Rapport de Stage
DEA Sémantique 1998/1999

Modélisation et spécification d'architectures logicielles

Benjamin Blanc

Direction : Michel Bidoit

Table des matières

Introduction

Ce stage a eu lieu au laboratoire LSV de l'Ecole Normale Supérieure de Cachan pendant les mois d'avril à septembre 1999. Ce laboratoire a pour projet scientifique la vérification en informatique, au niveau théorique et dans le cadre de contrats industriels. Ses motivations proviennent du fait que la part sans cesse croissante de l'informatique dans les systèmes (de télécommunications, de transport terrestres et aériens, de production et de transport d'énergie, etc.) conduit à accorder une attention renouvelée aux questions suivantes : dans quelle mesure peut-on faire confiance aux logiciels qui sont au coeur de ces systèmes, quels sont les moyens et les méthodes permettant d'étayer et d'accroître cette confiance ?

L'acuité du problème ainsi posé est évidente lorsque l'on considère les systèmes qualifiés de critiques, c'est-à-dire les systèmes dont les défaillances peuvent avoir des conséquences irréversibles, voire dramatiques, sur un environnement qu'ils contrôlent. A titre d'exemple citons les systèmes de contrôle-commande des centrales nucléaires, les systèmes de régulation du trafic ferroviaire, d'assistance au pilotage d'avions, etc. Le problème devient vite très difficile quand on considère des applications de grande taille ou des applications complexes. Cette complexité est typiquement induite par le caractère distribué de l'application considérée, ou par l'existence de contraintes temporelles strictes à satisfaire (systèmes réactifs, temps-réel).

L'objectif de ce stage était de comprendre comment peut se refléter, au niveau des spécifications, l'architecture de tels systèmes. Nous définirons dans le chapitre 1 ce qu'on entend par *l'architecture logicielle*, domaine dans lequel se place notre travail, à partir des articles de Garlan, Perry, Luckham, pour les plus importants. Une fois les notions préliminaires exposées, nous considérerons deux mécanismes possibles pour spécifier des architectures : partir du diagramme informel que l'on construit à partir du cahier des charges (*architecture de connexion*), ou bien refléter les possibilités d'un langage de programmation sous-jacent (*architecture d'implémentation*).

Le chapitre 2 expose trois langages de spécification d'architectures de connexion, dans leurs principes, leur syntaxe et leur sémantique. Nous illustrerons ces langages sur un exemple caractéristique des problèmes que l'on peut rencontrer au niveau architectural.

Une question se posait alors : quels sont les liens et les différences qui interviennent entre les concepts de connexion et d'implémentation ? La réponse à cette question est donnée par le chapitre 3 qui montre comment il est possible, dans un langage de spécification d'architectures d'implémentation, de décrire des architectures de connexion.

Chapitre 1

L'architecture logicielle

L'architecture logicielle (AL) est un domaine récent de la recherche en informatique. L'étude de ce domaine est devenue nécessaire à partir du moment où la taille et la complexité des logiciels ont engendré des problèmes particuliers : protocoles de communication entre les parties du programme, contrôle global de la structure, partage des fonctionnalités entre des modules de programme, etc. Ces problèmes sont connus depuis les années 60 mais les réponses nécessitent la définition de concepts abstraits et de formalismes pour les gérer qui ne sont pas encore bien définis.

Dans les années 70, l'attention se portait sur le *design* de logiciels en réponse au problème de taille ; les prémisses de la recherche ont été de décider que le design était une activité préliminaire à l'implémentation et demandait de nouvelles notations, de nouvelles techniques et de nouveaux outils. Dans les années 80, la recherche a porté sur l'intégration de la partie design au sein même du processus de développement et sur l'assimilation de cette partie dans les langages de programmation (programming-in-the-large, modules, objets). Une autre tendance de la recherche dans les années 80 a été d'améliorer les techniques de description et d'analyse des systèmes informatiques par des méthodes formelles.

L'évolution naturelle est alors d'intégrer dans ces méthodes formelles des mécanismes de structuration pour mettre en valeur les progrès réalisés dans le domaine du design. Des travaux fondateurs sont apparus au début des années 90 [?, ?, ?, ?, ?] pour précisément justifier que l'AL découle de cette évolution et pourra faciliter la description et l'analyse de systèmes informatiques complexes.

1.1 Motivations

Une question importante à soulever est celle des motivations : en quoi ces mécanismes de structuration sont-ils importants ? Autrement dit, *pourquoi* doit-on étudier ce domaine.

Si on veut essayer de répondre à cette question, il faut d'abord constater que l'évolution qui a conduit à l'étude de ces mécanismes a été engendrée par plusieurs mouvements. Tout d'abord celui provenant des développeurs qui possèdent en commun un répertoire de méthodes, techniques, motifs, jargons, ayant un contenu sémantique plus ou moins précis, qui permettent de La spécification des messages envoyés reprend aussi celle de l'article avec les adaptations minimales :

transmettre des informations sur le logiciel entre les différentes équipes prenant part à son développement. La possibilité de communiquer à un niveau abstrait accélère la répartition des tâches et la connaissance des mécanismes d'architectures autorise une recomposition correcte. Ensuite, par le besoin d'extraire des aspects communs à une famille de produits, ceci afin d'être plus efficace et rapide lorsqu'on voudra développer un nouveau produit appartenant à cette famille. Une famille peut être définie par l'étude d'un domaine particulier (aéronautique, télécommunications, etc), par un standard (architecture de transaction distribuée X/Open, etc), ou d'autres considérations permettant de relier des logiciels entre eux (style).

