{def}}{=} (a_i^{y_i})_{i>0} \quad \text{where } y_i \stackrel{\text{def}}{=} \begin{cases} \mathsf{p} & \text{if } (x_i^1 = \mathsf{p} \text{ and } x_i^2 = \mathsf{o}) \text{ or } (x_i^1 = \mathsf{o} \text{ and } x_i^2 = \mathsf{p}) \\ \mathsf{o} & \text{if } x_i^1 = x_i^2 = \mathsf{o} \\ \mathsf{e} & \text{if } x_i^1 = x_i^2 = \mathsf{e} \end{cases} \tag{5.14}$$

We leave $\tilde{\|}$ *undefined for all other pairs of labeled traces.*

Intuitively, this partial definition of $\tilde{\|}$ is sufficient, since rule ($\mathsf{CR_o}$) can be used at any moment during the execution of a process, and thus all possible hypothetical actions can be included in the semantics of a process. Therefore we leave $\tilde{\|}$ undefined on all other (pairs of) labeled traces of different kind. The extension of $\tilde{\|}$ to sets of labeled traces, which we note, by abuse of notation, as well by $\tilde{\|}$, is defined in the obvious way.

5.17 Definition ($\tilde{\|}$ on sets). *For two sets of labeled traces* T_1 *and* T_2 *we define:*

$$T_1 \,\tilde{\|}\, T_2 \stackrel{\text{def}}{=} \left\{ \mathsf{t} \,\middle|\, \exists \mathsf{t}_1 \in T_1, \exists \mathsf{t}_2 \in T_2 \text{ such that } \mathsf{t} = \mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2 \right\} \tag{5.15}$$

5.3.1.3 Non-Deterministic Choice: $\tilde{+}$

The operator of non-deterministic choice allows to execute either of the processes.

5.18 Definition ($\tilde{+}$). *For two sets of labeled traces* T_1 *and* T_2 *we define:*

$$T_1 \,\tilde{+}\, T_2 \stackrel{\text{def}}{=} T_1 \cup T_2 \tag{5.16}$$

5.3.1.4 Choice with Priority: $\tilde{\oplus}$

When executing $p_1 \oplus p_2$, the process p_2 is executed if and only if in the configuration in which the rule ($\mathsf{CR}_\oplus$) is applied, rule ($\mathsf{CR}'_\oplus$) cannot be applied. This means in terms of traces, that there is no trace for p_1 that makes the same assumptions as the sequence for p_2 and has its first action labeled p.

5.19 Example. *Consider the following two labeled traces* t_1 *and* t_2*:*

$$\mathsf{t}_1 \stackrel{\text{def}}{=} a^{\mathsf{e}}\,;b^{\mathsf{o}}\,;c^{\mathsf{o}}\,;\mathsf{t}' \tag{5.17a}$$

$$\mathsf{t}_2 \stackrel{\text{def}}{=} a^{\mathsf{e}}\,;b^{\mathsf{o}}\,;c^{\mathsf{p}}\,;\mathsf{t}'' \tag{5.17b}$$

where t' *and* t'' *are arbitrary labeled traces.*

Suppose that t_1 *(respectively,* t_2*) is a (labeled) trace in the semantics of a process term* p_1 *(respectively,* p_2*). Then* t_2 *is a labeled trace produced by an execution of* $p_1 \oplus p_2$ *under the hypothesis that there does not exist another labeled trace for* p_1 *which after the hypothetical actions* a^{e} *and* b^{o} *executes an action. Notice that similar to example 5.13 the hypothetical actions of the simple process term* p_2 *(i.e., the hypothetical actions of* t_2*) become hypothetical actions of the composed process term* $p_1 \oplus p_2$*.*

To define the semantical operator $\tilde{\oplus}$ formally, we introduce the notion of a *hypothetical prefix*, denoting the maximal prefix of a trace without any occurrence of the label p. Hence, the hypothetical prefix of a process describes the hypothetical actions before the process starts its execution.

5.20 Definition (hypothetical prefix). *The* hypothetical prefix *of a labeled sequence* $\mathsf{t} = a_1^{\ell_1}; \ldots ; a_{n-1}^{\ell_{n-1}} ; a_n^{\mathsf{p}} ; \mathsf{t}'$*, where all* $\ell_i \in \{\mathsf{o}, \mathsf{e}\}$ $(\forall i \in \{1; \ldots ; n-1\})$ *is the (finite) labeled sequence*

$$\mathsf{h}_{pref}(s) \stackrel{\text{def}}{=} a_1^{\ell_1}; \ldots ; a_{n-1}^{\ell_{n-1}} \tag{5.18}$$

Notice that the hypothetical prefix is not defined for all labeled traces. In particular, it is undefined for all labeled traces in the semantics of $\mathsf{success}$, *i.e.*, the labeled traces which do not contain any occurrence of the label p (see example 5.10.

Using hypothetical prefixes, we can define the semantical operator for choice with priority as follows.

5.21 Definition ($\tilde{\oplus}$). *For two sets of labeled traces* T_1 *and* T_2 *we define:*

$$T_1 \mathbin{\tilde{\oplus}} T_2 \stackrel{\text{def}}{=} T_1 \cup \left\{ \mathsf{t} \;\middle|\; \begin{array}{c} \mathsf{t} \in T_2 \text{ and } \forall \mathsf{t}' \in T_1 \text{ such that if} \\ \mathsf{h}_{pref}(\mathsf{t}) \text{ and } \mathsf{h}_{pref}(\mathsf{t}') \text{ are defined, then:} \\ \mathsf{h}_{pref}(\mathsf{t}) \neq \mathsf{h}_{pref}(\mathsf{t}') \end{array} \right\} \tag{5.19}$$

5.3.2 Compositionality of the Semantics $\mathcal{M}$

Using the semantical operators we can now state the main theorem of this chapter.

5.22 Theorem (Compositionality of $\mathcal{M}$). *Let* $\mathcal{C} \stackrel{\text{def}}{=} \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \mathrm{I}\!\mathrm{I}\mathcal{R}, p^{\mathtt{i}} \rangle$ *be a component,* p_1 *and* p_2 *two closed process terms (i.e.,* $p_1, p_2 \in \mathcal{P}^{\mathcal{N}}(\mathrm{C}\Sigma, \varnothing)$*) and* F *a store. Then the following equations hold:*

$$\mathcal{M}(p_1 \,;\, p_2,\, F) = \mathcal{M}(p_1,\, F) \mathbin{\tilde{;}} \mathcal{M}(p_2,\, F) \tag{5.20a}$$

$$\mathcal{M}(p_1 \parallel p_2,\, F) = \mathcal{M}(p_1,\, F) \mathbin{\tilde{\parallel}} \mathcal{M}(p_2,\, F) \tag{5.20b}$$

$$\mathcal{M}(p_1 + p_2,\, F) = \mathcal{M}(p_1,\, F) \mathbin{\tilde{+}} \mathcal{M}(p_2,\, F) \tag{5.20c}$$

$$\mathcal{M}(p_1 \oplus p_2,\, F) = \mathcal{M}(p_1,\, F) \mathbin{\tilde{\oplus}} \mathcal{M}(p_2,\, F) \tag{5.20d}$$

According to theorem 5.22, the semantics of a compound process term (and store) can be determined by "composing" the semantics of its constituents using the appropriate semantic combinator, the semantics $\mathcal{M}$ is *compositional*. This property implies that two processes with the same semantics behave in the same way when combined with a third process, since the semantics of the combination can be obtained by combining the semantics of the third process and the semantics of one of the two processes.

5.3.2.1 Auxiliary Lemmas

Before we prove theorem 5.22, we introduce some additional lemmas stating some properties of the semantics $\mathcal{M}$ and the associated transition system. Recall that throughout this and the following section, we suppose that we are given a component

$$\mathcal{C} = \langle sn, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \mathrm{I}\!\mathrm{I}\mathcal{R}, p^{\mathtt{i}} \rangle$$

We use the symbol (possibly with additional indexes and other markings) p (respectively, F) to denote process terms (respectively, stores) with respect to the component signature $C\Sigma$.

Our first lemma formalises the intuition that if a parallel composition of processes executes an action, this action is executed by one of the processes forming the parallel composition.

5.23 Lemma. *For process terms p_1 and p_2, a store F and an action a we have:*

$$\langle F, p_1 \parallel p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p'\rangle \;\Leftrightarrow\; \begin{array}{c} \exists p_1' : \big(p' \equiv (p_1' \parallel p_2) \text{ and } \langle F, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p_1'\rangle\big) \\ \text{or} \\ \exists p_2' : \big(p' \equiv (p_1 \parallel p_2') \text{ and } \langle F, p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p_2'\rangle\big) \end{array} \tag{5.21}$$

Proof. One of the implications, namely $\Leftarrow$, is a simple application of the inference rule for the parallel composition operator (see rules ($\mathsf{CR}_{\parallel}$) and ($\mathsf{CR}_{\equiv}$)). Therefore we do not detail the proof here, and focus on the other implication.

Consider the transition $\langle F, p_1 \parallel p_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p'\rangle$. Inspection of the inference rules in table 5.3 shows that this transition can only be inferred by either rule ($\mathsf{CR}_{\equiv}$) or ($\mathsf{CR}_{\parallel}$). In the case of rule ($\mathsf{CR}_{\parallel}$) the implication is obviously true. In the case of an application of rule ($\mathsf{CR}_{\equiv}$) we have two process terms, say p and p'', and a transition

$$\langle F, p\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p''\rangle \tag{5.22}$$

such that $p \equiv (p_1 \parallel p_2)$ and $p'' \equiv p'$. Without loss of generality[6], we suppose that transition (5.22) can be proved by using a rule different from ($\mathsf{CR}_{\equiv}$) as first inference rule. By the definition of the congruence relation $\equiv$ (see table 5.1) there are four different possibilities for the process term p, which we consider separately.

i) $p = \tilde{p}_1 \parallel \tilde{p}_2$ with $\tilde{p}_1 \equiv p_1$ and $\tilde{p}_2 \equiv p_2$:

Transition (5.22) can only be proved by using rule ($\mathsf{CR}_{\parallel}$) as first inference rule. Thus we have $p'' \equiv (\tilde{p}_1' \parallel \tilde{p}_2)$ and consequently the first condition of (5.21) holds, when we take $p_1' \stackrel{\mathsf{def}}{=} \tilde{p}_1'$ (the transition $\langle F, p_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p_1'\rangle$ is valid since it corresponds to the premise of rule ($\mathsf{CR}_{\parallel}$) which we used to infer the transition (5.22)) and we have $p' \equiv p'' \equiv (\tilde{p}_1' \parallel \tilde{p}_2) \equiv (p_1' \parallel \tilde{p}_2) \equiv (p_1' \parallel p_2)$.

ii) $p = \tilde{p}_2 \parallel \tilde{p}_1$ with $\tilde{p}_1 \equiv p_1$ and $\tilde{p}_2 \equiv p_2$:

Symmetric to case i) (with $p'' \equiv (\tilde{p}_2' \parallel \tilde{p}_1)$ and $p_2' \stackrel{\mathsf{def}}{=} \tilde{p}_2'$).

iii) $p = \tilde{p}_1$ with $\tilde{p}_1 \equiv p_1$:

In this case, we have that $p_2 = \mathsf{success}$. Thus by taking $p_1' \stackrel{\mathsf{def}}{=} p''$ we have that the first condition of (5.21) holds:
$p' \equiv p'' \equiv p_1' \equiv (p_1' \parallel \mathsf{success}) \equiv (p_1' \parallel p_2)$.

iv) $p = \tilde{p}_2$ with $\tilde{p}_2 \equiv p_2$:

Symmetric to case iii) (with $p_1 = \mathsf{success}$ and $p_2' \stackrel{\mathsf{def}}{=} p''$).

[6] Since rule ($\mathsf{CR}_{\equiv}$) is not an axiom scheme, there have to be process terms in the equivalence class of $(p_1 \parallel p_2)$ (respectively, p') such that we can apply another inference rule.

□

For a transition as the one on the left of equation (5.21), we say that "the action a has been executed by p_1 (respectively, p_2)" if the first (respectively, second) of the two conditions on the right of equation (5.21) holds. Notice that this statement is ambiguous, since both conditions on the right of equation (5.21) may hold (consider for instance processes of the form $p \parallel p$).

The following lemma formalises an intuition similar to the one of lemma 5.23, but for the operator of non-deterministic choice, namely that both processes might execute the action.

5.24 Lemma. *For process terms p_1 and p_2, stores F and F' and an action a we have:*

$$\langle F,\, p_1 + p_2\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle \;\Leftrightarrow\; \big(\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle\big) \text{ or } \big(\langle F,\, p_2\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle\big) \tag{5.23}$$

Proof. The implications $\Leftarrow$ is a simple application of the inference rule for the operator of non deterministic choice (see rule (CR_{+})).

Consider the transition $\langle F,\, p_1 + p_2\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle$. Inspection of the inference rules in table 5.3 shows that the transition can only be deduced by either rule $(\mathsf{CR}_{\equiv})$ or (CR_{+}). In the case of rule (CR_{+}) the implication is obviously true. In the case of an application of rule $(\mathsf{CR}_{\equiv})$ we have two process terms, say p and p'', and a transition

$$\langle F,\, p\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p''\rangle \tag{5.24}$$

such that $p \equiv (p_1 + p_2)$ and $p'' \equiv p'$. Similar to the proof for lemma 5.23 (see footnote 6 on page 151), we suppose without loss of generality that transition (5.24) can be proved by using a rule different from $(\mathsf{CR}_{\equiv})$ as first inference rule. By the definition of the congruence relation $\equiv$ (see table 5.1) there are two different possibilities for the process term p, namely $p = \tilde{p}_1 + \tilde{p}_2$ or $p = \tilde{p}_1 + \tilde{p}_2$ (where $\tilde{p}_1 \equiv p_1$ and $\tilde{p}_2 \equiv p_2$). Without loss of generality, we suppose the former, *i.e.*, $p = \tilde{p}_1 + \tilde{p}_2$. Thus according to rule (CR_{+}) there exists a transition $\langle F,\, \tilde{p}_1\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p''\rangle$. Since we have that $p'' \equiv p'$, the first condition of the right side of (5.23) holds. □

The next lemma formalises the intuition that if we have a transition for a sequential composition whose first process term is not equivalent to $\mathsf{success}$, than we can execute the same action with just the first process term (of the sequential composition).

5.25 Lemma. *For process terms p_1 and p_2, a store F and an action a we have:*

$$\begin{aligned}&\big(\langle F,\, p_1\,;p_2\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle \text{ and } p_1 \not\equiv \mathsf{success}\big)\\ &\quad\Rightarrow\quad \big(\exists p_1' \text{ such that } \langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p_1'\rangle \text{ and } p' \equiv (p_1'\,;p_2)\big)\end{aligned} \tag{5.25}$$

Proof. Notice that under the assumption that $p_1 \not\equiv \mathsf{success}$ the only inference rule allowing to infer the transition $\langle F,\, p_1\,;p_2\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p'\rangle$ is $(\mathsf{CR}_{;})$[7]. By inspection of this rule and its premise, we have that there exists a process term p_1' such that $\langle F,\, p_1\rangle \xrightarrow{a^{\mathsf{P}}} \langle F',\, p_1'\rangle$ and by rule $(\mathsf{CR}_{\equiv})$ we conclude that $p' \equiv (p_1'\,;p_2)$. □

[7]Similar to the proofs of the lemmas 5.23 and 5.24 (see footnote 6 on page 151), we do not consider rule $(\mathsf{CR}_{\equiv})$ without loss of generality.

The following lemma states that the $(i+1)$-th process term of a transition sequence is different from $\mathsf{success}$ if i is less than the p-index of the trace produced by the transition sequence. Intuitively, this signifies that the process has not terminated its execution.

5.26 Lemma. *For a transition sequence* $d \stackrel{\mathsf{def}}{=} \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \ldots$ *producing a labeled trace* t*, we have that for all* $i \in \mathbb{N}$*:*

$$i < \mathit{index}_{\mathsf{p}}(\mathsf{t}) \;\Rightarrow\; p_{i+1} \not\equiv \mathsf{success} \tag{5.26}$$

Proof. Suppose that there exists $i < \mathit{index}_{\mathsf{p}}(\mathsf{t})$ such that $p_{i+1} \equiv \mathsf{success}$. Let n_0 be the next action after i that is labeled with p, *i.e.*,

$$n_0 \stackrel{\mathsf{def}}{=} \min\{n \,|\, n > i \text{ and } \ell_n = \mathsf{p}\}$$

(n_0 exists according to definition 5.7). Examination of the inference rules defining the transition relation $\rightarrow$ of $\mathsf{CT}_{\mathcal{C}}$ (see table 5.3) shows that for all $n > i$, we have also that $p_n \equiv \mathsf{success}$ (since $p_i \equiv \mathsf{success}$). In particular, we have the transition

$$\langle F_{n_0}, \mathsf{success}\rangle \xrightarrow{a_{n_0}^{\mathsf{p}}} \langle F_{n_0+1}, \mathsf{success}\rangle$$

which is in contradiction to the inference rules of table 5.3 (see also example 5.10). □

The following lemma states that any action executed by a process can also be "executed hypothetically" using rule $(\mathsf{CR}_{\mathsf{o}})$.

5.27 Lemma. *For all stores* F *and* F'*, for all actions* a *and for all process terms* p *we have the following implication:*

$$\langle F, p\rangle \xrightarrow{a^{\mathsf{p}}} \langle F', p'\rangle \;\Rightarrow\; \langle F, p\rangle \xrightarrow{a^{\mathsf{o}}} \langle F', p\rangle \tag{5.27}$$

Proof. Notice that the rules $(\mathsf{CR}_{\mathit{action}})$, $(\mathsf{CR}_{\mathsf{o}})$ and $(\mathsf{CR}_{\mathsf{e}})$ are the only "axiom schemes" for the transition relation $\rightarrow$ of $\mathsf{CT}_{\mathcal{C}}$, that is to say, the only inference rules without premises (in table 5.3) which allow to *introduce* a transition (in contrary to the other inference rules which allow to *infer* new transitions from already introduced (or inferred) transitions). Since the premises of rules $(\mathsf{CR}_{\mathit{action}})$ and $(\mathsf{CR}_{\mathsf{o}})$ are exactly the same, we immediately have the validity of equation (5.27). □

The converse of lemma 5.27 does not hold in general. Indeed, consider a process term p in which the action a does not occur.

Last, but not least, the following lemma states that the process term does not matter for transitions labeled with actions labeled with either o or e. That is to say, for transitions with such labels, the process term can be replaced by any other process term, without invalidating the transition.

5.28 Lemma. *For all stores* F *and* F'*, for all actions*[8] a *and for all process terms* p_1 *and* p_2 *we have the following to equivalences.*

$$\langle F, p_1\rangle \xrightarrow{a^{\mathsf{o}}} \langle F', p_1\rangle \;\Leftrightarrow\; \langle F, p_2\rangle \xrightarrow{a^{\mathsf{o}}} \langle F', p_2\rangle \tag{5.28a}$$

$$\langle F, p_1\rangle \xrightarrow{a^{\mathsf{e}}} \langle F', p_1\rangle \;\Leftrightarrow\; \langle F, p_2\rangle \xrightarrow{a^{\mathsf{e}}} \langle F', p_2\rangle \tag{5.28b}$$

[8] More precisely *almost* all actions, since the guard of the action in equivalence (5.28b) has to be TRUE (see rule $(\mathsf{CR}_{\equiv})$).

Proof. Inspection of the inference rules ($\mathsf{CR_o}$) and ($\mathsf{CR_e}$) shows that the applicability of these inference rules does not depend on the process term. □

Exploiting that all labels of the actions in a hypothetical prefix are either o or e, we can extend lemma 5.28 in a straightforward manner to the following corollary.

5.29 Corollary. *Let* $d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \ldots$ *be a transition sequence (for a store* F_1 *and a process term* p_1*) producing a labeled trace* t *the hypothetical prefix of which has length* $n-1$*. Then we have that for any process term* p' *that*

$$d' \stackrel{\mathsf{def}}{=} \langle F_1, p'\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p'\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p'\rangle \xrightarrow{a_3^{\ell_3}} \ldots \xrightarrow{a_{n-1}^{\ell_{n-1}}} \langle F_n, p'\rangle$$

is a legal transition sequence (for F_1 *and* p'*) producing a labeled trace* t' *such that* $\mathsf{h}_{pref}(\mathsf{t}) = \mathsf{t}'$*.*

Notice that the labeled trace t' in corollary 5.29 is in general not of maximal length, *i.e.*, it can be extended. We leave the proof of corollary 5.29 as an exercise for the reader.

The following corollary of lemma 5.28 states that labeled traces that do not contain any action labeled with p are in the semantics of all process terms, whenever they are in the semantics of a process term (because this ensures that the guards of the hypothetical actions labeled with o are valid).

5.30 Corollary. *Let* $\mathsf{t} = (a_i^{\ell_i})_{i>0}$ *be a labeled trace such that for all* i *we have* $\ell_i \in \{\mathsf{o};\mathsf{e}\}$ *and that there exists a process term* p *and a store* F *with* $\mathsf{t} \in \mathcal{M}(p, F)$*. Then we have for all process terms* p' *that* $\mathsf{t} \in \mathcal{M}(p', F)$*.*

Proof. Consider a transition sequence d for p and F producing t

$$d \stackrel{\mathsf{def}}{=} \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \ldots$$

(*i.e.*, $F_1 \stackrel{\mathsf{def}}{=} F$ and $p_1 \stackrel{\mathsf{def}}{=} p_2$). Since for all i, $\ell_i \in \{\mathsf{o};\mathsf{e}\}$, we have (by the inference rules ($\mathsf{CR_o}$), ($\mathsf{CR_e}$) and ($\mathsf{CR}_{\equiv}$)) that $p_i \equiv p$ (for all i). Thus by lemma 5.28, we have that the transition sequence

$$d' \stackrel{\mathsf{def}}{=} \langle F_1, p'\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p'\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p'\rangle \xrightarrow{a_3^{\ell_3}} \ldots$$

is valid. Since clearly $d' \hookrightarrow \mathsf{t}$, we have that $\mathsf{t} \in \mathcal{M}(p', F)$ (notice that t is fair and maximal by definition). □

Combined with example 5.10, corollary 5.30 implies that for all stores F and all process term p we have that the semantics of $\mathsf{success}$ is a part of the semantics of *all* processes, *i.e.*,

$$\mathcal{M}_{\mathcal{C}}(\mathsf{success}, F) \subseteq \mathcal{M}_{\mathcal{C}}(p, F) \tag{5.29}$$

5.3.2.2 Proof of Theorem 5.22

Using the auxiliary lemmas of the preceding section, we prove the equations of theorem 5.22 one by one. This proof gives a formal justification of the definitions of the semantical operators in section 5.3.1.

To lighten the notation, we use $\hat{p}$ (respectively, $\hat{F}$) instead of p (respectively, F) in the equations we prove. This renaming allows to use the symbols p and F in other contexts, as for instance for the states of transition sequences which are most conveniently written as $\langle F_i, p_i\rangle$.

5.3.2.2.1 Sequential Composition. To prove equation (5.20a), we prove the following two set inclusions:

$\subset$ $\mathcal{M}(\hat{p}_1\,;\hat{p}_2, \hat{F}) \subseteq \mathcal{M}(\hat{p}_1, \hat{F})\,\tilde{;}\,\mathcal{M}(\hat{p}_2, \hat{F})$:

Consider a labeled trace $\mathsf{t} = (a_i^{\ell_i})_{i>0} \in \mathcal{M}(\hat{p}_1\,;\hat{p}_2, \hat{F})$. By definition of $\mathcal{M}$ (see definition 5.9) there exists a maximal fair transition sequence

$$d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$

(with $F_1 \stackrel{\mathsf{def}}{=} \hat{F}$ and $p_1 \stackrel{\mathsf{def}}{=} \hat{p}_1 \parallel \hat{p}_2$) such that $d \hookrightarrow \mathsf{t}$.

Using the transition sequence d (for $\hat{p}_1\,;\hat{p}_2$ and $\hat{F}$) as a starting point, we define two transition sequences d_1 (respectively, d_2) for $\hat{F}$ and $\hat{p}_1$ (respectively, $\hat{p}_2$).

We write $\langle F_i^j, p_i^j\rangle$ (respectively, $(a_i^j)^{\ell_i^j}$) for the i-th state (respectively, labeled action) of the transition sequence d_j ($j \in \{1;2\}$). The actions and stores of d_j are the same as for d, *i.e.*, $a_i^j \stackrel{\mathsf{def}}{=} a_i$ and $F_i^j \stackrel{\mathsf{def}}{=} F_i$ and we set $p_1^j \stackrel{\mathsf{def}}{=} \hat{p}_j$ ($j \in \{1;2\}$). Consequently, we have $p_1 \equiv p_1^1\,;p_1^2$. We define the labels ℓ_i^j and process terms p_i^j step by step. Suppose that we have already defined correct transitions for all $i < i_0$ such that $p_i \equiv p_i^1\,;p_i^2$ (for all $i \le i_0$)[9]. Consider the i_0-th transition of d, *i.e.*, $\langle F_{i_0}, p_{i_0}\rangle \xrightarrow{(a_{i_0})^{\ell_{i_0}}} \langle F_{i_0+1}, p_{i_0+1}\rangle$. If $\ell_{i_0} \in \{\mathsf{o};\mathsf{e}\}$ we can define $p_{i_0+1}^j \stackrel{\mathsf{def}}{=} p_{i_0}^j$ and $\ell_{i_0}^j \stackrel{\mathsf{def}}{=} \ell_{i_0}$ ($j \in \{1;2\}$) and have by lemma 5.28 that the corresponding transition is valid. In the case that $\ell_{i_0} = \mathsf{p}$, we distinguish the following two cases:

1. $p_{i_0}^1 \equiv \mathsf{success}$:
   We set $p_{i_0+1}^1 \stackrel{\mathsf{def}}{=} \mathsf{success}$ and $\ell_{i_0}^1 \stackrel{\mathsf{def}}{=} \mathsf{o}$. The corresponding transition is valid thanks to lemmas 5.27 and 5.28. Furthermore, since we have by assumption that $p_{i_0} \equiv (p_{i_0}^1\,;p_{i_0}^2) \equiv p_{i_0}^2$, we can set $p_{i_0+1}^2 \stackrel{\mathsf{def}}{=} p_{i_0+1}$ and $\ell_{i_0}^2 \stackrel{\mathsf{def}}{=} \ell_{i_0} = \mathsf{p}$. Clearly the i_0-th transition of d_2 is valid, and $p_{i_0+1} \equiv p_{i_0+1}^1\,;p_{i_0+1}^2$.
2. $p_{i_0}^1 \not\equiv \mathsf{success}$:
   Since we have, according to lemma 5.25, that there exists a process term $p_{i_0+1}^1$ such that $\langle F_{i_0}, p_{i_0}^1\rangle \xrightarrow{(a_{i_0})^{\mathsf{p}}} \langle F_{i_0+1}^1, p_{i_0+1}^1\rangle$, we can safely set $\ell_{i_0}^1 \stackrel{\mathsf{def}}{=} \mathsf{p}$. Using lemmas 5.27 and 5.28 we can prove the validity of the i_0-th transition of d_2 when setting $p_{i_0+1}^2 \stackrel{\mathsf{def}}{=} p_{i_0}^2$ and $\ell_{i_0}^2 \stackrel{\mathsf{def}}{=} \mathsf{o}$. Finally, lemma 5.25 tells us that $p_{i_0+1} \equiv (p_{i_0+1}^1\,;p_{i_0}^2) \equiv (p_{i_0+1}^1\,;p_{i_0+1}^2)$.

[9] For $i_0 = 1$ we have by the definitions above that $p_1 \equiv p_1^1\,;p_1^2$.

The transition sequences d_1 and d_2 are defined as the limits of the stepwise construction above. d_1 and d_2 produce two labeled traces, which we call t_1 and t_2.

It remains to prove that $\mathsf{t}_j \in \mathcal{M}(\hat{p}_j, \hat{F})$ $(j \in \{1;2\})$ and that $\mathsf{t} = \mathsf{t}_1 \tilde{;} \mathsf{t}_2$. To prove the latter, we define $n_0 \stackrel{\text{def}}{=} \mathit{index}_{\mathsf{p}}(\mathsf{t}_1)$. By construction, we have for all indexes i that

$$\ell_i^j = \mathsf{p} \;\Rightarrow\; \ell_i^{(j+1) \bmod 2} = \mathsf{o} \qquad (\forall j \in \{1;2\}) \tag{5.30a}$$

$$\ell_i = \mathsf{e} \;\Rightarrow\; \ell_i^1 = \ell_i^2 = \mathsf{e} \tag{5.30b}$$

$$\ell_i = \mathsf{o} \;\Rightarrow\; \ell_i^1 = \ell_i^2 = \mathsf{o} \tag{5.30c}$$

$$p_i^1 \not\equiv \mathsf{success} \;\Rightarrow\; \ell_i^2 \neq \mathsf{p} \tag{5.30d}$$

Since we have by lemma 5.26 for all $i < n_0$ that $p_{i+1}^1 \not\equiv \mathsf{success}$, we conclude (using (5.30d)) that $\ell_i^2 \neq \mathsf{p}$ $(\forall i \leq n_0)$. By definition of the p-index we have that for all $i > n_0$ we have $\ell_i^1 \neq \mathsf{p}$. We conclude by the definition of $\tilde{;}$ on pairs of labeled traces (see equation (5.12)) that $\mathsf{t} = \mathsf{t}_1 \tilde{;} \mathsf{t}_2$.

To prove $\mathsf{t}_j \in \mathcal{M}(\hat{p}_j, \hat{F})$ we remark first that by its definition, d_j is a maximal[10] transition sequence for $\hat{p}_j$ and $\hat{F}$ $(j \in \{1;2\})$. It remains to prove the fairness of t_1 and t_2. Reconsider the p-index of t_1, *i.e.*, n_0. If $n_0 \in \{0; \infty\}$, we have by definition 5.8 immediately the fairness of t_1. Suppose now that $0 < n_0 < \infty$. Obviously, by construction of d_1 we have that $n_0 \leq \mathit{index}_{\mathsf{p}}(\mathsf{t})$. Therefore, if $n_0 = \mathit{index}_{\mathsf{p}}(\mathsf{t})$, the fairness of t_1 is a consequence of the fairness of t. On the other hand, we have by construction of d_1 that $p_i^1 \equiv \mathsf{success}$ for all $i > n_0$ (otherwise $\ell^1_{(\mathit{index}_{\mathsf{p}}(\mathsf{t}))} = \mathsf{p}$ by construction of d_1, which leads to the contradiction $\mathit{index}_{\mathsf{p}}(\mathsf{t}_1) > n_0$), and it follows therefore by definition 5.8 the fairness of t_1. To prove the fairness of t_2, notice that in the case $\mathit{index}_{\mathsf{p}}(\mathsf{t}_1) \neq \infty$ we have by construction $\mathit{index}_{\mathsf{p}}(\mathsf{t}) = \mathit{index}_{\mathsf{p}}(\mathsf{t}_2)$, and have thus the fairness of t_2 due to the fairness of t. On the other hand, by construction of d_2, we have that $\mathit{index}_{\mathsf{p}}(\mathsf{t}_1) = \infty$ implies $\mathit{index}_{\mathsf{p}}(\mathsf{t}_2) = 0$ and we have immediately the fairness of t_2 (also by definition 5.8).

$\supset$ $\mathcal{M}(\hat{p}_1 ; \hat{p}_2, \hat{F}) \supseteq \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{;} \mathcal{M}(\hat{p}_2, \hat{F})$:

Let t be a labeled trace in $\mathcal{M}(\hat{p}_1, \hat{F}) \tilde{;} \mathcal{M}(\hat{p}_2, \hat{F})$. According to the definition of $\tilde{;}$ (over sets of labeled traces, see equation (5.13)), there exists $\mathsf{t}_1 \in \mathcal{M}(\hat{p}_1, \hat{F})$ and $\mathsf{t}_2 \in \mathcal{M}(\hat{p}_2, \hat{F})$ such that $\mathsf{t} = \mathsf{t}_1 \tilde{;} \mathsf{t}_2$. Thus by definition of $\mathcal{M}$ we have a maximal fair transition sequence d_1 (respectively, d_2) for $\hat{F}$ and $\hat{p}_1$ (respectively, $\hat{p}_2$) producing t_1 (respectively, t_2). We write $\langle F_i^j, p_i^j \rangle$ (respectively, $(a_i^j)^{\ell_i^j}$) for the i-th state (respectively, labeled action) of the transition sequence d_j (for $j \in \{1;2\}$).

Starting from d_1 and d_2 we construct a transition sequence d for $\hat{p}_1 ; \hat{p}_2$ and $\hat{F}$ producing t. We note $\langle F_i, p_i \rangle$ (respectively, $(a_i)^{\ell_i}$) the i-th state (respectively, labeled action) of the transition sequence d. Since the actions of d_1 and d_2 are the same (by definition of $\tilde{;}$, see equation (5.12)), we define $a_i \stackrel{\text{def}}{=} a_i^1$. Thus we can

[10] By construction, t_j is maximal, because t is maximal $(j \in \{1;2\})$.

also define $F_i \stackrel{\text{def}}{=} F_i^1$ because the execution of actions is deterministic (and thus also $F_i^1 = F_i^2$).

According to lemma 5.26 we have for all $n < \mathit{index}_{\mathsf{p}}(\mathsf{t}_1)$ that $p_{n+1}^1 \not\equiv \mathsf{success}$. Thus for all $n < \mathit{index}_{\mathsf{p}}(\mathsf{t}_1)$, the transition $\langle F_n, p_n^1 ; \hat{p}_2\rangle \xrightarrow{a_n^{\ell_n^1}} \langle F_{n+1}, p_n^1 ; \hat{p}_2\rangle$ is valid (using rule $(\mathsf{CR}_;)$ or lemma 5.28). Thus we define $(\forall n < \mathit{index}_{\mathsf{p}}(\mathsf{t}_1))\ p_n \stackrel{\text{def}}{=} p_n^1 ; \hat{p}_2$ and $\ell_n \stackrel{\text{def}}{=} \ell_n^1$. If $\mathit{index}_{\mathsf{p}}(\mathsf{t}_1) = \infty$, d is completely defined and we have clearly that $d \hookrightarrow \mathsf{t}$ and that t is fair (since $\mathit{index}_{\mathsf{p}}(\mathsf{t}) = \mathit{index}_{\mathsf{p}}(\mathsf{t}_1) = \infty$, see definition 5.8). We conclude that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 ; \hat{p}_2, \hat{F})$ (t is maximal because t_1 is maximal).

Suppose now, that $n_0 \stackrel{\text{def}}{=} \mathit{index}_{\mathsf{p}}(\mathsf{t}_1) \neq \infty$. Hence we can define $p_{n_0} \stackrel{\text{def}}{=} p_{n_0}^1 ; \hat{p}_2$, $\ell_{n_0} \stackrel{\text{def}}{=} \mathsf{p}$ and $p_{n_0+1} \stackrel{\text{def}}{=} p_{n_0+1}^1 ; \hat{p}_2$ (the former two only if $n_0 > 0$). Since d_1 is a fair transition sequence for t_1, we have that $p_{n_0+1}^1 \equiv \mathsf{success}$ (see definition 5.8), and thus $p_{n_0+1} \equiv \hat{p}_2$. By the definition of $\tilde{;}$ on pairs of labeled traces (see equation (5.12)), we have for all $n \leq n_0$ that $\ell_n^2 \in \{\mathsf{o}; \mathsf{e}\}$. Consequently, we have by rules (CR_o) and (CR_e) that $p_{n_0+1}^2 \equiv \hat{p}_2$, and we can define for all $n > (n_0 + 1)$: $p_n \stackrel{\text{def}}{=} p_n^2$ and $\ell_n \stackrel{\text{def}}{=} \ell_n^2$. Obviously, since d is a valid transition sequence and the fairness of t follows from the fairness of t_2, we have $\mathsf{t} \in \mathcal{M}(\hat{p}_1 ; \hat{p}_2, \hat{F})$.

5.3.2.2.2 Parallel Composition. To prove equation (5.20b), we have to prove the following two set inclusions.

$\subset$ $\mathcal{M}(\hat{p}_1 \parallel \hat{p}_2, \hat{F}) \subseteq \mathcal{M}(\hat{p}_1, \hat{F}) \mathbin{\tilde{\|}} \mathcal{M}(\hat{p}_2, \hat{F})$:

Consider a labeled trace $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \parallel \hat{p}_2, \hat{F})$. By definition of $\mathcal{M}$ (see definition 5.9) there exists a transition sequence (with $F_1 \stackrel{\text{def}}{=} \hat{F}$ and $p_1 \stackrel{\text{def}}{=} \hat{p}_1 \parallel \hat{p}_2$)

$$d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$

such that $d \hookrightarrow \mathsf{t}$. Using d as a starting point, we define a transition sequence d_1 (respectively, d_2) for $\hat{F}$ and $\hat{p}_1$ (respectively, $\hat{p}_2$), by considering the transitions of d one by one. We write $\langle F_i^j, p_i^j\rangle$ (respectively, $(a_i^j)^{\ell_i^j}$) for the i-th state (respectively, labeled action) of the transition sequence d_j ($j \in \{1; 2\}$). We define $a_i^j \stackrel{\text{def}}{=} a_i$ and $F_i^j \stackrel{\text{def}}{=} F_i$ ($j \in \{1; 2\}$).