Le problème de l'évolution est central. En effet, il suffit de voir les problèmes qui se présentent lorsqu'on veut faire évoluer ou customiser un système et de se rendre compte que la plupart de ces problèmes ne sont pas dûs aux algorithmes, mais plutôt spécifiquement à la structure du système. On peut concevoir que ces problèmes sont des conséquences de violations de principes architecturaux sous-jacents et que certaines modifications rendent une architecture inadaptée. Par exemple, si l'architecture d'un système est du type «pipe-filter» séquentiel et qu'on lui rajoute un filtre qui génère une boucle en reliant la sortie de l'un avec l'entrée de l'autre, on ne peux plus dire que l'architecture est du même type. Il faut donc pouvoir être sûr que les éléments que l'on rajoute sont bien conformes à l'idée de départ, car certaines propriétés sont dépendantes de l'architecture suivie (dans le cas de standards tout particulièrement). Par exemple, il est possible de définir des mesures qui permettront de choisir entre des architectures différentes pour un système donné pour améliorer ses performances [?, ?].

La connaissance de l'architecture d'un logiciel est donc un besoin dans le cas où le développement du système demande l'intervention d'un grand nombre de personnes : cela permet de partager les tâches tout en étant assuré d'avoir un système correct, de modifier la structure en connaissance de cause et de vérifier certaines propriétés (qui ne dépendent pas des algorithmes) avant l'implémentation des différentes parties.

1.2 Intuition

Une fois que l'on s'est convaincu que s'intéresser à l'AL est une bonne chose, il faudrait savoir ce qu'est l'AL, soit la question *quoi ?* Or il n'y a pas encore de définition précise de ce qu'est l'architecture logicielle. On rencontre ce mot dans plusieurs circonstances, avec un niveau plus ou moins grand d'abstraction. On peut essayer de se forger une intuition sur ce que doit ou ne doit pas être l'AL. Pour cela on effectue des comparaisons avec les autres domaines pour lesquels le mot «architecture» a un sens. Après ces analogies, on tentera de synthétiser les informations obtenues en une définition de l'AL.

1.2.1 L'architecture de bâtiments

C'est un domaine duquel on peut tirer beaucoup d'informations, car il a des points communs avec ce que l'on recherche et constitue donc une bonne approche.

Le premier point commun est dans l'utilisation d'un plan avant de construire le bâtiment. Ce plan fixe les méthodes et les matériaux à employer selon le but à atteindre et permet d'assigner les différentes tâches aux personnes compétentes. Dans cet objectif de partage du travail, on montrera sur des plans détaillés une seule partie de la construction : le plan des canalisations aux plombiers, le plan électrique aux électriciens, et ainsi de suite. Ces différentes vues de la construction permettent d'étudier à chaque fois un point particulier indépendamment des autres intervenants, tout en étant assuré que les canalisations, par exemple, seront placées au bon endroit à la fin. De la bonne conception du plan découlent enfin des propriétés globales (résistance, poids, possibilité de rajouter des pièces, ...) démontrées une fois pour toutes.

Dans le cas de l'AL, les plans détaillés servent à extraire des informations globales sur le système : plan des connexions, plan des dépendances, ... Chacune de ces particularités suit le plan global, qui permet d'assurer que, par exemple, les connexions se font entre les bonnes parties. Des propriétés comme l'absence de bloquage ou la performance doivent être démontrées au niveau de l'architecture.

Le deuxième point commun est dans la définition de *styles* en tant que méta-information regroupant des bâtiments. Le style est employé comme élément descriptif (en donnant des lignes générales) et prescriptif (en posant des limites).

1.2.2 L'architecture électronique

Dans le domaine de l'architecture électronique, on retrouve l'intérêt d'utiliser différentes approches qui soulignent chacune un aspect particulier de l'architecture, selon que l'on s'intéresse à l'ensemble des instructions (machines *RISC*) ou bien à la configuration (pipeline, multi-processeurs, ...).

On rencontre peu d'éléments de types différents (résistances, condensateurs, processeurs, mémoires) qu'il faut donc multiplier à l'identique pour obtenir de gros systèmes. Ces composants ont tous des caractéristiques et des propriétés connues. Ce dernier point différencie ce domaine avec l'AL puisque pour le logiciel on rencontrera plutôt de très nombreux composants sans duplication. Des tentatives ont été faites [?] pour définir des types de composants logiciels que l'on instancie ensuite pour obtenir le bout de programme désiré (de même qu'il faut choisir la valeur de la résistance que l'on veut utiliser une fois que l'on sait qu'il faut utiliser une résistance). Cette étude des *types* de composants permettrait de ne faire figurer dans l'architecture du logiciel que ces types avec leurs caractéristiques et leurs propriétés.

Une autre analogie possible avec l'architecture électronique est de concevoir l'architecture d'un logiciel comme étant le circuit imprimé sur lequel viendra se greffer les composants. Cette analogie s'appuie sur la séparation complète entre structure et comportement, ce qui n'est pas vraiment possible dans le cas de l'AL mais qui a l'avantage de nous aider à percevoir l'architecture comme étant une base solide pour le développement.

1.2.3 L'architecture réseau

Dans ce domaine on ne considère que des graphes représentant des configurations topologiques (bus, étoile, anneau, ...) dans lesquels les composants (principalement des machines) sont abstraits sous forme de noeuds.

Cette représentation d'un ensemble complexe sous forme graphique permet de mieux comprendre l'organisation du réseau et peut être reproduit au niveau des programmes. Cependant si l'on veut démontrer des propriétés sur un système, les informations contenues dans un graphe sont souvent insuffisantes. Certains chercheurs essayent de définir la structure minimale à apporter à cette représentation afin de pouvoir y effectuer des manipulations formelles [?]. Cette partie de la recherche a débouché sur la définition de langages de configuration dans lesquels on ne s'intéresse qu'au squelette du programme, cf. Darwin [?, ?].

Les configurations types des réseaux de communications peuvent être utilisées aussi pour l'AL toujours dans l'idée de regrouper des descriptions ayant certaines caractéristiques communes.

1.2.4 Synthèse

On peut essayer maintenant de synthétiser les trois intuitions en une définition de l'AL :

- de l'architecture de bâtiments, on retiendra l'utilisation de vues multiples, la possibilité de démontrer des propriétés avant la construction et de chercher à caractériser un bâtiment selon un *style* ;
- de l'architecture électronique, on retiendra, en plus de la nécessité d'utiliser différentes vues, la définition de *types* de composants ;
- de l'architecture réseau, on retiendra la représentation simplifiée sous forme de graphes, dans lesquels les noeuds correspondent à la structure minimale des composants : leur *interface*.

Voici une définition de ce que peut être l'architecture d'un logiciel :

Définition 1 *L'architecture d'un logiciel reflète sa structure de différentes manières, du texte aux graphes, et pour laquelle on ne précise que les types de composants par leur interface.*

1.3 Place de l'architecture

On essaye maintenant de préciser le contexte dans lequel l'AL interviendra : à quel moment du cycle de développement étudie-t-on l'architecture du système ? Cela correspond à la question : *quand ?* On peut envisager le déroulement suivant :

1. On aura en premier lieu le cahier des charges ou les exigences à satisfaire pour un programme/système donné.
2. On spécifie ensuite quelle architecture on utilise avec quels matériaux.
3. Puis vient le découpage du programme en modules, la définition des interfaces, les types de données utilisés.
4. Enfin on implémente les algorithmes satisfaisant la spécification globale.

Pourtant lorsqu'on regarde ce déroulement, une question importante émerge : doit-on connaître à l'avance la structure de tout le système ? Il y a évidemment une continuité entre les points, cependant on aimerait avoir un moyen de reproduire dans ce déroulement un aspect essentiel du développement : la construction progressive.

Pour pouvoir répondre à cette question, il faut tout d'abord introduire des concepts généraux.

1.3.1 Style

Un style est une abstraction de plus au dessus de l'AL, puisque l'on considère la plupart du temps qu'un style reprend des caractéristiques de plusieurs architectures, comme le vocabulaire, la topologie ou des propriétés particulières, pour constituer une famille bien définie. Ces caractéristiques imposent alors des contraintes sur les architectures, comme le font des standards, mais la définition d'un style n'est pas aussi précise car elle fait appel à l'intuition et à des connaissances communes.

Des exemples de style sont donnés dans [?], par exemple :

- le pipe-filter, dans lequel la transformation des données se fait séquentielement par des filtres indépendants ;
- les systèmes de couches, typique des protocoles de communication (TCP/IP) ;

– le Client-Serveur, dans lequel un nombre inconnu de clients essaye de se connecter à un ou plusieurs serveurs ;
– ...

Par opposition au style se trouve l'instance, qui elle sera précisément l'architecture du système étudié. On retrouve un phénomène connu au niveau des langages de programmation : le typage (le style jouant le rôle du type abstrait).

Des travaux ont porté sur les concepts permettant de classifier les styles architecturaux et de les utiliser lors du développement [?, ?]. Dans [?], M. Shaw donne des catégories de traits discriminant les styles :
– quels sont les sortes de composants et de connecteurs utilisés (voir [?]) ;
– comment le contrôle est partagé et transféré entre les composants ;
– comment les données sont communiquées dans le système ;
– quel type de raisonnement est compatible avec le style.

Cette notion de *style* pour les logiciels possède les mêmes caractéristiques que pour le bâtiment : il est à la fois élément descriptif (en donnant des lignes générales) et prescriptif (en posant des limites). Dans le contexte d'un développement progressif, il est intéressant de commencer par choisir à priori un style simple (séquentiel, ...), ce qui peut se faire très tôt et avec lequel on pourra démontrer certaines propriétés de façon plus facile (par exemple la relation entre l'entrée et la sortie d'un programme). Si on veut améliorer l'architecture, il est possible ensuite d'en modifier la structure si on peut effectuer un *raffinement* d'architectures.

1.3.2 Raffinement

Ce paragraphe fait le lien entre cette section et la suivante. En effet, le concept de *raffinement* est propre aux langages de spécification, c'est-à-dire qu'il est lié à des définitions formelles qu'on précisera plus loin. Nous introduirons seulement ici quelques termes essentiels à l'exposé du concept.

L'idée du développement progressif (stepwise) peut s'appliquer au cas d'architectures, on parlera alors de raffinement *global*, contrairement au raffinement *local* cherchant plutôt à préciser le contenu des composants en leur ajoutant des caractéristiques au fur et à mesure. Ce dernier est utilisé par la plupart des formalismes de spécification pour définir un composant : par exemple, l'ensemble des entiers relatifs se construit à partir d'une spécification des entiers (en disant que les entiers sont générés par 0 et par une opération de successeur), à laquelle on rajoute une opération caractéristique des entiers relatifs (l'opération de prédécesseur).

En ce qui concerne le raffinement global, il est possible d'utiliser les mêmes idées, en y apportant quelques modifications. Le principal travail sur le raffinement d'architectures a été effectué par Moriconi et Qian dans [?, ?]. L'exemple donné dans leurs articles est celui d'un compilateur dont on donne deux représentations : la première séquentielle ne donnant que le type de données transmises entre deux composants, la seconde faisant intervenir de la concurrence entre les différents modules et précisant la communication entre les composants. On voit dans cet exemple que le raffinement a lieu entre deux *styles* aussi bien qu'entre deux architectures. On peut donner un exemple de raffinement : le flot de données entre deux composants (le lexeur et le parseur) de la première est raffiné par une variable partagée : l'arbre construit par le lexeur est stocké dans une mémoire dans laquelle le parseur a accès.

Le raffinement global permet donc de passer d'un style à un autre de manière formelle et de concrétiser la continuité évoquée entre les différents points du développement.