Thus, the initial state of d_j is defined as $\langle F_1^j, p_1^j\rangle \stackrel{\text{def}}{=} \langle \hat{F}, \hat{p}_j\rangle$, and we clearly have that $p_1 = (\hat{p}_1 \parallel \hat{p}_2) \equiv (p_1^1 \parallel p_1^2)$. Along with the definition of the transition sequences d_j we prove that for all i:

$$p_{i+1} \equiv (p_{i+1}^1 \parallel p_{i+1}^2) \qquad \text{(INV)}$$

Clearly, (INV) holds for $i = 0$ (since $p_1 \equiv (p_1^1 \parallel p_1^2)$ by definition). Now, assuming that (INV) holds for all $i < i_0$, we define $p_{i_0+1}^j$ and $\ell_{i_0}^j$ such that (INV) holds for i_0 and the following is a correct transition with respect to $\mathsf{CT}_\mathcal{C}$:

$$\langle F_{i_0}^j, p_{i_0}^j\rangle \xrightarrow{(a_{i_0}^j)^{\ell_{i_0}^j}} \langle F_{i_0+1}^j, p_{i_0+1}^j\rangle$$

We distinguish the following two cases:

$\ell_{i_0} \in \{\mathsf{o}; \mathsf{e}\}$**:** Define $p^j_{i_0+1} \stackrel{\text{def}}{=} p_{i_0+1}$ and $\ell^j_{i_0} \stackrel{\text{def}}{=} \ell_{i_0}$ $(j \in \{1;2\})$. Obviously (INV) holds $\big(p_{i_0+1} \equiv p_{i_0} \equiv (p^1_{i_0} \parallel p^2_{i_0}) \equiv (p^1_{i_0+1} \parallel p^2_{i_0+1})\big)$ and the transitions are valid by lemma 5.28.

$\ell_{i_0} = \mathsf{p}$**:** By lemma 5.23 (and rule ($\mathsf{CR}_{\equiv}$)) we have three possibilities to consider: either the right, the left or both condition(s) of (5.21) hold. In the latter case, we choose (arbitrarily) one of the two other cases, since both are equally applicable. Without loss of generality, suppose that $p_{i_0+1} \equiv \big((p^1_{i_0})' \parallel p^2_{i_0}\big)$ and $\langle F_{i_0}, p^1_{i_0}\rangle \xrightarrow{a^{\mathsf{p}}_{i_0}} \langle F_{i_0+1}, (p^1_{i_0})'\rangle$. Define $p^1_{i_0+1} \stackrel{\text{def}}{=} (p^1_{i_0})'$, $p^2_{i_0+1} \stackrel{\text{def}}{=} p^2_{i_0}$, $\ell^1_{i_0} \stackrel{\text{def}}{=} \mathsf{p}$ and $\ell^2_{i_0} \stackrel{\text{def}}{=} \mathsf{o}$. By lemma 5.23 we have the validity of the transition and (INV), since $p_{i_0+1} \equiv \big((p^1_{i_0})' \parallel p^2_{i_0}\big) \equiv (p^1_{i_0+1} \parallel p^2_{i_0+1})$.

The transition sequences d_1 and d_2 are defined as the limits of the stepwise construction above. Let t_j be the labeled trace produced by d_j, *i.e.*, $d_j \hookrightarrow \mathsf{t}_j$ $(j \in \{1;2\})$.

Since by construction $\mathsf{t}_j \in \mathcal{M}(\hat{p}_j, \hat{F})$ (t_j (respectively, d) is maximal and fair, since t (respectively, d_j) is maximal and fair), it remains to prove that we have $\mathsf{t} = \mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2$. By construction, we have for all i, $a^1_i = a^2_i$, $(\ell^1_i = \mathsf{e}) \Leftrightarrow (\ell^2_i = \mathsf{e})$ and the following two implications $(\ell^1_i = \mathsf{p}) \Rightarrow (\ell^2_i = \mathsf{o})$ and $(\ell^2_i = \mathsf{p}) \Rightarrow (\ell^1_i = \mathsf{o})$. Consequently, $\mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2$ is defined and clearly equals t.

$\supset$ $\mathcal{M}(\hat{p}_1 \parallel \hat{p}_2, \hat{F}) \supseteq \mathcal{M}(\hat{p}_1, \hat{F}) \,\tilde{\|}\, \mathcal{M}(\hat{p}_2, \hat{F})$:

Let t be a labeled trace in $\mathcal{M}(\hat{p}_1, \hat{F}) \,\tilde{\|}\, \mathcal{M}(\hat{p}_2, \hat{F})$. By the definition of $\tilde{\|}$ (over sets of labeled traces, see equation (5.15)) there exists labeled traces t_1 $(\in \mathcal{M}(\hat{p}_1, \hat{F}))$ and t_2 $(\in \mathcal{M}(\hat{p}_2, \hat{F}))$ for the store $\hat{F}$ and the process term $\hat{p}_1$ respectively, $\hat{p}_2$ such that $\mathsf{t} = \mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2$. We construct a transition sequence d for $\hat{F}$ and $\hat{p}_1 \parallel \hat{p}_2$ producing t to show that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \parallel \hat{p}_2, \hat{F})$.

Since $\mathsf{t}_1 \in \mathcal{M}(\hat{p}_1, \hat{F})$ (respectively, $\mathsf{t}_2 \in \mathcal{M}(\hat{p}_2, \hat{F})$), there exists a maximal fair transition sequence d_1 (respectively, d_2) such that $d_1 \hookrightarrow \mathsf{t}_1$ (respectively, $d_2 \hookrightarrow \mathsf{t}_2$). As above we note the i-th state (respectively, labeled action) of d_j by $\langle F^j_i, p^j_i\rangle$ (respectively, $(a^j_i)^{\ell^j_i}$) $(j \in \{1;2\})$. Similarly, we write the i-th labeled action of t as $a^{\ell_i}_i$. Notice that $F^j_0 = \hat{F}$ $(j \in \{1;2\})$.

Since $\mathsf{t}_1 \,\tilde{\|}\, \mathsf{t}_2 = \mathsf{t}$, we have immediately that for every i: $a^1_i = a^2_i = a_i$. Since the execution of an action is deterministic, the execution of the same action has the same effect, and we conclude that $F^1_i = F^2_i$ (for all i). Hence we define the i-th state of d as follows: $F_i \stackrel{\text{def}}{=} F^1_i$ and $p_i \stackrel{\text{def}}{=} (p^1_i \parallel p^2_i)$. Since the labeled actions (including their labels) are already fixed by t and t is fair and maximal (by construction), it remains only to prove that d is a legal transition sequence, *i.e.*, we have to justify the transitions.

Consider the i-th transition, *i.e.*, $\langle F_i, p^1_i \parallel p^2_i\rangle \xrightarrow{a^{\ell_i}_i} \langle F_{i+1}, p^1_{i+1} \parallel p^2_{i+1}\rangle$. We distinguish two different cases:

$\ell_i \in \{\mathsf{o}; \mathsf{e}\}$**:** By lemma 5.28 we have the validity of the transition.

$\ell_i = \mathsf{p}$**:** By the definition of $\tilde{\|}$ (equation 5.14), we have either ($\ell_i^1 = \mathsf{p}$ and $\ell_i^2 = \mathsf{o}$) or ($\ell_i^1 = \mathsf{o}$ and $\ell_i^2 = \mathsf{p}$). Without loss of generality, we consider the former.

Thus we have the transition $\langle F_i, p_i^1\rangle \xrightarrow{a_i^{\mathsf{p}}} \langle F_{i+1}, p_{i+1}^1\rangle$ and $p_{i+1} \equiv (p_{i+1}^1 \parallel p_i^2)$ (since according to rule ($\mathsf{CR_o}$) necessarily $p_{i+1}^2 \equiv p_i^2$), so that we can apply lemma 5.23 to prove the validity of the transition.

5.3.2.2.3 Nondeterministic Choice. To prove equation (5.20c), we prove the following two set inclusions:

$\subset$ $\mathcal{M}(\hat{p}_1 + \hat{p}_2, \hat{F}) \subseteq \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{+} \mathcal{M}(\hat{p}_2, \hat{F})$:

Let $\mathsf{t} = (a_i^{\ell_i})_{i>0}$ be a fair maximal labeled trace in $\mathcal{M}(\hat{p}_1 + \hat{p}_2, \hat{F})$. If t does not contain any action labeled with p, we have by corollary 5.30 that $\mathsf{t} \in \mathcal{M}(\hat{p}_j, \hat{F})$ ($j \in \{1;2\}$), and we have by definition of $\tilde{+}$ that $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{+} \mathcal{M}(\hat{p}_2, \hat{F})$.

Suppose now that the length of the hypothetical prefix of t is $n-1$. Consider a transition sequence for $\hat{p}_1 + \hat{p}_2$ and $\hat{F}$ (*i.e.*, $F_1 \stackrel{\mathsf{def}}{=} \hat{F}$ and $p_1 \stackrel{\mathsf{def}}{=} \hat{p}_1 + \hat{p}_2$)

$$d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$

producing t.

Consider the n-th transition of d, *i.e.*, $\langle F_n, p_n\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1}, p_{n+1}\rangle$. Since by rules ($\mathsf{CR_o}$) and ($\mathsf{CR_e}$) $p_n \equiv \hat{p}_1 + \hat{p}_2$ we have by lemma 5.24 (and rule ($\mathsf{CR}_{\equiv}$)) that $\langle F_n, \hat{p}_1\rangle \xrightarrow{a^{\mathsf{p}}} \langle F_{n+1}, p_{n+1}\rangle$ or $\langle F_n, \hat{p}_2\rangle \xrightarrow{a^{\mathsf{p}}} \langle F_{n+1}, p_{n+1}\rangle$. Without loss of generality, we suppose the former.

We define the transition sequences $d' \stackrel{\mathsf{def}}{=} \langle F_1, \hat{p}_1\rangle \xrightarrow{a_1^{\ell_1}} \dots \xrightarrow{a_{n-1}^{\ell_{n-1}}} \langle F_n, \hat{p}_1\rangle$ and

$$d'' \stackrel{\mathsf{def}}{=} \langle F_n, \hat{p}_1\rangle \xrightarrow{a_n^{\ell_n}} \langle F_{n+1}, p_{n+1}\rangle \xrightarrow{a_{n+1}^{\ell_{n+1}}} \langle F_{n+2}, p_{n+2}\rangle \xrightarrow{a_{n+2}^{\ell_{n+2}}} \dots$$

(notice that d' is valid by corollary 5.29 and d'' is valid since it is a part of d). Thus $d \cdot d''$ is a fair maximal transition sequence for $\hat{p}_1$ and $\hat{F}$ producing t. We conclude that $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F})$ and consequently $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{+} \mathcal{M}(\hat{p}_2, \hat{F})$.

$\supset$ $\mathcal{M}(\hat{p}_1 + \hat{p}_2, \hat{F}) \supseteq \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{+} \mathcal{M}(\hat{p}_2, \hat{F})$:

Let t be a fair maximal labeled trace in $\mathcal{M}(p_1, \hat{F}) \tilde{+} \mathcal{M}(p_2, \hat{F})$. Thus according to the definition of $\tilde{+}$ (see equation (5.16)), $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F})$ or $\mathsf{t} \in \mathcal{M}(\hat{p}_2, \hat{F})$. Without loss of generality, we suppose that $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F})$. Let

$$d \stackrel{\mathsf{def}}{=} \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$

be a transition sequence for $\hat{p}_1$ and $\hat{F}$ (*i.e.*, $p_1 = \hat{p}_1$ and $F_1 = \hat{F}$) producing t. If the hypothetical prefix of t does not exist, we have by corollary 5.30 that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 + \hat{p}_2, \hat{F})$. Let now be $n-1$ the length of the hypothetical prefix of t.

By corollary 5.29 we construct a transition sequence

$$d' \stackrel{\mathsf{def}}{=} \langle F_1, \hat{p}_1 + \hat{p}_2\rangle \xrightarrow{a_1^{\ell_1}} \langle F_1, \hat{p}_1 + \hat{p}_2\rangle \xrightarrow{a_2^{\ell_2}} \dots \xrightarrow{a_{n-1}^{\ell_{n-1}}} \langle F_n, \hat{p}_1 + \hat{p}_2\rangle$$

Using rule (CR_+) we have that the transition $\langle F_n,\, \hat{p}_1 + \hat{p}_2\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1},\, p_{n+1}\rangle$ is valid. Consequently, defining

$$d'' \stackrel{\mathsf{def}}{=} \langle F_n,\, \hat{p}_1 + \hat{p}_2\rangle \xrightarrow{a_n^{\ell_n}} \langle F_{n+1},\, p_{n+1}\rangle \xrightarrow{a_{n+1}^{\ell_{n+1}}} \langle F_{n+2},\, p_{n+2}\rangle \xrightarrow{a_{n+2}^{\ell_{n+2}}} \ldots$$

$d' \cdot d''$ is a fair maximal transition sequence for $\hat{p}_1 + \hat{p}_2$ and $\hat{F}$ producing t, and we conclude $\mathsf{t} \in \mathcal{M}(\hat{p}_1 + \hat{p}_2,\, \hat{F})$.

5.3.2.2.4 Choice with Priority. To prove equation (5.20d), we prove the following two set inclusions:

$\subset$ $\mathcal{M}(\hat{p}_1 \oplus \hat{p}_2,\, \hat{F}) \subseteq \mathcal{M}(\hat{p}_1,\, \hat{F}) \,\tilde{\oplus}\, \mathcal{M}(\hat{p}_2,\, \hat{F})$:

Consider a fair maximal labeled trace $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \oplus \hat{p}_2,\, \hat{F})$. By the definition of $\mathcal{M}$ we have a transition sequence (with $F_1 \stackrel{\mathsf{def}}{=} \hat{F}$ and $p_1 \stackrel{\mathsf{def}}{=} \hat{p}_1 \oplus \hat{p}_2$)

$$d = \langle F_1,\, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2,\, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3,\, p_3\rangle \xrightarrow{a_3^{\ell_3}} \ldots$$

producing t.

If the hypothetical prefix of t is not defined, t does not contain any action labeled with p and we have by corollary 5.30 that $\mathsf{t} \in \mathcal{M}(\hat{p}_1,\, \hat{F})$, and we have by definition of $\tilde{\oplus}$ (equation (5.19)) that $\mathsf{t} \in \mathcal{M}(\hat{p}_1,\, \hat{F}) \,\tilde{\oplus}\, \mathcal{M}(\hat{p}_2,\, \hat{F})$.

Let now be $n-1$ the length of the hypothetical prefix of t. Notice that by rules (CR_o) and (CR_e) we have that $p_n \equiv \hat{p}_1 \oplus \hat{p}_2$. We distinguish two cases:

1. $\exists \hat{p}_1'$ such that $\langle F_n,\, \hat{p}_1\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1},\, \hat{p}_1'\rangle$:
   We remark that according to rules ($\mathsf{CR}_\oplus$) and ($\mathsf{CR}_\equiv$) there exists a process term $\hat{p}_1''$ such that $p_{n+1} \equiv \hat{p}_1''$ and $\langle F_n,\, \hat{p}_1\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1},\, \hat{p}_1''\rangle$ (this is the premise necessary for proving the n-th transition of d). Consequently,

   $$d' \stackrel{\mathsf{def}}{=} \langle F_n,\, p_n\rangle \xrightarrow{a_n^{\ell_n}} \langle F_{n+1},\, p_{n+1}\rangle \xrightarrow{a_{n+1}^{\ell_{n+1}}} \langle F_{n+2},\, p_{n+2}\rangle \xrightarrow{a_{n+2}^{\ell_{n+2}}} \ldots$$

   is a transition sequence for the process term $\hat{p}_1$ and the store F_n producing the labeled trace $(a_i^{\ell_i})_{i \geq n}$. By corollary 5.29 we can construct a transition sequence d'' for $\hat{p}_1$ and $\hat{F}$ such that the concatenation of d'' and d', *i.e.*, $d'' \cdot d'$, is a transition sequence for $\hat{p}_1$ and $\hat{F}$ producing t. This proves that $\mathsf{t} \in \mathcal{M}(\hat{p}_1,\, \hat{F})$, and by the definition of $\tilde{\oplus}$ (equation (5.19)) we can conclude: $\mathsf{t} \in \mathcal{M}(\hat{p}_1,\, \hat{F}) \,\tilde{\oplus}\, \mathcal{M}(\hat{p}_2,\, \hat{F})$.

2. $\nexists \hat{p}_1'$ such that $\langle F_n,\, \hat{p}_1\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1},\, \hat{p}_1'\rangle$:
   In this case, there exists, according to the premise of rule ($\mathsf{CR}'_\oplus$), a process term $\hat{p}_2'$ such that $\hat{p}_2' \equiv p_{n+1}$ and $\langle F_n,\, \hat{p}_2\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1},\, \hat{p}_2'\rangle$. Thus we can prove similarly to the preceding case 1 that $\mathsf{t} \in \mathcal{M}(\hat{p}_2,\, \hat{F})$.
   According to the definition of $\tilde{\oplus}$ (see equation (5.19)), it remains to prove that for all labeled traces $\mathsf{t}_1 \in \mathcal{M}(\hat{p}_1,\, \hat{F})$ such that the hypothetical prefix of t_1 exists, we have that $\mathsf{h}_{pref}(\mathsf{t}_1) \neq \mathsf{h}_{pref}(\mathsf{t})$.

Suppose now that there exists a labeled trace $\mathsf{t}_1 \in \mathcal{M}(\hat{p}_1, \hat{F})$ such that we have $\mathsf{h}_{pref}(\mathsf{t}_1) = \mathsf{h}_{pref}(\mathsf{t})$. Thus we have a transition $\langle F_n, \hat{p}_1\rangle \xrightarrow{(a')^{\mathsf{p}}} \langle F'_n, \hat{p}'_1\rangle$ with an action a' and a new state $\langle F'_n, \hat{p}'_1\rangle$ (recall that the length of $\mathsf{h}_{pref}(\mathsf{t}) = \mathsf{h}_{pref}(\mathsf{t}_1)$ is $n-1$). But this transition invalidates the n-th transition of d, namely $\langle F_n, \hat{p}_1 \oplus \hat{p}_2\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1}, \hat{p}'_2\rangle$, since the side-condition of rule $(\mathsf{CR}'_{\oplus})$ would not be valid. Because the latter transition is part of the (correct) transition sequence d we started with, we reached a contradiction, and the labeled trace t_1 cannot exist.

$\supset$ $\mathcal{M}(\hat{p}_1 \oplus \hat{p}_2, \hat{F}) \supseteq \mathcal{M}(\hat{p}_1, \hat{F}) \tilde{\oplus} \mathcal{M}(\hat{p}_2, \hat{F})$:

Let a fair maximal labeled trace $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}_1) \tilde{\oplus} \mathcal{M}(\hat{p}_2, \hat{F}_2)$. Then we have according to the definition of $\tilde{\oplus}$ (see equation (5.19)) either $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}_1)$ or $\mathsf{t} \in \mathcal{M}(\hat{p}_2, \hat{F}_2)$. We analyse these two cases separately.

1. $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F})$:
   Consider a transition sequence
   $$d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$
   for $\hat{p}_1$ and $\hat{F}$ (*i.e.*, $p_1 \stackrel{\mathsf{def}}{=} \hat{p}_1$ and $F_1 \stackrel{\mathsf{def}}{=} \hat{F}$) producing t. If the hypothetical prefix of t is not defined, we have by corollary 5.30 that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \oplus \hat{p}_2, \hat{F}_1)$. Suppose now that the length of the hypothetical prefix of t is $n-1$. By corollary 5.29 we have for all $i \in \{1; \dots; n-1\}$ that the following transition is valid $\langle F_i, \hat{p}_1 \oplus \hat{p}_2\rangle \xrightarrow{a_i^{\ell_i}} \langle F_{i+1}, \hat{p}_1 \oplus \hat{p}_2\rangle$ and we call the corresponding transition sequence d'. Furthermore, the transition $\langle F_n, \hat{p}_1 \oplus \hat{p}_2\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1}, p_{n+1}\rangle$ is valid by application of rule $(\mathsf{CR}_{\oplus})$. Writing d'' for the transition sequence
   $$d = \langle F_n, \hat{p}_1 \oplus \hat{p}_2\rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1}, p_{n+1}\rangle \xrightarrow{a_{n+1}^{\ell_{n+1}}} \langle F_{n+2}, p_{n+2}\rangle \xrightarrow{a_{n+2}^{\ell_{n+2}}} \dots$$
   we conclude that $d' \cdot d''$ is a transition sequence for $\hat{p}_1 \oplus \hat{p}_2$ and $\hat{F}$. Since clearly $(d' \cdot d'') \hookrightarrow \mathsf{t}$, we have that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \oplus \hat{p}_2, \hat{F})$.
2. $\mathsf{t} \notin \mathcal{M}(\hat{p}_1, \hat{F})$ (*i.e.*, $\mathsf{t} \in \mathcal{M}(\hat{p}_2, \hat{F})$):
   Consider a transition sequence
   $$d = \langle F_1, p_1\rangle \xrightarrow{a_1^{\ell_1}} \langle F_2, p_2\rangle \xrightarrow{a_2^{\ell_2}} \langle F_3, p_3\rangle \xrightarrow{a_3^{\ell_3}} \dots$$
   for $\hat{p}_2$ and $\hat{F}$ (*i.e.*, $p_1 \stackrel{\mathsf{def}}{=} \hat{p}_2$ and $F_1 \stackrel{\mathsf{def}}{=} \hat{F}$) producing t. If the hypothetical prefix of t is not defined, we have by corollary 5.30 that $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}_1)$, and this situation has already be handled in the preceding case 1.
   Suppose now that the length of the hypothetical prefix of t is $n-1$. We define the transition sequence d' as above, using corollary 5.29.
   Suppose that there exists a process term $\hat{p}'_1$, a store F'_{n+1} and an action a'_n such that we have the transition $\langle F_n, \hat{p}_1 \oplus \hat{p}_2\rangle \xrightarrow{(a'_n)^{\mathsf{p}}} \langle F'_{n+1}, \hat{p}'_1\rangle$. Given

this transition, we can construct a transition sequence $\tilde{d}$ for $\hat{p}_1$ and $\hat{F}$ such that for the labeled trace $\tilde{\mathsf{t}}$ produced by $\tilde{d}$ holds $\mathsf{h}_{pref}(\tilde{\mathsf{t}}) = \mathsf{h}_{pref}(\mathsf{t})$ and $\tilde{\mathsf{t}} \in \mathcal{M}(\hat{p}_1, \hat{F})$. But this is a contradiction, since we have that $\mathsf{t} \notin \mathcal{M}(\hat{p}_1, \hat{F})$, $\mathsf{t} \in \mathcal{M}(\hat{p}_2, \hat{F})$ and $\mathsf{t} \in \mathcal{M}(\hat{p}_1, \hat{F}_1) \tilde{\oplus} \mathcal{M}(\hat{p}_2, \hat{F}_2)$, and thus by definition of $\tilde{\oplus}$ (see equation (5.19)) no labeled trace in $\mathcal{M}(\hat{p}_1, \hat{F})$ (for which the hypothetical prefix is defined) has the same hypothetical prefix as t. Thus we conclude, that we can apply rule $(\mathsf{CR}'_{\oplus})$ to prove the validity of the transition $\langle F_n, \hat{p}_1 \oplus \hat{p}_2 \rangle \xrightarrow{a_n^{\mathsf{p}}} \langle F_{n+1}, \hat{p}'_2 \rangle$, where $\hat{p}'_2 \equiv p_{n+1}$.

Defining d'' as in case 1 above, we conclude that $d' \cdot d''$ is a transition sequence for $\hat{p}_1 \oplus \hat{p}_2$ and $\hat{F}$. Since clearly $(d' \cdot d'') \hookrightarrow \mathsf{t}$, we have that $\mathsf{t} \in \mathcal{M}(\hat{p}_1 \oplus \hat{p}_2, \hat{F})$.

QUOD ERAT DEMONSTRANDUM.

——— ⋆ ——— ⋆ ——— ⋆ ———

In this chapter we have presented a semantics based on labeled traces for a component which we have proven to be compositional, motivated by an example showing that a semantics based on the traces of the operational semantics is not compositional. Our semantics associates to a process term p and an initial store F the set of all traces that can be observed upon execution of the process term p starting in the initial store F. The definition of (labeled) traces we used in this chapter (see definitions 5.1 and 5.6) differs from other definitions of traces, as can be found for instance in [DR95], where a trace is used in order to *specify* a process. In these approaches, a trace is considered as the *equivalence class* of strings (which correspond to our traces) according to a *dependency relation* D.

Compared to the compositional semantics based on labeled traces for ccp as presented in [dBP91], our semantics uses *three* different labels, distinguishing the actions executed by the process, other processes of the same component, and processes from other components. Furthermore, our semantics has operators for general sequential composition and choice with priority which do not exist in the model considered in [dBP91]. Also, the semantics of [dBP91] is restricted to *finite* traces, since all traces are required to have a termination mode. Similarly, the compositional semantics for a non-monotonic extension of ccp presented in [dBKPR93] considers only traces which are *finite* (possibly with a marker for deadlock δ corresponding to an inconsistent store) and *maximal*. Using the marker and the finiteness of the traces, [dBKPR93] define the semantical operator for (general) sequential composition by a simple concatenation of the traces. However, as [dBP91], [dBKPR93] does not provide a operator of choice with priority. With respect to previous version of our compositional semantics [Ser98, ES99, ES], we give a "complete" proof[11], and extend the compositional semantics to a transition system in the style of the CHAM, *i.e.*, using a congruence relation.

Notice that the semantics $\mathcal{M}$ considers fairness in the sense as expressed in definition 5.8, which means informally, that a fair trace represents either a terminating

[11] Notice that the compositionality of the semantics presented in [dBP91] is not proven.

execution of a process (under the necessary assumptions about its context) or no execution at all. One might be tempted to require that the traces retained for $\mathcal{M}$ respect a more restrictive *fairness* condition, as for instance that there has to be at least one action labeled with p. However, due to the presence of both, an operator for general sequential composition and non terminating processes, a process might always be composed sequentially after a process that never terminates, and therefore we are forced to admit labeled traces in the semantics of a process which do contain no action labeled with p. Nevertheless, an implementation should ensure that the execution of a parallel composition is fair, in the sense that all parallel (or concurrent) processes (of a component) have equal chances to proceed their execution. This can be achieved by standard techniques known from operating systems (*e.g.*, time-slicing).

Chapter 6

Secrecy Analysis

The need for *secrecy* increases as more and more private data (credit card numbers, personal medical files, *etc.*) migrate through the Internet, since one needs to ensure that sensible data is not publicly accessible, *i.e.*, remains in some restricted, controlled area. One way to define secrecy from a theoretical point of view is to assign *privacy levels* to data used by a program. High levels (of secrecy) denote highly private data while low levels represent public data. The aim of a secrecy analysis is to show how high and low level data interact with each other. Secrecy is achieved when information may only flow from low levels to higher ones. In other words, public data may influence private data whereas the converse is forbidden, *i.e.*, any modification of private data should not be observable at the public level.

To our knowledge, this kind of approach to secrecy has been first studied in the context of imperative programing in [SVI96]. The extension to concurrent programming is not straightforward as [SV98] shows. Indeed, control-flow may be turned into information-flow. Consider the following sequential, *i.e.*, non concurrent program[1]:

```
while (PIN > 0) loop
   null;
end loop;
SPY := 0;
```

This program verifies secrecy as defined in [SVI96] because at the end of every execution, the value of variable `SPY` is the same, and thus is not influenced by the value of variable `PIN`. Nevertheless, it is clear that in a concurrent setting, it is possible, by a smart combination of many of such processes, to gather informations about the value of `PIN`. Indeed, consider the processes α, β and γ in table 6.1, which are the direct translation of the example given in [BC01b, figure 1][2]. The processes α and β are translations of the `while`-loop shown above, using recursion[3]. Execution of the process term $\alpha \parallel \beta \parallel \gamma$

[1] We use the syntax of ADA [Ada95], where `null` is the instruction which does not do anything, corresponding to the elementary action skip.

[2] Notice that this example itself is a simplification of the example given in [SV98], by using *bool*ean values instead of natural numbers.

[3] Notice that in our framework, we could define α simply by

$$\alpha \Leftarrow [c_\alpha = \mathit{true} \Rightarrow \mathit{SPY} := \mathit{false}\,;\ c_\beta := \mathit{true}]\,;\ \mathsf{success}$$

since a process blocks until its guard becomes valid. While this specification avoids "busy waiting", it

$$
\begin{array}{lcl}
\alpha & \Leftarrow & [c_\alpha = \mathit{false} \Rightarrow \mathsf{skip}]\,;\ \alpha \\
& \oplus & [c_\alpha = \mathit{true} \Rightarrow \mathit{SPY} := \mathit{false}\,;\ c_\beta := \mathit{true}]\,;\ \mathsf{success} \\
\beta & \Leftarrow & [c_\beta = \mathit{false} \Rightarrow \mathsf{skip}]\,;\ \beta \\
& \oplus & [c_\beta = \mathit{true} \Rightarrow \mathit{SPY} := \mathit{true}\,;\ c_\alpha := \mathit{true}]\,;\ \mathsf{success} \\
\gamma & \Leftarrow & [\mathit{PIN} = \mathit{true} \Rightarrow c_\alpha := \mathit{true}]\,;\ \mathsf{success} \\
& \oplus & [\mathit{PIN} = \mathit{false} \Rightarrow c_\beta := \mathit{true}]\,;\ \mathsf{success}
\end{array}
$$

PIN, *SPY*, c_α and c_β are constants of sort *bool* (which is defined by the two constructors *true* and *false*). c_α and c_β are initialised to *false*.

Table 6.1: Information Flow through Control Flow

copies the secret value of *PIN* into the public variable *SPY*.

To circumvent these problems, [SV98] suggest to forbid guards testing secret data in `while` loops, that is to say to allow only tests on low level data in `while` loops. This condition is a bit drastic, and has been relaxed by the introduction of a subtler type system in [BC01a, BC01b]. The idea is that if a guard has a high level of secrecy then all following assignments must be performed at a higher or equal level. Therefore the maximal levels of loop guards have to be taken into account for sequential compositions. To implement this idea, program types in [BC01b] are defined as pairs where the first component is the upper bound of secrecy level of guards, while the second component records the lower bound of the secrecy levels of assigned variables.

6.1 Formalisation of Secrecy

In this section we precisely define the notion of secrecy we consider in this chapter. Informally, we associate to each symbol (of a signature) a secrecy (or privacy) level indicating its status: the higher the privacy level of a symbol, the more private is the status of this symbol. We say that a program respects secrecy (or privacy), if information cannot flow from high privacy levels towards lower privacy levels. In other words, we consider secrecy from a non-interference point of view: a program respects secrecy (or is safe), if there are no interferences from higher secrecy levels towards lower secrecy levels.

Before we give the formal definition of secrecy in section 6.1.2, we present briefly the part of our computation model which we consider in this chapter.

6.1.1 Simplified Model of a Component

We restrict our analysis to the processes of a single component, the store of which is described using the simple declarative language presented in section 3.1.2, on which we use only the five actions tell, del, :=, new and skip. Furthermore, we do not consider process functions. Thus we get the following simplified definition of a component.

does not correspond exactly to the example as given in [BC01b].

6.1 Definition (simplified component). *A* simplified component *is a five-tuple* $\mathcal{C} \stackrel{\text{def}}{=} \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}} \rangle$ *of a signature* Σ*, process symbols* P*, rewrite rules* $\mathcal{R}$*, process definitions* $\mathcal{R}^{\mathrm{p}}$ *and an initial process term* $p^{\mathtt{i}}$*.*

Recall from definition 3.1 that we note the set of terms over a signature Σ and a set of variables X as $T(\Sigma, X)$. In this chapter, we write $\mathcal{P}(\mathcal{C})$ the set of process terms for the simplified component $\mathcal{C}$. Similarly, we note the set of actions defined for a component as $\mathcal{A}(\mathcal{C})$.

The operational semantics of a process term is described by the transition system sT, the rules of which are shown in the tables on page 167. Table 6.6 recalls the axioms defining the congruence relation $\equiv$ on process terms, and table 6.4 recalls the inference rules defining the transition relation $\rightarrow$. Contrary to section 4.1.1, we define the execution of actions in this chapter by an additional transition relation, namely $\twoheadrightarrow$, the inference rules for which are shown in table 6.3.

In the sequel, we sometimes label the transition relations with the (elementary) action executed by the transition. We have omitted the labels in the tables to enhance the readability. We write also $\xrightarrow{a \cdot \mathcal{A}(\mathcal{C})^*}$ a finite sequence of transitions the first of which is labeled with the action a, and the subsequent actions are labeled with any action $\in \mathcal{A}(\mathcal{C})$.

The following straightforward lemma states that we can construct corresponding execution sequences for a same process term on different stores, provides the guards of the actions are valid.

6.2 Lemma. *Let* $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}} \rangle$ *be a component. Then, for all stores* F_1 *and* F_2*, process terms* p *and actions* a*, we have that the two transitions* $\langle F_1, p \rangle \xrightarrow{a} \langle F_1', p' \rangle$ *and* $\langle F_2, p \rangle \xrightarrow{a} \langle F_2', p'' \rangle$ *imply that we have also a transition* $\langle F_2, p \rangle \xrightarrow{a} \langle F_2', p' \rangle$*.*

Proof. Consider two transition sequences $\langle F_1, p \rangle \xrightarrow{a} \langle F_1', p' \rangle$ and $\langle F_2, p \rangle \xrightarrow{a} \langle F_2', p'' \rangle$. Notice that the second transition implies that $F_2 \vdash g$, where g is the guard of the action a, *i.e.*, $a = [g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_n]$. Since the process term p is the same in both transitions, we have, by using the same inference rules as for the first transition, that $\langle F_2, p \rangle \xrightarrow{a} \langle F_2', p' \rangle$. □

6.1.2 Formalisation of Secrecy

The idea of our formalisation of secrecy is to assign a *privacy level* to every symbol of a signature $\Sigma = \langle S, \Omega \rangle$, *i.e.*, all elements of Ω. Following [BC01b] we require the privacy levels to have the structure of a *lattice*.

6.3 Definition (privacy lattice). *A* privacy lattice $\mathfrak{L}$ *is a lattice of privacy levels.*

We note $\sqsubseteq$ (respectively, $\sqsubset$) the (strict) order defined on a privacy lattice $\mathfrak{L}$ and we make no differences between the lattice and the set of its elements, *i.e.*, $\mathfrak{L}$ denotes in the same time the lattice and its carrier. If π_1, π_2 are two elements of $\mathfrak{L}$ and $\pi_1 \sqsubseteq \pi_2$, we say that π_1 is more *private* than π_2. We write $\sqcup$ for the join operation (least upper bound) and $\sqcap$ for the meet operation (greatest lower bound). The maximum of $\mathfrak{L}$ is $\top$ and its minimum is $\bot$. In the rest of this chapter, we consider mostly a privacy lattice of two elements $\top$ and $\bot$, where $\top$ represents secret data, and $\bot$ public data.

$$\begin{array}{rcl} \mathsf{success}\,;p & \equiv & p \\ \mathsf{success} \parallel p & \equiv & p \end{array} \qquad (Unit_{\equiv})$$

$$\begin{array}{c} p_1 \parallel p_2 \equiv p_2 \parallel p_1 \\ p_1 + p_2 \equiv p_2 + p_1 \end{array} \qquad (Comm_{\equiv})$$

Table 6.2: Axiom Schemes Defining the Structural Congruence $\equiv$ on Process Terms

$$\left\langle \langle \Sigma,\, \mathcal{R} \rangle,\, \mathsf{tell}(l \to r \mid c)\,;\mathsf{a} \right\rangle \twoheadrightarrow \left\langle \left\langle \Sigma,\, \mathcal{R} \cup \{l \to r \mid c\} \right\rangle,\, \mathsf{a} \right\rangle \qquad (\mathsf{sR}_{\mathsf{tell}})$$

$$\left\langle \langle \Sigma,\, \mathcal{R} \rangle,\, \mathsf{del}(l \to r \mid c)\,;\mathsf{a} \right\rangle \twoheadrightarrow \left\langle \left\langle \Sigma,\, \mathcal{R} \setminus \{l \to r \mid c\} \right\rangle,\, \mathsf{a} \right\rangle \qquad (\mathsf{sR}_{\mathsf{del}})$$

$$\left\langle \langle \Sigma, \mathcal{R} \rangle,\, (d := v)\,;\mathsf{a} \right\rangle \twoheadrightarrow \left\langle \left\langle \Sigma,\, \left\{ l \to r \mid c \;\middle|\; \begin{array}{c} (l \to r \mid c) \in \mathcal{R} \quad \text{and} \\ l \text{ is not of the form } d(t_1, \ldots, t_n) \end{array} \right\} \cup \left\{ d \to v{\downarrow_F} \right\} \right\rangle,\, \mathsf{a} \right\rangle \qquad (\mathsf{sR}_{:=})$$

$$\left\langle \langle \Sigma,\, \mathcal{R} \rangle,\, \mathsf{new}(c,\, s)\,;\mathsf{a} \right\rangle \twoheadrightarrow \left\langle \langle \Sigma \uplus \{c : s\},\, \mathcal{R} \rangle,\, \mathsf{a} \right\rangle \qquad (\mathsf{sR}_{\mathsf{new}})$$

$$\langle F,\, \mathsf{skip}\,;\mathsf{a} \rangle \twoheadrightarrow \langle F,\, \mathsf{a} \rangle \qquad (\mathsf{sR}_{\mathsf{skip}})$$

Table 6.3: Axiom Schemes describing the Execution of Actions

$$\frac{p \equiv p' \qquad \langle F,\, p' \rangle \to \langle F',\, p'' \rangle \qquad p'' \equiv p'''}{\langle F,\, p \rangle \to \langle F',\, p''' \rangle} \qquad (\mathsf{sR}_{\equiv})$$

$$\frac{\left\langle F,\, \mathsf{a}_1; \ldots ; \mathsf{a}_n\,;\mathsf{skip} \right\rangle \overset{*}{\twoheadrightarrow} \left\langle F',\, \mathsf{skip} \right\rangle \qquad F \vdash g}{\left\langle F,\, [g \Rightarrow \mathsf{a}_1; \ldots ; \mathsf{a}_n] \right\rangle \to \langle F',\, \mathsf{success} \rangle} \qquad (\mathsf{sR}_{action})$$

$$\frac{\left(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{i=1}^{m} (a_i\,;p_i)\right) \in \mathcal{R}^{\mathrm{p}} \qquad \left\langle F,\, \left(\bigoplus_{i=1}^{m} \mathit{rename}(a_i\,;p_i)\right)[v_j/x_j] \right\rangle \to \langle F',\, p' \rangle}{\left\langle F,\, \mathrm{q}(v_1, \ldots, v_n) \right\rangle \to \langle F',\, p' \rangle} \qquad (\mathsf{sR}_{call})$$

$$\frac{\langle F,\, p_1 \rangle \to \langle F',\, p_1' \rangle}{\langle F,\, p_1\,;p_2 \rangle \to \langle F',\, p_1'\,;p_2 \rangle} \qquad (\mathsf{sR}_{;})$$

$$\frac{\langle F,\, p_1 \rangle \to \langle F',\, p_1' \rangle}{\langle F,\, p_1 \parallel p_2 \rangle \to \langle F',\, p_1' \parallel p_2 \rangle} \qquad (\mathsf{sR}_{\parallel})$$

$$\frac{\langle F,\, p_1 \rangle \to \langle F',\, p_1' \rangle}{\langle F,\, p_1 + p_2 \rangle \to \langle F',\, p_1' \rangle} \qquad (\mathsf{sR}_{+})$$

$$\frac{\langle F,\, p_1 \rangle \to \langle F',\, p_1' \rangle}{\langle F,\, p_1 \oplus p_2 \rangle \to \langle F',\, p_1' \rangle} \qquad (\mathsf{sR}_{\oplus})$$

$$\frac{\langle F,\, p_2 \rangle \to \langle F',\, p_2' \rangle}{\langle F,\, p_1 \oplus p_2 \rangle \to \langle F',\, p_2' \rangle} \quad \text{if } \nexists\, p_1',\, \nexists\, F'', \text{ such that } \langle F,\, p_1 \rangle \to \langle F'',\, p_1' \rangle \qquad (\mathsf{sR}'_{\oplus})$$

Table 6.4: Simplified Transition System sT for a single Component

The privacy level of a symbol in a signature is determined by a privacy map. This map is extend to terms by taking the least upper bound of the privacy levels of all the symbols occurring in the term.

6.4 Definition (privacy map). *Let $\Sigma = \langle S, \Omega \rangle$ be a signature. We call a* privacy map *any map ℓ from symbols of Ω towards a privacy lattice $\mathfrak{L}$. We extend naturally any privacy map ℓ to all terms in $T(\Sigma, X)$, in the following way:*

$$\ell(f(t_1, \ldots, t_n)) = \big(\bigsqcup_{i=1}^{n} \ell(t_i)\big) \sqcup \ell(f) \tag{6.1a}$$

$$\ell(x) = \bot \qquad \textit{where } x \textit{ is a variable} \tag{6.1b}$$

In the following we suppose that we are given a privacy map ℓ. When we talk of the privacy of a term t we intend $\ell(t)$.

We now define the notion of safe rewrite rules, *i.e.*, rules which respect secrecy. Informally, a rewrite rule is safe whenever no information may flow from a higher level towards a lower one. Suppose that *PIN* is a constant denoting an information of the highest level: $\ell(PIN) = \top$, whereas *SPY* is a constant of the lowest level, $\ell(SPY) = \bot$. Then rewrite rules as for instance $SPY \to PIN$, or $SPY \to v_1 \mid (PIN = v_2)$ are not safe, since they allow the use of private or secret information in the evaluation of a public term or expression. Indeed, when the right hand side or the conditions are of a higher privacy level than the left hand side of a rewrite rule, the result of the application of the rule depends on information of a higher privacy level.

6.5 Definition (safe rewrite rule). *A conditional rewrite rule* $\mathsf{lhs} \to \mathsf{rhs} \mid \mathsf{cond}$ *is called* safe *with respect to a privacy map ℓ whenever the following conditions hold:*

$$\ell(\mathsf{rhs}) \sqsubseteq \ell(\mathsf{lhs}) \quad \textit{and} \quad \ell(\mathsf{cond}) \sqsubseteq \ell(\mathsf{lhs}) \tag{6.2}$$

In other words, rewrite rules are safe when the result of a rewrite or reduction step is a term of a lower privacy level than the original term, or a condition on which the reduction depends. This is in line with our informal presentation of the preservation of secrecy, where we stated that information may only flow from low levels towards high levels. In the following, the privacy level of a safe rewrite rule $l \to r \mid c$ is written $\ell\big(l \to r \mid c\big)$ and is defined as $\ell(l)$. We extend the notion of safe rewrite rules to safe stores. A *store* $F = \langle \Sigma, \mathcal{R} \rangle$ is called *safe* (with respect to a privacy map ℓ), if all rewrite rules $R \in \mathcal{R}$ are safe (with respect to ℓ).

The extension of a privacy map ℓ to actions is defined by the following equations:

$$\ell\big([g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_n]\big) \stackrel{\mathsf{def}}{=} \prod_{i=1}^{n} \ell(\mathsf{a}_i) \tag{6.1c}$$

$$\ell\big(\mathsf{tell}(R)\big) \stackrel{\mathsf{def}}{=} \ell\big(\mathsf{tell}(R)\big) \stackrel{\mathsf{def}}{=} \ell(R) \tag{6.1d}$$

$$\ell(c := v) \stackrel{\mathsf{def}}{=} \ell(c) \tag{6.1e}$$

$$\ell(\mathsf{skip}) \stackrel{\mathsf{def}}{=} \ell\big(\mathsf{new}(c, s)\big) \stackrel{\mathsf{def}}{=} \top \tag{6.1f}$$

Intuitively, the privacy level of a guarded action denotes the minimal privacy level of the rules that are changed by the execution of the action.

In order to define what we intend by safe processes, we have first to define the notion of equivalence between stores up to a given privacy level with respect to a privacy map.

6.6 Definition ($\langle \ell, \pi \rangle$-equivalence). *Let F_0, F_1 be two stores, ℓ a privacy map (into the privacy lattice $\mathfrak{L}$) and π a privacy level, i.e., $\pi \in \mathfrak{L}$. We say that F_0 and F_1 are $\langle \ell, \pi \rangle$-*equivalent*, written as $F_0 \cong^{\ell}_{\pi} F_1$, if and only if for all rules $R_i \in \mathcal{R}_i$ with $\ell(R_i) \sqsubseteq \pi$ there exists a rule $R_{(1-i)} \in \mathcal{R}_{(1-i)}$, such that $R_i = R_{(1-i)}$ (up to a bijective renaming of free variables) for all $i \in \{0; 1\}$.*

Notice that according to definition 6.6, we have for an action a with $\pi \sqsubseteq \ell(a)$ that for all stores F_1 and F_2 with $F_1 \cong^{\ell}_{\pi} F_2$, $\langle F, p \rangle \xrightarrow{a} \langle F', p' \rangle$ implies that also $F'_1 \cong^{\ell}_{\pi} F'_2$.

The following lemma states that the evaluation of a term is independent from the rules of higher privacy in a safe store. Intuitively, the lemma holds, since the evaluation of a term t uses only rules of a lower privacy than t.

6.7 Lemma. *Let π be a privacy level, ℓ a privacy map and $F_i = \langle \Sigma, \mathcal{R}_i \rangle$ ($i \in \{1; 2\}$) two safe stores with the same signature Σ such that $F_1 \cong^{\ell}_{\pi} F_2$. Then for all terms $t \in T(\Sigma, \varnothing)$ with $\ell(t) \sqsubseteq \pi$ we have that a reduction step $t \rightarrow_{\mathsf{p}, R_1} t'$ (with $R_1 \in \mathcal{R}_1$ and a position p) implies the existence of a term t'' such that $t \rightarrow_{\mathsf{p}, R_2} t''$ (with $R_2 \in \mathcal{R}_2$), t' is equal to t'' up to renaming and both, $\ell(t') = \ell(t'') \sqsubseteq \pi$.*

Proof. The safety of the stores together with the definition of the privacy map ensure that the privacy levels of all rules[4] used in the reduction step $t \rightarrow t'$ are lower or equal to π. By definition of $\cong^{\ell}_{\pi}$ we have thus for each these rules ($\in \mathcal{R}_1$) the existence of a rule ($\in \mathcal{R}_2$) which is equal up to renaming. Thus the reduction steps are with respect to the same stores (modulo renaming of the variables in the rewrite rules). □

An immediate consequence of lemma 6.7 is that the normal forms of a term t (with $\ell(t) \sqsubseteq \pi$) are the same with respect to F_1 and F_2 if $F_1 \cong^{\ell}_{\pi} F_2$. In the sequel we note $t{\downarrow_F}$ the normal form of the term t with respect to the store or program F.

We now extend the notion of bisimulation of processes to take into account the privacy levels of the stores. Thus we define a *bisimulation* up to a privacy map ℓ and a privacy level π. Informally two processes p_1, p_2 are bisimilar with respect to ℓ and π ($\langle \ell, \pi \rangle$-bisimilar) when, executed on $\langle \ell, \pi \rangle$-equivalent stores, they remain $\langle \ell, \pi \rangle$-bisimilar. In other words, either one can execute p_1 and then for each execution of p_1 there is an execution of p_2 such that the resulting pairs of stores and process terms remain $\langle \ell, \pi \rangle$-bisimilar, or p_1 can't be executed. In this case, we ensure that every execution of p_2 only affect parts of the store with an higher privacy than π, *i.e.*, all stores in an execution of p_2 remain $\langle \ell, \pi \rangle$-equivalent to the "initial" store.

6.8 Definition ($\langle \ell, \pi \rangle$-bisimulation). *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}} \rangle$ be a component, ℓ a privacy map and π a privacy level. A relation $\mathcal{B}^{\ell}_{\pi}$ on pairs of stores and process terms is called a $\langle \ell, \pi \rangle$-*bisimulation *if $\mathcal{B}^{\ell}_{\pi}$ is symmetric and if $\langle F_1, p_1 \rangle \, \mathcal{B}^{\ell}_{\pi} \, \langle F_2, p_2 \rangle$ implies:*

1. *$F_1 \cong^{\ell}_{\pi} F_2$*
2. *$\langle F_1, p_1 \rangle \xrightarrow{a} \langle F'_1, p'_1 \rangle$ implies that*

[4] Notice that due to conditions in the rewrite rules, more than one rule are used in a reduction step.

- *either there exist* F_2' *and* p_2 *such that we have* $\langle F_2, p_2\rangle \xrightarrow{a} \langle F_2', p_2'\rangle$ *and* $\langle F_1', p_1'\rangle \mathcal{B}_\pi^\ell \langle F_2', p_2'\rangle$
- *or* $\langle F_2, p_2\rangle \overset{a}{\not\rightarrow}$ *and for all* $F_1^\sharp$ *and* $p_1^\sharp$ *such that* $\langle F_1, p_1\rangle \xrightarrow{a\cdot\mathcal{A}(\mathcal{C})^*} \langle F_1^\sharp, p_1^\sharp\rangle$ *we have that* $F_1^\sharp \cong_\pi^\ell F_2$.

For the rest of this chapter, we consider only the largest $\langle \ell, \pi\rangle$-bisimulation, which we note as $\approx_\pi^\ell$.

We are now ready to define the notion of a safe process term. Intuitively, a process term is safe, *i.e.*, respects secrecy map ℓ, if it is bisimilar to itself for every privacy level π on $\langle \ell, \pi\rangle$-equivalent stores.

6.9 Definition (safe process term). *Let* $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}}\rangle$ *be a component. A process term* $p \in \mathcal{P}(\mathcal{C})$ *is called* safe *with respect to a privacy map* ℓ*, if for all privacy levels* $\pi \in \mathfrak{L}$ *and for all stores* F_1*,* F_2 *such that* $F_1 \cong_\pi^\ell \langle \Sigma, \mathcal{R}\rangle \cong_\pi^\ell F_2$ *we have that* $\langle F_1, p\rangle \approx_\pi^\ell \langle F_2, p\rangle$.

Notice that up to a given privacy level π the stores occurring during the execution of a safe process term do not depend on the initial rules (*i.e.*, the rules of the initial store) the privacy of which is greater than π.

We conclude this section with the characterisation of the properties of an unsafe process term, *i.e.*, a process term which is not $\langle \ell, \pi\rangle$-bisimilar to itself. Intuitively, if a process term is not bisimilar to itself, than we have that the execution of the process term on equivalent stores leads to one of two situations: either the execution (of the same actions) leads to two non-equivalent stores or one of the executions blocks and the other execution continues to a non-equivalent store.

6.10 Lemma. *Let* $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}}\rangle$ *be a component. Let* F_0^1*,* F_0^2 *be two stores (for the signature* Σ*),* $p_0 \in \mathcal{P}(\mathcal{C})$ *a process term,* ℓ *a privacy map and* π *a privacy level. If* $\langle F_0^1, p_0\rangle \not\approx_\pi^\ell \langle F_0^2, p_0\rangle$ *then there exist* $N \geq 0$ *and two transition sequences*

$$\begin{aligned} &\langle F_0^1, p_0\rangle \xrightarrow{a_1} \langle F_1^1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F_N^1, p_N\rangle \\ &\langle F_0^2, p_0\rangle \xrightarrow{a_1} \langle F_1^2, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F_N^2, p_N\rangle \end{aligned} \tag{6.3}$$

such that $F_j^1 \cong_\pi^\ell F_j^2$ *for all* $j < N$ *and that we have one of the following two situations*

1. $F_N^1 \ncong_\pi^\ell F_N^2$*, or*
2. *there exist a store* $F^\sharp$ *and a process term* $p^\sharp$ *such that* $\langle F_N^j, p_N\rangle \xrightarrow{a\cdot\mathcal{A}(\mathcal{C})^*} \langle F^\sharp, p^\sharp\rangle$ *but* $\langle F_N^{(j+1 \bmod 2)+1}, p_N\rangle \overset{a}{\not\rightarrow}$ *and* $F^\sharp \ncong_\pi^\ell F_N^{(j+1 \bmod 2)+1}$ *(for* $j \in \{1;\ 2\}$*).*

Proof. Without loss of generality, consider, in the situation of lemma 6.10, a *maximal* transition sequence d_1 for F_0^1 and p_0 (see definition 5.2). We distinguish between the case of a finite and an infinite execution trace.

1. The maximal transition sequence d_1 is *finite*, *i.e.*, we have $n \geq 0$ such that

$$\langle F_0^1, p_0\rangle \xrightarrow{a_1} \langle F_1^1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F_n^1, p_n\rangle \overset{a}{\not\rightarrow} \tag{6.4}$$

We prove the lemma by induction on the length n of the transition sequence.

Base Case. In the case that $n = 0$, if we have $F_0^1 \not\approx_\pi^\ell F_0^2$ the lemma is obviously true (for $N = 0$). Otherwise, by definition of $\approx_\pi^\ell$, there exist $F^\sharp$, $p^\sharp$ such that $\langle F_0^2, p_0\rangle \rightarrow^+ \langle F^\sharp, p^\sharp\rangle$ with $F_0^1 \not\approx_\pi^\ell F^\sharp$. Thus the lemma holds for $N = 0$.

Induction Step. Suppose now, that lemma 6.10 holds for maximal transition sequences for $\langle F_0^1, p_0\rangle$ of length shorter than n. Consider a maximal transition sequence of length n. If $F_0^1 \not\approx_\pi^\ell F_0^2$, the lemma is obviously true (for $N = 0$). Otherwise we distinguish two further cases:

- $\langle F_0^2, p_0\rangle \stackrel{a_1}{\not\longrightarrow}$. Assume that $F_0^2 \cong_\pi^\ell F_i^1$ for all $i \in \{1; \ldots; n\}$. Thus we have by definition 6.8 that $\langle F_0^1, p_0\rangle \approx_\pi^\ell \langle F_0^2, p_0\rangle$ which is in contradiction to our assumption. Thus there exists i_0 such that $\langle F_0^1, p_0\rangle \xrightarrow{a_1 \cdot \mathcal{A}(\mathcal{C})^*} \langle F_{i_0}^1, p_{i_0}^1\rangle$ but $\langle F_0^2, p_0\rangle \stackrel{a_1}{\not\longrightarrow}$ and $F_{i_0}^1 \not\approx_\pi^\ell F_0^2$, *i.e.*, we are in the second situation of lemma 6.10 for $N = 0$.
- there exists a process term p_0' such that[5] $\langle F_0^2, p_0\rangle \xrightarrow{a_1} \langle F_1^2, p_0'\rangle$. Notice that in this case, we have by lemma 6.2 also the transition $\langle F_0^2, p_0\rangle \xrightarrow{a_1} \langle F_1^2, p_1\rangle$. Hence we have $\langle F_1^1, p_1\rangle \not\approx_\pi^\ell \langle F_1^2, p_1\rangle$ and can apply the hypothesis of the induction, since the considered maximal transition sequence for $\langle F_1^1, p_1\rangle$ has length $n - 1$.

2. The maximal transition sequence d_1 is *infinite*, *i.e.*, we have

$$\langle F_0^1, p_0\rangle \xrightarrow{a_1} \langle F_1^1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_i} \langle F_i^1, p_i\rangle \xrightarrow{a_{i+1}} \cdots \tag{6.5a}$$

Consider the maximal transition sequence d_2 for F_0^2 and p_0 which corresponds to an execution of (a prefix of) the action sequence $(a_i)_{i>0}$. If d_2 is finite, the proof is symmetric to case 1. Thus we suppose we have an infinite transition sequence[6]

$$\langle F_0^2, p_0\rangle \xrightarrow{a_1} \langle F_1^2, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_i} \langle F_i^2, p_i\rangle \xrightarrow{a_{i+1}} \cdots \tag{6.5b}$$

Notice that we cannot have $F_i^1 \cong_\pi^\ell F_i^2$ for all $i \geq 0$ since this implies that $\langle F_0^1, p_0\rangle \approx_\pi^\ell \langle F_0^2, p_0\rangle$, in contrary to our assumptions. Thus, we choose N as the least index such that $F_n^1 \not\approx_\pi^\ell F_n^2$.

□

6.2 Analysis: Abstraction and Constraint Generation

Our analysis is based on an *abstract execution* of a component. This abstract execution yields a constraint system, *i.e.*, conditions on the privacy levels assigned to the symbols of the signature. We prove in theorem 6.29 that if a privacy map respects the constraint system, an execution of the program respects secrecy. Hence our approach is a particular case of *abstract interpretation* [CC77]. In this section, we define the abstraction as well as its use for the analysis, *i.e.*, the generation of the constraint system.

à revoir !

[5] The transition yields the store F_1^2 since the execution of actions is deterministic.

[6] We have the same process terms in the transition sequences d_1 and d_2 by a similar reasoning as above.

6.2.1 Abstraction

We define the abstraction of a component $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p \rangle$ by means of a set of abstraction functions, which we note, by abuse of notation, all as $\mathcal{A}bs$. Intuitively, $\mathcal{A}bs$ maps a signature Σ into an abstract signature which contains only two sorts, namely `Priv` and `Truth`. Each of the symbols of Σ is mapped into a new constant symbol in the abstract signature. These new symbols are to be interpreted as privacy levels for the symbols of Σ. Besides these symbols, the abstract signature contains the standard operators of a lattice together with their specification. Rewrite rules are mapped into pairs of rewrite rules representing the conditions on the privacy levels of their constituents. Finally, a term is mapped to an expression over privacy levels denoting the maximal privacy level of any symbol occurring in the term.

6.11 Definition (abstract store). *The* abstract store $\mathcal{A}bs(F)$ *for a store* $F = \langle \Sigma, \mathcal{R} \rangle$ *is defined as a pair* $\big\langle \mathcal{A}bs(\Sigma),\ \mathcal{A}bs(\mathcal{R}) \big\rangle$ *of a signature* $\mathcal{A}bs(\Sigma)$ *and a set of rules* $\mathcal{A}bs(\mathcal{R})$ *by means of the* abstraction function $\mathcal{A}bs$ *which is defined by the following equations*

$$\mathcal{A}bs(\Sigma) \stackrel{\text{def}}{=} \left\langle \begin{array}{l} \{\texttt{Priv};\ \texttt{Truth}\} \\ \big\{\, \tilde{f} \,\big|\, f \in \Sigma \big\} \uplus \left\{ \begin{array}{l} \sqsubseteq,\ =\colon \texttt{Priv} \times \texttt{Priv} \to \texttt{Truth};\ \tilde{\text{TRUE}} : \texttt{Truth}; \\ \top,\ \bot\colon \texttt{Priv};\ \sqcup,\ \sqcap : \texttt{Priv} \to \texttt{Priv} \end{array} \right\} \end{array} \right\rangle \tag{6.6a}$$

$$\mathcal{A}bs(\mathcal{R}) \stackrel{\text{def}}{=} \mathcal{R}^{\mathfrak{L}} \cup \bigcup_{R \in \mathcal{R}} \mathcal{A}bs(R) \tag{6.6b}$$

$$\mathcal{A}bs(l \to r \mid c) \stackrel{\text{def}}{=} \big\{\, \mathcal{A}bs(r) \sqsubseteq \mathcal{A}bs(l) \to \tilde{\text{TRUE}};\quad \mathcal{A}bs(c) \sqsubseteq \mathcal{A}bs(l) \to \tilde{\text{TRUE}} \,\big\} \tag{6.7}$$

$$\mathcal{A}bs\big(f(t_1, \ldots, t_n)\big) \stackrel{\text{def}}{=} \tilde{f} \sqcup \Big(\bigsqcup_{i=1}^{n} \mathcal{A}bs(t_i) \Big) \tag{6.8a}$$

$$\mathcal{A}bs(x) \stackrel{\text{def}}{=} x \qquad \textit{where } x \textit{ is a variable} \tag{6.8b}$$

where $\mathcal{R}^{\mathfrak{L}}$ *is a set of rewrite rules axiomatising a lattice (of privacy levels).*

In the sequel, we write for short TRUE instead of $\tilde{\text{TRUE}}$. Furthermore, we abbreviate rewrite rules of the form $t_1 \sqsubseteq t_2 \to \text{TRUE}$ to $t_1 \sqsubseteq t_2$. As a shorthand and in analogy to the notation of the operators in the abstract signature, we write in the sequel $\tilde{F}$ (respectively, $\tilde{\Sigma}$, $\tilde{R}$ and $\tilde{t}$) for the abstraction of a store F (respectively, a signature Σ, a rule R and a term t).

The abstraction of an elementary action consists in the action, where the arguments have been abstracted. Abusing the notation, we note the abstraction of (elementary) actions also by the function $\mathcal{A}bs$.

6.12 Definition (abstract action). *The abstraction of a guarded action is the abstraction of the guard and the sequence of the abstractions of the elementary actions:*

$$\mathcal{A}bs\big([g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_n]\big) \stackrel{\text{def}}{=} \big[\mathcal{A}bs(g) \Rightarrow \mathcal{A}bs(\mathsf{a}_1); \ldots; \mathcal{A}bs(\mathsf{a}_n)\big] \tag{6.9}$$

The abstraction of an elementary action is defined by the following equations

$$\mathcal{A}bs\big(\mathsf{tell}(R)\big) \stackrel{\text{def}}{=} \mathsf{tell}\big(\mathcal{A}bs(R)\big) \tag{6.10a}$$
$$\mathcal{A}bs\big(\mathsf{del}(R)\big) \stackrel{\text{def}}{=} \mathsf{del}\big(\mathcal{A}bs(R)\big) \tag{6.10b}$$
$$\mathcal{A}bs\big(c := t\big) \stackrel{\text{def}}{=} \tilde{c} := \mathcal{A}bs(t) \tag{6.10c}$$
$$\mathcal{A}bs\big(\mathsf{new}(c,\, s)\big) \stackrel{\text{def}}{=} \mathsf{new}(\tilde{c},\, s) \tag{6.10d}$$
$$\mathcal{A}bs(\mathsf{skip}) \stackrel{\text{def}}{=} \mathsf{skip} \tag{6.10e}$$

Similar to the abstraction of an action, the abstraction of a process call is defined by the call with abstracted arguments. The only modification concerning the abstraction of process terms is that $\mathcal{A}bs$ maps the operator $\oplus$ to $+$. This is motivated by the fact that in the abstract operational semantics we consider all guards to be valid[7], so that we do not need to distinguish between $\oplus$ and $+$. We continue our abuse of notation concerning the abstraction function $\mathcal{A}bs$.

6.13 Definition (abstraction of process terms). *Besides the abstraction of a guarded action which is defined in definition 6.12, the abstraction of a process term is defined by the following equations:*

$$\mathcal{A}bs\big(\mathrm{q}(t_1,\, \ldots\, ,\, t_n)\big) \stackrel{\text{def}}{=} \mathrm{q}\big(\mathcal{A}bs(t_1)\,,\, \ldots\, ,\, \mathcal{A}bs(t_n)\big) \tag{6.11a}$$
$$\mathcal{A}bs(p_1\,;\, p_2) \stackrel{\text{def}}{=} \mathcal{A}bs(p_1)\,;\, \mathcal{A}bs(p_2) \tag{6.11b}$$
$$\mathcal{A}bs(p_1 \parallel p_2) \stackrel{\text{def}}{=} \mathcal{A}bs(p_1) \parallel \mathcal{A}bs(p_2) \tag{6.11c}$$
$$\mathcal{A}bs(p_1 + p_2) \stackrel{\text{def}}{=} \mathcal{A}bs(p_1) + \mathcal{A}bs(p_2) \tag{6.11d}$$
$$\mathcal{A}bs(p_1 \oplus p_2) \stackrel{\text{def}}{=} \mathcal{A}bs(p_1) + \mathcal{A}bs(p_2) \tag{6.11e}$$

The operational semantics of an abstracted component is used in order to generate a constraint system, which describes the conditions under which the execution of the initial process term respects secrecy (with respect to a privacy map). Therefore, we define the operational semantics of an abstracted component by a new transition system, namely ST, the rules of which are presented on page 174.

The goal of the abstract operational semantics is to accumulate the constraints which guarantee that a concrete execution respects secrecy. According to lemma 6.10, there are two possibilities to violate secrecy, namely by executing an inherently bad action or by modifying a public information after checking a secret guard. The first case is handled by the constraints imposed on the actions executed by a process term. In order to capture the second case, we follow the idea of [BC01b] and associate a process term with a privacy level σ, representing the maximal privacy level of all the guards checked so far. All actions executed by process are then required to modify only data of a higher privacy than σ. Notice that defining the states of the abstract transition system as triples of an abstract store, abstract process term and σ would be too restrictive, as the following example shows.

[7] Notice that a guard is abstracted to a privacy level. Thus, checking the validity of a guard amounts to check if the abstracted guard is a privacy level, which holds by definition.

$$\langle p_1 \parallel p_2,\, \sigma\rangle \rightrightarrows \langle p_1,\, \sigma\rangle \;\|^{\mathsf{s}}\; \langle p_2,\, \sigma\rangle \qquad (\|^{\mathsf{s}}\text{-}\mathit{Intro})$$

$$\langle p_1 + p_2,\, \sigma\rangle \rightrightarrows \langle p_1,\, \sigma\rangle \;+^{\mathsf{s}}\; \langle p_2,\, \sigma\rangle \qquad (+^{\mathsf{s}}\text{-}\mathit{Intro})$$

$$\langle p_1\,;p_2,\, \sigma\rangle \rightrightarrows \langle p_1,\, \sigma\rangle \;;^{\mathsf{s}}\; \langle p_2,\, \bot\rangle \qquad (;^{\mathsf{s}}\text{-}\mathit{Intro})$$

$$\langle \mathsf{success},\, \sigma_1\rangle \;\|^{\mathsf{s}}\; \langle \mathsf{success},\, \sigma_1\rangle \rightrightarrows \langle \mathsf{success},\, \sigma_1 \sqcup \sigma_2\rangle \qquad (\|^{\mathsf{s}}\text{-}\mathit{Elim})$$

$$\langle \mathsf{success},\, \sigma_1\rangle \;;^{\mathsf{s}}\; \langle p_2,\, \sigma_2\rangle \rightrightarrows \langle p_2,\, \sigma_1\rangle \qquad (;^{\mathsf{s}}\text{-}\mathit{Elim})$$

Table 6.5: Axiom Schemes for $\rightrightarrows$

$$\begin{array}{c}\langle p_1,\, \sigma_1\rangle \;\|^{\mathsf{s}}\; \langle p_2,\, \sigma_2\rangle \equiv^{\mathsf{s}} \langle p_2,\, \sigma_2\rangle \;\|^{\mathsf{s}}\; \langle p_1,\, \sigma_1\rangle \\ \langle p_1,\, \sigma_1\rangle \;+^{\mathsf{s}}\; \langle p_2,\, \sigma_2\rangle \equiv^{\mathsf{s}} \langle p_2,\, \sigma_2\rangle \;+^{\mathsf{s}}\; \langle p_1,\, \sigma_1\rangle\end{array} \qquad (\mathit{Comm}_{\equiv^{\mathsf{s}}})$$

Table 6.6: Axiom Schemes Defining the Structural Congruence $\equiv^{\mathsf{s}}$ on Abstract Process Terms, *i.e.*, Pairs of Process Terms and Secrecy Levels

$$\left\langle \tilde{F},\, \mathsf{tell}(\tilde{l} \to \tilde{r} \mid \tilde{c})\,;\, \mathsf{a},\, \sigma\right\rangle \rightharpoonup \left\langle \tilde{F} \cup \{(\tilde{r} \sqsubseteq \tilde{l});\, (\tilde{c} \sqsubseteq \tilde{l});\, (\sigma \sqsubseteq \tilde{l})\},\, \mathsf{a},\, \sigma\right\rangle \qquad (\mathsf{SR}_{\mathsf{tell}})$$

$$\left\langle \tilde{F},\, \mathsf{del}(\tilde{l} \to \tilde{r} \mid \tilde{c})\,;\, \mathsf{a},\, \sigma\right\rangle \rightharpoonup \left\langle \tilde{F} \cup \{(\sigma \sqsubseteq \tilde{l})\},\, \mathsf{a},\, \sigma\right\rangle \qquad (\mathsf{SR}_{\mathsf{del}})$$

$$\left\langle \tilde{F},\, (\tilde{c} := \tilde{v})\,;\, \mathsf{a},\, \sigma\right\rangle \rightharpoonup \left\langle \tilde{F} \cup \{(\tilde{v} \sqsubseteq \tilde{c});\, (\sigma \sqsubseteq \tilde{c})\},\, \mathsf{a},\, \sigma\right\rangle \qquad (\mathsf{SR}_{:=})$$

$$\left\langle \tilde{F},\, \mathsf{new}(c,\, s)\,;\, \mathsf{a},\, \sigma\right\rangle \rightharpoonup \left\langle \tilde{F},\, \mathsf{a},\, \sigma\right\rangle \qquad (\mathsf{SR}_{\mathsf{new}})$$

$$\langle \tilde{F},\, \mathsf{skip}\,;\, \mathsf{a},\, \sigma\rangle \rightharpoonup \langle \tilde{F},\, \mathsf{a},\, \sigma\rangle \qquad (\mathsf{SR}_{\mathsf{skip}})$$

Table 6.7: Axiom Schemes describing the Abstract Execution of Actions

$$\frac{ap{\downarrow\downarrow} \equiv^{\mathsf{s}} ap' \qquad \langle \tilde{F},\, ap'\rangle \rightharpoonup \langle \tilde{F}',\, ap''\rangle \qquad ap''{\downarrow\downarrow} \equiv^{\mathsf{s}} ap'''}{\langle \tilde{F},\, ap\rangle \rightharpoonup \langle \tilde{F}',\, ap'''\rangle} \qquad (\mathsf{SR}_{\equiv^{\mathsf{s}}})$$

$$\frac{\left\langle \tilde{F},\, \mathsf{a}_1;\ldots;\mathsf{a}_n\,;\mathsf{skip},\, \sigma \sqcup \tilde{g}\right\rangle \overset{*}{\rightharpoonup} \left\langle \tilde{F}',\, \mathsf{skip},\, \sigma \sqcup \tilde{g}\right\rangle}{\left\langle \tilde{F},\, \langle[g \Rightarrow \mathsf{a}_1;\ldots;\mathsf{a}_n],\, \sigma\rangle\right\rangle \rightharpoonup \left\langle \tilde{F}',\, \langle \mathsf{success},\, \sigma \sqcup \tilde{g}\rangle\right\rangle} \qquad (\mathsf{SR}_{action})$$

$$\frac{\left(\mathrm{q}(x_1,\ldots,x_n) \Leftarrow \bigoplus_{i=1}^{m}(a_i\,;p_i)\right) \in \mathcal{R}^{\mathrm{p}} \quad \left\langle \tilde{F},\, \langle \mathcal{A}bs\big(\bigoplus_{i=1}^{m} a_i\,;p_i\big)[\tilde{v}_j/x_j],\, \sigma\rangle\right\rangle \rightharpoonup \left\langle \tilde{F}',\, \langle p',\, \sigma'\rangle\right\rangle}{\left\langle \tilde{F},\, \langle \mathrm{q}(\tilde{v}_1,\ldots,\tilde{v}_n),\, \sigma\rangle\right\rangle \rightharpoonup \left\langle \tilde{F}',\, \langle p',\, \sigma'\rangle\right\rangle} \qquad (\mathsf{SR}_{call})$$

$$\frac{\langle \tilde{F},\, ap_1\rangle \rightharpoonup \langle \tilde{F}',\, ap_1'\rangle}{\langle \tilde{F},\, ap_1 ;^{\mathsf{s}} ap_2\rangle \rightharpoonup \langle \tilde{F}',\, ap_1' ;^{\mathsf{s}} ap_2\rangle} \qquad (\mathsf{SR}_{;^{\mathsf{s}}})$$

$$\frac{\langle \tilde{F},\, ap_1\rangle \rightharpoonup \langle \tilde{F}',\, ap_1'\rangle}{\langle \tilde{F},\, ap_1 \|^{\mathsf{s}} ap_2\rangle \rightharpoonup \langle \tilde{F}',\, ap_1' \|^{\mathsf{s}} ap_2\rangle} \qquad (\mathsf{SR}_{\|^{\mathsf{s}}})$$

$$\frac{\langle \tilde{F},\, ap_1\rangle \rightharpoonup \langle \tilde{F}',\, ap_1'\rangle}{\langle \tilde{F},\, ap_1 +^{\mathsf{s}} ap_2\rangle \rightharpoonup \langle \tilde{F}',\, ap_1'\rangle} \qquad (\mathsf{SR}_{+^{\mathsf{s}}})$$

Table 6.8: Transition System ST for the Secrecy Analysis

6.14 Example. *Consider a process term $p_1 \parallel p_2$ such that p_i executes the action a_i ($i \in \{1; 2\}$). Suppose further that a_1 tests a secret guard and modifies only secret data, whereas a_2 tests a public guard and handles public data. Then the execution of a_1* before *a_2 is correct and should be accepted by the analysis. However, a single σ for the complete $p_1 \parallel p_2$ would not allow this, since both p_1 and p_2 would be required to use the same privacy level σ.*

Therefore we introduce a new kind of process terms for the use in the analysis, where all parallel processes are paired with a privacy level representing the least upper bound of the guards tested by this process.

6.15 Definition (abstract process terms). *The following grammar defines the set of* abstract process term*s:*

$$ap ::= \langle p, \sigma \rangle \mid ap \parallel^{\mathsf{s}} ap \mid ap +^{\mathsf{s}} ap \mid ap ;^{\mathsf{s}} ap \tag{6.12}$$

Notice that we distinguish between *abstractions of process terms* (see definition 6.13) and *abstract process terms*, *i.e.*, abstractions of process terms paired with privacy levels (see definition 6.15). The transformation of a pair of an abstraction of a process term and a privacy level σ is described by the relation $\rightrightarrows$, which is completely defined by the axiom schemes shown in table 6.5. The introduction of the new operators $\parallel^{\mathsf{s}}$ and $+^{\mathsf{s}}$ dispatches the privacy level σ over the subterms (see rules ($\parallel^{\mathsf{s}}$-*Intro*) and ($+^{\mathsf{s}}$-*Intro*)). Since the operator $;^{\mathsf{s}}$ is not commutative, the second process term cannot execute, so that the privacy level associated to it does not matter. Therefore, we set the privacy level associated to the second process term to $\bot$ in rule ($;^{\mathsf{s}}$-*Intro*).

The operator $\parallel^{\mathsf{s}}$ is removed when both processes have terminated successfully (see rule ($\parallel^{\mathsf{s}}$-*Elim*)), taking as new privacy level the least upper bound of the two privacy levels before. Notice that we do not need an elimination rule for the operator $+^{\mathsf{s}}$ thanks to the definition of the transition relation $\rightharpoonup$ (see rule ($\mathsf{SR}_{+^{\mathsf{s}}}$)). Notice that we do not need to consider the privacy level associated to the process term after the $;^{\mathsf{s}}$, since this process has not yet tested any guard, since it has not yet executed any action. This also explains why we set in rule ($;^{\mathsf{s}}$-*Intro*) the privacy level associated to p_2 to $\bot$. Notice that since the relation $\rightrightarrows$ is completely defined by the axiom schemes of table 6.5 and rule ($\mathsf{SR}_{;^{\mathsf{s}}}$) does not allow the reduction of the second argument of $;^{\mathsf{s}}$, we have that the second argument of $;^{\mathsf{s}}$ will always be of the form $\langle p, \sigma \rangle$. We note the normal form of a pair of a process term p and a privacy level σ as $\langle p, \sigma \rangle \downarrow\downarrow$. Notice that the rules of table 6.5 are only applied to the top of an abstract process term[8].

The congruence $\equiv^{\mathsf{s}}$ is similar to $\equiv$, besides the fact that, due to the use of $\rightrightarrows$, we do no longer need the axiom schemes defining $\mathsf{success}$ as unit element for $\parallel$ and ;, since these properties are already expressed in the rules defining the relation $\rightrightarrows$.