1.4 Spécification

Ce paragraphe s'intéresse à la question *comment ?*, i.e. comment formaliser les notions des sections précédentes dans des langages formels. Ce besoin provient de deux tendances dont nous avons parlé dans l'introduction : la nécessité de s'intéresser à la structure et l'obligation d'utiliser des méthodes formelles dans le développement de systèmes critiques. Il existe un grand nombre de langages s'intéressant à la description des architectures : Rapide [?], Aesop, MetaH, UniCon[?], Darwin [?, ?], Wright [?, ?], C2, SADL [?, ?], CHAM [?, ?], COMMUNITY [?, ?], CASL [?]. Ces langages utilisent deux sortes de mécanismes que nous présentons dans le paragraphe suivant, mais tout d'abord nous essayons de savoir ce qu'ils doivent décrire.

1.4.1 Impératifs

Les langages de spécification doivent refléter les points mis en valeur par les comparaisons avec les autres domaines de l'architecture. Un point essentiel est la composition (ou recomposition) du programme après implémentation : on aimerait être assuré que tous les composants sont *compatibles*. Pour cela, nous définissons comment les spécifications doivent être écrites pour faciliter la composition, puis quels sont les mécanismes pour composer des spécifications. La plupart des définitions proviennent de [?].

Dans la définition que l'on a donné de l'architecture d'un logiciel, on a cité l'*interface* d'un composant. Voici une définition pour une notion d'interface *complète* :

Définition 2 *Une interface complète d'un composant doit comporter l'ensemble des traits proposés avec une spécification de leurs comportements, l'ensemble des traits nécessaires muni aussi d'une spécification de leurs comportements, ainsi qu'une spécification des interactions mutuelles entre des traits de catégories différentes.*

Cette définition permet de vérifier le principe de «Communication Integrity» : tous les liens entre les composants sont précisés dans leur interface. Ce principe permet d'éviter des problèmes lors de la recomposition finale des composants. Par exemple, si on raffine un composant en lui rajoutant des fonctionnalités, il doit être possible de voir rapidement (au niveau architectural) quel autre composant sera affecté par ce raffinement. Un autre problème pouvant intervenir est l'échange entre deux implémentations d'une même spécification : on doit être sûr qu'il peut être effectué sans dommages. On peut exprimer ce principe par la formule suivante dans laquelle la relation $\equiv$ signifie l'égalité syntaxique et $\sim$ une similitude sémantique [?] :

$$\forall x, y : programme. \forall S : systeme. \forall I : interface \bullet I(x) \equiv I(y) \Leftrightarrow S[x/y] \sim S(x)$$

Il est important de savoir que la composition des programmes se passera *bien*. Le principe décrit ci-dessus donne déjà une certaine indication sur la façon dont les spécifications doivent être écrites pour faciliter la composition, mais il ne donne pas le mécanisme en lui-même. Pour cela, deux stratégies sont possibles :

- la «Connectabilité» : ce principe définit la composition entre deux composants comme une compatibilité syntaxique, par des propriétés de typage, ou de signature. Si une interface contient des fonctions typées, on peut définir la connectabilité comme une relation entre les types des fonctions de plusieurs interfaces (f est une fonction proposée, g est une fonction requise) par : $Type(f) = Type(g)$ ou $Type(f) <: Type(g)$.

- la «Connection correcte» : ce principe suppose la donnée d'une sémantique formelle au langage, car il se base sur la conservation de certaines propriétés sous-jacentes à cette sémantique. Si on poursuit l'exemple ci-dessus, on a : si $f :\rightarrow Nat$ et $g :\rightarrow Int$ mais que g ne rende que des nombres premiers et f que des nombres pairs, alors remplacer g par f ne convient pas pour ce principe, bien que ces deux fonctions soient *connectables*.

Les différents langages de spécification que nous décrirons dans le chapitre suivant ont pris des directions différentes par rapport à ces principes, que nous détaillerons alors. Nous allons décrire tout d'abord les deux mécanismes principaux concernant la description des architectures par des méthodes formelles.

1.4.2 Architectures d'implémentation

Lorsqu'on veut refléter dans un langage de spécification la structure d'un système, une première possibilité consiste à se servir des mécanismes des structuration des langages de programmation (Ada, Standard ML, ...). Ces mécanismes (modules, objets) sont définis plus ou moins formellement selon les langages, il faut donc les simplifier quelque peu pour pouvoir les intégrer dans un langage formel.

Si l'on reprend le plan de la section 1.3 on se rend compte qu'une fois le cahier des charges en main, l'écriture d'une spécification formelle décrivant le système nécessite des mécanismes de structuration puissants. Le premier pas dans cette démarche sera la plupart du temps de décrire un diagramme de style «boîte et lignes» donnant une première idée de comment sera organisé le logiciel. On espère ensuite qu'une architecture spécifie les modules ainsi que la manière dont on les assemble pour satisfaire les demandes. On peut donner une définition générale sur ce que peut être une architecture d'implémentation :

Définition 3 *Une architecture d'implémentation est une architecture construite à partir des mécanismes prévus par des langages de programmation dans laquelle on ne spécifie que les «boîtes».*

On peut citer comme exemple de langage utilisant le concept d'architecture d'implémentation : CASL [?] ou ExtendedML [?]. Ce concept comporte plusieurs avantages :

- sa facilité de compréhension : une fois que l'on connaît les mécanismes proposés par les langages de programmation, l'adaptation aux mécanismes du langage de spécification est rapide ;
- la notion de raffinement local est bien connue pour ces langages, ce qui permet de retourner aux langages de programmation de façon formelle ;
- l'indépendance des composants est assurée au niveau des spécifications, donc l'échange d'implémentations est possible, ce qui est un point essentiel dans l'étude de l'architecture.

Cependant, les mécanismes de structuration des langages de programmation ont aussi des limitations vis-à-vis des idées exposées précédemment. Il faudrait corriger ces défauts pour définir un formalisme totalement conforme. Par exemple, une partie de la structure d'un système est cachée à l'intérieur des modules (ou tout autre mécanisme) : en effet on découpe un programme en unités d'exécution avec des interfaces pour communiquer les unes avec les autres, principalement par *définition-use* (une fonction peut être définie dans un autre module que celui où elle est utilisée) ; puis on peut combiner ces unités par *name-matching* (le nom d'une librairie est donné dans laquelle on va chercher les définitions, par exemple dans un header). Le principe de «Communication Integrity» n'est pas vérifié car certaines fonctionnalités requises ne sont pas explicites, mais nous verrons dans le chapitre 3 comment

CASL corrige ce problème.

Un second inconvénient est que certains types de connexions sont plus complexes que ceux proposés par les langages de programmation. De plus, la composition des modules, par exemple dans une expression de modules ou un fichier make, est limitée aux faits : la nature de la connexion est inconnue. Si on a besoin en plus de reconfigurer dynamiquement le système, aucune facilitée n'est proposée.

Un des objets de ce mémoire est de montrer comment faire pour exprimer des types de connexions complexes ainsi que de préserver la nature des connexions lors de la composition dans un langage de spécification suivant ce concept d'architecture d'implémentation, en l'occurrence CASL.

1.4.3 Architectures de connexions

Comme nous l'avons vu précédemment, une spécification complète doit utiliser différentes techniques, du dessin en passant par du texte jusqu'aux formules mathématiques. Une fois le diagramme «boîte et lignes» écrit, une possibilité est d'essayer de formaliser les concepts de «boîtes» aussi bien que ceux de «lignes». Cette idée caractérise le concept d'architecture de connexions. Le point important de ce concept est le fait de décrire les mécanismes de communication, de les expliciter pour en faire des citoyens de première classe. À ce niveau d'abstraction, on modélise la plupart du temps le comportement du système comme un ensemble de processus concurrents. Voici une définition d'une architecture de connexions :

Définition 4 *Une architecture de connexions est une architecture construite à partir de la description informelle du cahier des charges sous forme de diagramme, dans laquelle on spécifie à la fois les «boîtes» et les «lignes». Le comportement du système est vu dans le paradigme concurrent.*

Des exemples de langages suivant ce concept sont donnés dans le chapitre suivant. On regardera alors leur comportement par rapport aux idées décrites auparavant sur la recomposition des spécifications.

On peut dégager des avantages dans l'utilisation de ce concept :

- la facilité de description : après avoir donné un diagramme informel décrivant l'organisation du système, le passage à une écriture formelle est simplifié si on a la possibilité de spécifier aussi les connexions entre les boîtes. Les descriptions contiennent toute l'intuition des ingénieurs à un plus ou moins haut niveau, ce qui permet aussi la communication entre les différentes équipes ;
- la séparation claire entre la *coordination* (ensemble des principes gouvernant les relations entre des composants) et le calcul : en effet les spécifications apparaissant à ce niveau ne concerneront que les modes de communication (synchrones par appels de fonctions ou asynchrones par génération d'événements) entre les composants. Ceci peut nécessiter cependant de rajouter des informations de comportement pour se rapprocher de la notion d'interface complète ;
- la possibilité de décrire des architectures dynamiques : on distingue deux formes de dynamisme au niveau architectural. Soit les composants restent identiques mais les interactions changent (calcul mobile), soit les composants changent mais les interactions restent identiques (lorsque plusieurs clients se connectent à un serveur). Ces modifications interviennent au niveau de l'exécution mais doivent aussi se présenter au niveau de la spécification, car elles peuvent modifier la coordination (particulièrement quand

de nouveaux composants sont introduits). Ce trait du concept de connexion le distingue particulièrement du concept précédent ;
- l'analyse : si on veut vérifier des propriétés sur le système, on a besoin d'un certain niveau d'abstraction dans lequel les propriétés principales sont conservées, et surtout dans lequel cette analyse est possible malgré la complexité. Cette analyse sera simplifiée si on donne de l'importance aux connexions. Par exemple, si on veut vérifier qu'un flot de données est bien transmis d'un filtre à un autre dans une architecture pipe-filtre, il suffit de vérifier une seule fois que le connecteur (pipe) est correct vis-à-vis de cette propriété puis que les connexions sont aussi correctes selon un des deux principes évoqués dans la section précédente.

Ce concept présente donc des avantages importants par rapport au précédent, mais il y a un point qui fait que, de toutes façons, le premier concept reste fondamental. Une fois que nous sommes en possession de la spécification de l'architecture du système étudié sous forme *connective* et que des propriétés ont été démontrées sur cette spécification, comment peut-on utiliser cette spécification pour construire le programme ? Le raffinement de cette spécification débouchera forcément sur une spécification de style *implémentation* qui elle se projettera aisément et formellement sur un langage de programmation. Les spécifications des connexions seront raffinées progressivement d'un style à l'autre jusqu'à être diffusées à l'intérieur des spécifications des «boîtes». Ce raffinement global n'est pas tout le temps possible ce qui donne un handicap certain au formalisme ne le permettant pas.

Conclusion

En conclusion, une spécification architecturale doit permettre de décrire une architecture dans plusieurs vues différentes, d'être simple à écrire à partir du cahier des charges, de permettre un développement indépendant des différentes parties puis de les composer formellement, de séparer la *coordination* du calcul pour pouvoir effectuer une analyse de dépendance et de conformance (quelles sont les interdépendances entre les architectures, entre les parties d'une architecture ? Est-ce qu'une architecture est conforme au style choisi ?) et d'être raffinable vers un modèle exécutable.

Une illustration de ces deux mécanismes de formalisation des idées architecturales est donnée dans les chapitres suivants. Le chapitre 2 donne trois exemples de langages de spécifications suivant l'approche *connective*, le chapitre 3 montre les principes d'architectures d'*implémentation* dans le langage CASL.