The rules of table 6.7 describe the execution of abstract actions by the transition relation $\twoheadrightarrow$. The constraints or rewrite rules[9] added to the abstract store $\tilde{F}$ ensure that the action does not modify a part of the store the privacy level of which is strictly lower than σ (see the last condition in the rule ($\mathsf{SR}_{\mathsf{tell}}$), ($\mathsf{SR}_{\mathsf{del}}$) and ($\mathsf{SR}_{:=}$)) and that the rules added to the store are safe[10]. The addition of a new symbol to the signature

[8] That is to say, we do *not* consider a congruence relation generated by the axioms of table 6.5.

[9] We abbreviate rewrite rules of the form $x \sqsubseteq y \rightarrow \text{TRUE}$ to $x \sqsubseteq y$.

[10] A formal proof of these two properties can be found in the lemmas 6.27 and 6.28.

of a store does not modify the rules of the store. Therefore, rule ($\mathsf{SR}_{\mathsf{new}}$) does not add any constraints to the abstract store.

The execution of an abstract process term ap is described by the transition relation $\rightharpoonup$ defined by the inference rules in table 6.8. Notice that in rule (SR_{call}), we do not rename the new symbols introduced by the new action in the rules of a process definition. Indeed, we consider that all symbols that are created by the same new action must have the same privacy level. Furthermore, the first premise of rule (SR_{call}) is identical to the first premise of rule (sR_{call}) (an uses the operator $\oplus$) since we do not abstract the process definitions.

In order to state the correspondence between concrete and abstract executions formally, we introduce the notion of a *weak equivalence* between abstract process terms for the analysis. We call this equivalence weak, since it does not consider the privacy levels associated with abstract process terms. Furthermore, we wish to consider equivalent concrete process terms as weak equivalent. Thus, we introduce a congruence $\equiv^{\mathsf{a}}$ on abstractions of process terms which is similar to $\equiv$.

6.16 Definition (weak equivalence). *Two abstract process terms ap_1 and ap_2 are called* weakly equivalent*, written as $ap_1 \mathrel{\widetilde{\equiv}^{\mathsf{s}}} ap_2$, if the corresponding process terms are equivalent, i.e.,*

$$ap_1 \mathrel{\widetilde{\equiv}^{\mathsf{s}}} ap_2 \qquad \Leftrightarrow \qquad \mathcal{U}(ap_1) \equiv^{\mathsf{a}} \mathcal{U}(ap_2) \tag{6.13}$$

where the function $\mathcal{U}$ is defined by the following four equations

$$\mathcal{U}\big(\langle p, \sigma\rangle\big) \stackrel{\text{def}}{=} p \tag{6.14a}$$

$$\mathcal{U}\big(ap_1 \parallel^{\mathsf{s}} ap_2\big) \stackrel{\text{def}}{=} \mathcal{U}(ap_1) \parallel \mathcal{U}(ap_2) \tag{6.14b}$$

$$\mathcal{U}\big(ap_1 +^{\mathsf{s}} ap_2\big) \stackrel{\text{def}}{=} \mathcal{U}(ap_1) + \mathcal{U}(ap_2) \tag{6.14c}$$

$$\mathcal{U}\big(ap_1 ;^{\mathsf{s}} ap_2\big) \stackrel{\text{def}}{=} \mathcal{U}(ap_1) ; \mathcal{U}(ap_2) \tag{6.14d}$$

and $\equiv^{\mathsf{a}}$ is the congruence on abstractions of process terms generated by the following axiom schemes

$$\mathsf{success} ; ap \equiv^{\mathsf{a}} ap \qquad \mathsf{success} \parallel ap \equiv^{\mathsf{a}} ap \qquad (Unit_{\equiv^{\mathsf{a}}})$$

$$ap_1 + ap_2 \equiv^{\mathsf{a}} ap_2 + ap_1 \qquad ap_1 \parallel ap_2 \equiv^{\mathsf{a}} ap_2 \parallel ap_1 \qquad (Comm_{\equiv^{\mathsf{a}}})$$

Since the axiom schemes of $\equiv^{\mathsf{a}}$ and $\equiv$ are the same, we have obviously that $p_1 \equiv p_2$ implies that $\mathcal{A}bs(p_1) \equiv^{\mathsf{a}} \mathcal{A}bs(p_2)$[11]. We have further that the application of $\rightrightarrows$ does not modify weak equivalence, since

$$\mathcal{U}(ap{\downarrow\downarrow}) \equiv^{\mathsf{a}} \mathcal{U}(ap) \qquad (\forall ap) \tag{6.15}$$

Using the notion of weak equivalence, we can express the correspondence between concrete and abstract transitions. Intuitively, we have that for every concrete transition from p to p', there exists also a corresponding abstract transition, that is to say an transition from the abstraction of p to a process term which is weakly equivalent to the abstraction of p'.

[11] The converse is false, as the example of the process terms $p_1 + p_2$ and $p_1 \oplus p_2$ shows.

6.17 Lemma. *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p^{\mathtt{i}} \rangle$ be a component, F a store and $p \in \mathcal{P}(\mathcal{C})$ a process term. Then we have for all concrete transitions*

$$\langle F, p\rangle \xrightarrow{a} \langle F', p'\rangle \tag{6.16a}$$

that there exists, for all privacy levels σ, a process term $\hat{p}$, an abstract store $\tilde{F}$ and an abstract process term $\tilde{p}$ such that $\hat{p} \equiv p$, $\langle \mathcal{A}bs(p'), \sigma\rangle \mathrel{\tilde{\equiv}^{\mathsf{s}}} \tilde{p}$ and we have the abstract transition

$$\big\langle \mathcal{A}bs(F),\ \langle \mathcal{A}bs(\hat{p}), \sigma\rangle\big\downarrow\big\downarrow \big\rangle \overset{\mathcal{A}bs(a)}{\rightharpoonup} \langle \tilde{F}, \tilde{p}\rangle \tag{6.16b}$$

Proof. We prove lemma 6.17 by induction on the height of the inference tree for the concrete transition. That is to say, we prove for all inference rules for $\rightarrow$ (see table 6.4) that, if the lemma holds for the premises, than it also holds for the conclusion of the inference rule.

Base Case. The only inference rule in table 6.4 without any occurrence of $\rightarrow$ in the premise is rule (sR_{action}). By inspection of the axiom schemes for $\twoheadrightarrow$ (see table 6.3) and $\rightharpoonup$ (see table 6.7), we have that $\big\langle F,\ \mathsf{a}_1; \ldots ; \mathsf{a}_n\,;\mathsf{skip}\big\rangle \overset{*}{\twoheadrightarrow} \big\langle F',\ \mathsf{skip}\big\rangle$ implies that for all σ there exists $\tilde{F}$ such that $\big\langle \mathcal{A}bs(F),\ \mathcal{A}bs(\mathsf{a}_1); \ldots ; \mathcal{A}bs(\mathsf{a}_n)\,;\mathsf{skip},\ \sigma\big\rangle \overset{*}{\rightharpoonup} \big\langle \tilde{F},\ \mathsf{skip},\ \sigma\big\rangle$. Thus, defining $\hat{p} \stackrel{\mathsf{def}}{=} p$ and $\tilde{p} \stackrel{\mathsf{def}}{=} \langle \mathsf{success}, \sigma\rangle$, we have by rule ($\mathsf{SR}_{action}$) that lemma 6.17 holds.

Induction Step. We consider the remaining inference rules one by one, under the hypothesis that lemma 6.17 holds for the transitions occurring in the premise.

($\mathsf{sR}_{\equiv}$)**:** Suppose that the premises of rule ($\mathsf{sR}_{\equiv}$) hold, *i.e.*, we have that $p \equiv p'$, $p'' \equiv p'''$ and $\langle F, p'\rangle \rightarrow \langle F', p''\rangle$. Using the hypothesis of the induction, we have thus that for all σ there exist $\hat{p}'$, $\tilde{F}$ and $\tilde{p}''$ such that $\hat{p}' \equiv p'$, $\langle \mathcal{A}bs(p''), \sigma\rangle \mathrel{\tilde{\equiv}^{\mathsf{s}}} \tilde{p}''$ and $\big\langle \mathcal{A}bs(F),\ \langle \mathcal{A}bs(\hat{p}'), \sigma\rangle\big\downarrow\big\downarrow \big\rangle \overset{\mathcal{A}bs(a)}{\rightharpoonup} \langle \tilde{F}, \tilde{p}''\rangle$. Since we have $p \equiv p' \equiv \hat{p}'$, we define $\hat{p} \stackrel{\mathsf{def}}{=} \hat{p}'$ and have obviously that $\langle \mathcal{A}bs(\hat{p}), \sigma\rangle\big\downarrow\big\downarrow \equiv^{\mathsf{s}} \langle \mathcal{A}bs(\hat{p}'), \sigma\rangle\big\downarrow\big\downarrow$, corresponding to the first premise of rule ($\mathsf{SR}_{\equiv^{\mathsf{s}}}$). We define $\tilde{p}''' \stackrel{\mathsf{def}}{=} \tilde{p}''\big\downarrow\big\downarrow$ and have an abstract transition by rule ($\mathsf{SR}_{\equiv^{\mathsf{s}}}$). Thus it remains to prove that $\langle \mathcal{A}bs(p'''), \sigma\rangle \mathrel{\tilde{\equiv}^{\mathsf{s}}} \tilde{p}'''$. This holds, since we have

$$\mathcal{U}\big(\langle \mathcal{A}bs(p'''), \sigma\rangle\big) = \mathcal{A}bs(p''') \equiv^{\mathsf{a}} \mathcal{A}bs(p'') \equiv^{\mathsf{a}} \mathcal{U}(\tilde{p}'') \equiv^{\mathsf{a}} \mathcal{U}(\,\tilde{p}''\big\downarrow\big\downarrow) = \mathcal{U}(\tilde{p}''') \tag{6.17}$$

($\mathsf{sR}_{\|}$)**:** The premise of ($\mathsf{sR}_{\|}$) is $\langle F, p_1\rangle \xrightarrow{a} \langle F', p_1'\rangle$. Hence, according to the hypothesis of the induction, for all privacy levels σ, there exist $\hat{p}_1$, $\tilde{F}$ and $\tilde{p}_1$ such that $\hat{p}_1 \equiv p$, $\big\langle \mathcal{A}bs(p_1'),\ \sigma\big\rangle \mathrel{\tilde{\equiv}^{\mathsf{s}}} \tilde{p}_1$ and $\big\langle \mathcal{A}bs(F),\ \langle \mathcal{A}bs(\hat{p}_1), \sigma\rangle\big\downarrow\big\downarrow \big\rangle \overset{\mathcal{A}bs(a)}{\rightharpoonup} \langle \tilde{F}, \tilde{p}_1\rangle$. We define $\hat{p} \stackrel{\mathsf{def}}{=} p$ and $\tilde{p} \stackrel{\mathsf{def}}{=} \tilde{p}_1 \|^{\mathsf{s}} \langle \mathcal{A}bs(p_2), \sigma\rangle$. Since $\langle \mathcal{A}bs(\hat{p}), \sigma\rangle\big\downarrow\big\downarrow = \langle \mathcal{A}bs(p_1), \sigma\rangle \|^{\mathsf{s}} \langle \mathcal{A}bs(p_2), \sigma\rangle$, we can infer that $\big\langle \mathcal{A}bs(F),\ \langle \mathcal{A}bs(\hat{p}), \sigma\rangle\big\downarrow\big\downarrow \big\rangle \overset{\mathcal{A}bs(a)}{\rightharpoonup} \langle \tilde{F}, \tilde{p}_1 \|^{\mathsf{s}} \langle \mathcal{A}bs(p_2), \sigma\rangle\rangle$ using rule ($\mathsf{SR}_{\|^{\mathsf{s}}}$).

Since $\mathcal{A}bs(p_1') \equiv^{\mathsf{a}} \mathcal{U}(\tilde{p}_1)$ we also have that $\mathcal{A}bs(p_1') \parallel \mathcal{A}bs(p_2) \equiv^{\mathsf{a}} \mathcal{U}(\tilde{p}_1) \parallel \mathcal{A}bs(p_2)$ (since $\equiv^{\mathsf{a}}$ is a congruence). Hence, we conclude that $\langle \mathcal{A}bs(p_1' \parallel p_2), \sigma\rangle \mathrel{\tilde{\equiv}^{\mathsf{s}}} \tilde{p}$.

(sR_{call}), ($\mathsf{sR}_{;}$), (sR_{+}), ($\mathsf{sR}_{\oplus}$) or ($\mathsf{sR}'_{\oplus}$): Since these cases are proven in a similar way as the case for rule ($\mathsf{sR}_{\|}$), we omit them here. □

Lemma 6.17 can be extended in a straightforward manner to finite sequences of transitions. We get the following corollary the proof of which we leave to the reader.

6.18 Corollary. *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p \rangle$ be a component, F_0 a store and $p_0 \in \mathcal{P}(\mathcal{C})$ a process term. Then we have for all concrete transition sequences*

$$\langle F_0, p_0 \rangle \xrightarrow{a_1} \langle F_1, p_1 \rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F_n, p_n \rangle \tag{6.18a}$$

that there exist abstract stores $\tilde{F}_i$ and abstract process term ap_i such that the following is a valid abstract transition sequence

$$\langle \tilde{F}_0, ap_0 \rangle \overset{\mathcal{A}bs(a_1)}{\rightsquigarrow} \langle \tilde{F}_1, ap_1 \rangle \overset{\mathcal{A}bs(a_2)}{\rightsquigarrow} \cdots \overset{\mathcal{A}bs(a_n)}{\rightsquigarrow} \langle \tilde{F}_n, ap_n \rangle \tag{6.18b}$$

where $\tilde{F}_0 \overset{\mathsf{def}}{=} \mathcal{A}bs(F_0)$ and $ap_i \mathrel{\tilde{\equiv}^{\mathsf{s}}} \langle p_i, \sigma_i \rangle \downarrow\downarrow$ for all $i \in \{0; \ldots; n\}$ and a (growing) sequence of privacy levels σ_i.

6.2.2 Secrecy Analysis

The overall idea of our program analysis is to perform all possible abstract executions of the initial process term of a component and to collect all constraints computed in the abstract stores by this execution. The point is that we can define an abstract execution which is *finite*, *i.e.*, which always terminates, since after a certain point no more privacy inequations are created by further execution of the abstract process term. This comes from the fact that abstract elementary actions don't modify values of the store but add only new constraints to the store. Since the number of symbols appearing in a component is finite, the number of inequalities that can be generated by this program is also finite. Therefore it is possible to consider the complete execution tree of a program (infinite branches being cut whenever no more new information can be collected). The union of all abstract stores of the leafs of this program abstract execution is the result of the analysis. The idea for implementing the decision if a branch of an abstract execution can be cut is based on checking for each process call, if the call has already been executed. Therefore, we define the analysis reduction $\ggg$ with respect to an abstract store, an abstract process term and a set of the process calls already executed, called *history*.

6.19 Definition (analysis reduction ($\ggg$)). *The* analysis reduction $\ggg$ *is a relation between triples of the form $\langle \tilde{F}, ap, \mathcal{H} \rangle$, where $\mathcal{H}$ is a set of process calls. $\ggg$ is defined by the same inference rules as $\rightharpoonup$, but by replacing rule (SR_{call}) by the following two inference rules.*

$$\frac{\begin{array}{l}\mathrm{q}(v_1, \ldots, v_n) \mathrel{\tilde{\notin}} \mathcal{H} \\ \big(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{i=1}^{m} (a_i \,; p_i)\big) \in \mathcal{R}^{\mathrm{p}} \\ \Big\langle \tilde{F}, \big\langle \big(\sum_{i=1}^{m} a_i \,; p_i\big)[v_j/x_j], \sigma \big\rangle, \big(\mathcal{H} \cup \{\mathrm{q}(v_1, \ldots, v_n)\}\big) \Big\rangle \ggg \langle \tilde{F}', \langle p', \sigma' \rangle, \mathcal{H}' \rangle\end{array}}{\langle \tilde{F}, \langle \mathrm{q}(v_1, \ldots, v_n), \sigma \rangle, \mathcal{H} \rangle \ggg \langle \tilde{F}', \langle p', \sigma' \rangle, \mathcal{H}' \rangle} \quad (\mathsf{aSR}_{call})$$

$$\frac{\mathrm{q}(v_1, \ldots, v_n) \mathrel{\tilde{\in}} \mathcal{H}}{\langle \tilde{F}, \langle \mathrm{q}(v_1, \ldots, v_n), \sigma \rangle, \mathcal{H} \rangle \ggg \langle \tilde{F}', \langle \mathsf{success}, \sigma \rangle, \mathcal{H} \rangle} \quad (\mathsf{aSR}^{cut}_{call})$$

where the relation $\widetilde{\in}$ of membership of a call in a history is defined as follows (we note $\simeq$ the equivalence of abstract terms using the standard axioms of a lattice, e.g., the associativity, commutativity and idempotency of $\sqcup$ and $\sqcap$ or the absorption laws for $\top$ and $\bot$):

$$\mathrm{q}(v_1, \ldots, v_n) \,\widetilde{\in}\, \mathcal{H} \quad \Leftrightarrow \quad \exists \mathrm{q}(v'_1, \ldots, v'_n) \in \mathcal{H} \text{ such that } \forall i \in \{1; \ldots; n\} : \; v_i \simeq v'_i \tag{6.19}$$

Thus $\ggg$ is the same as $\rightharpoonup$ but allows to stop the execution of process calls if a call with the same parameters has already been executed. Intuitively, the execution of $\ggg$ terminates always since the set of possible parameters for process calls is finite.

6.20 Lemma. *All executions according to $\ggg$ terminate.*

Proof. Notice that the abstract signature is finite, and that only a finite number of new symbols is added to the signature during the abstract executions, since according to rule (SR_{call}) we do not rename the symbols introduced by `new` elementary actions, such that these symbols are added only once. Using the standard axioms of a lattice for the operators $\sqcap$, $\sqcup$, $\top$ and $\bot$, we conclude that the set of terms of sort `Priv` is finite. Consequently the set of possible process calls is also finite. Since recursive process calls are the only way for writing non-terminating processes, the finiteness of the set of possible calls implies lemma 6.20 by rule ($\mathsf{aSR}^{cut}_{call}$). □

We now define the set of constraints computed by the analysis. Informally, it is the collection of all possible abstract stores that can be computed using $\ggg$ for the abstraction of the initial store, an abstract process term and the empty history.

6.21 Definition (constraint system). *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p \rangle$ be a component, $F = \langle \Sigma, \mathcal{R} \rangle$ the initial store and p a process term. The* constraint system $\Delta^p_F[\pi]$ *associated to F and p is defined as follows:*

$$\Delta^p_F[\pi] \stackrel{\mathsf{def}}{=} \bigcup_{\substack{\tilde{F} \text{ such that} \\ \langle \mathcal{A}bs(F), \langle p, \pi \rangle, \varnothing \rangle \stackrel{*}{\ggg} \langle \tilde{F}, \langle \mathsf{success}, \sigma \rangle, \mathcal{H} \rangle}} \tilde{F} \tag{6.20}$$

In the sequel, we write simply Δ^p_F instead of $\Delta^p_F[\bot]$.

According to definition 6.21, Δ^p_F is the set of rules that can be generated by an abstract execution of p. Notice that $\mathcal{A}bs(F)$, the abstraction of the initial store F, is a part of Δ^p_F.

In order to state the "monotonicity" for our analysis, we define the notion of a *weaker* constraint system. Intuitively, Δ_1 is weaker than Δ_2 if for all constraints $(x \sqsubseteq y)$ in Δ_1 there is the same or a stronger constraint in Δ_2, where a stronger constraint is obtained by increasing the level of x.

6.22 Definition. *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p \rangle$ be a component. Consider two stores $F_i = \langle \Sigma, \mathcal{R}_i \rangle$ and two process terms $p_i \in \mathcal{P}(\mathcal{C})$. We say that the constraint system Δ_1 is* weaker *than Δ_2, written as $\Delta_1 \Subset \Delta_2$, if for all rules $R_1 = (t_1 \sqsubseteq t' \rightarrow \mathrm{TRUE}) \in \Delta_1$*

there exists $R_2 = (t_2 \sqsubseteq t' \to \text{TRUE}) \in \Delta_2$ *such that* $\mathcal{S}ymb(t_1) \subseteq \mathcal{S}ymb(t_2)$*, where the set of symbols occurring in a term* t *is defined as follows:*

$$\begin{aligned} \mathcal{S}ymb(t_1 \sqcup t_2) &= \mathcal{S}ymb(t_1) \cup \mathcal{S}ymb(t_2) && (6.21a)\\ \mathcal{S}ymb(\tilde{f}) &= \{\tilde{f}\} \qquad \textit{where } \tilde{f} \textit{ is a constant symbol} && (6.21b)\\ \mathcal{S}ymb(x) &= \varnothing \qquad \textit{where } x \textit{ is a variable} && (6.21c) \end{aligned}$$

Notice that the condition in definition 6.22 includes the case that $R_1 \in \Delta_2$.

The following lemma proves that our analysis is *monotone* (with respect to the relation $\Subset$) in the sense that the constraints computed starting from the initial process term include the constraints that can be computed starting from all process terms that can be reached by executing the initial process term.

6.23 Lemma. *Let* $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p\rangle$ *be a component with initial store* $F = \langle \Sigma, \mathcal{R}\rangle$*. Then we have that* $\langle F, p\rangle \xrightarrow{*} \langle F^\sharp, p^\sharp\rangle$ *implies that there exists a privacy level* $\sigma^\sharp$ *such that* $\Delta^{p^\sharp}_{F^\sharp}[\sigma^\sharp] \Subset \Delta^p_F$*.*

Proof. Notice first, that all executions of $\langle F^\sharp, p^\sharp\rangle$ are a part (more precisely a suffix) of a execution of $\langle F, p\rangle$. Thus, for all privacy levels σ and σ', we have that all abstract actions executed by (an execution of) $\big\langle \mathcal{A}bs(F^\sharp),\ \langle \mathcal{A}bs(p^\sharp), \sigma'\rangle\big\rangle$ are also executed by the corresponding execution of $\big\langle \mathcal{A}bs(F),\ \langle \mathcal{A}bs(p), \sigma\rangle\big\rangle$. By corollary 6.18 we have further an abstract execution for $\mathcal{A}bs(p)$ leading to an abstract process term $\tilde{p}^\sharp$. Let $\sigma^\sharp$ be the greatest lower bound of all privacy levels occurring in $\tilde{p}^\sharp$. Thus we have for all constraints added by the abstract execution of $\langle p^\sharp, \sigma^\sharp\rangle$ that they (or a stronger constraint) are also added by an execution of $\langle p, \bot\rangle$. Consequently, we have $\big(\Delta^{p^\sharp}_{F^\sharp}[\sigma^\sharp] \setminus \mathcal{A}bs(F^\sharp)\big) \Subset \Delta^p_F$.

To prove $\mathcal{A}bs(F^\sharp) \Subset \Delta^p_F$, consider a rule $R \in \mathcal{R}^\sharp$ (we suppose that $F^\sharp = \langle \Sigma^\sharp, \mathcal{R}^\sharp\rangle$). If $R \in \mathcal{R}$, the lemma obviously holds. Otherwise, R has been added to the store by the execution of an elementary action. By corollary 6.18 and inspection of the axiom schemes of table 6.7 we have that Δ^p_F contains the rules corresponding to $\mathcal{A}bs(R)$. □

Notice that all rules in Δ^p_F are of the form $x \sqsubseteq y \to \text{TRUE}$ (by definition of $\twoheadrightarrow$ and $\mathcal{A}bs$). Therefore, we can define the formula $\widetilde{\Delta^p_F}$ as the conjunction of the left hand sides of the rules in Δ^p_F:

$$\widetilde{\Delta^p_F} \stackrel{\text{def}}{=} \bigwedge_{(x \sqsubseteq y \to \text{TRUE}) \in \Delta^p_F} (x \sqsubseteq y) \qquad (6.22)$$

In the sequel, we will confound $\widetilde{\Delta^p_F}$ and Δ^p_F.

6.24 Example. *The non-trivial*[12] *part of the constraint system computed for the example of* [BC01b] *mentioned at the beginning of this chapter (see table 6.1) is*

$$\begin{gathered} \mathit{false} \sqsubseteq \mathit{SPY} \wedge (c_\alpha \sqcup \mathit{true}) \sqsubseteq \mathit{SPY} \wedge \mathit{true} \sqsubseteq c_\beta \wedge (c_\alpha \sqcup \mathit{true}) \sqsubseteq c_\beta \wedge \\ \mathit{true} \sqsubseteq \mathit{SPY} \wedge (c_\beta \sqcup \mathit{true}) \sqsubseteq \mathit{SPY} \wedge \mathit{true} \sqsubseteq c_\alpha \wedge (c_\beta \sqcup \mathit{true}) \sqsubseteq c_\beta \wedge \\ \mathit{true} \sqsubseteq c_\alpha \wedge (\mathit{PIN} \sqcup \mathit{true}) \sqsubseteq c_\beta \wedge \mathit{true} \sqsubseteq c_\beta \wedge (\mathit{PIN} \sqcup \mathit{true}) \sqsubseteq c_\beta \end{gathered}$$

[12] We have omitted all constraints of the form $\bot \sqsubseteq x$ since they are valid by definition.

The link between the respect of secrecy (for a privacy map ℓ) and our analysis (*i.e.*, the constraint system Δ_F^p) is defined by the notion of compatibility between a privacy map and a constraint system. Informally, a privacy map ℓ is compatible with a constraint system Δ_F^p, if all the conjuncts of $\widetilde{\Delta_F^p}$ are valid for ℓ.

6.25 Definition (compatible). *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p\rangle$ be a component, $F = \langle \Sigma, \mathcal{R}\rangle$ the initial store and p a process term. A privacy map ℓ (for a privacy lattice $\mathfrak{L}$) is* compatible *with the constraint system Δ_F^p associated to F and p if $\langle \mathfrak{L}, \widetilde{\ell}\rangle$ is a model of Δ_F^p, where $\widetilde{\ell}$ maps $\sqsubseteq$, $\sqcup$, $\sqcap$, $\top$ and $\bot$ to the corresponding functions and elements of $\mathfrak{L}$, and abstract symbols to the privacy levels of the corresponding symbols, i.e., $\widetilde{\ell}(\tilde{f}) \stackrel{\mathsf{def}}{=} \ell(f)$.*

Notice that the compatibility of a privacy map ℓ with Δ_F^p implies that the initial store F is safe with respect to ℓ.

6.26 Example. *Consider the privacy map ℓ, defined by the following equations:*

$$\ell(PIN) \stackrel{\mathsf{def}}{=} \ell(c_\alpha) \stackrel{\mathsf{def}}{=} \ell(c_\beta) \stackrel{\mathsf{def}}{=} \top \tag{6.23a}$$

$$\ell(SPY) \stackrel{\mathsf{def}}{=} \ell(true) \stackrel{\mathsf{def}}{=} \ell(false) \stackrel{\mathsf{def}}{=} \bot \tag{6.23b}$$

We have obviously that ℓ is not compatible with the constraint system of example 6.24, since for instance $\top = \ell(c_\alpha) \not\sqsubseteq \ell(SPY) = \bot$.

6.3 Correctness of the Analysis

In this section, we show that our analysis is correct in the sense that if the analysis tells us that the initial process p of a component is safe, *i.e.*, respects secrecy, than all execution of p respect secrecy. Before we prove the correctness of our analysis in theorem 6.29, we introduce some auxiliary lemmas.

The following lemma states that the constraint system implies that the stores are always safe during the execution of a process (with respect to a privacy map compatible with the constraint system).

6.27 Lemma. *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p\rangle$ be a component and Δ_F^p the constraint system associated to $F = \langle \Sigma, \mathcal{R}\rangle$ and p. Let ℓ be a privacy map for Σ which is compatible with Δ_F^p. Then a transition sequence $\langle F, p\rangle \xrightarrow{*} \langle F^\sharp, p^\sharp\rangle$ implies that the store $F^\sharp$ is safe.*

Proof. Notice that by definition 6.21, the constraints Δ_F^p imply that F is safe. Consider a transition sequence $\langle F, p\rangle \xrightarrow{a_1} \langle F_1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F_n, p_n\rangle$ (with $n > 0$) such that for all $i < n$, the store F_i is safe, but F_n is not safe. Thus the execution of action a_n transforms the safe store F_{n-1} into the unsafe store F_n. Let $a_n = [g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_m]$. According to rule ($\mathsf{sR}_{action}$), the execution of a_n is described by the sequence of the executions of the elementary actions a_j (where $F_{n-1}^0 \stackrel{\mathsf{def}}{=} F_{n-1}$ and $F_{n-1}^m \stackrel{\mathsf{def}}{=} F_n$)

$$\langle F_{n-1}^0, \mathsf{a}_1; \ldots; \mathsf{a}_m; \mathsf{skip}\rangle \twoheadrightarrow \langle F_{n-1}^1, \mathsf{a}_2; \ldots; \mathsf{a}_m; \mathsf{skip}\rangle \twoheadrightarrow \cdots \twoheadrightarrow \langle F_{n-1}^m, \mathsf{skip}\rangle \tag{6.24}$$

Consider the greatest index j_0 such that $F_{n-1}^{j_0-1}$ is safe but F_{n-1}^j is not safe for all $j \in \{j_0; \ldots; m\}$. Thus the execution of a_{j_0} transforms a safe store in an unsafe store. We consider the different elementary actions one by one.

$\mathsf{tell}(l \to r \mid c)$**:** Since F_{n-1} is safe, the rule $(l \to r \mid c)$ is unsafe, since it is the only difference between F_{n-1} and F_n, according to rule $(\mathsf{sR}_{\mathsf{tell}})$. By corollary 6.18, we have that Δ_F^p implies the constraints added by rule $(\mathsf{SR}_{\mathsf{tell}})$, in particular $\tilde{r} \sqsubseteq \tilde{l}$ and $\tilde{c} \sqsubseteq \tilde{l}$. This is a contradiction to the hypothesis, since these constraints imply the safety of $(l \to r \mid c)$.

$\mathsf{del}(l \to r \mid c)$**:** According to rule $(\mathsf{sR}_{\mathsf{del}})$, the rules of the store F_n are a subset of the rules of the store F_{n-1}. Thus F_n is safe, in contradiction to the hypothesis.

$(c := v)$**:** Since the removal of rules by the assignment cannot transform a safe store in an unsafe store, we have only to consider the rule added according to rule $(\mathsf{sR}_{:=})$. By corollary 6.18, we have that Δ_F^p implies the constraints added by rule $(\mathsf{SR}_{:=})$, in particular $\tilde{v} \sqsubseteq \tilde{c}$. This is a contradiction to the hypothesis, since this constraint implies the safety of the rewrite rule $(c \to v{\downarrow})$.

$\mathsf{new}(c,\, s)$ or skip**:** This case is impossible, since according to rules $(\mathsf{sR}_{\mathsf{new}})$ and $(\mathsf{sR}_{\mathsf{skip}})$, the rules of the store are not modified.

□

Recall that the sequence of elementary actions of a guarded action is executed *atomically*. Thus, the action $[\text{TRUE} \Rightarrow SPY := PIN\,;\ SPY := 0]$ respects secrecy, since the temporary violation is invisible. This explains why we did consider the *greatest* index (or *last* elementary action) in the proof of lemma 6.27.

The following lemma states that the actions executed by a process are of a higher privacy level than the first guard checked by the process.

6.28 Lemma. *Let* $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p_0 \rangle$ *be a component and* $\Delta_{F_0}^{p_0}$ *the constraint system associated to* $F_0 \stackrel{\text{def}}{=} \langle \Sigma, \mathcal{R} \rangle$ *and* p_0*. Let* ℓ *be a privacy map for* Σ *which is compatible with* $\Delta_{F_0}^{p_0}$*. Then, for all transition sequences*

$$\langle F_0,\, p_0 \rangle \xrightarrow{a_1} \langle F_1,\, p_1 \rangle \xrightarrow{a_2} \cdots \xrightarrow{a_m} \langle F_m,\, p_m \rangle \xrightarrow{a_{m+1}} \cdots \tag{6.25}$$

where $a_1 = [g^1 \Rightarrow \mathsf{a}_{1_1}^1; \ldots; \mathsf{a}_{n_1}^1]$*, we have that* $\ell(g) \sqsubseteq \ell(a_i)$ *for all* $i > 0$*.*

Proof. Let i_0 be the least index such that $\ell(a_{i_0}) \sqsubset \ell(g)$. By corollary 6.18 we have an abstract execution corresponding to (6.25), *i.e.*,

$$\big\langle \mathcal{A}bs(F_0),\ \langle p_0,\, \sigma \rangle{\downarrow}{\downarrow} \big\rangle \xrightarrow{\mathcal{A}bs(a_1)} \big\langle \tilde{F}_1,\ ap_1 \big\rangle \xrightarrow{\mathcal{A}bs(a_2)} \cdots \xrightarrow{\mathcal{A}bs(a_{i_0})} \big\langle \tilde{F}_{i_0},\ ap_{i_0} \big\rangle \tag{6.26}$$

Notice first, that by inspection of the rules describing the execution of abstract process terms (see table 6.8), we have that $\ell(g) \sqsubseteq \sigma_i$ for all σ_i occurring in the process terms ap_i $(\forall i > 0)$. Let $a_{i_0} = [g^{i_0} \Rightarrow \mathsf{a}_1^{i_0}; \ldots; \mathsf{a}_{n_{i_0}}^{i_0}]$. Thus we have an abstract transition $\big\langle \tilde{F}_{i_0-1},\ \mathsf{a}_1^{i_0}; \ldots; \mathsf{a}_{n_{i_0}}^{i_0}; \mathsf{skip},\ \sigma_{i_0} \big\rangle \xrightarrow{*} \big\langle \tilde{F}_{i_0},\ \mathsf{skip},\ \sigma_{i_0} \big\rangle$.

According to equation (6.1c), we conclude from $\ell(a_{i_0}) \sqsubset \ell(g)$ that there exists $j \in \{1; \ldots; n_{i_0}\}$ such that $\ell(\mathsf{a}_j^{i_0}) \sqsubset \ell(g)$. We consider the different possibilities for the elementary action $\mathsf{a}_j^{i_0}$ one by one.

$\mathsf{tell}(l \to r \mid c)$**:** According to rule ($\mathsf{SR}_{\mathsf{tell}}$), $\Delta^{p_0}_{F_0}$ implies $\sigma_{i_0} \sqsubseteq \ell(l)$, which is in contradiction to the hypothesis, since $\ell\big(\mathsf{tell}(l \to r \mid c)\big) = \ell(l)$ (by equation (6.1d)).

$\mathsf{del}(l \to r \mid c)$**:** According to rule ($\mathsf{SR}_{\mathsf{del}}$), $\Delta^{p_0}_{F_0}$ implies $\sigma_{i_0} \sqsubseteq \ell(l)$, which is in contradiction to the hypothesis, since $\ell\big(\mathsf{del}(l \to r \mid c)\big) = \ell(l)$ (by equation (6.1d)).

$(c := v)$**:** According to rule ($\mathsf{SR}_{:=}$), $\Delta^{p_0}_{F_0}$ implies $\sigma_{i_0} \sqsubseteq \ell(c)$, which is in contradiction to the hypothesis, since $\ell(c := v) = \ell(c)$ (by equation (6.1e)).

$\mathsf{new}(c, s)$ or skip**:** In this case we have $\ell(\mathsf{a}^{i_0}_j) = \top$ (according to equation (6.1f)), which is in contradiction to our assumption.

□

Using the preceding lemmas, we can prove the correctness of our analysis, *i.e.*, we can prove that if a concrete execution is unsafe with respect to a privacy map ℓ, than the ℓ is not compatible with the constraint system Δ^p_F.

6.29 Theorem (correctness). *Let $\mathcal{C} = \langle \Sigma, P, \mathcal{R}, \mathcal{R}^{\mathrm{p}}, p\rangle$ be a component, $\mathfrak{L}$ a privacy lattice and Δ^p_F the constraint system associated to $F = \langle \Sigma, \mathcal{R}\rangle$ and p. Then we have for all privacy maps ℓ for Σ that if ℓ is compatible with Δ^p_F then $\langle F, p\rangle$ is safe with respect to ℓ.*

Proof. Suppose that the privacy map ℓ compatible with Δ^p_F and that $\langle F, p\rangle$ is not safe. According to definition 6.9, there exists thus a privacy level $\pi \in \mathfrak{L}$ and two sets of rewrite rules $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $F^1_0 \cong^\ell_\pi F \cong^\ell_\pi F^2_0$ but $\langle F^1_0, p_0\rangle \not\cong^\ell_\pi \langle F^2_0, p_0\rangle$, where $F^i_0 \stackrel{\mathsf{def}}{=} \langle \Sigma, \mathcal{R}_i\rangle$ $(i \in \{1;2\})$ and $p_0 \stackrel{\mathsf{def}}{=} p$. According to lemma 6.10, we have a natural number n[13] and two transition sequences

$$\begin{aligned} &\langle F^1_0, p_0\rangle \xrightarrow{a_1} \langle F^1_1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F^1_n, p_n\rangle \\ &\langle F^2_0, p_0\rangle \xrightarrow{a_1} \langle F^2_1, p_1\rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle F^2_n, p_n\rangle \end{aligned} \tag{6.27}$$

such that $F^1_i \cong^\ell_\pi F^2_i$ for all $i < n$, and that one of the following to cases holds.