Chapitre 2

Langages de description d'architectures (ADL)

Dans ce chapitre, nous décrivons trois langages de spécification basés sur la formalisation des architectures sous forme connective. Notre choix s'est porté sur Wright, car ce langage développé par Allen et Garlan fut un des premiers à poser des bases pour la vision connective ; COMMUNITY, car la structure catégorique de ses programmes en fait un modèle général ; CHAM, car le principe de configuration défini dans ce langage nous a paru intéressant. Le principe de ce chapitre est de donner pour chacun de ces langages une description de sa syntaxe et de sa sémantique (peu détaillée), puis d'écrire les spécifications correspondant à un exemple décrit plus loin, et enfin de les comparer par rapport à un cadre défini ci-dessus.

Il existe déjà plusieurs surveys sur les ADL [?, ?, ?] et des articles synthétiques sur les besoins des ADL [?], dans lesquels un cadre est donné pour permettre une comparaison entre les différents langages. Ce cadre a été employé en partie pour mettre en évidence les caractéristiques des architectures de connexions dans le chapitre précédent, en voici les traits essentiels :

- connecteurs : est-ce que le langage propose une construction explicite pour les connecteurs ?
- interfaces : comment sont définies les interfaces des composants ? On cherchera à savoir si les interfaces sont complètes, c'est-à-dire si elle notifient les fonctionnalités des composants, les fonctionnalités nécessaires au composant, ainsi qu'une relation entre ces deux types de fonctionnalités.
- configuration : est-elle implicite ou explicite ?
- composition : suit-elle le principe de «connectabilité» ou de «connexion correcte» (cf. Chapitre 1) ?
- modularité : a-t-on un moyen de structurer les spécifications ?
- raffinement : le langage possède-t-il un mécanisme de raffinement formel appuyé sur sa sémantique ?
- dynamisme : peut-on spécifier des architectures dynamiques ?
- analyse : quel type d'analyse peut-on effectuer sur les spécifications ?

Les informations seront compilées dans un tableau récapitulatif à la fin de ce chapitre.

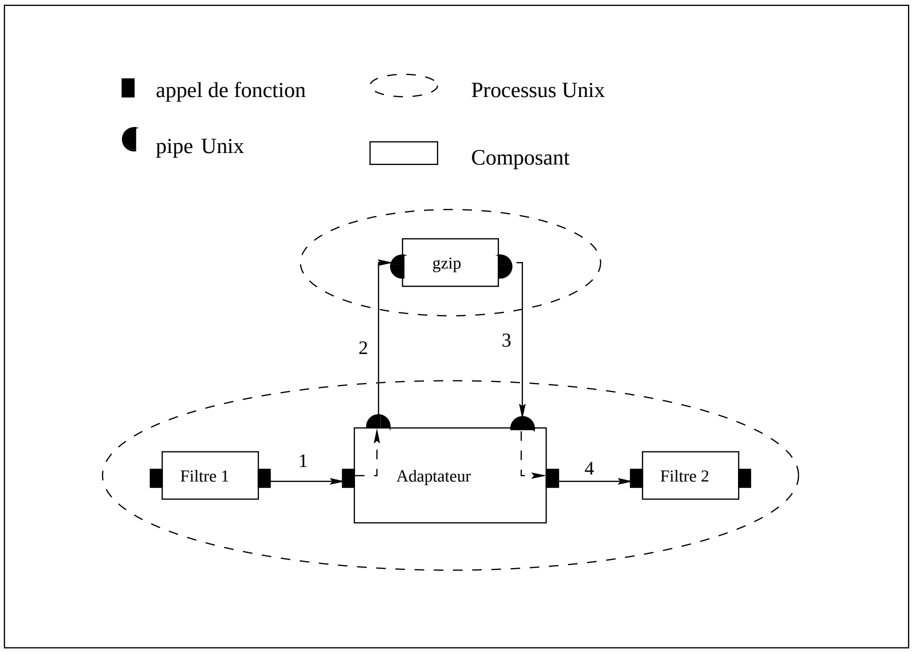


Fig. 2.1 – Compressing Proxy

Le problème du Compressing Proxy

Afin de pouvoir comparer ces formalismes, on utilisera un exemple commun, relativement simple, décrit informellement à partir de [?].

Pour améliorer la performance des transactions des browsers du WWW sur des réseaux lents, basés sur Unix, une possibilité est de créer un serveur HTTP qui compresse les données envoyées sur le réseau et les décompresse à la lecture. Le but du Compressing Proxy est d'ajouter le programme de compression/décompression **gzip** au standard HTTP.

Le diagramme «boîtes et lignes» du standard HTTP est formé des différentes fonctionnalités du programme rangées dans des filtres reliés par des lignes modélisant les appels de fonctions entre ces filtres. En effet, la transformation des données par le standard HTTP se fait par l'application de fonctions sur ce flot de données, qui passe donc successivement d'un filtre à un autre. Il est donc naturel de considérer que les filtres procèdent par lecture/écriture des données à travers ce flot et que les lignes représenteront ces lectures/écritures. On dira que le filtre F_2 «lit» des données dès que le filtre F_1 appelle une fonction présente dans l'interface de F_2. Le filtre F_1 a aussi la possibilité de fermer le flot. Puisque le moyen de communication est basé sur l'appel de fonctions, tous les filtres doivent être contenus dans un même processus Unix.

Le programme **gzip** peut aussi être considéré comme un filtre, mais au niveau d'un processus Unix ; il utilise donc l'interface standard entrée/sortie d'Unix (*pipe*). Une différence importante entre **gzip** et les filtres du serveur HTTP est que **gzip** choisit quand il veut lire tandis que les filtres du serveur HTTP sont obligés de lire lorsqu'un filtre supérieur leur envoie des données.

Pour pouvoir assembler le Compressing Proxy à partir des filtres du serveur HTTP et du programme **gzip**, il est nécessaire de créer un adaptateur (voir figure 2.1). Cet adaptateur sert à coordonner les interfaces : il agira comme un pseudo-filtre qui communiquera avec les filtres par un appel de fonction et avec **gzip** par des *pipes*.

Sans une compréhension juste des hypothèses faites par chaque composant, une erreur dans le déroulement peut se produire quand on intègre tous les composants dans un seul système. On peut décider de créer l'adaptateur de telle sorte qu'il transmet des données à **gzip** dès qu'il en reçoit du filtre précédent et qu'une fois le flot fermé par le filtre précédent, l'adaptateur lit les données compressées provenant de **gzip** et les pousse dans le filtre suivant. Cette vision de l'adaptateur est incorrecte puisqu'un bloquage peut surgir.

Le problème provient de ce que **gzip** utilise un algorithme en une passe et peut essayer d'écrire une partie des données (si son buffer interne est plein) avant que l'adaptateur soit prêt et ainsi bloquer le système. Une façon de résoudre le problème est d'autoriser l'adaptateur à manipuler les données incrémentalement et d'avoir des lectures/écritures concurrentes, ce qui permettra à l'adaptateur de lire des données de **gzip** lorsqu'il ne peut plus lui écrire.

L'avantage de cet exemple est d'avoir une solution bien comprise et de mettre en relief les concepts de connexion, d'interface et de connexion correcte. Il représente de plus un problème typique de l'évolution d'un protocole déjà existant ainsi qu'un exemple problématique d'intégration lié à l'architecture. Nous regarderons pour chacun des langages si le problème est présent, mais nous ne donnerons pas les solutions à apporter pour le corriger.

2.1 Wright

Ce langage a été proposé par David Garlan et Robert Allen en 1994 [**?**, **?**]. C'est un langage de spécification d'architectures d'interaction, dont la plupart des idées ont été mises à profit en ce qui concerne la description d'architectures de connexion. Wright est une construction au-dessus de CSP [**?**], que les auteurs ont utilisé car ce langage permet d'exprimer le comportement dynamique de la communication inter-composants, de spécifier quel composant sera responsable d'une prise de décision, de composer le système facilement (garder une trace de chacun des composants après la composition et pouvoir raisonner séparément sur chacun d'entre eux), et enfin de pouvoir vérifier que le système ne sera pas bloqué lors de son exécution (la sémantique de CSP est fully-abstract par rapport au deadlock [**?**]).

2.1.1 Le langage

Voici une description abstraite d'architecture en Wright, pour spécifier la communication entre deux filtres par un pipe :

System Example

  Composant Filter

    port provide *[provide protocol]*

    port request *[request protocol]*

    spec *[Filter specification]*

  Connector Pipe

    role input *[input protocol]*

    role output *[output protocol]*

    glue *[glue protocol]*

Instances
    f1, f2, f3 : Filter ;
    p1, p2 : Pipe Connector.
Attachments
    f1.provide **as** p1.input ;
    f2.request **as** p1.output ;
    f2.provide **as** p2.input ;
    f3.request **as** p2.output.
end.

La première partie décrit des types de composants et de connecteurs. Le type d'un composant est formé d'un ou plusieurs *ports* considérés comme autant de points d'interaction avec le système, et d'une spécification de son fonctionnement. Les ports constituent l'interface du composant, c'est-à-dire l'ensemble des fonctionnalités que l'on décide d'observer ; la spécification demandée pour chaque type de composant précise les relations entre ces fonctionnalités.

Le type d'un connecteur est formé de *rôles* décrivant le comportement local des parties en jeu (les événements des ports que l'on décide d'observer), et d'une *glue* expliquant la coordination entre les activités des rôles. La principale distinction entre un composant et un connecteur est qu'un connecteur ne possède pas de fonctionnalité propre, c'est-à-dire qu'un connecteur est une spécification d'un composant qui n'effectue pas de *calcul*.

En faisant une analogie avec l'électronique, un connecteur peut être comparé à un câble reliant les composants ; dans ce cas les rôles correspondent aux prises mâles avec un certain format, la glue décrit comment l'information sera gérée par ce câble et les ports correspondent aux prises femelles. Un point important à préciser dans cette analogie, c'est qu'il existe plusieurs types de prises femelles pouvant être branchées sur ce câble. En effet les types femelles compatibles doivent avoir *au minimum* les orifices complémentaires des plots proposés par le connecteur, rien n'empêche qu'un composant ait des prises femelles plus complexes ou même plusieurs prises femelles qui ne sont pas toutes utilisées. Cette possibilité permet de plus aux composants de communiquer avec plusieurs composants à la fois en reliant leurs prises par différents connecteurs.

Les instances servent à la configuration du système qui est donnée par les *Attachments* qui relient les connecteurs aux composants en associant les rôles aux ports. Nous verrons plus loin le principe de composition utilisé à cet endroit.

Le principe de «Communication Integrity» est vérifié par ce langage puisque les seules connexions autorisées entre les composants sont celles définies par leurs interfaces. La formalisation de la notion de connecteur permet de rester très proche du diagramme informel schématisant le cahier des charges et l'utilisation de types autorise une réutilisation optimale des spécifications.

Les spécifications données dans les composants et dans la glue sont écrites en CSP, dont elles n'utilisent qu'un sous-ensemble de constructions :
- événements : e. Ces événements montrent les aspects de la modélisation auxquels on s'intéresse, ils peuvent transmettre des données (e !x) ou en recevoir (e ?y).
- préfixe : $e \rightarrow P$. Cela signifie que le processus Q attend l'action e puis se comporte comme le processus P. On note § le processus d'arrêt avec succès.
- choix externe (déterministe) : $P \,[]\, Q$. Le choix entre P et Q est effectué par l'environnement de ce processus.