Case 1: $F^1_n \not\cong^\ell_\pi F^2_n$. Suppose that the guarded action $a_n = [g \Rightarrow \mathsf{a}_1; \ldots ; \mathsf{a}_m]$ is composed of the guard g and the sequence of elementary actions a_i $(i \in \{1; \ldots ; m\})$. Thus we have by rule ($\mathsf{sR}_{action}$) the following sequences of transitions describing the execution of the elementary actions a_i:

$$\begin{aligned} \big\langle F^j_{n,0},\ \mathsf{a}_1; \ldots ; \mathsf{a}_m ; \mathsf{skip}\big\rangle &\twoheadrightarrow \big\langle F^j_{n,1},\ \mathsf{a}_2; \ldots ; \mathsf{a}_m ; \mathsf{skip}\big\rangle \\ &\twoheadrightarrow \big\langle F^j_{n,2},\ \mathsf{a}_3; \ldots ; \mathsf{a}_m ; \mathsf{skip}\big\rangle \\ &\ \ \vdots \\ &\twoheadrightarrow \big\langle F^j_{n,m},\ \mathsf{skip}\big\rangle \end{aligned} \tag{6.28}$$

where $F^j_{n,0} \stackrel{\mathsf{def}}{=} F^j_{n-1}$ (for $j \in \{1;2\}$). Let i_0 be such that $F^1_{n,i_0-1} \cong^\ell_\pi F^2_{n,i_0-1}$ and $F^1_{n,i} \not\cong^\ell_\pi F^2_{n,i}$ for all $i \geq i_0$. Notice that we have $i_0 > 0$ and that i_0 exists, since

[13] We use a here a lowercase n instead of the uppercase N used in lemma 6.10 to enhance the readability.

(by assumption) $F^1_{n,0} \cong^\ell_\pi F^2_{n,0}$ and $F^1_{n,m} \not\cong^\ell_\pi F^2_{n,m}$. We consider the different possibilities for the elementary action a_{i_0} one by one, and show that each of them leads to a contradiction with the assumptions.

$\mathsf{tell}(R)$**:** According to lemma 6.17, we have that the abstract action $\mathsf{tell}(\tilde{R})$ has been executed during the analysis, *i.e.*, the computation of the constraints Δ^p_F. We distinguish the following two cases:

- $\tilde{R} \sqsubseteq \pi$**:** The rule R added to the store is the same for both, F^1_{n,i_0-1} and F^2_{n,i_0-1}. Thus, we have that $F^1_{n,i_0} \cong^\ell_\pi F^2_{n,i_0}$, in contradiction to the assumption.
- $\pi \sqsubset \tilde{R}$**:** In this case, the rules of a lower or equal privacy level than π are not modified, and thus we cannot have $F^1_{n,i_0} \not\cong^\ell_\pi F^2_{n,i_0}$.

$\mathsf{del}(R)$**:** We distinguish two cases:

- $\tilde{R} \sqsubseteq \pi$**:** In this case, the rule R is present in F^1_{n,i_0-1} if and only if R is present in F^2_{n,i_0-1}. Thus the removal of R has the same effect, and we have that $F^1_{n,i_0} \cong^\ell_\pi F^2_{n,i_0}$, in contradiction to the assumption.
- $\pi \sqsubset \tilde{R}$**:** Since the rules of the store which have a lower or equal privacy level than π are not modified by the execution of this action, we have $F^1_{n,i_0} \cong^\ell_\pi F^2_{n,i_0}$, in contradiction to the assumption.

$c := v$**:** According to lemma 6.17, we have that the abstract action $\tilde{c} := \tilde{v}$ has been executed during the analysis. We distinguish the following two cases:

- $\tilde{l} \sqsubseteq \pi$**:** Since $F^1_{n,i_0-1} \cong^\ell_\pi F^2_{n,i_0-1}$, we have that $v{\downarrow_{F^1_{n,i_0-1}}} = v{\downarrow_{F^2_{n,i_0-1}}}$ (using lemma 6.7; by lemma 6.27 we have that the stores F^1_{n,i_0-1} and F^2_{n,i_0-1} are safe). Consequently $F^1_{n,i_0} \cong^\ell_\pi F^2_{n,i_0}$, in contradiction to the assumptions.
- $\pi \sqsubset \tilde{l}$**:** In this case, the assignment does modify only rules of a higher privacy level then π, and thus we cannot have $F^1_{n,i_0} \not\cong^\ell_\pi F^2_{n,i_0}$.

$\mathsf{new}(c,\, s)$**:** This is impossible, since new does not modify the rules of the store (see rule ($\mathsf{sR}_{\mathsf{new}}$)).

skip**:** This is impossible, since skip does not modify the store (see rule ($\mathsf{sR}_{\mathsf{skip}}$)).

Case 2: there exists a process term $p^\sharp$ and a store $F^\sharp$ such that $\langle F^1_n,\, p_n\rangle \rightarrow^+ \langle F^\sharp,\, p^\sharp\rangle$ but $\langle F^2_n,\, p\rangle \not\rightarrow$ and $F^\sharp \not\cong^\ell_\pi F^2_n$. Thus we have a transition sequence

$$\langle F^1_n,\, p_n\rangle \xrightarrow{a_{n+1}} \langle F^1_{n+1},\, p_{n+1}\rangle \xrightarrow{a_{n+2}} \cdots \xrightarrow{a_{n+m}} \langle F^1_{n+m},\, p_{n+m}\rangle \tag{6.29}$$

such that $F^1_{n+j} \cong^\ell_\pi F^2_n$ for all $j \in \{1; \ldots; m-1\}$ and $F^1_{n+m} \not\cong^\ell_\pi F^2_n$. Consequently, we have that the privacy level of a_{n+m} is lower than π, *i.e.*, $\ell(a_{n+m}) \sqsubseteq \pi$. Suppose that the guarded action $a_{n+1} = [g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_k]$ is composed of the guard g and the sequence of elementary actions a_i ($i \in \{1; \ldots; k\}$). Notice that $\pi \sqsubset \ell(g)$, since $\langle F_2,\, p\rangle \xrightarrow{a_{n+1}} \not\rightarrow$ (otherwise we have a contradiction to lemma 6.7 for the evaluation of g which yields the same result with respect to both stores). Thus we have that $\ell(a_{n+m}) \sqsubset \ell(g)$. On the other hand, we have by lemma 6.28 that the compatibility of ℓ with $\Delta^{p_n}_{F^1_n}$ implies that $\ell(g) \sqsubseteq \ell(a_{n+m})$. Thus, using lemma 6.23, we reach a contradiction.

Q.E.D.

Informally, the two cases in the proof of theorem 6.29 correspond to the two different possibilities for information flow. Case 1 handles a direct information flow, whereas an information flow induced by a control flow is handled in case 2. Notice that in the first case we consider, as already in the proof of lemma 6.27 the last elementary action, to take into account the atomic execution of the sequence of elementary actions of a guarded action.

The following examples show that there exist process term which respect secrecy, but which are rejected by our analysis.

6.30 Example. *Consider a process term p_1 which respects privacy (for a privacy map ℓ), and a process term p_2 which does not respect secrecy (for ℓ). Let g be an arbitrary guard (for the component for which p_1 and p_2 are defined). Then the process term $\big([g \Rightarrow \mathsf{skip}]\,;p_1\big) \oplus \big([g \Rightarrow \mathsf{skip}]\,;p_2\big)$ obviously respects secrecy (for ℓ) since p_2 will be never executed. However, our analysis rejects this process term.*

6.31 Example. *Consider the process term $\big[\text{TRUE} \Rightarrow SPY := PIN\,;\ SPY := 0\big]$ Due to the atomic execution of the sequence of elementary actions, the "bad" elementary action $SPY := PIN$ has no observable effect. However, our analysis yields a constraint system which is incompatible with the privacy map ℓ, where ℓ is defined by $\ell(SPY) \stackrel{\text{def}}{=} \bot$ and $\ell(PIN) \stackrel{\text{def}}{=} \top$.*

6.32 Example. *Consider the following process term*

$$\begin{array}{rcl} \mathrm{q} & \Leftarrow & [PIN = 42 \Rightarrow SPY := 0\,;\ \mathsf{skip}]\,;\ \mathsf{success} \\ & \oplus & [PIN \neq 42 \Rightarrow SPY := 0\,;\ \mathsf{skip}]\,;\ \mathsf{success} \end{array}$$

is analysed as not respecting secrecy for a privacy map that assigns, as in example 6.31, $\top$ to PIN and $\bot$ to SPY, whereas in fact the modification of the store by q *does not depend on the actual value of PIN.*

——— ⋆ ——— ⋆ ——— ⋆ ———

Secrecy has been well investigated in many different programming paradigms: imperative programming [SVI96], functional programming or more precisely the λ-calculus, *e.g.*, [ABHR99, Pro00, HR98], concurrent programming [Aba97, HR00, HVY00]. Nevertheless the line of approach of these works is rather theoretical, since basic computation models are considered, *e.g.*, the λ-calculus, π-calculus or other theoretical models which do not aim to be a support for programming of practical applications. We have defined an analysis for our computation model which is closer to reality: in chapter 7, we describe a prototype implementation of a platform supporting the presented computation model. That is to say, the language we consider is more expressive than the one used in [BC01a, BC01b] and [SV98] in two respects. First, we provide a general sequential composition operator, *i.e.*, the process term $(p_1 \parallel p_2)\,;p_3$ is legal in our model, in contrary to the language used in [BC01b] and [SV98]. Second, our framework allows the dynamic creation of parallel processes.

Similar to the type system presented in [BC01b], our analysis is finer than the one of [SV98], since we do not restrict the guards to be of the lowest privacy level. Our analysis is based on an abstract execution instead of a type system. Therefore we do not need to assign a privacy level to a process corresponding to the τ in [SV98, BC01b]. Notice finally, that the examples 6.30 and 6.32 are also rejected by the type systems of [SV98, BC01b]. Example 6.31 is not directly comparable, since the languages considered in [SV98, BC01b] do not offer a construct for atomic grouping of actions.

Chapter 7

Implementation: Sabir

In order to prove the feasibility of our computation model, we have implemented a first prototype of a multiparadigm programming platform based on the computation model presented in the preceding chapters. This chapter presents this prototype which we call Sabir[1]. First we present in the following section the general architecture and implementation principles of our interpreter for components. Then we illustrate in section 7.2 the use of our prototype by means of some examples of execution traces of components using Sabir.

7.1 Presentation of Sabir

The current version of Sabir is an interpreter for *one* component, that is to say Sabir takes a description of a component as input and interprets or executes the component. Therefore, to interpret a system consisting of several components, a programmer has to start several instances of Sabir at the same time. Since these components communicate via the Internet, *i.e.*, using TCP/IP connections, it is possible to execute a distributed system, *i.e.*, a system the components of which are located on different computers.

As we have seen in chapter 3, a component $\mathcal{C}$ is defined by an eight-tuple, *i.e.*, $\mathcal{C} = \langle \widehat{sn}, \mathrm{C}\Sigma, \mathcal{R}, \mathfrak{A}, \mathcal{T}r, \mathcal{R}^{\mathrm{p}}, \mathrm{I}\!\mathrm{I}\mathcal{R}, p^{\mathrm{i}} \rangle$ (see definition 3.62 for more details). To simplify the effective description of a component, in Sabir a component is described by a set of five different parts or files, as is symbolised in figure 7.1, namely:

F the (initial) *store* or declarative program,

A the definitions of the *actions* that are executable on the store F,

P the definitions of *processes*, together with the initial process term,

T the definitions of *translations* for communicating values of the store to other stores that are possibly written in a different language and

I the *imported* (respectively, *exported*) definitions from (respectively, to) the environment, *i.e.*, other components of the system.

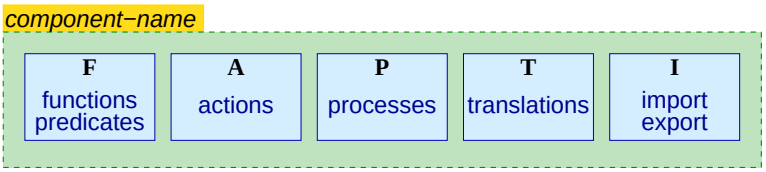

Figure 7.1: A Programmer's View of a Component

The motivation of this separation is to group the declaration of symbols with the corresponding definitions, by splitting the component signature according to the different kinds of definitions. Recall that the symbols of a component are defined in a component signature which is defined as an eight-tuple $\mathfrak{C}\Sigma = \langle \Sigma, \mathfrak{M}\Sigma_{\mathcal{L}}, A, I, E, Trans, P, \Pi \rangle$. Part **F** describes the initial store F of the component, *i.e.*, $F = \langle \Sigma, \mathcal{R} \rangle$, a pair of a signature Σ and a set of rules $\mathcal{R}$. The action signature $\mathfrak{A}\Sigma$ is grouped with the definitions of actions $\mathfrak{A}$ in part **A**. Recall (from section 3.2.1) that an action signature $\mathfrak{A}\Sigma$ includes, besides a family of action names, also a meta-signature, allowing the representation of the store as an abstract data type (ADT) and the definition of action in an action definition language (ADL). Processes are grouped with process functions and the initial process term: Part **P** contains the declaration of the process symbols P and the process function symbols Π (from the component signature) and the process definitions $\mathcal{R}^{\mathrm{p}}$, the definitions of process functions $\Pi\mathcal{R}$ and the initial process term p from the component. Part **T** defines the translations, *i.e.*, it groups the declaration of the translation symbols *Trans* with their definitions $\mathcal{T}r$. The remaining symbols of the component signature are left undefined, since they are imported from other components (where they are supposed to be defined). The imported declarations I and the storenames $\mathcal{SN}$ of the remote stores form the part **I**, together with the declaration of the exported symbols, *i.e.*, the symbols (declared in the component at hand) that remote components are allowed to use. The grammars describing the syntax of these five different files are given in appendix A.

The general scheme of the interpretation process for a single component in Sabir is shown in figure 7.2. The interpreter takes as an argument the name of the component, and tries to read the (five) files corresponding to the parts of the description of a component mentioned above[2]. These files are processed in the order indicated by the numbers in figure 7.2. Using the information contained in the files, the "compiler" produces an "abstract forest" (or parse-tree), *i.e.*, an internal, intermediate representation of the component (which might be written into a file and stored). Finally, the abstract forest is passed to an "interpreter" which executes the component.

In the following paragraphs we present in more detail both, the contents of the different files describing a component and the different steps in the execution of a component. Our prototype Sabir is implemented in the programming language `ocaml`

[1]Etymologically, the word "Sabir" stems from the Spanish "saber" (to know). Historically, "Sabir" denotes a language created for the communication between people of different mother tongues. Thus Sabir denotes a mixed language, most notably the *lingua franca* used in Mediterranean harbours.

[2]The file-names of the files corresponding to these parts are obtained by adding suffixes to name of the component, namely `.store` for part **F**, `.actions` for part **A**, `.procs` for part **P**, `.import` for part **I** and `.trans` for part **T**.

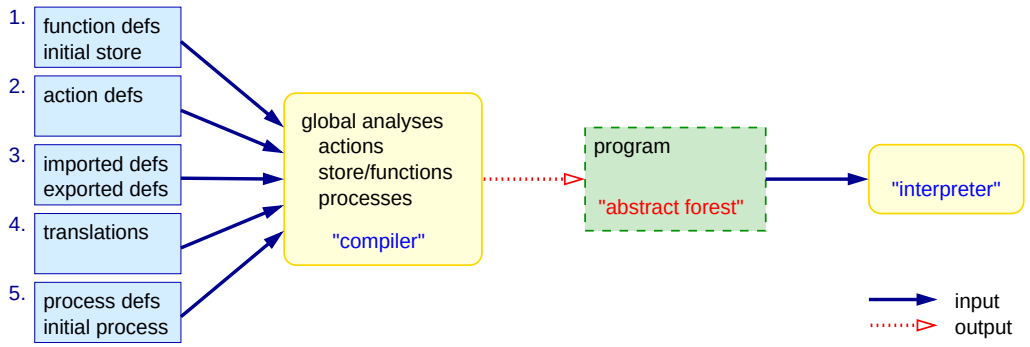

Figure 7.2: Global Vision of the Interpretation Process of a Component in Sabir

[LDG+01].

7.1.1 Part F

The stores of components in Sabir are programs in declarative languages following the principles presented in section 3.1.2. One of the principal motivation in implementing a rather simple declarative language on our own was the need for an ADL for the definition of actions. While the corresponding (meta-) data types necessarily to exist as well for existing declarative languages, they are more difficult to access, not as well documented as the use of the language itself and more sophisticated than a straightforward, naive implementation, since they need to handle additional features as for example I/O.

We have chosen a syntax close to Curry [HAK+00b] and Haskell [PJHA+99], since the definition of a function by several equations is rather close to the rewrite rules of section 3.1.2. However, to simplify our parser, we require all terms to be parenthesised completely, and do not provide local definitions or `where`-clauses. Furthermore, we require that all symbols are declared before they are used, which allows a simple parser (in a single pass) and avoids the need to *infer* types. Finally, as in section 3.1.2, we use a simple sorting instead of a polymorphic type system with higher-order functions. The corresponding extensions of our prototype are under development.

The operational semantics is based on a conditional version of weakly needed narrowing [AEH00], an optimal evaluation strategy, representing the rules of functions internally by definitional trees [Ant92]. In the current version of the first, simple declarative language used in Sabir, non-determinism is solved by parallel evaluation, *i.e.*, in the case of several possibilities for the reduction of a term, the evaluation process is split up and all possibilities are continued in parallel. Consequently, the current implementation is complete, in the sense that every possible answer is computed.

7.1.2 Part A

As the declarative language for stores (see sections 3.1.2 and 7.1.1) is implemented in `ocaml` [LDG+01], it seems natural to chose `ocaml` as an ADL. In fact, by using the implementation language of the declarative language, we have a straightforward access to all the data types representing the meta-signature. Therefore an action is imple-

mented by a classical `ocaml`-function using the same modules for the representation of programs as the interpreter for the declarative language of Sabir.

To allow for a more efficient execution of actions during interpretation of a component, we do not *interpret* actions, but exploit the possibilities offered by the `Dynlink`-library of `ocaml` [LDG+01, chapter 26], which allows to link (previously) compiled `ocaml`-code dynamically, *i.e.*, at run-time, into a program. Thus, actions are used in form of (pre-) *compiled* `ocaml`-functions that are (dynamically) linked to the interpreter. The `ocaml`-functions defining the actions have to be compiled using the standard `ocaml`-compiler and a library containing the necessary definitions of Sabir, before we can start the interpretation of a component using these actions.

7.1.3 Part I

In the current prototype of Sabir, this part is mainly used for the specification of the storenames *SN* with respect to which the component is defined, since the current prototype of Sabir does support only very limited control about imported and exported symbols, in order to simplify its implementation. Currently, *all* declarations of stores are considered as imported (respectively, exported), such that we only need to parse the files describing the stores of the remote components to know all imported symbols (of the stores). However, the imported set of actions executable on the remote store has to be mentioned explicitly.

Concerning the specification of the storenames *SN* of the other components in the system, this part specifies besides the symbolic names also the port number and IP-address of the remote components. These addresses are used by the interpreter of a component to establish the communication links between components.

7.1.4 Part T

To simplify our current prototype implementation, we suppose that all languages for the stores supported by the interpreter have a syntax similar to the one used for the description of the store used in part **F**[3]. Thus the specification of translations can use the same syntax for rules as part **F**.

7.1.5 Part P

The part describing the processes of a component contains besides the process definitions (see definition 3.58) also the initial process term. The concrete syntax (see appendix A) for process terms enforces that the initial process, as well as the process terms in the rules of the process definitions, are *restricted* process terms.

The current implementation of Sabir does not include process functions and p-rules, nor a type checker for process expressions. In fact, as mentioned in section 3.5.2, we consider process functions as a means to *facilitate* the description of processes. Thus we have, by lack of time, excluded these features in our first prototype. The corresponding extensions are under progress. Consequently, the only simplification of

[3]Otherwise, we would need a parser that can handle expressions written in a syntax mixing those of different languages.

process expressions which takes place in the current version of Sabir is the application of translations.

7.1.6 Execution of a Component

The execution of a component proceeds in several steps. First, in order to prepare the execution of the component as a part of a system, a handler for the mail-box is set up so that the component is ready to accept connections from other components of the system. Then the interpreter tries to establish the connections to the other components corresponding to the declared storenames $\mathcal{SN}$ (which are specified by part **I** as mentioned above). When all connections are working (or after a rather long timeout), the execution strictly speaking the strict sense of the component is started.

The execution of a component strictly speaking consists in the execution of an interactive interpreter for the declarative program or store, together with the execution or interpretation of process expressions. This interpretation follows rather closely the operational semantics as presented in chapter 4. First, execution of the initial process is started, and in parallel the process handling the incoming mailbox is unlocked, so that messages arriving from other components can effectively be handled. Additionally two (interactive) interpreters are launched. The first is an interpreter of the declarative language for the store and the second is an interpreter for the interactive execution of actions on the store. The former can be used for interactive use of the theory described by the store, and the second allows a user to update the theory (or program) remotely, similar to the primitives for exchanging the code of modules in Erlang [AVWW96].

The mutual exclusion of the execution of actions is ensured in Sabir by protecting the stores with a *monitor* [Han72, Hoa74], a concept first defined for the design of operating systems which is now implemented as libraries for a range of programming languages, as for instance JAVA [Lea99, secion 3.3.2] or `ocaml` [LDG+01, chapter 23]. Informally, all procedures of monitor are guaranteed to be executed in mutual exclusion. We use this property to ensure that at most one process (including the interactive interpreters and the process handling the mailbox) has access to the store. Additionally, a procedure of the monitor may suspend its execution and *wait* for a resource to become free or, in our case, a condition or guard to become true. In terms of monitors, this is means to wait on a so-called *condition variable*. When a process has finished the execution of an action, it releases the store and *signals* the condition variable, that is to say, wakes one (or all) of the waiting processes up. When woken up, a process resumes its execution just after the *wait* instruction, but *before* another process can enter the monitor. Therefore we can implement a process blocking on a guard by putting the test of the validity in a loop: while the guard is not valid, the process has to wait on the condition variable and to recheck the guard as soon as it is woken up (which means that another process has modified the store such that the guard might have become TRUE), otherwise it can proceed with the execution of the action.

7.2 Example of a Lift Controller

In the rest of the chapter, we present samples of descriptions of components in Sabir and their execution traces. In contrary to the examples in chapters 1 and 3, we use here

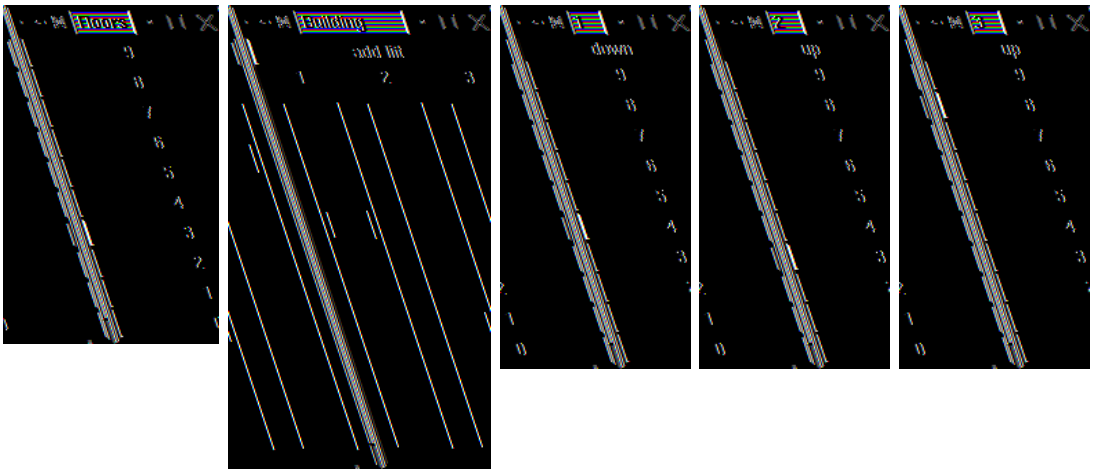

Figure 7.3: Windows of the Lift Controller Application

the concrete syntax of the current prototype of Sabir (see appendix A). Lines starting with `--` are comments, and are not considered by the parser.

Our first example is inspired from a very vague specification in [Abr96a]. Consider a building with m floors in which a system of n lifts is installed. On every floor there is a button for requesting a lift, and each lift is equipped with buttons for ordering the lift to stop at a given level. Suppose we have to model this system, *i.e.*, control the lifts, ensuring that all orders and requests are eventually handled. For simplicity, we do not consider the capacity of the lifts, assuming that it is always sufficient for the requests to handle.

Figure 7.3 shows a snapshot of an execution of our lift control system for a building with $m = 10$ floors (numbered from 0 to 9) and $n = 3$ lifts. We see five windows. The first window, entitled Floors, represents the ten buttons on the floors. A second window, entitled Building, visualises the current positions of the lifts[4], where a black (respectively, white) rectangle symbolises a lift the doors of which are closed (respectively, open). The remaining three windows, entitled 1, 2 and 3, represent the button inside the lifts, as well as the current direction of the corresponding lift (*i.e.*, up or down).

The intuitive idea of our model is as follows. We model the lifts as independent processes that share the information of requests coming from the floors. Hence, our model does not depend on the numbers of floors and lifts, and we can, for instance, easily add an additional lift[4]. This feature might be useful when we want to use the system for the evaluation of the number of lifts actually needed for the building (in this case, it would be necessary to consider the capacity of the lifts as well).

We distinguish between the `requests` issued by the buttons on the floors which can be handled by any of the lifts and `orders` which have to be handled by a given lift. `orders` are either issued by the buttons inside the lift or requests that have been assigned to the lift. In order to optimise the assignment of requests to lifts, we try to

[4]Clicking on the button add lift in the window Building also allows to dynamically add a new lift.

make the lifts move as long as possible in one direction, *i.e.*, either `up` or `down`. Thus we can model a `lift` as a four-tuple `(L ready direction position orders)` containing its current level or `position` (represented by an `integer`), its current `direction` and a list of `orders` to handle, represented as a list of floors-numbers (*i.e.*, `integer`s). The flag `ready` is a boolean value indicating if the lift is ready to move after the doors have been opened in order to handle a request or order. One of the conditions that have to be satisfied in order to consider a lift a ready is, for instance, that the doors of the lifts have to be closed. Orders from the buttons inside a lift are directly put in the list of `orders` of the lift, while requests on the floors are put into a list `requests` shared by all lifts. A process controlling the lift `ln` moves a request from the list `requests` into the list of `orders` of `ln`, if the request is in the current direction of `ln` and if `ln` is among the lifts that are `nearest` to the request. Whenever a lift handles an order on a floor, all requests for this floor are handled (*i.e.*, removed from the list `requests`) as well.

As in the example of the multiple counters (see example 3.63), we suppose that, besides the component `lifts` (modelling the lifts), we have a second component, namely `X`, which acts as a GUI for the lifts. The actions executable on the component `X` are the following:

```
action move_to_floor :: Name(lift) -> integer -> store -> store.
action open_doors    :: Name(lift) -> integer -> store -> store.
action close_doors   :: Name(lift) -> integer -> store -> store.
```

Their first parameter denotes the (name of the) lift, and the second the current level of this lift.

Tables 7.1 and 7.2 give the declarations of types and functions (*i.e.*, the signature) of the store for the component `lifts`. The corresponding rules are presented in table 7.3 and 7.4. Finally, the process definitions of the component `lifts` are given in tables 7.5 and 7.6. The initial process term of the component `lifts` is a call to the process `start` (see table 7.6). In the following, we comment the program shown in tables 7.1 to 7.6, using the process definitions as a guideline, that is to say, we explain the functions of the store in the context of the process where they are used. The explanations of some of the simpler functions are omitted, since for instance the classical sort of lists (`int_list`) has already been presented in chapter 3 in the example of the multiple counters.

The definition of the process `lift_ctrl` (see table 7.5) controlling a lift named `ln` is shown in table 7.5 (where we denote by `s^` the `Name` of a symbol `s`, and the symbol associated to a name `n` by `(deref n)`[5]). The process is defined by four rules or clauses. The guard of the first rule, *i.e.*, `(order_to_handle (deref ln))`[6], checks if there is an order or request to handle on the current floor (of the lift `ln`), *i.e.*, `order_to_handle` tests if the current position of the lift `ln` is a `member` of the concatenation (`append`) of the list of `orders` of the lift `ln` and the global `requests`. Handling an order (respectively, request) simply means to remove it from the lists of orders `(ord ln)` (respectively, requests `requests`). This is expressed by the first two actions, which

[5] In section 3.1.3 we used the notations $\hat{\mathtt{s}}$ (respectively, $\mathtt{n}\uparrow$).

[6] In the case of a *boolean* expression, say `b`, as a guard, we abbreviate the notation and write simply `b` instead of `b = true`.

```
-- type of list of naturals
type int_list = nil | cons integer int_list.

-- classical functions on lists
head   :: int_list -> integer
tail   :: int_list -> int_list
append :: int_list -> int_list -> int_list

-- classical predicates on lists
member  :: integer -> int_list -> bool
is_nil  ::            int_list -> bool
not_nil ::            int_list -> bool


-- return the list without all (the first) occurrence(s)
remove    :: integer -> int_list -> int_list
rm_first  :: integer -> int_list -> int_list

-- return the sub-list of greater (smaller) elements
get_above :: integer -> int_list -> int_list
get_below :: integer -> int_list -> int_list

-- return the list of differences (with the parameter)
get_dists :: integer -> int_list -> int_list

-- test if there are elements greater (smaller)
orders_above :: integer -> int_list -> bool
orders_below :: integer -> int_list -> bool

-- test if all elements of the list are greater
all_above :: integer -> int_list -> bool

-- is the position "a" nearest to a request "b" ?
nearest :: integer -> integer -> bool


-- type of directions
type direction = up | down.

-- compute the next floor for a floor and direction
next :: integer -> direction -> integer


-- the "limit-floors" of the building
base :: integer
top  :: integer
```

Table 7.1: Signature of the store for the component `lifts`: Part 1

```
-- type of lifts
-- a lift is a 4-tuple <ready, direction, current floor, "orders">
-- ready is a flag for the control of the opening/closing of the doors
type lift = L bool direction integer int_list.

-- accessors to the fields of a lift
fin :: lift -> bool
dir :: lift -> direction
pos :: lift -> integer
ord :: lift -> int_list

-- "operations" on the lifts:
-- move to the next floor
next_pos    :: lift -> lift
-- change the direction
opposite    :: lift -> lift
-- remove the orders
rm          :: lift -> lift
-- toogle the ready-flag
set_timer   :: lift -> lift
clear_timer :: lift -> lift

-- add an order
add_lift_ord :: integer -> lift -> lift

-- return the sub-list of all elements in the direction of the lift
get_in_dir :: lift -> int_list -> int_list
-- return the sub-list of all elements not in the direction of the lift
rm_in_dir  :: lift -> int_list -> int_list

-- some "predicates" for the guards for the lift control process
-- order to handle on the current floor?
order_to_handle    :: lift -> bool
-- further orders in the current direction of the lift?
order_in_direction :: lift -> bool
-- new request in the direction of the lift?
new_req_in_dir     :: lift -> bool
-- any other requests ?
further_requests   :: lift -> bool

-- global constants:
-- a list containing the current positions of the lifts
lifts_pos :: int_list
-- the list of the (floor) requests
requests :: int_list
-- create a new lift?
create :: bool
```

Table 7.2: Signature of the store for the component `lifts`: Part 2

```
-- rules for the classical list functions
head (cons x y) = x
tail (cons x y) = y

append nil        a = a
append (cons a b) c = (cons a (append b c))

is_nil  nil        = true
not_nil (cons a b) = true

member a (cons b c) | (a = b) = true
member a (cons b c) | (a < b) = (member a c)
member a (cons b c) | (a > b) = (member a c)

-- compute particular sublists
get_above a nil                   = nil
get_above a (cons b c) | (a <  b) = (cons b (get_above a c))
get_above a (cons b c) | (a >= b) =         (get_above a c)

get_below a nil                   = nil
get_below a (cons b c) | (a >  b) = (cons b (get_below a c))
get_below a (cons b c) | (a <= b) =         (get_below a c)

get_dists a nil        = nil
get_dists a (cons b c) = (cons (abs (a - b)) (get_dists a c))

-- predicates on lists of requests/orders
requests_above a e = (not_nil (get_above a e))
requests_below a e = (not_nil (get_below a e))

all_above a nil                   = true
all_above a (cons b c) | (a <= b) = (all_above a c)

-- removal of requests/orders from a list
remove a nil                  = nil
remove a (cons b c) | (a = b) =         (remove a c)
remove a (cons b c) | (a < b) = (cons b (remove a c))
remove a (cons b c) | (a > b) = (cons b (remove a c))

rm_first a nil                    = nil
rm_first a (cons b c) | (a = b)   = c
rm_first a (cons b c) | (a < b) = (cons b (rm_first a c))
rm_first a (cons b c) | (a > b) = (cons b (rm_first a c))

-- is the position "a" nearest to a request "b" ?
nearest a b = (all_above (abs (a - b)) (get_dists b lifts_pos))
```

Table 7.3: Rules of the store for the component `lifts`: Part 1

```
-- next floor in a direction
next x up   | (x < top)  = (x + 1)
next x down | (x > base) = (x - 1)

-- accessors to the fields of a lift
fin (L r d p o) = r
dir (L r d p o) = d
pos (L r d p o) = p
ord (L r d p o) = o

-- "modifiers" of lists
rm (L r d p o) = (L r d p (remove p o))

next_pos (L r d p o) = (L r d (next p d) o)

opposite (L r up   p o) = (L r down p o)
opposite (L r down p o) = (L r up   p o)

set_timer   (L r d p o) = (L true  d p o)
clear_timer (L r d p o) = (L false d p o)

add_lift_ord x (L r d p o) = (L r d p (cons x o))

-- predicates for the guards
order_to_handle (L r d p o) = (member p (append requests o))

order_in_direction (L r up   p (cons o os)) =
       (requests_above p (cons o os))
order_in_direction (L r down p (cons o os)) =
       (requests_below p (cons o os))

further_requests (L r d p  nil       ) = (not_nil requests)
further_requests (L r d p (cons o os)) = true

new_req_in_dir (L r up   p o)
       | (nearest p (head requests)) = (p <= (head requests))
new_req_in_dir (L r down p o)
       | (nearest p (head requests)) = (p >= (head requests))

-- initialisation of "global" constants
requests = nil

create = false

base = 0
top  = 9

lifts_pos = nil
```

Table 7.4: Rules of the store for the component `lifts`: Part 2

```
process lift_ctrl ln :-
  [(order_to_handle (deref ln)) =>
   {lifts @ (assign requests^ (remove (pos (deref ln)) requests))};
   {lifts @ (assign ln (rm (deref ln)))};
   {X     @ (open_doors ln (pos (deref ln)))}
  ];
  ((random_wait 7); (signal_timeout ln)) || (handle ln),

  [(new_req_in_dir (deref ln)) =>
   {lifts @ (assign ln (add_lift_ord (head requests) (deref ln)))};
   {lifts @ (assign requests^ (tail requests))}
  ];
  (lift_ctrl ln),

  [(order_in_direction (deref ln)) =>
   {lifts @ (assign lifts_pos^ (rm_first (pos (deref ln)) lifts_pos))};
   {lifts @ (assign ln (next_pos (deref ln)))};
   {lifts @ (assign lifts_pos^ (cons (pos (deref ln)) lifts_pos))};
   {X     @ (move_to_floor ln (pos (deref ln)))}
  ];
  (lift_ctrl ln),

  [(further_requests (deref ln)) =>
   {lifts @ (assign ln (opposite (deref ln)))};
   {X     @ (show_direction ln (string_of_direction (dir (deref ln))))}
  ];
  (lift_ctrl ln)
end

process handle ln :-
  [(order_to_handle (deref ln)) =>
   {lifts @ (handle_request (pos (deref ln)))};
   {lifts @ (assign requests^ (remove (pos (deref ln)) requests))};
   {lifts @ (assign ln (rm (deref ln)))};
   {X     @ (open_doors ln (pos (deref ln)))}
  ];
  (handle ln),

  [(new_req_in_dir (deref ln)) =>
   {lifts @ (assign ln (add_lift_ord (head requests) (deref ln)))};
   {lifts @ (assign requests^ (tail requests))}
  ];
  (handle ln),

  [(fin (deref ln)) =>
   {lifts @ (assign ln (clear_timer (deref ln)))};
   {X     @ (close_doors ln (pos (deref ln)))}
  ];
  (lift_ctrl ln)
end
```

Table 7.5: Process Definitions for the component `lifts`: Part 1

```
process signal_timeout ln :-
  [true => {lifts @ (assign ln (set_timer (deref ln)))}]; success
end

process start :-
  [create =>
   {lifts @ (assign create^ false)};
   {lifts @ (new ln lift)};
   {lifts @ (assign ln (L false up base nil))};
   {lifts @ (assign lifts_pos^ (cons base lifts_pos))};
   {X     @ (create_lift_window ln lifts)}
  ];
  (lift_ctrl ln) || start
end
```

Table 7.6: Process Definitions for the component `lifts`: Part 2

modify the store of the component `lifts`accordingly. The removal of the order in the lift is encapsulated inside the function `rm`. Notice that the removal of the request in both lists is *atomic*, so that each request is handled exactly one lift. The third action displays the handling in the GUI by executing the action `open_doors` on the component `X`. After executing the actions, the process becomes the parallel composition of, on the one hand, the process `handle` and, on the other hand, the sequential composition of the special process `random_wait` and the process `signal_timeout`. The predefined process `random_wait` needs a random amount[7] of time for its successful termination, and the process `signal_timeout` (see table 7.6) sets the ready flag of the lift `ln`. The process `handle` is similar to the process `lift_ctrl` and presented later on. Intuitively, it represents a lift the doors of which are open, but which can still handle requests and orders on the current floor. In a real lift, this situation corresponds for instance to persons arriving just before the doors close.

The second rule of the process `lift_ctrl` describes the transformation of the first request of the list `requests` into an order of the lift. The guard `new_req_in_dir` checks if the first element of the list of `requests` is a request in the direction of the lift such that the lift is among the lifts that are `nearest` to the floor of the request. The boolean function `nearest` takes two parameters, namely the position of the lift and the request, and checks if the other lifts are all at least as far as the current lift, using the global constant `lifts_pos`, which corresponds to the list of positions of all the lifts. The two actions of the second rule describe the addition of the request as an order to the lift and the removal of the request from the list `requests`. Similar to the removal of an order, the addition of an order to the list of orders of a lift is encapsulated inside the function `add_lift_ord`.

The third rule describes the movement of the lift. Obviously, a lift should continue to move in its current direction, if it has pending orders in its current direction. For an upwards (respectively, downwards) moving lift, the guard `order_in_direction` checks if the sub-list of orders that are greater (respectively, smaller) than the current

[7]The amount is randomly chosen between 0 and the number of second specified by the parameter.

position of the lift is not empty. The movement of the lift is described by three actions. First, the current position of the lift is removed from the list `lifts_pos` recording the current positions of all lifts (see the previous paragraph). The second action uses the function `next_pos` which encapsulates the changement of the position of the lift properly speaking. The third action adds the new position of the lift to `lifts_pos`. The GUI is updated by means of the fourth action.

Finally, the last rule of the definition of the process `lift_ctrl` describes the situation where a lift changes the direction. The guard `further_requests` is `true` if either the list of orders of the lift or the list of `requests` is not empty. Consequently, this guard also holds when one of the previous guards is valid. However, since the rules of a process definition are ordered by priority, the fourth rule is only chosen when the others rules can not be applied. Since the other guards consider all situations where a request or order exists in the current direction, it is reasonable to change the direction of the lift in case that there is another request pending. Notice that if none of the guards applies, the process `lift_ctrl` suspends until one of them becomes true. The fourth guard ensures that a lift does not suspend as long as there are requests pending in the system.

The process `handle` is the counter-part of the process `lift_ctrl`, in the sense that `lift_ctrl` describes the behaviour of a lift that is ready to move, whereas `handle` describes a lift that is handling a request, *i.e.*, stopped at a floor with open doors. Consequently, instead of the rules describing the movement and the change of the direction, `handle` has a further rule which describes the closing of the doors. The boolean function `fin` checks the ready-flag of the lift. This flag is set by the process `signal_timeout` which is started concurrently to `handle` whenever a request is handled by a lift. If the flag is set, it is cleared, the command to close the doors is sent to the GUI, and the process behaves afterwards as the process `lift_ctrl`.

Besides this rule, the process `handle` contains also the rules for handling a request (since persons might press on the button on the floor while the lift is there) and for selecting new requests from other floors. Notice that the rule describing the closing of the doors has to have a lower priority than the handling of request. In a real lift, this behaviour corresponds to the fact, that a person, which arrives just at the moment when the doors close, can still get on the lift by forcing the doors to reopen by pushing the button on the floor.

The process `signal_timeout` has already been explained along with the description of the first rule of the process `lift_ctrl` on page 199. The remaining process `start` controls the creation of new lifts as response to clicks on the "add lift"-button in the Building-window of the GUI (see figure 7.3). Its only rule is similar to the rules of the process cnt_ctrl describing the creation of a new counter window in the example of the multiple counters (see example 3.59 on page 114). We suppose that a click on the "add lift"-button sets the global boolean constant `create` to `true`. In this case, we have to reset `create` to false in order to allow further requests for the creation of lifts, create and initialise a new lift (which is placed on the lowest floor `base` and directed upwards), add the position of the new lift (*i.e.*, `base`) to the list `lifts_pos`, sent the command to create a new lift to the GUI and to continue waiting for a click on the "add lift"-button.

To execute the lifts, we have to start the interpretation of two components, namely

```
% sabir lifts

[?] > requests;;

NIL with {}

[?] > :p;;
<snip>
function s_a_b_i_r__ln_1 :: ( -> lift)
<snip>
{s_a_b_i_r__ln_1 -> (L false UP 0 NIL)}
<snip>

[?] > requests;;

(CONS 8 (CONS 6 (CONS 4 (CONS 2 NIL)))) with {}

[?] > s_a_b_i_r__ln_1;;

(L false UP 4 (CONS 8 (CONS 8 (CONS 6 NIL)))) with {}
```

Figure 7.4: Example of an Interactive Session for the Component `lifts`

`lifts` and `X`, by executing the interpreter `sabir` with the name of the component as argument. Under the hypothesis that the component `X` is started, Figure 7.4 shows a transcript of an interactive session for the component `lifts`. The output of Sabir is printed in `type-writer`, and the input of the user in **`bold type-writer`**.

The start of the interpreter for the declarative program presented in tables 7.1 to 7.1 is signaled by the prompt "`[?] >`". At the beginning, the evaluation of the function `requests` yields the empty list `NIL`[8]. The result of an evaluation is followed by the answer substitution, here the identity substitution `{}`. The primitive `:p` allows to print the current program. This primitive allows to detect the creation of a constant `s_a_b_i_r__ln_1` of sort `lift`, corresponding to the creation of a new lift[9]. Before the function `requests` is evaluated for the second time, the user has clicked on the buttons generating requests for the floors `2`, `4`, `6` and `8`, as can be seen from the result of the second evaluation. The evaluation of the function `s_a_b_i_r__ln_1` shows that, in the mean time, the lift has moved to floor `2` and is handling the corresponding request.

——— ⋆ ——— ⋆ ——— ⋆ ———

In this chapter we have described our prototype implementing a platform for the computation model presented in the preceding chapters. Besides a description of the different parts of a program which have to be produced by a user, *i.e.*, a programmer, we have given a brief overview of the architecture of the prototype and illustrated its use through

[8] In the output of the interpreter, constructors are printed in uppercase letters, but case does not matter (see appendix A).

[9] "`<snip>`" indicates omissions from the transcript corresponding to other lines of the store.

examples. Due to lack of time, the current version does not include all of the features of our computation model framework. However, it shows clearly the feasibility of the platform and gives hints for further improvements. Among these, the improvement of the efficiency of the implementation of the functional logic programming language used for the stores is already under progress. Further improvements are mentioned in the conclusion of the thesis (see chapter 9).

Chapter 8

Comparison with Related Work

In this chapter compare our computation model as presented in the preceding chapters to some related programming models and languages. Given the large amount of existing research in the field, we cannot compare our work to all the propositions in the area, but focus on those we are aware of and that are closely related to our model.

The chapter follows roughly the structure of the presentation of related work in chapter 2. However, as we already mentioned there, the order chosen is completely arbitrary, and some of the work might have been discussed in another section of this chapter. Furthermore, we omit a detailed comparison with some of the related work we already criticised in chapter 2 and mention some work which we did not present there. Finally, in order to reduce redundancy, we also refer the reader to chapter 2 for a presentation of the formalisms which we mention here.

8.1 Declarative Programming

The main difference with existing concurrent extensions of declarative programming languages is that we distinguish clearly between the notions underlying the declarative language and processes. Our motivation for this separation is to avoid the need to *encode* one concept by another. We strongly believe that this leads to more structured programs which in consequence are easier to write, read and understand. Furthermore, our approach is more general in the sense that it can be applied to extend most declarative languages with concurrency, since we do not rely on concepts to be available in the declarative language, but provide processes as an *additional* notion to those which are already offered by the declarative language. In fact, our approach requires only the conditions mentioned in sections 3.1 and 3.2.

However, besides this fundamental difference, there are some common points, which we point out for some selected examples of concurrent extensions of declarative programming languages we are aware of.

8.1.1 Logic Programming

The rules defining our processes (see definition 3.58) are syntactically similar to the clauses defining predicates in logic programming, with the difference that we provide more than just conjunction to combine the process terms of the bodies of the "clauses"

and that we explicitly order the clauses defining a process by priority. Therefore, definitions of processes are less "declarative" as definitions of predicates in logic programming. However, a process is a different concept which has to be distinguished from predicates, and the order of the execution of actions by a process matters, requiring a more imperative description of processes.

The built-in "predicates" `assert` and `retract` of Prolog [DEDC96] are similar to our actions. In fact, CIAO [CH99] and ESP [Cia94] interpret these predicates as Linda [Gel85] coordination primitives on a database of atoms. While this view does not improve our understanding of the semantics of this mix of predicates and actions, [CH99] presents nice implementation techniques for the execution of these particular actions.

$\mathcal{TR}$ [BK94, BK98a] extends the view of `assert` and `retract` as database updates by the introduction of *transactions*, *i.e.*, sequences of actions, which are, as in our computation model, to be executed *atomically*. A further similarity with our model is the notion of a shared store (respectively, database) which is modified by processes (respectively, transactions) which are defined separately by process definitions (respectively, in a transaction base). However, in contrary to our model, $\mathcal{TR}$ requires all actions to be *reversible* in order to allow the roll-back of an transaction. This requirement is not necessary in our model, since all actions of a guarded action (our abstraction corresponding to a transaction in $\mathcal{TR}$) are required to be executable if the guard is valid. Furthermore, the updates in $\mathcal{TR}$ are restricted to the modification of atoms. Thus, in contrary to our framework, the update of functions cannot be expressed directly.

The execution model of our components is closely related to the one of concurrent constraint programming (ccp) [Sar93], namely a (constraint) store shared by a number of processes. The main differences of our model to basic ccp are that the store of ccp can only be modified in a *monotonic* manner, and that the set of actions available is fixed (for a particular language) cannot be defined by the programmer.

Most of the semantics proposed for the family of ccp-languages consider the resulting, final store as the semantics of a process. Thus most of the semantics for ccp consider only *finite*, terminating executions. The only semantics for ccp considering explicitly infinite computations we are aware of are presented in [dBG97] and [FRS98], but even these approaches are concerned with the final, resulting store, which is modeled as a least fix-point[1]. We prefer a trace based semantics in order to model non-terminating processes controlling external devices.

The compositional (and fully abstract) semantics based on labeled execution traces for ccp presented in [dBP90, dBP91] inspired our compositional semantics of chapter 5. Several differences are worth to be mentioned. First, our language includes an operator for general sequential composition (instead of only action-prefixing which is a particular case of sequential composition, since the first process has to be a single action) and introduces an operator of choice with priority. Second, we are interested in possibly infinite executions, and do not have the notion of a *termination mode*. Finally, we distinguish *three* (instead of two) different labels, in order to emphasise the difference of local and distributed computation. Other investigations of the semantics of ccp concern the definition of a process algebra for ccp, *i.e.*, an equational characterisation

[1] Notice that this requires a monotone evolution of the store.

of a congruence relation for ccp processes, which considers only *terminating* processes [dBP92]. A behaviour similar to our operator of choice with priority can be achieved by means of the if-then-else-like construction "**now** c **then** A **else** B" construct of the timed extension of ccp presented in [dBG95].

Several non-monotonic extensions of ccp have been suggested, either by providing new built-in actions [dBKPR93, CR95] or by using non-monotonic logics, as a logic with defaults [SJG95, SJG96, GJS96] or linear logic [SL92, BdBP97, RF97, FRS98]. The user-definable actions in our computation model allow a greater degree of flexibility than the specific new built-in primitives of [dBKPR93] and [CR95]. Two of the solutions to the problem of the Dining Philosophers of [dBKPR93, CR95] use an additional arbiter process (second solution of [dBKPR93] and the one of [CR95]). The first solution of [dBKPR93] uses a specific constraint system which allows to model the atomic removal of two forks[2]. Similarly, the actions provided by the approaches based on linear logic are in our opinion less intuitive, since they rely on an *implicit* removal (of the constraints used for the proof of entailment of the guard), instead of specifying *explicitly* the constraints to be removed (as in our computation model). As most linear logic programming languages, linear Janus [Tse94], the implementation of the framework suggested in [SL92] does not distinguish between processes, stores and formulæ. The semantics of a process in [BdBP97] is defined by a *history*, which can be seen informally as a graph representing *all* possible executions and taking into account the causal dependencies between occurrences of basic actions. In contrary to most semantics for process calculi, this semantics does not need to interleave all actions in a sequential manner and is thus a truly concurrent semantics. Although this semantics considers only *finite*, *i.e.*, either successful terminating or deadlocking, computations, [BdBP97] sketches a solution to the (non-terminating) problem of the Dining Philosophers (see example 1.1.5), using an additional semaphore[3]. Since [FRS98] (similar to [SL92]) provides only an operator for prefixing a process with a guard, sequential composition, *i.e.*, imposing an order on the execution of tell operations has to be encoded. As an example, consider the solution suggested for the Dining Philosophers in [FRS98, section 2.2] which is defined as follows:

```
philosopher(I,N) =
   fork(I) ⊗ fork(I+1 mod N) ⟶
      (tell(eat(I,N)) ‖
```

[2]Informally, the predicate $use(x, leftfork)$ (respectively, $use(x, rightfork)$) models the fact that philosopher x uses the fork on his left (respectively, right). The constraint system is such that

$$use(x_1, leftfork) \wedge use(x_2, rightfork) = use(x_2, leftfork) \wedge use(x_3, rightfork) = \ldots = use(x_n, leftfork) \wedge use(x_1, rightfork) = false$$

Thus, using an atomic *atell*, a philosopher can only tell the use of his both forks, otherwise the constraint store would become *false*.

[3][BdBP97] sketches two further solutions to the problem of the Dining Philosophers. The first one is a translation to ccp (with an *atomic* atell) of the one suggested in [Sha89, page 1245], which is similar to ours, since a philosopher can take both forks (or sticks) in a single atomic step. The synchronisation is expressed by incrementally instantiating streams associated to the forks. The second solution is a translation to standard ccp of the one presented in [Rin88] which is rather long, complicated (70 lines of code, which is still incomplete) and requires the use of an additional semaphore for ensuring that no deadlocks occur. Both solutions do not support interactive goal-solving.

$$\texttt{eat(I,N)} \longrightarrow (\texttt{tell(fork(I)} \otimes \texttt{fork(I+1 mod N))} \longrightarrow \texttt{philosopher(I,N)}))$$

where the sequential composition "eat and then think" is encoded as

$$\texttt{tell(eat(I,N))} \parallel \texttt{eat(I,N)} \longrightarrow \texttt{P}$$

Notice that this program is close to our solution (see example 1.1.5) since the atomic removal of both forks (or sticks) is possible, and that no additional semaphore is needed. Indeed, the main difference is the implicit removal of the predicates `fork(I)`.

The notion of *ports* as a many-to-one communication medium has been introduced in AKL [JMH93]. It is argued that the introduced port primitives have a "logical reading" (as a special (port-) constraint[4]) and preserve the monotonicity of the constraint store. In our non-monotonic setting, we can provide the behaviour of ports via appropriate (elementary) actions. For instance, we can model the streams as lists of messages. The reception of a messages corresponds to access the head of the list, and sending amounts to simply add a new message to the end of the list. Notice that we could easily specify other communication schemes, for example we might want to introduce priorities of the messages. In this case, we would just have to modify the definition of the action send.

The programming system MOzart [Moz], based on the (multiparadigm) programming language Oz [Smo95b, VRHB+97], combines, in a distributed setting, a number of different programming paradigms, namely declarative programming, object-oriented programming, constraint programming and concurrent programming. Unfortunately, the formal semantical descriptions of the Oz language we are aware of [Smo94, Smo95a, Smo95b, Smo98], focus on particular aspects of the language and do not cover the framework as a whole. The assignment operation is modeled in Oz by the update of a *cell*, *i.e.*, by modifying the references contained in the cell to a binding to a new variable. Our actions allow to express the modification of the theory without the need to introduce the additional notion of cells. As in AKL, communication in (distributed) Oz is based on the notion of ports, the behaviour of which can be *defined* using the stateful features of Oz [Smo95b, section 9, pages 333 – 334].

Reactive CLP [FFS95, FFS98] investigates the dynamic change of the theory used for constraint or goal solving. The technique proposed is based on a rearrangement of the search tree in order to keep all reduction steps which depend exclusively on parts of the theory that have not been modified. It seems interesting to investigate the adaptation of these techniques in order to improve rule (P) (see page 131).

[4]We do not completely understand the semantics of this constraint, especially the (incremental?) modification of the stream of messages according to the execution of send constraints. Unfortunately, the port constraint is not defined formally, as the following quotation shows.

> "the interpretation in terms of constraints is not a complete characterisation of the behaviour of ports, [...]. In particular, it does not account for message multiplicity, nor for their "relevance", i.e., it does not "minimise" the ports to the messages that appear in a computation.
>
> A logic with resources could possibly help, e.g., Linear Logic [Gir87]. The don't care nondeterministic and resource sensitive behaviour of ports can easily be captured by LL. The automatic closing requires much more machinery. If such an exercise would aid our understanding remains to be seen." [JMH93, end of section 3.2]

8.1.2 Functional Programming

In most concurrent extensions of functional programming languages, processes are modeled by means of functions. Therefore, a process is required to return a value, even if the value is discarded (or simply does not matter) in most of the proposals we are aware of [Rep91, AVWW96, PJGF96, TLK96a, Rep99, LDG+01]. One of the motivations to consider processes just as (a special kind of) a function seems to be that this allows to apply standard functional programming techniques to the description of processes (see for instance, [TLK96b, page 285]). Our process and action expressions together with process functions (see section 3.5.2) allow to use similar techniques in our computation model.

The design of Concurrent Haskell (CH) [PJGF96] was guided by the research for a "minimal" set of primitive operations which would allow to provide concurrent programming in the functional language Haskell [PJHA+99], such that, using the rich set of abstraction features of functional programming, more friendly abstraction can be defined. Thus, the confusion between the notions of functions and processes is one of the design principles of CH. In fact, only the type system allows to distinguish between a (pure) function and a state transformer (or process), *i.e.*, a function the result type of which is necessarily of the form `IO t`[5]. However, the introduction of concurrency renders the interpretation of monadic I/O as abstract descriptions of a state transformers "untenable" [PJGF96, section 2.1] for CH, since the execution of the side-effects denoted by the actions cannot wait until the program has finished (and the description of the actions to be executed has been computed completely). Thus, the operational semantics of Haskell has to be extended.

The operational semantics of CH is stratified in two layers, namely the deterministic reduction of expressions (*i.e.*, the operational semantics of Haskell) and the concurrent reaction modeling the execution of processes, or reduction of functions of type `IO ()`. While this separation in two levels is similar to our operational semantics as presented in chapter 4, there are several differences. First, CH does not provide operators for sequential composition and choice between processes, since these operators are not *primitive* in the sense that they can be simulated using the operators available in CH. We have included these operators, since we did not have the goal of designing a *minimal* extension to a particular language nor a minimal calculus allowing to model any problem, but rather searched to combine the best features of both, declarative programming and process calculi, where these operators are widely used. Second, communication between processes in CH uses `MVar`s (besides the implicit synchronisation on shared expressions due to the lazy evaluation strategy). In the operational semantics, `MVar`s are modeled as a special kind of process. Thus this communication scheme is a particular case of the communication using a store, since the parallel composition of `MVar`s can be considered as a store (containing only atomic formulæ). Third, our operational semantics presents the execution of all actions or state-transformers at the level of the processes, whereas the operational semantics of CH integrates at least[6] the description of the operational behaviour of the operator of sequential composition, *i.e.*,

[5] Since the value returned by a function representing a process is discarded, the type `t` is mostly the empty type `()`.

[6] Due to lack of space, the other primitive actions are not considered in [PJGF96, section 6].

`>>=`, into the operational semantics of the declarative program. Last, but not least, CH does not provide a symmetrical operator for parallel composition of processes (as is usual in most process calculi), but a primitive action which has the side-effect of launching a process[7].

Therefore, the main difference between our computation model and the closely related model of CH is that we provide operators taken from process calculi for the description of processes, instead of encoding processes as a particular kind of functions. Thus we allow the direct use of the appropriate description tools for each concept, without the need of encoding them. For instance, our solution to the example of the Dining Philosophers (see example 1.1.5) needs the atomic test of two guards, which is not directly provided in CH.

CML [Rep91, PR96, Rep99] differs from our model (of a component) in two basic design choices. On the one hand, communication in CML is based on message passing, whereas the our processes share a common store. On the other hand, our programming model is asynchronous, whereas processes in CML synchronise on *events*, a new data type introduced in CML. Furthermore, since the behaviour is specified by functions which might be defined by means of processes the definition of which might use functions, CML clearly does not distinguish between processes and functions.

In contrary to our computation model and to most process calculi, processes in CML are named, *i.e.*, the creation of a process (which is, as in CH, a side-effect of executing a particular built-in function, namely `spawn`), returns the (unique) identifier of the process, similar to the UNIX [RT78] operating system. While this allows the definition of primitives for the control of the execution of processes, as for instance the possibility to *kill* (*i.e.*, stop) a running process, we are not aware of a clear semantics for such a model. Notice that the solution to the problem of the Dining Philosophers given in [Rep99, page 186] as an example for the implementation of the Linda [Gel85] coordination principles in CML needs an additional semaphore to ensure that at most $n-1$ philosophers are seated around the table, since there is no direct support for the atomic test and update of two tuples, as we used in example 1.1.5.

Similar to CML (and in contrary to our computation model), functions and processes can use each other mutually in Facile [TLK96a]. Indeed, the behaviour of processes if defined by means of process scripts, which can be transformed into an expression using the primitive function `script` and the execution of a script is a side-effect of executing the functions `spawn`. As in CML, communication in Facile is based on message passing using synchronous channels, so that additional primitive functions for choosing between different communication events. Another aspect of Facile, namely the notion of a *node*, is closer to our computation model. Roughly speaking a node of Facile corresponds to a component in our model, and in Facile we also find two levels of concurrency, namely between processes (of a same node) and between nodes. However, our current model does not allow to start processes on a remote component nor does it consider the dynamic creation of nodes. On the other hand, the components of a system in our computation model may be written in different languages, whereas nodes in Facile have to be written in Facile.

[7] It is argued in [PJGF96, section 2.2] that a symmetrical fork, as for instance the primitive `symFork` of [JH93]) would have forced the synchronisation on the termination of the forked process, that is to say, the introduction of a general sequential operator.

Processes in Erlang [AVWW96] are defined by means of (untyped) functions, with additional built-in functions implementing asynchronous communication via message passing. Similar to our computation model, the Erlang runtime systems allows the interaction with the system, including the modification of the code executed by the processes, by replacing a module by a new version of the module[8]. When defining a function, the syntax of Erlang allows to distinguish between a call to a function in the same version of the module or in the most recent version of the module[9]. Since Erlang can handle at most two versions of a module (called the "old" and "new" version), the standard runtime system kills all processes that are still executing the "old" version, before the "new" version becomes the new "old" one, and the new module is loaded into the new "new" version. When a new version of a module is loaded into the runtime-system, the signature of the module is allowed to change, such that the complications related to the removal of a symbol mentioned in section 3.2.2.4 apply also to Erlang, where they are solved by killing the corresponding processes.

Eden [BLOMP98] distinguishes between (static) functions and (dynamic) processes [BLOM95, section 1.4], which are modeled as functional transformations from a set of inputs to a set of outputs. As in our computation model, the creation of new channels allows for a varying communication structure. However, the coordination model of Eden [BLOMP97] is based on message passing, and since the communication channels of Eden are restricted to one-to-one connections, an additional built-in process `MERGE` is needed, in contrary to our computation model.

Concurrency in concurrent Clean [NSvEP91, PvE98] is mainly aimed at improving the operational behaviour by means of a parallel execution. For instance, processes are created by means of annotations on the subexpressions in the right hand sides of the rules defining the functions. Furthermore, Clean provides different annotations for processes depending if the new process should work on the graph representing the term to be reduced, or on a copy of it. However, an extension of Clean for the simulation of concurrent interactive processes has been suggested in [AP95a], where processes are modeled as state transforming functions. Similar to CH and CML, the creation of processes is the side-effects of a specific action. Since processes are allowed to share (a part of) their state, the communication scheme is similar to the one of our computation model. However, the sequential execution of the processes is controlled by the reception of messages corresponding to events in the GUI, which restricts the flexibility of the definition of processes.

8.1.3 Functional Logic Programming

Since the declarative language sketched in section 3.1 which is used in our prototype for the description of the stores is a functional logic programming language, our approach is naturally closely related to concurrent functional logic programming languages. Unfortunately, we are aware of only a small number of languages extending the functional logic paradigm with concurrency, which might by due to the fact that the functional

[8]This is not really a restriction, since update at the level of functions can be easily obtained by encapsulating all functions in separate modules.

[9]The former is achieved by explicitly prefixing the module-name to the call of the function; omitting the prefix results in the latter [AVWW96, page 123].

logic paradigm by itself is far less established than the functional or logic paradigm separately. In fact, the only concurrent ones we are aware of are Curry and Escher.

Similar to logic programming, processes in Curry [HAK+00b] are represented by constraints, using a concurrent interpretation of conjunction. As a means for communication, the notion of ports of [JMH93] has been introduced in Curry and been extended to *named ports* which allow for distributed programming [Han99]. Our communication scheme allows to model the behaviour of ports (by appropriate actions) so that we do not need to introduce them into our computation model. Due to the use of monadic I/O, processes in Curry are not allowed to perform I/O actions, *i.e.*, to interact with the external world. This implies that Curry provides two different operators for sequential composition, namely `>>=` (which is used for the composition of action in the `IO` monad) and `&>` (which corresponds to the sequential interpretation of conjunction used for modelling the sequential composition of processes). The example program of the Dining Philosophers which comes with the distribution of PAKCS[10] [HAK+00a] uses an additional semaphore to ensure that only $n-1$ philosophers are seated around the table at the same time such that the system does not deadlock.

The concurrent extension of Escher [Llo] has a similar execution model as our computation model, since processes also communicate by means of a shared memory, called the blackboard. However, the execution of processes (which are modeled, similar to CH, as functions of type `IO ()`) is simulated by means of "the primitive function `ensemble`, that cannot be written directly in Escher" [Llo, section 3, second phrase]. As CH, concurrent Escher uses the monadic operators `>>` and `>>=` to model sequential composition and does not provide an operator for nondeterministic choice. Finally, since concurrent Escher does not allow the encapsulation of a guard and several actions into one single atomic step, the program modeling the Dining Philosophers given in [Llo, figure 5] needs an additional semaphore.

8.1.4 Linear Logic Programming

As already pointed out in section 2.1.4, the proof theoretic behaviour of the logical connectives in linear logic is quite similar to the operational behaviour of operators in process algebra. In most linear logic programming languages, the execution of a processes corresponds, roughly speaking, to the *search* of a proof in linear logic. Therefore, in terms of linear logic programming, the execution of a process aims at a (final) result, namely the proof which is searched. On the other hand, some processes, as for instance those controlling an external system, are designed to never terminate or to return a result. In our opinion, the modeling of these processes in linear logic programming as a nonterminating search for a proof (of the initial formula) is not as intuitive as a process algebraic description. Furthermore, the initial formula describes the system completely, in the sense that the process described does not interact with an environment external to the formula. Thus the model of a system in linear logic programming is *closed*, whereas our computation model is *open* in the sense that a component may receive (via the mailbox) at any moment any executable action from the outside of the system. Nevertheless, in the light of the similarities of the rules, the investigation of a

[10] PAKCS is the acronym for the Portland Aachen Kiel Curry System which is available for download at the URL `http://www.informatik.uni-kiel.de/~pakcs`.

description of the semantics of our computation model in terms of linear logic might be worth further investigation, but this is beyond the scope of this thesis.

8.2 Concurrent Programming

The description of processes in our computation model is based on process calculi, and by means of appropriate actions we can model most[11] of the communication mechanisms of these calculi. However, our computation model distinguishes clearly between the notions of processes and those underlying declarative languages, such as functions and predicates. This distinction does not exists in process calculi, and functions or predicates have to *encoded*. We consider therefore our computation model to be more convenient for programming, since these different notions can be expressed directly using an appropriate formalism. In this section we compare our computation model to some programming languages based on process calculi. Following the calculi they are based on, most of these languages require the *encoding* of the notions of functions and predicates by means of processes.

The combination of algebraic specification with a process calculus similar to CCS [Mil80] and CSP [Hoa87] provided by LOTOS [LOT00, ELO01] distinguishes between functions and processes. However, in contrary to our proposal, the definitions of the functions cannot be changed. In fact the store is mainly used to specify the types of the messages. The communication mechanism of LOTOS by synchronisation on ports has to be simulated in our computation model. On the other hand, a broadcast is natural in our model, but rather difficult to obtain in LOTOS.

The parallel and functional programming language FP2 [Jor84, Jor85] is similar to LOTOS. In both languages, the functional part is used for the specification of the data types used in the communication between processes. Similar to our computation model, the behaviour of processes is specified by transition rules, which in our model modify the state. In contrary to our model however, the state is local to a process and consist only of a closed atoms instead of descriptions of theories. The process forms of FP2 are similar to our process expressions. However, the set of operators for the combination of processes of FP2 differs from ours. While FP2 provides three different kinds of parallel composition (corresponding to the different modes of interaction between the composed processes), sequential composition has to be encoded in FP2. As for LOTOS, the main differences to our computation model is the communication scheme and the fact that the definitions of the functions cannot be modified in FP2, in contrary to our stores.

Our action `new` (which allows the dynamic creation of channels) together with the parameterised sort `Name` (which allows channel names to be passed) allows to model mobility in the same way as the (asynchronous) π-calculus, *i.e.*, by passing communication links. Since the encoding of functions by means of processes in the π-calculus is rather complicated and not very intuitive, an integration of the λ-calculus and the π-calculus has been proposed [Bou97], by combining the operational behaviour of functions and processes, whereas in our model, functions and processes are clearly distinguished no-

[11] An example of a calculus we cannot model, is the full synchronous π-calculus, which offers an external choice between send and receive on the same channel. However, we are not aware of any implementation of such a powerful calculus.

tions. Furthermore, our model is a conservative extension of declarative programming, such that interactive goal solving with respect to the current store is possible.

The programming language Pict [PT97, PT98] is based on the asynchronous π-calculus. Contrary to Pict (and the π-calculus), our guards allow the atomic reception on several channels, whereas in the π-calculus processes can only wait on a single channel. Extensions of the (asynchronous) π-calculus without this restriction are the join-calculus [FG96] and $\mathcal{L}_\pi$ [CM98].

jocaml [FFMS01] is a language based on the join-calculus where processes can be seen as communicating via a multiset of messages: sending a message corresponds to place it in the multiset, and the "joint reception" of several messages is blocking and removes the received messages from the multiset. Thus broadcast is not provided directly and has to be encoded as in any language based on the π-calculus. Since jocaml is implemented on top of `ocaml` [LDG$^+$01], jocaml-programs can use the facilities of `ocaml` for the definition of data structures and functions[12]. However, we are not aware of a complete description of the theoretical foundations of this integration.

In $\mathcal{L}_\pi$, [CM98], processes communicate as in the join-calculus via a multiset, but additionally may have guards, *i.e.*, a $\mathcal{L}_\pi$-process that, when executed in an encapsulated environment, has to terminate successfully. Using these guards, the solution to the problem of the Dining Philosophers given in [Cai99, secion 2.1, pages 23 – 24][13] allows a philosophers to take two sticks in a single atomic action, similar to our solution (see example 1.1.5). However, $\mathcal{L}_\pi$ does not distinguish between predicates and processes, and like all the other dialects of the π-calculus mentioned above does not support interactive goal solving.

Synchronous programming languages, as for instance Esterel [BG92] or Lustre [HCRP89], model reactive systems as functions that periodically associate to a (finite) set of input signals a (finite) set of output signals, where the computation of the output signals takes no time (or is at least completed before the next input signals arrive). [Bon95] suggests the specification of these function by means of the process algebra CoReA. Basic processes of CoReA are guarded emissions of (output) signals, where the guards check on the presence (or absence) of input signals. Since the sequential ordering is imposed by the synchronous model, CoReA provides only two combinators for processes, namely parallel composition and a choice with priority (similar to our operator $\oplus$).

8.3 Coordination

Most of the coordination languages presented in section 2.3 are targeted to the coordination of imperative programs, that is to say, they provide a set of predefined coordination primitives which have to be incorporated as basic procedures or actions in imperative programming languages. On the other hand, those coordination models

[12] "To explore the expressive power of message-passing in jocaml, we now consider the encoding of some data structures. In practice however, one would use the state-of-the-art built-in data structures inherited from `ocaml`, rather than their jocaml internal encodings." [FFMS01, introduction of chapter 1.6]

[13] Notice that the specification in [Cai99, page 24] does not contain a generic description of a philosopher process.

designed for combination with declarative programming languages (see section 2.3.2) consider mostly a store restricted to contain only atoms (or atomic formulæ). In all cases, the programs to be coordinated are required to share the common language used for the description of the tuples or atoms in the shared data space.

Similar to our computation model, a system in KLAIM [NFP98] is composed of several components (called "nets" in KLAIM) which themselves are structured by means of concurrent, parameterised processes. Therefore the actions executed by processes in KLAIM are located, *i.e.*, paired with the location (of the net) where they are to be executed. This is similar to our pairs of storenames and elementary actions. However, the stores of KLAIM are not declarative programs, but multisets of tuples which are accessed or selected (as in Linda [Gel85], see section 2.3.1) by means of pattern matching. A further similarity is the separation of the operational semantics in two levels, corresponding to the locations (components) and the nets (system). However, while KLAIM provides a global transition system describing the semantics of a system, we prefer to view a system as a parallel composition of several transition system, in order to reflect that we cannot know the states of all components at the same moment. Furthermore, KLAIM does not distinguish between tuples and processes: on the one hand, KLAIM supports the notion of active tuples, *i.e.*, tuples representing processes[14], and on the other hand, the operational semantics of KLAIM presented in [NFP98] models tuples as processes. This way of modelling messages as processes can also be found in the asynchronous π-calculus (see section 2.2.2) or the join-calculus calculus (see section 2.2.2) where sending a message corresponds to spawn a parallel process which synchronises on the reception of the message and terminates. Finally, we have voluntary restricted ourselves in this thesis to the precise description of components and their interactions, and model system as a static set of components. KLAIM does not have this restriction and allows the dynamic creation of locations (*i.e.*, components) by means of particular builtin actions. In our opinion, the creation of components should be distinguished from actions, since the former modify the *set* of stores in the system, whereas the latter modify stores, *i.e.*, *members* of the before mentioned set.

The operational semantics of **MANIFOLD** [BAdB$^+$00] is defined in two levels, similar to the one of our computation model as presented in chapter 4. The first level defines the behaviour of processes and the second level defines the interaction between the different transition systems. Whereas in our approach the behaviour of processes and components is explicitly defined, the atomic (computation) processes in **MANIFOLD** are considered as black boxes such that only a specific set of events can be observed, namely silent steps (τ), raising (*raise*) and reception (*receive*) of events, input (*get*) and output (*put*) on streams and termination (*halt*).

In contrary to $\mathcal{ACLT}$ and Tuple Centres [ODN95, DNO97, DO99], our model provides a clear definition of processes. The benefits of separating the specification of the communication medium from the description of the communicating agents is illustrated by the example of the Dining Philosophers [DNO97, section 3]. In a first solution, the definition of reaction rules allows to encapsulate the removal of both sticks inside a single atomically executed action. Consider now the following modification of the problem.

[14]Notice that this leads to two different ways of introducing a concurrent process: Either by putting an active tuple into the tuple space by means of the tuple space operation **eval** or by the operator $\|$ of parallel composition of processes.

Each philosopher disposes now of a set of different sticks, one for each meal during the day, *e.g.*, a stick for each, breakfast, lunch and diner. In $\mathcal{ACLT}$, the program can be adapted correspondingly, by changing only the reaction rules and the representation of the sticks (or forks) in the tuple space and thus by keeping the same descriptions for the philosophers. Notice that the same holds in our framework, where we would have to change the definition of the elementary actions and the representation of the situation in the store.

Similarly to our model, definitions of processes and definitions of predicates are distinguished in *μLog* [JdB94, dBJ96], since the clauses defining processes are allowed to contain blackboard primitives, whereas the clauses defining predicates are not. Nevertheless, the execution of processes is seen as the solving of the initial goal in *μLog*, which is also witnessed by the definition of a sound and complete declarative semantics for (terminating) goals [dBJ96, section 4]. The distinction between fore- and background processes in *μLog* has no equivalent in our computation model, since we do not consider processes which return a result. Thus all our processes can be considered as background processes in the sense of *μLog*, and predicates (or functions of result-sort `Truth`) correspond to the foreground processes of *μLog*. Since the blackboard primitives of *μLog* modify only the blackboard (or logical tuple space), the *clauses* defining predicates (or conditions) in *μLog* cannot be modified, in contrary to our elementary actions which are allowed to modify the rules defining functions. Notice also that *μLog* is not a conservative extension of logic programming since the predicates in the initial goal are not allowed to share variables[15]. Finally, our implementation of the store is similar to the implementation of blackboards in *μLog*, in the sense that our stores are a *passive* data-structure, and not a particular process that manages the store[16].

The behaviour of coordinators in CLF [AFP96, APPPr98] is defined by means of scripts, *i.e.*, collections of rules in a subset of LinLog [And92]. Thus the handling of the resources involved in the interaction between components is expressed implicitly in the rules (the head h of a rule $h \diamond\!\!- b$ is removed and the body b is added). Similar to our guarded actions, the execution of a rule in CLF is atomic, allowing to model the example of the Dining Philosophers in a similar way as in section 1.1.5 [APPPr98, section 2.5, pages 192 – 195]. Furthermore the heads of CLF-rules are similar to our guards: both correspond to a formula which has to be valid (at the current instant). In contrary to our model, the only action available in a CLF rule is the addition of resources (or formulæ) to the (stores of the) components. Another difference is that the head of a CLF-rule may contain predicates defined in different components, so that CLF needs an elaborate protocol in order to ensure the atomic execution of rules. Finally, as our model, CLF considers the interaction between components written in different languages. However, as in CORBA [COR01] or MLP [HS87], all components and languages are required to support the common communication primitives of CLF. For this purpose, Mekano[17] [AAP+99], a library of standard CLF components has been developed for the programming languages Python and JAVA.

[15]This restriction ensures that the communication between processes uses only the blackboard and not streams encoded using logical variables.

[16]However, the interactive interpreter for the store is a particular process.

[17]Mekano is an abbreviation of "Multi-platform Environment for Knowledge Applications in Networked Organisations".

8.4 Specifications

Both, programming and specification languages allow the description of a system by means of a formal language. However, while programming languages aim at an *executable* program, specification languages and methods focus rather on an abstract, high-level description which is not necessarily executable.

The modifiers of the extension of algebraic specifications with implicit state (AS-IS) presented in [DG94, Kho96] are similar to our (elementary) actions. However, they cannot be defined by the programmer (as our actions), but are predefined and only applicable to their corresponding function. Furthermore, AS-IS focuses on the description of the modification of the state (*i.e.*, the actions) and needs to be extended in order to allow the convenient description of processes which execute these changes and allow to structure the description of the system.

Similar to our approach, d-oids [AZ92, AZ95, Zuc99] model the semantics of a system by a transition system. While we use stores, *i.e.*, declarative programs, as the states (of the transition system), d-oids describe modifications of (instant) algebras which, informally, correspond to structures classically used in algebraic specifications. A further similarity to our computation model is that the signatures of d-oids are dynamic, allowing the creation of new functions, similar to our elementary action new. As for AS-IS, we are not aware of a concurrent extension of d-oids. In addition, we are not aware of a tool or language allowing to develop programs based on d-oids.

The states of Abstract State Machines (ASMs), see for instance [Gur97], are *algebras* which can be considered as *models* of the theory descriptions or stores in our computation model. These algebras are modified by means of elementary updates which are similar to our actions, in the sense that the execution of an update (or an action) results in the change of the function definitions of the current state. However, similar to AS-IS, the basic model of ASMs does not provide process algebraic combinators for the description of several concurrent processes. In the framework of communicating evolving algebras [GR93b], processes communicate as in our computation model by means of a shared store or state. Interaction between different units in IASMs [dAMdIdSB98] uses message passing, similar to the sending of actions between components in our model. IASMs distinguish three kinds of dynamic functions[18], namely internal, external and shared functions. Internal functions can only be modified by updates of the unit itself, external functions only by the environment and shared functions by both. The distinction between external and shared functions is not necessary in our model, since we do not need to encode the environment by means of partially specified external functions. Finally, we are not aware of an implementation of IASMs.

The combination of AS-IS with ASMs presented in [GKZ99] as well as DADT's [EO94] share with our proposal the structuring of the description of a system (or rather a single component) description in several levels, where the lower levels correspond to the description of a static structure, and the higher levels define the modifications of these structures. However, in contrary to the combination of AS-IS with ASMs and DADT's, we suggest to use process algebras for the description of the dynamic part of a system, taking advantage of the theoretical studies of concurrency and mobility developed in the context of process algebras.

[18] These functions are called *dynamic* in contrast to the *static* functions which cannot be modified.

Similar to our model, algebraic state machines [BW00] distinguish between the static and dynamic parts of a system, in order to enhance the readability of specifications. Thus, algebraic state machines also use different layers in the description of a system. Indeed, the states of an algebraic state machine are algebraic specifications, and the transitions between states are expressed by particular transition rules or axioms. Compared to our computation model, algebraic state machines are closer to *specifications*, *i.e.*, they are not necessarily executable. This is witnessed by the fact that the transition rules are considered as specifications of a logical relation between states, and arbitrary logical formulæ are allowed in the pre- and post-conditions of the transition rules. Furthermore, algebraic state machines do not use process algebra combinators for their composition, but a single composition operator which is based on the data flow using the input and output channels, considering single algebraic state machines as black boxes. Thus, notions of processes (or algebraic state machines) and components are not as clearly distinguished as in our model. Finally, the use of process algebraic operators allows us to express the dynamic creation of processes, since we allow arbitrary process terms in the rules of process definitions, whereas the transitions of algebraic state machines cannot change the number of processes.

8.5 Multiparadigm Programming

Our approach shares most of its motivations with existing proposals for multiparadigm programming, namely the accommodation of a multitude of programming styles inside a single computation model, such that for every part of a system the most appropriate paradigm can be used. However, most of the approaches to multiparadigm programming presented in section 2.5 lack a theoretical model or semantics for the complete approach. Thus these approaches [Bud95, CLSM96, Con88, Cop98, NL95, Pla91, Spi94, Zav89, Zav91, ZJ96] do not benefit from the advantages associated with classical declarative programming, which are related to the sound theoretical foundations of these paradigms. The multiparadigm languages mentioned in section 2.5 where we are aware of a formally defined semantics are Alma-0 [ABPS98], $\mathcal{OLI}$ [LP96, LP97], LIFE [AKP93] and Maude [CDE+99].

The combination of logic and imperative programming represented in Alma-0 is based on the *sequential* interpretation of conjunction. Thus, in contrary to our computation model, procedures (or processes) and predicates are not distinguished. The examples of programs given in [ABPS98] illustrate the use of Alma-0 for the expression of algorithms involving the necessity for the exploration of "search tree" allowing both, a fine control on the sequential order of the exploration and an abstract description of the search. However, we are not aware of a concurrent extension of Alma-0.

Similar to our motivations, $\mathcal{OLI}$ [LP96, LP97] aims at a clear distinction between the different notions integrated in the framework. An advantage of this separation is that, on the one hand, the resulting language is a conservative extension of the combined languages, and, on the other hand, the integration does not try to encode a concept by another but rather provides both. A further similarity between $\mathcal{OLI}$ and our model is that modifications of the theory are defined in a part of the program separated from the predicates defining the theory. Thus the objects in $\mathcal{OLI}$ play a

similar rôle as our processes, albeit we are not aware of a concurrent implementation of $\mathcal{OLI}$. Finally, while a (sound and complete) formal semantics for the logic part of $\mathcal{OLI}$ is described in [LP96, LP97] (considering objects as constants, using the notion of an *enriched Herbrand universe*), the operational semantics of the object oriented part is based on an informal description.

The foundation of LIFE [AKP93] on ψ-terms allows to conveniently represent objects, functions and predicates in a uniform framework. However, we are not aware of a concurrent extension of LIFE.

The use of rewriting logic as the foundation of the programming language Maude [CDE+99] allows the unification of different programming paradigms in Maude, by viewing the different programming styles as particular instances of a rewriting logic [Mes92]. Since rewriting logic is *reflective*, Maude allows the definition of a special module, called `META-LEVEL` [CDE+98, CDE+99], which allows to *reify* programs, *i.e.*, to represent programs (or modules) as terms in Maude itself. The module `META-LEVEL` introduces the notion of a *reflective tower* [CDE+99, page 35], since the terms representing modules can also be reified. Notice that the reification of a meta-representation of a module, *i.e.*, a term of the module `META-LEVEL`, yields a term of the module `META-LEVEL`. Consequently, reification of meta-terms does not change the sort, in the sense that a reified meta-term is also a meta-term. In fact, most of the time[19], only the two lowest levels of the reflective tower are used. In our model, the use of the third level would allow the modification of the definitions of actions, which we did not need in the examples we have considered so far. Thus, while in Maude the notions of functions and processes are represented in the same way, namely by operators defined by (conditional) rewrite rules, they can be distinguished since they are on different levels of the reflective tower. Finally, we are not aware of a theoretical model for *interaction* of a Maude program with its environment (which is not part of the program). In fact, the only input/output primitives provided by the standard library are in the `LOOP-MODULE` [CDE+99, section 2.8] which allows the definition of interactive read-eval-print loops.

[Spi94] distinguishes different approaches to multiparadigm programming, namely new languages, languages extensions, theoretical approaches and multiparadigm frameworks. Theoretical approaches, which achieve the unification of paradigms by a unification of the theoretical bases of the paradigms, are criticised as being both, limited to only a few paradigms as well as being impractical since too difficult to implement [Spi94, section 2.4.3][20]. In our opinion, this critic misses the point, since there is no reason not to attempt the combination of particular paradigms, just because the development of a sound theoretical foundation seemsdifficult.

Classical imperative programming languages, as for instance C [KR88], ADA [Ada95] or JAVA [GJSB00], lack most of the nice features of declarative programming languages. First, these languages do not have a well-defined formal semantics which allows the formal reasoning about properties of programs[21]. Second, the high level of abstraction

[19] We are aware of only one example where more than two levels are needed [CDE+98, section 4.4], namely the implementation of theorem provers using modifiable strategies for theory transformations.

[20] On the other hand, the lack of an unifying model, not to speak of an underlying theory, for their own approach of is acknowledged [Spi94, sections 2.4.5 and 7.1.2].

[21] Using ASMs (see section 2.4.2), the semantics of C [GH93] and JAVA [SSB01] has been defined. However, since these semantics have been defined after the language, they are, in our opinion, rather

available in declarative languages allows the programmer to focus on the *declaration* of the problem, rather than explicitly describing a solution step by step. Notice further, that our computation model combines the advantages of declarative languages in the description of static theories, such as data types, functions or predicates, with imperative programming for the description of concurrent interactive processes.

The use of reflection in JAVA [GJSB00, McC98] is restricted to the *inspection* of the properties of the code which has been downloaded, in order to check which method can be called, *etc.*. This kind of reflection is needed in the component framework of JAVABEANS [Eng97]. In this framework, components are exchanged in *binary*, *i.e.*, compiled, form. Thus, for the integration of these components into a running program, as for instance a development environment, the program needs to access information about the *interface* of the included component. This is to be distinguished from the reflection in Maude [CDE+99], which allows to model the modification of stores as required by the definition of our actions.

——— ⋆ ——— ⋆ ——— ⋆ ———

In this chapter we have compared our computation model to the related work presented in chapter 2. This comparison shows that our model combines ideas and motivations from a wide range of fields. However, the comparison also points out the distinguishing features of our approach, namely a clear separation of the different programming styles in a conservative manner, while keeping a formally defined semantics.

Note finally, that our language for the definition of actions (see section 3.2) should not be confused with action semantics [Mos92] which is a method for defining the *semantics* of a given programming language, whereas we want to define the modification of programs.

complicated (since the language has not been designed with the goal of a simpler semantics in mind).

Chapter 9

Conclusion and Perspectives

Aiming at an appropriate tool for executable, formal descriptions and specifications of complex systems, we have presented in this thesis a first step towards a computation model which provides a component based approach for constructing systems by combining declarative (*i.e.*, functional, logic and functional-logic) programming with concurrency, expressed in form of mobile processes. Roughly speaking, we suggest to model a system as a collection of components where each component consists of a store, *i.e.*, a declarative program, and a set of processes modifying the store of the components in the system by the execution of actions.

A distinctive feature of the proposed model is that the different notions, as for instance functions, predicates, processes, actions, *etc.* are clearly distinguished from each other. As pointed out in the comparison in chapter 8, this is the main difference with respect to most of the other models we are aware of. Indeed, most of these models *encode* one concept or notion by another. We claim that less encoding is a gain in readability and maintainability, which should result in both, augmented programmer productivity and less errors or bugs. As further advantage of distinguishing the different concepts, we have that our approach is generic in the sense that it can be applied to extend almost any declarative language with concurrent processes, since it does not depend on a particular encoding of the notion of processes by means of the concepts provided by the declarative language.

Further noteworthy features of our computation model are the user-definable actions (see section 3.2) and the specification of translations in the interaction between components (see section 3.4.2). Both allow a programmer to modularise the description of the system by separating specific parts. Notice also, that we give a (precise) definition of a component, in contrary to most component based approaches where a component is characterised indirectly or solely by its interface.

The definition of a formal computation model is motivated by the possibility of defining formal analyses. Therefore, we have defined in chapter 5 a compositional semantics for processes of a component, which allows to analyse a process by analysing its parts separately. In chapter 6 we have presented an example of an analysis, namely the analysis of secrecy for the processes of a component from a point of view of non-interference. This analysis is based on an abstract execution of a process, and ensures that no secret information can flow to places where it can be accessed publicly.

Last, but not least, the reader may keep in mind that a complete description of an

inherently complex system cannot be expressed in a simple way, since otherwise the system would not be complicated. Our computation model may seem complicated, due to the many different notions and definitions presented in chapter 3, but it is designed to allow to describe, at an appropriate level of abstraction, systems which have to interact with their complex environment.

The computation model presented in this thesis is, as already mentioned, only a first step towards a long-term research goal. Thus it provides plenty of possibilities for future research and improvements. We conclude this thesis by mentioning some of them.

A first line of research concerns the extension and deeper exploration of the theoretical model underlying the framework, as for instance, the development of a process algebra for our processes or the investigation of a denotational semantics. The former has already been attempted in a setting with a fixed set of predefined actions [Ser98]. For the latter, a semantics based on Kripke-structures where the worlds would correspond to the states of the transition system defining the operational semantics of a component seems to be a promising approach. Another possibility would be to follow ideas of linear logic programming languages and investigate the relationships of our operational semantics with the proof theoretical behaviour of the operators in linear logic. With respect to the specification of actions and translations, a closer investigation of dedicated formalism or languages is necessary, in order to help a programmer to analyse and ensure the properties required in chapter 3, as for instance totality, termination, confluence, *etc.* Concerning the operational semantics of our components, rule (P) provides a multitude of possible improvements. In our opinion, a significant improvement of rule (P) cannot be achieved in a general model but is necessarily tailored to a particular declarative language and its operational semantics. Indeed, the improvement of rule (P) involves the manipulation of the search tree and strategy used by the operational semantics of the declarative language.

There are also a number of interesting extensions of the framework, since we have chosen to exclude some concepts in this first attempt to define a computation model. For instance, the current model does not take into account temporal notions at all. However, these notions are necessary for modeling control systems, *e.g.*, when an emergency is to be signaled to the operators if some events do not occur inside a particular time interval. An integration of temporal properties with declarative programming has been suggested in [BE01], using a *synchronous* programming model, where all processes act at the same time, and actions have no duration. It seems interesting to investigate the integration of the temporised programs of [BE01] as stores in our computation model. Besides enriching the computation model with temporal notions, this integration involves the combination of the synchronous and asynchronous computation model.

Another concept we did not consider in this thesis, is the notion of *objects* which is used to structure and organise programs in object-oriented programming. One of the main advantages of the object oriented programming paradigm is the ease of reusing existing programs, most prominently by the use of inheritance and dynamic linking.

Mobility in our current model is restricted to the passing of communication links as in the π-calculus. An interesting extension concerns the modeling of mobile processes

which actually change the component where they are executing. This extension requires to locate processes, *i.e.*, to associate to processes the component where it executes. Furthermore, the communication between components has to be extended, since a component has to be able to receive not only actions, but entire processes. Notice that we therefore need to consider different forms of parallel composition, since the parallel composition of the launch of two processes on a remote component behaves not necessarily the same way as the launch of the parallel composition of the two processes on the remote component. Similarly, our model does not provide higher-order processes, *i.e.*, processes which take processes as arguments. Notice, that our action and process expressions (as well as all the other rewrite systems) could be easily extended to deal with higher-order functions. However, the relationship of this extension to higher-order process calculi needs further investigation.

Last, but not least, the relationship of our framework with reflection needs further explorations. We have tried to stratify our computation model, that is to say, to ensure that modifications of one layer can only be made at a superior layer. For instance, processes are allowed to modify the functions of the store, since processes are running on a meta-level with respect to the store. On the other hand, processes are not allowed to modify the definitions of process functions. Reflection allows a unified view of these different layers, facilitating the exchanges between layers while keeping them distinct. This is particularly interesting for the arguments of the actions which we require to be on the meta-level with respect to the store (by using implicit reifications when necessary). A better understanding of reflection would allow more flexible control about these transformations.

Obviously, as any program (and in particular as any prototype), the current implementation of the framework could be improved in several respects. For instance, the implementation of the declarative language could be greatly improved by including dedicated constraint solvers, higher-order functions, a polymorphic type-checker, *etc.* But it should also be possible and interesting to include existing declarative languages for the description of stores. Another area of improvement is the implementation of the operational semantics. The current implementation tests the guards of the processes sequentially and locks the complete store. The use of a finer granularity for the locks should come with at least two advantages. First, there should be more processes that could be executed in parallel, such that on a multi-processor system the execution would become faster. Second, since we have a finer locking mechanism, the implementation can control better which of the waiting processes to wake up. Currently, all processes reevaluate their guards, but if we know that a different part of store has been modified such that the validity of the guard has not been changed, there is no need to recheck the guard. This also should increase the efficiency of the implementation.

Besides the improvement of the implementation itself, the development of a surrounding collection of tools is mandatory if we are to tackle realistic problems of a significant size. A first set of such tools which comes to mind are an environment for program development, debuggers and libraries. However, these tools could include different kinds of program analyses. For instance, we would need analysers to check the required conditions on the definitions of actions and process functions (*e.g.*, confluence, termination, *etc.*). Further analyses could verify the properties of a component or even a complete system, such as safety, liveness or non-interference. These analyses could

use abstract interpretation or a more elaborate type system than the one currently used.

Bibliography

[AAP+99] Jean-Marc Andreoli, Damián Arregui, François Pacull, Michel Rivière, Jean-Yves Vion-Dury and Jutta Willamowski. CLF/Mekano: A framework for building virtual-enterprise applications. In *Proceedings of EDOC '99*, Mannheim, September 1999.

[AB99] Krzysztof R. Apt and Marc Bezem. Formulas as programs. In K. R. Apt, V. Marek, M. Truszczynski and D. S. Warren, editors, *The Logic Programming Paradigm: A 25 Years Perspective*, Artificial Intelligence Series, pages 75 – 107. Springer-Verlag, 1999.

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In Martín Abadi and Takayasu Ito, editors, *Proceedings of the* 3^{rd} *Internation Symposium on Theoretical Aspects of Computer Software (TACS 1997)*, volume 1281 of *Lecture Notes in Computer Science*, pages 611 – 638, Sendai, September 1997. Springer Verlag. Open lecture.

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the* 26^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 147 – 160, San Antonio, January 1999.

[ABPS98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington and Andrea Schaerf. Alma - 0: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014 – 1066, September 1998.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1 – 2):3 – 57, April 1993.

[Abr96a] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] Jean-Raymond Abrial. *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*, chapter Steam-Boiler Control Specification Problem, pages 500 – 510. Springer Verlag, 1996.

[ACG86] Sudhir Ahuja, Nicholas Carriero and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26 – 34, August 1986.

[Ada95] ISO. *Ada 95 Reference Manual (Language and Standard Libraries)*, 1995. ISO/IEC 8652:1995.

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher Order and Symbolic Computation*, 11(1):7 – 105, August 1998.

[AEH94] Sergio Antoy, Rachid Echahed and Michael Hanus. A needed narrowing strategy. In *Proceedings of the* 21^{st} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '94)*, pages 268 – 279, Portland, 1994.

[AEH97] Sergio Antoy, Rachid Echahed and Michael Hanus. Parallel evaluation strategies for functional logic languages. In *Proceedings of the International Conference on Logic Programming (ICLP '97)*, pages 138 – 152, Portland, 1997. The MIT Press.

[AEH00] Sergio Antoy, Rachid Echahed and Michael Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776 – 822, July 2000.

[AFP96] Jean-Marc Andreoli, Steve Freeman and Remo Pareschi. The coordination language facility: Coordination of distributed objects. *Theory and Practice of Object Systems*, 2(2):77 – 94, 1996.

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press series in Artificial Intelligence. The MIT Press, 1986. fifth printing, 1990.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 1991.

[AKKB99] Egidio Astesiano, Hans-Jörg Kreowski and Bernd Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer Verlag, 1999.

[AKP93] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3):195 – 234, July/August 1993.

[AMST92] Gul Agha, Ian A. Mason, Scott Smith and Carolyn Talcott. Towards a theory of actor computation. In Rance Cleaveland, editor, *Proceedings to the* 3^{rd} *International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 565 – 579. Springer Verlag, 1992.

[AMST97] Gul Agha, Ian A. Mason, Scott Smith and Carolyn Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1 – 72, 1997.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297 – 347, June 1992.

[And01] Jean-Marc Andreoli. *Paradigmes de programmation et fondements logiques.* Habilitation á diriger des recherches, Universitè Joseph Fourier, Grenoble, June 2001.

[Ant92] Sergio Antoy. Definitional trees. In Hélène Kirchner and Giorgio Levi, editors, *Proceedings of the* 3^{rd} *International Conference on Algebraic and Logic Programming (ALP 1992)*, volume 632 of *Lecture Notes in Computer Science*, pages 143 – 157, Volterra, September 1992. Springer Verlag.

[AP91] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3 – 4):445 – 473, 1991. selected papers from the 7^{th} International Conference on Logic Programming, 1990.

[AP92] Jean-Marc Andreoli and Remo Pareschi. Linear Objects: a logic framework for open system programming. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR '92)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 448 – 450, SAt. Petersburg, July 1992. Springer Verlag.

[AP95a] Peter M. Achten and M. J. Plasmeijer. Concurrent interactive processes in a pure functional language. In Vliet van Utrecht, editor, *Proceedings of Computing Science in the Netherlands*, pages 10 – 21, Stichting Mathematisch Centrum, 1995.

[AP95b] Peter M. Achten and M. J. Plasmeijer. The ins and outs of concurrent clean i/o. *Journal of Functional Programming*, 5(1):81 – 110, 1995.

[APPPr98] Jean-Marc Andreoli, François Pacull, Daniele Pagani and Remo Pareschi. Multiparty negotiation for dynamic distributed object services. *Science of Computer Programming*, 31(2 – 3):179 – 203, July 1998.

[Apt00] Krzysztof R. Apt. A denotatinal semantics for first-order logic. In John Lloyd, Veronika Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv and Peter J. Stuckey, editors, *Proceedings of the* 1^{st} *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 53 – 69, London, July 2000. Springer Verlag. invited talk.

[Arm96] Joe Armstrong. Erlang – a survey of the language and its industrial applications. In *Proceedings of the* 9^{th} *Exhibitions and Symposium on Industrial Applications of Prolog (INAP '96)*, Hino, Tokyo, October 1996.

[Art01] Thomas Arts. Functional programming and logic decrease the use of the most important part of your system. In Michael Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 5 – 16, Kiel, September 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.

[AS97] Krzysztof R. Apt and Andrea Schaerf. Search and imperative programming. In *Proceedings of the* 24^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 67 – 79, 1997.

[ASRA97] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A semantic framework for functional logic programming with algebraic polymorphic types. In Michel Bidoit and Max Dauchet, editors, *Proceedings of the* 7^{th} *International Joint Conference on Theory and Practice of Software Development (TAPSOFT '97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 453 – 464, Lille, April 1997. Springer Verlag. (extended abstract).

[AV90] Joe Armstrong and Robert Virding. Erlang – an experimental telephony programming language. In *Proceedings of the* 13^{th} *International Switching Symposium*, Stockholm, May/June 1990.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikstrom and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.

[AZ92] Egidio Astesiano and Elena Zucca. A semantic model for dynamic systems. In U. W. Lipeck and B. Thalheim, editors, *Workshop on Modelling Database Dynamics*, Workshops in Computing, pages 63 – 83, Volkse, 1992. Springer Verlag.

[AZ95] Egidio Astesiano and Elena Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5(2):257 – 282, June 1995.

[BAdB+00] Marcello M. Bonsangue, Farhad Arbab, J. W. de Bakker, Jan J. M. M. Rutten, A. Secutella and Gianluigi Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. *Theoretical Computer Science*, 240(1):3 – 47, June 2000.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North Holland Publishing Company, 1984. revised edition.

[Bar92] Geoff Barrett. *occam 3 reference manual*, March 1992. draft.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the* 17^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '90)*, pages 81 – 94, San Francisco, January 1990. ACM.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217 – 248, 1992. selected papers of the 2^{nd} Workshop on Concurrency and Compositionality, San Miniato, March 1990.

[BB01] Hubert Baumeister and Didier Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, Formal Approaches to Computing and Information Technology (FACIT). Springer Verlag, 2001.

[BC91] Antonio Brogi and Paolo Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99 – 123, January 1991.

[BC01a] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In Fernando Orejas, Paul G. Spirakis and Jan van Leeuwen, editors, *Proceedings of the* 28^{th} *International Colloquium on Automata, Languages and Programming (ICALP 2001)*, volume 2076 of *Lecture Notes in Computer Science*, pages 382 – 395, Heraklion, July 2001. Springer Verlag.

[BC01b] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. Technical Report 4254, Institut National de Recherche en Informatique et en Automatique (INRIA), September 2001. Accepted for publication in Theoretical Computer Science.

[BdBP97] Eike Best, Frank de Boer and Catuscia Palamidessi. Partial order and SOS semantics for linear constraint programs. In D. Garlan and D. Le Métayer, editors, *Proceedings of the* 2^{nd} *International Conference on Coordination: Languages and Models (Coordination '97)*, volume 1282 of *Lecture Notes in Computer Science*, pages 256 – 273, Berlin, September 1997. Springer Verlag.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal and Clemens A. Szyperski. What characterizes a (software) component? *Software: Concepts and Tools*, 19(1):49 – 56, June 1998.

[BE01] Jérémie Blanc and Rachid Echahed. Adding time to functional logic programs. In Michael Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 31 – 44, Kiel, September 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.

[BES98] Jérémie Blanc, Rachid Echahed and Wendelin Serwe. Towards reactive functional logic programming languages. In Herbert Kuchen, editor, *Proceedings of the* 7^{th} *International Workshop on Functional and Logic Programming (WFLP '98)*, Bad Honnef, April 1998. Institut für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster.

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[BGZ97] Nadia Busi, Roberto Gorrieri and Gianluigi Zavattaro. On the Turing equivalence of Linda coordination primitives. *Electronic Notes in Theoretical Computer Science*, 7, 1997.

[BGZ98] Nadia Busi, Roberto Gorrieri and Gianluigi Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167 – 199, February 1998.

[BHR84] Stephen D. Brookes, Charles Antony Richard Hoare and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560 – 599, July 1984.

[BJ98] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of linda-like concurrent languages. *Electronic Notes in Theoretical Computer Science*, 16(2), 1998.

[BJ99] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination models. In Paolo Ciancarini and Alexander L. Wolf, editors, *Procedings of the 3^{rd} International Conference on Coordination Languages and Models (Coordination '99)*, volume 1594 of *Lecture Notes in Computer Science*, pages 134 – 149, Amsterdam, April 1999. Springer Verlag.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1 – 3):109 – 137, January/February/March 1984.

[BK86] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Science*, 32(3):323 – 362, June 1986.

[BK94] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205 – 265, October 1994. Special issue on the Workshop on Formal Methods in Databases and Software Engineering.

[BK96] Anthony J. Bonner and Michael Kifer. Concurrency and communication in transaction logic. In Michael J. Maher, editor, *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming*, pages 142 – 156, Bonn, September 1996. The MIT Press.

[BK98a] Anthony J. Bonner and Michael Kifer. Results on reasoning about updates in transaction logic. In Burkhard Freitag, Hendrik Decker, Michael Kifer and Andrei Voronkov, editors, *Transactions and Change in Logic Databases: Invited Surveys and Selected Papers of the International Seminar on Logic Databases and the Meaning of Change and ILPS '97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97)*, volume 1472 of *Lecture Notes in Computer Science*, pages 166 – 196, Schloß Dagstuhl and Port Jefferson, 1998. Springer Verlag.

[BK98b] Anthony J. Bonner and Michael Kifer. The state of change: A survey. In Burkhard Freitag, Hendrik Decker, Michael Kifer and Andrei Voronkov, editors, *Transactions and Change in Logic Databases: Invited Surveys and Selected Papers of the International Seminar on Logic Databases and the Meaning of Change and ILPS '97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97)*, volume 1472 of *Lecture Notes in Computer Science*, pages 1 – 36, Schloß Dagstuhl and Port Jefferson, 1998. Springer Verlag.

[BLOM95] Silvia Breitinger, Rita Loogen and Yolanda Ortega-Mallén. Concurrency in functional and logic programming. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*. World Scientific, 1995.

[BLOMP97] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén and Ricardo Peña. The Eden coordination model for distributed memory systems. In *Proceedings of*

High-Level Parallel Programming Models and Supportive Environments (HIPS '97). IEEE Press, 1997.

[BLOMP98] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén and Ricardo Peña. Eden: Language definition and operational semantics. Technical Report 96-10, Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, 1998. revised version, April 7, 1998.

[BM96] Jean-Pierre Banâtre and David Le Métayer. Gamma and the chemical reaction model: ten years after. In Jean-Marc Andreoli, Chris Hankin and David Le Métayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 3 – 41. Imperial College Press, August 1996.

[Bon95] Frédéric Boniol. Synchronous communicating reactive processes. In *Proceedings of the* 2^{nd} *AMAST Workshop on Real-Time Systems*, Bordeaux, June 1995.

[Bou88] Luc Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Informatica*, 25(2):179 – 201, February 1988.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus. Research Report 1702, Institut National de Recherche en Informatique et en Automatique (INRIA), May 1992.

[Bou97] Gérard Boudol. The π-calculus in direct style. In *Proceedings of the* 24^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 228 – 241, 1997.

[Bro98] Manfred Broy. A uniform mathematical concept of a component. *Software: Concepts and Tools*, 19(1):57 – 59, 1998. Appendix to [BDH$^+$98].

[Bud95] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley Publishing Company, 1995.

[BW88] Richard Bird and Philip Wadler. *Introduction to functional programming*. Prentice Hall, 1988.

[BW90] Jos C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[BW00] Manfred Broy and Martin Wirsing. Algebraic state machines. In Teodor Rus, editor, *Proceedings of the* 8^{th} *International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 89 – 118, Iowa, May 2000. Springer Verlag. Invited talk.

[BWW90] John Backus, John H. Williams and Edward L. Wimmers. An introduction to the programming language FL. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 219 – 247. Addison-Wesley Publishing Company, 1990.

[Cai99] Luís Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, Universidade Nova de Lisboa, July 1999.

[CAS01] The CoFI Task Group on Language Design. CASL*: The Common Algebraic Specification Language: Summary*, March 15, 2001. version 1.0.1.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the* 4^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '77)*, pages 238 – 252, Los Angeles, January 1977. ACM.

[CDE+98] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet and José Meseguer. Metalevel computation in maude. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the* 2^{nd} *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[CDE+99] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and José Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, March 1999.

[CdKC94] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295 – 336, September 1994.

[CG86] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1 – 49, January 1986.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[CGZ94] Nicholas Carriero, David Gelernter and Lenore Zuck. Bauhaus Linda. In Paolo Ciancarini, Oscar Nierstrasz and Akinori Yonezawa, editors, *Proceedings of the Workshop on Models and Languages for Coordination of Parallelism and Distribution*, volume 924 of *Lecture Notes in Computer Science*, Bologna, July 1994. Springer Verlag.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Proceedings of FPCA*, 1993.

[CH98] Magnus Carlsson and Thomas Hallgren. *Fudgets: Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg University, 1998.

[CH99] Manuel Carro and Manuel Hermenegildo. Concurrency in prolog using threads and a shared database. In *Proceedings of the* 16^{th} *International Conference on Logic Programming (ICLP '99)*, Las Cruces, November 1999. The MIT Press.

[Cha95] Daniel Chandler. *The Act of Writing*. University of Wales, Aberystwyth, 1995.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346 – 366, 1932.

[Chu36] Alonzo Church. An unsolvable problem in elementary number theory. *American Journal of Mathematics*, 58:345 – 363, 1936.

[Cia94] Paolo Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251 – 284, 1994.

[Cli81] William D Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.

[CLSM96] Anna Ciampolini, Evelina Lamma, Cesare Stefanelli and Paola Mello. Distributed logic objects. *Commputer Languages*, 22(4):237 – 258, 1996.

[CM98] Luís Caires and Luís Monteiro. Verifiable and executable logic specifications of concurrent objects in $\mathcal{L}_\pi$. In Chris Hankin, editor, *Proceedings of the* 7^{th} *European Symposium on Programming (ESOP '98)*, volume 1381 of *Lecture Notes in Computer Science*, pages 42 – 56, Lisbon, March – April 1998. Springer Verlag.

[COM95] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, October 24, 1995. version 0.9, available at `http://www.microsoft.com/com/resources/comdocs.asp`.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396 – 408, July 1963.

[Con88] John S. Conery. Logical objects. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the* 5^{th} *International Conference and Symposium on Logic Programming*, volume 1, pages 420 – 434, Seattle, August 1988. The MIT Press.

[Cop98] James O. Coplien, editor. *Multi-Paradigm Design for C++*. Addison-Wesley Publishing Company, October 1998.

[COR01] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.4.2 edition, February 2001. available at `http://www.omg.org/cgi-bin/doc?formal/01-02-33`.

[Cos01] Pascal Coste. *Conception des systèmes hétérogènes multilangages*. thèse de doctorat, Université Joseph Fourier Grenoble I, January 2001. in french.

[CR95] Philippe Codognet and Francesca Rossi. Nmcc programming: Constraint enforcement and retraction in cc programming. In *Proceedings of ICLP '95*. The MIT Press, 1995.

[dAMdIdSB98] Marcelo de Almeida Maia, Vladimir Oliveira di Iorio and Roberto da Silva Bigonha. Interacting abstract state machines. In *Proceedings of the* 28^{th} *Annual Conference of the Gesellschaft für Informatik*, Technical Report. Universität Magdeburg, 1998. extended abstract.

[dAMdSB98] Marcelo de Almeida Maia and Roberto da Silva Bigonha. Formal semantics for interacting abstract state machines. Technical Report RT 005/98, Universidade Federal de Minas Gerais, September 1998.

[Dav65] Martin Davis, editor. *The Undecidable: Basic Papers On Undecidable Propositions, Unsolvable Problems And Uncomputable Functions.* Raven Press Books, Hewlett, New York, 1965.

[dBG95] Frank S. de Boer and Maurizio Gabbrielli. Modeling real-time in concurrent constraint programming. In John W. Lloyd, editor, *Proceedings of the International Logic Programming Symposium (ILPS '95)*, pages 528 – 542, Portland, December 1995.

[dBG97] Frank S. de Boer and Maurizio Gabbrielli. Infinite computations in concurrent constraint programming. In *Proceedings of the* 13^{th} *Conference on Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, Pittsburgh, 1997.

[dBJ93] Koen de Bosschere and Jean-Marie Jacquet. Multi-Prolog: Definition, operational semantics and implementation. In David Scott Warren, editor, *Proceedings of the* 10^{th} *International Conference on Logic Programming (ICLP '93)*, pages 299 – 313, Budapest, 1993. The MIT Press.

[dBJ96] Koen de Bosschere and Jean-Marie Jacquet. Extending the μLog framework with local and conditional blackboard operations. *Journal of Symbolic Computation*, 21(4):669 – 697, April / May / June 1996.

[dBKPR93] Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi and Jan J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *Proceedings of the International Symposium on Logic Programming (ILPS '93)*, pages 315 – 334. The MIT Press, 1993.

[dBP90] Frank S. de Boer and Catuscia Palamidessi. On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of the* 1^{st} *International Conference on Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 99 – 114, Amsterdam, August 1990. Springer Verlag.

[dBP91] Frank S. de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT '91), Volume 1, Colloquium on Trees in Algebra and Programming (CAAP '91)*, volume 493 of *Lecture Notes in Computer Science*, pages 296 – 319, Brighton, UK, April 1991. Springer Verlag.

[dBP92] Frank S. de Boer and Catuscia Palamidessi. A process algebra of concurrent constraint programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 463 – 477, Washington, USA, 1992. The MIT Press.

[dBP94] Frank S. de Boer and Catuscia Palamidessi. *Advances in Logic Programming Theory*, chapter 2: From Concurrent Logic Programming to Concurrent Constraint Programming, pages 55 – 113. Oxford University Press, 1994.

[DEDC96] Pierre Deransart, AbdelAli Ed-Dbali and Laurent Cervoni. *Prolog: The Standard, Reference Manual*. Springer Verlag, 1996.

[DG94] Pierre Dauchy and Marie-Claude Gaudel. Algebraic specification with implicit state. Technical Report 887, Laboratoire de Recherche en Informatique, Université de Paris-Sud, F - 91405 Orsay Cedex 2, February 1994.

[dG95] Philippe de Groote, editor. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de logique*. Academia-Bruylant, Louvain-la-Neuve, 1995.

[Dij71] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. In Charles Antony Richard Hoare and R. H. Perrott, editors, *Proceedings of a Seminar on Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 72 – 93, Belfast, 1971. Acdemic Press.

[Dij75] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453 – 457, 1975.

[Dij82] Edsger Wybe Dijkstra. How do we tell truths thatmight hurt? In *Selected Writings on Computing: A Personal Perspective*, pages 129–131. Springer Verlag, 1982. Originally written in June 1975.

[Dij01] Edsger Wybe Dijkstra. The end of computing science? *Communications of the ACM*, 44(3):92, March 2001.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243 – 320. Elsevier, Amsterdam, 1990.

[DNO97] Enrico Denti, Antonio Natali and Andrea Omicini. Programmable coordination media. In David Garlan and David Le Métayer, editors, *Proceedings of the* 2^{nd} *International Conference on Coordination Languages and Models (Coordination '97)*, volume 1282 of *Lecture Notes in Computer Science*, pages 274 – 288, Berlin, September 1997. Springer Verlag.

[DNO98] Enrico Denti, Antonio Natali and Andrea Omicini. On the expressive power of a language for programming coordination media. In *Proceedings of the ACM Symposium on Applied Computing (SAC '98)*, Atlanta, February 1998.

[DNOV96] Enrico Denti, Antonio Natali, Andrea Omicini and Marco Venuti. An extensible framework for the development of coordinated applications. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the* 1^{st} *International Conference on Coordination Languages and Models (Coordination '96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 305 – 320, Cesena, April 1996. Springer Verlag.

[DO99] Enrico Denti and Andrea Omicini. From tuple spaces to tuple centres. Technical Report DEIS-LIA-99-001 – LIA Series No. 35, Dipartimento di Elettronica, Informatica e Sistemistica, Universitá di Bologna, 1999.

[DR95] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.

[Ede99] Kerstin Eder. A study of the operational behaviour of Escher and its implementation. In Rachid Echahed, editor, *Proceedings of the* 8^{th} *International Workshop on Functional and Logic Programming (WFLP '99)*, pages 182 – 194, Grenoble, June 1999. Institut IMAG, Rapport de Recherche RR 1021-I-.

[EHRLR80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213 – 253, June 1980.

[EJ99] Rachid Echahed and Jean-Christophe Janodet. Parallel graph narrowing. In *Proceedings of the* 8^{th} *International Workshop on Functional and Logic Programming (WFLP '99)*, pages 269 – 281, Grenoble, June 1999.

[ELO01] ISO/IEC. *Enhancements to* LOTOS, September 2001. ISO 15437:2001.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs in Theoretical Computer Science.* Springer Verlag, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications an Constraints*, volume 21 of *EATCS Monographs in Theoretical Computer Science.* Springer Verlag, 1990.

[EN86] Uffe H. Engberg and Mogens Nielsen. A calculus of communicating systems with label passing. Technical Report DAIMI PB-208, University of Aarhus, Department of Computer Science, May 1986.

[Eng97] Robert Englander. *Developing Java Beans.* The Java Series. O'Reilly & Associates, Inc., June 1997.

[EO94] Hartmut Ehrig and Fernando Orejas. Dynamic abstract data types: An informal proposal. *Bulletin of the European Association for Theoretical Computer Science*, 53, June 1994.

[EPd92] Dick Eriksson, Mats Persson and Kerstin ŸÖdling. Switching software architecture prototype using real time declarative language. In *Proceedings of the* 14^{th} *International Switching Symposium*, Yokohama, 1992.

[ES] Rachid Echahed and Wendelin Serwe. A concurrent extension of functional logic programming languages. Internal Report, available at `ftp://ftp.imag.fr/pub/LEIBNIZ/PMP/conc_ext_flp.ps.gz`.

[ES99] Rachid Echahed and Wendelin Serwe. A concurrent extension of functional logic programming languages. extended abstract. In *Preproceedings of the* 9^{th} *International Workshop on Logic-based Program Synthesis and Transformation*, pages 189 – 198, Venezia, September 1999. Universita Ca Foscari di Venezia, Technical Report CS-99-16.

[ES00] Rachid Echahed and Wendelin Serwe. Combining mobile processes and declarative programming. In John Lloyd, Veronika Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv and Peter J. Stuckey, editors, *Proceedings of the* 1^{st} *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, July 2000. Springer Verlag.

[ES01a] Rachid Echahed and Wendelin Serwe. A component-based approach to concurrent declarative programming. In Michael Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 285 – 298, Kiel, September 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.

[ES01b] Rachid Echahed and Wendelin Serwe. Integrating action definitions into concurrent declarative programming. In Michael Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 299 – 312, Kiel, September 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.

[FFMS01] Cédric Fournet, Fabrice Le Fessant, Luc Maranget and Alan Schmitt. *The JoCaml language (beta release): Documentation and User's Manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), January 2001. available at `http://pauillac.inria.fr/jocaml/htmlman/index.html`.

[FFS95] François Fages, Julian Fowler and Thierry Sola. A reactive constraint logic programming scheme. In *Proceedings of the* 12^{th} *International Conference on Logic Programming (ICLP '95)*, 1995.

[FFS98] François Fages, Julian Fowler and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1 – 3):185 – 212, October 1998.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the* 23^{rd} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 372 – 385, St. Petersburg Beach, Florida, January 1996.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the* 7^{th} *International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406 – 421, Pisa, August 1996. Springer Verlag.

[FHJ91] Torkel Franzén, Seif Haridi and Sverker Janson. An overview of the andorra kernel language. In Lars-Henrik Eriksson, Lars Hallnäs and Peter Schroeder-Heister, editors, *Proceedings of the* 2^{nd} *International Workshop on Extensions of Logic Programming (ELP '91)*, volume 596 of *Lecture Notes in Computer Science*, pages 163 – 179, Stockholm, January 1991. Springer Verlag.

[FLP85] Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374 – 382, April 1985.

[Fok00] Wan Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer Verlag, 2000.

[FPJ95] Sigbjorn Finne and Simon L. Peyton Jones. Composing Haggis. In Remco C. Veltkamp and Edwin H. Blake, editors, *Proceedings of the Eurographics Workshop on Programming Paradigms in Graphics*, pages 85 – 101, Maastricht, September 1995. Springer Verlag.

[FRS98] François Fages, Paul Ruet and Sylvain Soliman. Phase semantics and verification of concurrent constraint programs. In *Proceedings of the* 13^{th} *Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, 1998.

[Fu97] Yuxi Fu. A proof theoretical approach to communication. In Pierpaolo Degano, Robert Gorrieri and Alberto Marchetti-Spaccamela, editors, *Proceedings of the* 24^{th} *International Colloquium on Automata, Languages and Programming (ICALP '97)*, volume 1250 of *Lecture Notes in Computer Science*, pages 325 – 335, Bologna, July 1997. Springer Verlag.

[GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95 – 146, January 1992.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96 – 107, February 1992.

[GCR99] Yan Georget, Philippe Codognet and Francesca Rossi. Constraint retraction in CLP(FD): Formal framework and performance results. *Constraints*, 4(1):5 – 42, February 1999.

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

[Gel89] David Gelernter. Multiple tuple spaces in Linda. In Eddy Odijk, Martin Rem and Jean-Claude Syre, editors, *Proceedings of the Conference on Parallel Architecture and Languages Europe (PARLE '89)*, volume 366 of *Lecture Notes in Computer Science*, pages 20 – 27, Eindhoven, June 1989. Springer Verlag.

[Gen35] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68 – 131. North Holland Publishing Company, 1935. 1969.

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini and Michael M. Richter, editors, *Selected papers from the* 6^{th} *Workshop on Computer Science Logic (CSL '92)*, volume 702 of *Lecture Notes in Computer Science*, pages 274 – 308, San Miniato, 1993. Springer Verlag.

[GHLR96] J. C. González-Moreno, M. Teresa Hortalá-González, Francisco Javier López-Fraguas and Mario Rodríguez-Artalejo. A rewriting logic for declarative programming. In Hanne Riis Nielson, editor, *Programming Languages and Systems. Proceedings of the* 6^{th} *European Symposium on Programming (ESOP '96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 156 – 172, Linköping, April 1996. Springer Verlag.

[GHLR99] J. C. González-Moreno, M. Teresa Hortalá-González, Francisco Javier López-Fraguas and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47 – 87, July 1999.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 102, 1987.

[Gir95] Jean-Yves Girard. Linear logic, its syntax and semantics. In Jean-Yves Girard, Yves Lafont and Laurent Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 1 – 43. Cambridge University Press, June 1995.

[GJS96] Vineet Gupta, Radha Jagadeesan and Vijay Saraswat. Models for concurrent constraint programming. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the* 7^{th} *International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 66 – 83, Pisa, August 1996. Springer Verlag.

[GJSB00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc, second edition, June 2000.

[GKZ99] Marie-Claude Gaudel, Carole Khoury and Alexandre V. Zamulin. Dynamic systems with implicit state. In Jean-Pierre Finance, editor, *Proceedings of the* 2^{nd} *Internationsl Conference on Fundamental Approaches to Software Engineering (FASE '99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 114 – 128, Amsterdam, March 1999. Springer Verlag. Held as part of the European Joint Conferences on the Theory and Practice of Software (ETAPS '99).

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[GR93a] Emden R. Gansner and John H. Reppy. A multithreaded higher-order user interface toolkit. In *User Interface Software*, volume 1 of *Software Trends*, pages 61 – 80. John Wiley & Sons, 1993.

[GR93b] Paola Glavan and Dean Rosenzweig. Communicating evolving algebras. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini and Michael M. Richter, editors, *Selected papers from the* 6^{th} *Workshop on Computer Science Logic (CSL '92)*, volume 702 of *Lecture Notes in Computer Science*, pages 182 – 215, San Miniato, 1993. Springer Verlag.

[Gui95] GUI Fest '95 Post-Challenge: multiple counters. available at `http://www.cs.chalmers.se/~magnus/GuiFest-95/`, July 24 – July 28 1995. organized by Simon Peyton Jones and Phil Gray as a part of the Glasgow Research Festival.

[Gur85] Yuri Gurevich. A new thesis. *American Mathematical Society Abstracts*, page 317, August 1985. abstract 85T-68-203.

[Gur91] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the European Association for Theoretical Computer Science*, 43:264 – 284, February 1991. preliminary version of [Gur93].

[Gur93] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In Grzegorz Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266 – 292. World Scientific, 1993. A previous version appeared as [Gur91].

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford University Press, 1995.

[Gur97] Yuri Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan, May 1997.

[Gur00] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77 – 111, July 2000.

[Hai86] Brent Hailpern. Multiparadigm languages and environments. *IEEE Software*, 3(1):6 – 9, January 1986. Guest Editor's Introduction.

[HAK+00a] Michael Hanus [Ed.], Sergio Antoy, Johannes Koj, Philip Niederau, Ramin Sadre and Frank Steiner. *PAKCS 1.3 User Manual*, December 4, 2000. available at `http://www.informatik.uni-kiel.de/~pakcs`.

[HAK+00b] Michael Hanus [Ed.], Sergio Antoy, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno Navarro and Frank Steiner. Curry: An integrated functional logic language. available at `http://www.informatik.uni-kiel.de/~mh/curry/report.html`, June 6, 2000. Version 0.7.1.

[Han] Michael Hanus. FlatCurry: An intermediate representation for Curry programs. URL `http://www.informatik.uni-kiel.de/~curry/flat`.

[Han72] Per Brinch Hansen. Strutured multiprogramming. *Communications of the ACM*, 15(7):574 – 578, July 1972.

[Han94] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19 & 20:583 – 628, 1994.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *Proceedings of the* 24^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '97)*, 1997.

[Han99] Michael Hanus. Distributed programming in a multi-paradigm declarative language. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 188 – 205, Paris, 1999. Springer Verlag.

[HCRP89] Nicolas Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Special issue on Another Look at Real-Time Systems*, Proceedings of the IEEE, 1989.

[Hen82] Peter Henderson. Purely functional operating systems. In J. Darlington, P. Henderson and D. A. Turner, editors, *Functinal Programming and its Applications: an advanced course*, CREST advanced courses from Camdridge University Press. Cambridge university press, July, 20 – 31 1982. reprinted 1983.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, June 1977.

[HI93] Matthew Hennessy and Anna Ingólfsdóttir. A theory of communicating processes with value passing. *Information and Computation*, 107(2):202 – 236, December 1993.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computation model. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the* 7^{th} *International Conference on Logic Programming (ICLP 1990)*, pages 31 – 46, Jerusalem, June 1990. The MIT Press.

[HJ94] Paul Hudak and Mark Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... an experiment in software prototyping productivity. available at `http://www.cs.yale.edu/homes/hudak-paul.html`, July 1994.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327 – 365, May 1994.

[Hoa74] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549 – 557, October 1974.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666 – 677, August 1978. Corrigendum: CACM 21(11): 958; Reprint: CACM 26(1): 100 – 106.

[Hoa85] Charles Antony Richard Hoare. *Communicating sequential processes.* International Series in Computer Science. Prentice Hall, 1985.

[Hoa87] Charles Antony Richard Hoare. *Processus Sequentiels Communicants*. Manuels Informatiques Masson. Masson, 1987. french translation of [Hoa85] by Alain Kermarrec.

[HPW96] James Harland, David J. Pym and Michael Winikoff. Programming in lygon: An overview. In Martin Wirsing and Maurice Nivat, editors, *Proceedings of the* 5^{th} *International Conference on Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 391 – 405, Munich, July 1996. Springer Verlag.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the* 25^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 365 – 377, San Diego, January 1998.

[HR00] M. Hennessy and J. Riely. Information flow vs. ressource access in the asynchronous pi-calculus. In Ugo Montanari, José D. P. Rolim and Emo Welzl, editors, *Proceedings of the* 27^{th} *International Colloquium on Automata, Languages and Programming (ICALP 2000)*, volume 1853 of *Lecture Notes in Computer Science*, pages 415 – 427, Geneva, July 2000. Springer Verlag.

[HS87] Roger Hayes and Richard D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13(12):1254 – 1264, December 1987.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the* 5^{th} *European Conference on Object-Oriented Programming (ECOOP '91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133 – 147, Geneva, July 1991. Springer Verlag. revised version of August 1991.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 17 – 42. Addison-Wesley Publishing Company, 1990.

[HVY00] Kohei Honda, Vasco Thudichum Vasconcelos and Nobuko Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *Proceedings of the* 9^{th} *European Symposium on Programming (ESOP '2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180 – 199, Berlin, March 2000. Springer Verlag.

[Jan94] Sverker Janson. *AKL – A Multiparadigm Programming Language*. PhD thesis, Uppsala Theses in Computing Science 19, June 1994. SICS Dissertation Series 14.

[JdB94] Jean-Marie Jacquet and Koen de Bosschere. On the semantics of μLog. *Future Generation Computer Systems Journal*, 10:93 – 135, February 1994.

[JH93] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Yale University, August 1993.

[JH94] Sverker Janson and Seif Haridi. An introduction to AKL – a multiparadigm programming language. In *Constraint Programming*, volume 131 of *NATO-ASI Series*. Springer-Verlag, 1994.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the* 14^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '87)*, pages 111 – 119, München, January 1987. ACM.

[JMH93] Sverker Janson, Johan Montelius and Seif Haridi. Ports for objects in concurrent logic programs. In Gul A. Agha, Peter Wegner and Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[Jon99] Mark P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, France, October 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[Jor84] Philippe Jorrand. FP2: Functional parallel programming based on term substitution. In Wolfgang Bibel and B. Petkoff, editors, *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA '84)*, pages 95 – 112, Varna, September 1984. Elsevier Science Publishers.

[Jor85] Philippe Jorrand. Term rewriting as a basis for the design of a functional and parallel programming language. a case study: The language FP2. In Wolfgang Bibel and Philippe Jorrand, editors, *Fundamentals of Artificial Intelligence: An Advanced Course*, volume 232 of *Lecture Notes in Computer Science*, pages 221 – 276, Vignieu, July 1985. Springer Verlag.

[KdRB91] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, chapter 5 and 6. The MIT Press, 1991.

[Kho96] Carole Khoury. Spécifications algébriques avec état implicite. In *Journées du GDR Programmation, CNRS*, Orléans, November, 20 – 22 1996.

[KK71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3/4):227 – 260, 1971.

[Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay and Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1 – 112. Oxford University Press, 1992.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of Information Processing 74*, pages 569 – 574, Stockholm, August 1974. International Ferderation for Information Processing (IFIP), North Holland Publishing Company.

[KP97] Philipp W. Kutter and Alfonso Pierantonio. Montages: Specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416 – 442, 1997.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988. ANSI C.

[LDG+01] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. *The Objective Caml system: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), July 2001. Release 3.02.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley Publishing Company, second edition, november 1999.

[LF92] Francisco Javier López-Fraguas. A general scheme for constraint functional logic programming. In Héléne Kirchner and Giorgio Levi, editors, *Proceedings of the* 3^{rd} *Internatinal Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 213 – 227, Volterra, September 1992. Springer Verlag.

[LFSH99] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proceedings of the* 10^{th} *International Conference on Rewriting Techniques and Applications (RTA '99)*, pages 244 – 247. Springer Verlag, LNCS 1631, 1999.

[Llo] John W. Lloyd. Interaction and concurrency in a declarative programming language. to appear.

[Llo95] John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Departement of Computer Science, University of Bristol, June 1995.

[Llo99] John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.

[LOT00] ISO JTC 1/SC 7. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, June 2000. ISO 8807:1989.

[LP96] Jimmy H. M. Lee and Paul K. C. Pun. An overview of the $\mathcal{OLI}$ multiparadigm programming language and its semantics. In Dilip Patel, Yuan Sun and Shushma Patel, editors, *Proceedings of the International Conference on Object Oriented Information Systems (OOIS '96)*, pages 79 – 92, London, December 1996. Springer Verlag.

[LP97] Jimmy H. M. Lee and Paul K. C. Pun. Object logic integration: A multiparadigm design methodology and a programming language. *Commputer Languages*, 23(1):25 – 42, 1997.

[LSV99] Luís Lopes, Fernando M. A. Silva and Vasco Thudichum Vasconcelos. A virtual machine for a process calculus. In Gopalan Nadathur, editor, *Proceedings of the Conference on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 244 – 260, Paris, 1999. Springer Verlag.

[Mah87] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proceedings of the 4^{th} International Conference on Logic Programming (ICLP '87)*, pages 858 – 876, Melbourne, May 1987. The MIT Press.

[McC98] Glen McCluskey. Using Java reflection. available at `http://developer.java.sun.com /developer/technicalArticles/ALT/Reflection`, January 1998.

[Mes92] José Meseguer. Multiparadigm logic programming. In Hélène Kirchner and Giorgio Levi, editors, *Proceedings of the 3^{rd} Internatinal Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 158 – 200, Volterra, September 1992. Springer Verlag.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348 – 375, December 1978.

[Mil80] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

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

[Mil92a] Dale Miller. The pi-calculus as a theory in linear logic: Preliminary results. In Evelina Lamma and Paola Mello, editors, *Proceedings of the 3^{rd} International Workshop on Extensions of Logic Programming (ELP '92)*, volume 660 of *Lecture Notes in Computer Science*, pages 242 – 264, Bologna, February 1992. Springer Verlag.

[Mil92b] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119 – 141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[Mil93a] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):78 – 89, January 1993. Turing Award Lecture.

[Mil93b] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, 1991)*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mil95] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63 – 67, December 1995.

[Mil96] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201 – 232, September 1996.

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

[ML95] Naftaly H. Minsky and Jerrold Leichter. Law-governed Linda as a coordination model. In Paolo Ciancarini, Oscar Nierstrasz and Akinori Yonezawa, editors, *selected papers of the Workshop on Models and Languages for Coordination of Parallelism and Distribution (ECOOP '94)*, volume 924 of *Lecture Notes in Computer Science*, pages 125 – 146, Bologna, July 1995. Springer Verlag.

[Mod96] ISO/IEC JTC 1/SC 22/WG 13. *Modula-2, Base Language*, June 1996. ISO 10514-1:1996.

[MOM01] Narciso Martí-Oliet and José Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 2001. To appear.

[Mos92] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Moz] The mozart programming system. `http://www.mozart-oz.org/`.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*, volume 1. Springer, 1991.

[MPW92] Robin Milner, Joachim G. Parrow and David J. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1 – 77, September 1992.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The Definition of Standard ML – Revised*. The MIT Press, 1997.

[Mul86] Special issue on multiparadigm languages and environments. *IEEE Software*, 3(1):6 – 77, January 1986.

[NFP98] Rocco De Nicola, Gian Luigi Ferrari and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315 – 330, May 1998.

[Nie96] Flemming Nielson, editor. *ML with Concurrency: Design, Analysis, Implementation and Application*. Monographs in Computer Science. Springer Verlag, 1996.

[NL95] Kam-Wing Ng and Chi-Keung Luk. I+: A multiparadigm language for object-oriented declarative programming. *Computer Languages*, 21(2):81 – 100, 1995.

[NM95] Joachim Niehren and Martin Müller. Constraints for free in concurrent computation. In Kanchana Kanchanasut and Jean-Jacques Lévy, editors, *Proceedings of the Asian Computing Science Conference*, volume 1023 of *Lecture Notes in Computer Science*, pages 171 – 186, Pathumthani, December 1995. Springer Verlag.

[NP96] Rocco De Nicola and Rosario Pugliese. A process algebra based on Linda. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the* 1^{st} *International Conference on Coordination Languages and Models (Coordination '96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 160 – 178, Cesena, April 1996. Springer Verlag.

[NS94] Joachim Niehren and Gert Smolka. A confluent relational calculus for higher-order programming with constraints. In Jean-Pierre Jouannaud, editor, *Proceedings of the* 1^{st} *International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 89 – 104, München, September 1994. Springer Verlag.

[NSvEP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen and M. J. Plasmeijer. Concurrent CLEAN. In Aarts, Leeuwen and Rem, editors, *Proceedings of Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 202 – 219, Eindhoven, 1991. Springer Verlag.

[O'D85] Michael J. O'Donnell. *Equational Logic as a Programming Language*. The MIT Press, 1985.

[Ode00] Martin Odersky. Functional nets. In Gert Smolka, editor, *Programming Languages and Systems, Proceedings of the* 9^{th} *European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 1 – 25, Berlin, March 2000. Springer Verlag. invited paper.

[ODN95] Andrea Omicini, Enrico Denti and Antonio Natali. Agent communication and control through logic theories. In Marco Gori and Giovanni Soda, editors, *Proceedings of the* 4^{th} *Congress of the Italian Association for Artificial Intelligence (AI*IA '95)*, volume 992 of *Lecture Notes in Computer Science*, pages 439 – 450, Florence, 1995. Springer Verlag.

[PA98a] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical Report SEN-R9834, CWI, December 1998.

[PA98b] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329 – 400. Acdemic Press, August 1998.

[Pal97] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proceedings of the* 24^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 256 – 265, Paris, January 1997. ACM.

[Pik89] Rob Pike. A concurrent window system. *Computing Systems*, 2(2):133 – 153, Spring 1989.

[PJGF96] Simon L. Peyton Jones, Andrew D. Gordon and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the* 23^{rd} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 295 – 308, St Petersburg Beach, Florida, January 1996.

[PJHA+99] Simon Peyton Jones [Ed.], John Hughes [Ed.], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John

Peterson, Alastair Reid, Colin Runciman and Philip Wadler. Report on the programming language Haskell 98: A non-strict, purely functional language. available at `http://haskell.systemsz.cs.yale.edu/definition`, February 1999.

[Pla91] John Placer. The multiparadigm language *G*. *Commputer Languages*, 16(3/4):235 – 258, 1991.

[PR96] Prakash Panangaden and John H. Reppy. The essence of concurrent ML. In Nielson [Nie96], chapter 2, pages 5 – 29.

[Pro00] Fédéric Prost. A static calculus of dependencies for the λ-cube. In *Proceedings of the* 15^{th} *Annual IEEE Symposium on Logic in Computer Science (LICS '2000)*, pages 267 – 276, Santa Barbara, 2000. IEEE Computer Society Press.

[Pro01] Fédéric Prost. Types for static analyses of mobile processes. *Cahiers du laboratoire Leibniz*, 29, August 2001.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, CSCI, Indiana University, March 1997. Dedicated to Robin Milner on the occasion of his 60^{th} birthday.

[PT98] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[PV97] Joachim Parrow and Björn Victor. The update calculus. In Michael Johnson, editor, *Proceedings of AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 409 – 423. Springer Verlag, 1997. Full version available as Technical report DoCS 97/93, Uppsala University.

[PV98] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of Logig in Computer Science*, pages 176 – 185. IEEE Computer Society Press, 1998.

[PvE98] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean language report. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, June 1998. Version 1.3.

[Rei80] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132, 1980.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, SIGPLAN Notices, pages 293 – 305, Toronto, June 1991. ACM Press.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RF97] Paul Ruet and François Fages. Concurrent constraint programming and non-commutative linear logic. In *Selected Annual Conference of the European Association for Computer Science Logic*, Aarhus, August 1997.

[Rin88] G. A. Ringwood. Parlog86 and the dining logicians. *Communications of the ACM*, 31(1):10 – 25, January 1988.

[RS81] J. A. Robinson and E. E. Sibert. The LOGLISP user's manual. Technical Report 12/81, Syracuse University, New York, 1981.

[RT74] Dennis Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365 – 375, July 1974.

[RT78] Dennis Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6, part 2), July/August 1978. Update of [RT74].

[San98] Davide Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, INRIA - Sophia Antipolis, August 1998.

[Sar93] Vijay Anand Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards. The MIT Press, 1993.

[Sce90] Andre Scedrov. A guide to linear logic. *Bulletin of the European Association for Theoretical Computer Science*, 41:154 – 165, June 1990. updated version.

[Sch98] Wolfgang Schönfeld. Interacting abstract state machines. In *Proceedings of the* 28^{th} *Annual Conference of the Gesellschaft für Informatik*, Technical Report. Universität Magdeburg, 1998.

[SDE95] Diomidis D. Spinellis, Sophia Drossopoulou and Susan Eisenbach. Object-oriented technology in multiparadigm language implementation. *Journal of Object-Oriented Programming*, 8(1):33 – 38, March/April 1995.

[Ser98] Wendelin Serwe. Étude d'une sémantique pour les langages logico-fonctionnels réactifs. mémoire de DEA, Université Joseph Fourier, Grenoble and Institut National Polytechnique de Grenoble, June 24 1998.

[Sha83] Ehud Shapiro. A subset of concurrent prolog and its interpreter. Technical report, ICOT, March 1983.

[Sha86] Ehud Shapiro. Concurrent prolog: A progress report. *IEEE Computer*, 19(8):44 – 58, August 1986.

[Sha87] Ehud Shapiro, editor. *Concurrent Prolog*, volume 1. The MIT Press, 1987.

[Sha89] Ehud Shapiro. Technical correspondence: Linda in context. *Communications of the ACM*, 32(10):1244 – 1249, October 1989.

[SJG95] Vijay Anand Saraswat, Radha Jagadeesan and Vineet Gupta. Default timed concurrent constraint programming. In *Proceedings of the* 22^{nd} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 272 – 285, San Francisco, January 1995.

[SJG96] Vijay Anand Saraswat, Radha Jagadeesan and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5 – 6):475 – 520, November – December 1996.

[SKL90] Vijay Anand Saraswat, Kenneth M. Kahn and Jacob Levy. Janus – a step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*, Austin, October 1990. extended abstract.

[SL92] Vijay Anand Saraswat and Patrick Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, Xerox PARC, 1992.

[Sla74] James R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *Journal of the ACM*, 21(4):622 – 642, October 1974.

[SML98] Standard ML of New Jersey user's guide. available at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/index.html`, 1998.

[Smo94] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *Proceedings of the* 1^{st} *International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50 – 72, München, September 1994. Springer Verlag.

[Smo95a] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251 – 292. Springer Verlag, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324 – 343. Springer Verlag, 1995.

[Smo98] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Proceedings of the* 7^{th} *European Symposium on Programming: Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 1 – 11, Lisbon, March / April 1998. Springer Verlag.

[Spi94] Diomidis D. Spinellis. *Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, London, February 1994.

[SR90] Vijay Anand Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the* 17^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '90)*, pages 232 – 245, San Francisco, January 1990. extended abstract.

[SRP91] Vijay Anand Saraswat, Martin Rinard and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the* 18^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '91)*, pages 333 – 353, New York, 1991. ACM. Preliminary Report.

[SSB01] Robert Stärk, Joachim Schmid and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer Verlag, June 2001.

[Ste90] Guy L. Steele, Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.

[Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, October 1985.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the* 25^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 355 – 364, San Diego, January 1998.

[SVI96] Geoffrey Smith, Dennis M. Volpano and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167 – 187, 1996.

[Szy98] Clemens A. Szyperski. Emerging component software technologies – a strategic comparison. *Software: Concepts and Tools*, 19(1):2 – 10, June 1998.

[Tho90] Simon Thompson. Interactive functional programs. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 249 – 285. Addison-Wesley Publishing Company, 1990.

[Tic95] Evan Tick. The deevolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2):89 – 124, May 1995.

[TLK96a] Bent Thomsen, Lone Leth and Tsung-Min Kuo. FACILE — from toy to tool. In Nielson [Nie96], chapter 5, pages 97 – 144.

[TLK96b] Bent Thomsen, Lone Leth and Tsung-Min Kuo. A facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the* 7^{th} *International Conference onConcurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 278 – 298, Pisa, August 1996. Springer Verlag.

[Tse94] Clifford Sheung-Ching Tse. The design and implementation of an actor language based on linear logic. Master's thesis, Massachusetts Institute of Technology, May 1994.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, ser. 2, vol. 42:230 – 265, November 1936. reprinted in [Dav65, pages 115 – 153].

[Tur39] Alan M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, ser. 2, vol. 45:161 – 228, 1939. reprinted in [Dav65, pages 154 – 222].

[Ued85] Kazunori Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pages 168 – 179. Springer Verlag, 1985.

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the* 8^{th} *European Conference on Object-Oriented Programming (ECOOP '94)*, volume 821 of *Lecture Notes in Computer Science*, pages 100 – 117, Bologna, July 1994. Springer Verlag.

[vEdLF82] M. H. van Emden and G. J. de Lucena Filho. Predicate logic as a language for parallel programming. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, volume 16 of *APIC Studies in Data Processing*, pages 189 – 198. Acdemic Press, 1982.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733 – 742, October 1976.

[VP98] Björn Victor and Joachim Parrow. Concurrent constraints in the fusion calculus. In Kim G. Larsen, Sven Skyum and Glynn Winskel, editors, *Proceedings of the* 25^{th} *International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 455 – 469, Aalborg, July 1998. Springer Verlag.

[VRHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804 – 851, September 1997.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*, volume 711 of *Lecture Notes in Computer Science*, Gdansk, August 1993. (invited talk).

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240 – 263, September 1997.

[Weg98] Peter Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315 – 351, February 1998.

[WFF98] Carl-Alexander Wichert, Burkhard Freitag and Alfred Fent. Logical transactions and serializability. In Burkhard Freitag, Hendrik Decker, Michael Kifer and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*, pages 134 – 165. Springer Verlag, 1998.

[Win97] Michael David Winikoff. *Logic Programming with Linear Logic*. PhD thesis, School of Electrical Engineering and Computer Science, University of Melbourne, 1997.

[Wir90] Martin Wirsing. *Handbook of Theoretical Computer Science*, volume B, chapter Algebraic Specification, pages 675 – 788. North Holland Publishing Company, 1990.

[WW88] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Proceedings of the* 15^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '88)*, pages 169 – 179, San Diego, January 1988.

[Zam98] Alexandre V. Zamulin. Specification of dynamic systems by typed gurevich machines. In Zdzislaw Bubnicki and Adam Grzech, editors, *Proceedings of the* 13^{th} *International Conference on System Science*, pages 160 – 167, Wroclaw, September 15 – 18, 1998.

[Zav89] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15 – 25, September 1989.

[Zav91] Pamela Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212 – 225, March 1991.

[Zei95] Stephen F. Zeigler. Comparing development costs of C and Ada. whitepaper, Rational Software Corporation, March 30, 1995.

[ZJ96] Pamela Zave and Michael Jackson. Where do operations come from? a multiparadigm specification technique. *IEEE Transactions on Software Engineering*, XXII(7):508 – 528, July 1996.

[Zuc99] Elena Zucca. From static to dynamic abstract data-types: An institution transformation. *Theoretical Computer Science*, 216(1 – 2):109 – 157, March 1999.

Total number of references: 341.

Appendix A

Concrete Syntax of Sabir

In this appendix we give the grammars defining the concrete syntax used for the examples in chapter 7. We use the symbol ::= for the definition of a non-terminal, the symbol ::= for an alternative and the write $\{a\}$ for an optional a. Furthermore a^* (respectively, a^+) stands for the repetition of a a natural (respectively, positive) number of times. We do not present the (classical) definitions for identifiers, number and character strings. The `typewriter font` is used for the keywords and "special" symbols. The current implementation does not distinguish between upper and lower case letters, and considers any undeclared identifier as a variable.

Since the actions are specified as classical `ocaml`-functions, we refer the reader to the `ocaml` documentation for further details. As mentioned in section 7.1.3, the import of declarations from another component amounts in the current implementation to read the corresponding files. Consequently, we do not need to present the syntax for imports separately, and present only the grammars for stores, translations and processes.

A.1 Grammars for Stores

The grammar shown in table A.1 on page 253 defines the syntax of the declarative language used in the description of the stores of a component, as it is used in the examples of the lifts (see section 7.2), the multiple counters (see example 3.63) and the Dining Philosophers (see section 1.1.5). We do not show the built-in infix operations (such as `+`, `-`), nor do we present the classical syntax for identifiers (idf), numbers and character strings (which are enclosed by `"`).

A.2 Translations

The syntax for the definition of translations is shown in table A.2 on page 253.

A.3 Grammar for Processes

Table A.3 on page 254 shows the grammar which defines the syntax used in the description of the processes, *i.e.*, for the process definitions and the initial process term.

store	::=	`DECL` *declaration** `RULES` *rule**	(A.1.1)
declaration	::=	`TYPE` idf `=` *constructors* `.`	(A.1.2)
	|	`TYPE` idf `.`	(A.1.3)
	|	idf `::` *type-expression* `.`	(A.1.4)
constructors	::=	idf *simple-type** $\big($`|` *constructors*$\big)^*$	(A.1.5)
type-expression	::=	*simple-type* | *functional-type* | *named-type*	(A.1.6)
simple-type	::=	idf	(A.1.7)
functional-type	::=	`(` *type-expression* `->` *type-expression* `)`	(A.1.8)
named-type	::=	`NAME (` *type-expression* `)`	(A.1.9)
rule	::=	*lhs* $\{$`|` *expression*$^*\}$ `=` *expression*	(A.1.10)
lhs	::=	idf *expression**	(A.1.11)
expression	::=	*constant* | *application* | `(` *expression* `)`	(A.1.12)
constant	::=	idf | number | string	(A.1.13)
application	::=	`(` idf *expression** `)`	(A.1.14)

Table A.1: Syntax for Stores: Language 1

translation	::=	`TRANSLATION` idf *translation-rule*$^+$ `.`	(A.2.1)
translation-rule	::=	*expression* `=` *expression*	(A.2.2)
expression	::=	*constant* | *application* | `(` *expression* `)`	(A.2.3)
constant	::=	idf | number | string	(A.2.4)
application	::=	`(` idf *expression** `)`	(A.2.5)

Table A.2: Syntax for Translations

processes	::=	`DECL` *declaration** `DEF` *definition** `INITIAL` *process-term**	(A.3.1)
declaration	::=	`PROCESS` idf `::` *parameter-type** `.`	(A.3.2)
definition	::=	`PROCESS` idf variable* `:-` *process-rules*$^+$ `END`	(A.3.3)
process-rules	::=	*guarded-action* `;` *process-term* $\big($`,` *process-rules*$\big)^*$	(A.3.4)
guarded-action	::=	`[` *expression*$^+$ `=>` *actions* `]`	(A.3.5)
actions	::=	`{` idf `@` *expression* `}` $\big($`;` *actions*$\big)^*$	(A.3.6)
process-term	::=	`SUCCESS` | `WAIT` | `RANDOM_WAIT` | *application*	(A.3.7)
	|	*process-term* `||` *process-term*	(A.3.8)
	|	*process-term* `;` *process-term*	(A.3.9)
expression	::=	*constant* | *application* | `(` *expression* `)`	(A.3.10)
constant	::=	idf | idf `^` | number | string	(A.3.11)
application	::=	`(` idf *expression** `)`	(A.3.12)

Table A.3: Syntax for Process Definitions and Process Terms

The non-terminal *parameter-type* occurring in rule (A.3.2) is defined as the non-terminal *type-expression* of table A.1 (rule (A.1.6)).

Contents

List of Tables

List of Figures

Index

Étude de la programmation logico-fonctionnelle concurrente

La construction de programmes nécessite l'utilisation d'outils adaptés. Un outil particulier est le langage de programmation. Les langages logico-fonctionnels sont des langages de programmation dits déclaratifs qui se basent sur les notions mathématiques de *fonction* et de *prédicat*. Ce fondement théorique solide facilite à la fois la description d'un système à un niveau proche de la spécification ainsi que la validation de programmes. Néanmoins, les concepts sous-jacents aux langages logico-fonctionnels sont insuffisants pour la description aisée de systèmes complexes qui nécessitent l'interactivité, la concurrence et la distribution. Pour la modélisation de ces systèmes, la notion de *processus* a été introduite. Dans le contexte des *algèbres de processus*, un processus est caractérisé par les *actions* qu'il est capable d'exécuter. Cependant, les langages fondés uniquement sur les algèbres de processus doivent être étendus afin d'éviter le codage de fonctions et de prédicats en termes de processus.

Dans cette thèse nous proposons un modèle de calcul qui intègre la programmation concurrente et déclarative. Nous suggérons de modéliser un système par un ensemble de composants. Chacun de ces composants comporte un programme déclaratif, appelé *store*, et un ensemble de processus interagissant par l'exécution d'actions. De plus, un composant peut contenir de nouvelles actions définissables par le programmeur. L'interaction entre composants est fondée sur le même principe, c.-à.-d. un processus peut exécuter des actions sur les stores des autres composants. Les différents composants d'un système peuvent utiliser des langages déclaratifs différents pour la description de leurs stores respectifs, ce qui nécessite la *traduction* des valeurs communiquées. Nous donnons une sémantique compositionnelle ainsi qu'une analyse de la confidentialité pour les processus d'un composant, et présentons les principes d'un prototype implanté.

Spécialité : « Informatique : Systèmes et Communications »

Mots clefs : Langages de programmation, langages déclaratifs, langages logico-fonctionnels, programmation concurrente, algèbres de processus, mobilité, programmation multiparadigme, action, composants, compositionnalité, confidentialité.

On Concurrent Functional Logic Programming

The construction of programs needs the use of appropriate tools. A particular kind of tool are programming languages. Functional-logic programming languages are declarative languages based on the mathematical notions of *functions* and *predicates*. This solid formal foundation facilitates both, the concise description of systems at a level close to specifications, as well as the validation of programs. Nevertheless, the underlying concepts are insufficient for a convenient description of complex systems where interactivity, concurrency and distribution are needed. In order to model such systems, the notion of *processes* has been introduced. In the context of process algebræ, a process is characterised, informally, by the *actions* it might execute. However, languages based solely on process algebræ have to be extended with the notions of functions and predicates to avoid the encoding of these notions by means of processes.

In this thesis, we study an integration of declarative and concurrent programming. We model a system as a set of components. Each component contains a declarative program, called *store*, and a set of processes which interact by the execution of actions and are described by means of process algebræ. Additionally, a component might contain definitions of further actions definable by the programmer. Interaction between components uses the same scheme: Processes can execute actions on the store of the other components. The different components of a system can use different declarative languages for the description of their stores, so that the *translation* of the communicated values is necessary. We give a compositional semantics and a secrecy analysis for the processes of a component and present the outlines of a prototype implementation.

Keywords: (Declarative) Programming Languages, Functional-Logic Programming, Concurrent Programming, Process Algebra, Mobility, Multiparadigm Programming, Action, Components, Compositionality, Secrecy.

Laboratoire leibniz – Institut imag, 46, avenue Félix Viallet, 38031 Grenoble Cedex, france