– choix interne (non-déterministe) : P $\sqcap$ Q. Le choix est fait par le processus lui-même.
– composition parallèle : P $\parallel$ Q. Cet opérateur synchronise les actions par noms, celles qui ne sont pas communes peuvent avoir lieu indépendamment.

2.1.2 Sémantique

La sémantique du langage Wright est entièrement plongée dans CSP : les constructions spécifiques du langage comme les composants ou les connecteurs ont une sémantique basée sur l'opérateur de composition parallèle de CSP.

Un processus CSP comportant possiblement du non-déterminisme a pour sémantique un triplet ($\mathcal{A}$, $\mathcal{F}$, $\mathcal{D}$), qui correspondent respectivement à son alphabet $\mathcal{A}$ (les actions qui sont prises en compte par le processus), à son ensemble de traces *failures* $\mathcal{F}$ (celles qui mèneront à un bloquage), et son ensemble de traces *divergentes* $\mathcal{D}$ (celles après lesquelles le processus agira de façon chaotique, c'est-à-dire de façon totalement imprévisible. Le processus chaotique est l'élément neutre pour toutes les opérations sur les processus utilisées dans Wright).

On a fait l'hypothèse dans la section précédente que la glue et les rôles se combinaient de telle sorte que la glue coordonne le comportement des rôles. Pour pouvoir formaliser cette hypothèse dans CSP, il suffit de considérer les rôles comme étant des processus indépendants contraints uniquement par la glue. La sémantique d'un connecteur est donnée par le processus CSP :

$$glue \parallel R_1 \parallel .... \parallel R_n,$$

où *glue* est le *[glue protocol]* et les R_i sont les *[roles protocol]*. On suppose que l'alphabet de la glue est constitué de l'union des alphabets de tous les rôles.

Lorsqu'on voudra connecter des composants avec un connecteur, la signification de la connexion suit celle du connecteur. La sémantique d'un système est donnée par :

$$glue \parallel P_1 \textbf{ as } R_1 \parallel .... \parallel P_n \textbf{ as } R_n.$$

où les P_i sont les *[port protocol]* liés aux rôles R_i et «**as**» est l'opération d'attachement. On discutera plus loin de quel principe de connexion il s'agit.

2.1.3 Compressing Proxy

En considérant la description du problème, on dégage quelques idées pour la construction de la spécification. Il y aura deux types de composants : un type pour les filtres, un autre pour **gzip**. Les communications entre les composants se passent de deux façons :
– par lecture/écriture entre les filtres et l'adaptateur ;
– par des *pipes* entre **gzip** et l'adaptateur.

On peut synthétiser ces deux formes de communications dans un unique type de connecteur en considérant que l'adaptateur gère ces communications. Dans un esprit de simplification, le comportement des communications entre **gzip** et l'adaptateur sous forme de *pipe* ne sera pas précisé (ce sera une simple entrée/sortie sans tampon). On laissera au lecteur le soin de compléter les spécifications suivantes.

Filtres

On trouvera deux ports pour le type des filtres : un port de lecture composé d'un événement (*lit*) associé à l'appel d'une fonction du filtre et de la réponse à un événement (*fermeture*) émis par le filtre précédent, un port d'écriture composé d'un événement (*écrit*) associé à l'appel d'une fonction du filtre suivant et d'un autre signifiant la fin de la transmission (fin). La spécification du filtre est simplement esquissée, car pour mettre en évidence le problème architectural, une spécification précise apporterait trop de détails. On ne précise que l'événement *calcul* représentant la fonctionnalité du filtre.

La différence entre le choix déterministe ($[]$) et le choix non-déterministe ($\sqcap$) reflète le fait que les filtres sont obligés de lire dès qu'on leur soumet des données (choix $[]$) ; en revanche, ils ont la possibilité de choisir quand ils veulent écrire (choix $\sqcap$).

Composant Filtre =

port Lecture = lit ?x $\rightarrow$ Lecture $[]$ fermeture $\rightarrow$§

port Écriture = écrit !y $\rightarrow$ Écriture $\sqcap$ fin $\rightarrow$ §

spec Filtre = lit ?x $\rightarrow$ calcul $\rightarrow$ écrit ?y $\rightarrow$ Filtre $[]$ lit ?x $\rightarrow$ fermeture $\rightarrow$ calcul $\rightarrow$ écrit !y $\rightarrow$ fin $\rightarrow$ §

end.

Gzip

Le composant GZip possède deux ports pour l'entrée et la sortie des données du programme. Contrairement aux filtres du protocole HTTP, **gzip** choisit indépendamment quand il veut lire ou écrire : le choix est non-déterministe dans les deux cas.

Les événements $\mathbf{end}_i$ et $\mathbf{end}_o$ servent à la communication entre l'adaptateur et GZip en plus des transmissions de données, pour se synchroniser la fin des transmissions avec l'adaptateur. On modélise la possibilité qu'a GZip d'interrompre la transmission des données de façon autonome par des actions particulières (fin_i et fin_o).

Dans la spécification du composant GZip on ne s'intéresse pas à l'algorithme de compression, mais uniquement à la manière dont se déroule la lecture puis l'écriture des données.

Composant Gzip =

port Entrée = (in !x $\rightarrow$ Entrée $[]$ $\mathbf{end}_i$ $\rightarrow$ §) $\sqcap$ fin_i $\rightarrow$ §

port Sortie = (out ?y $\rightarrow$ Sortie $\sqcap$ $\mathbf{end}_o$ $\rightarrow$ §) $\sqcap$ fin_o $\rightarrow$ §

spec gzip =

let P = (Sortie.out ?y $\rightarrow$ P $\sqcap$ $\mathbf{end}_o$ $\rightarrow$ **gzip**) $\sqcap$ fin_o $\rightarrow$ **gzip**

in (Entrée.in ?x $\rightarrow$ **gzip** $[]$ $\mathbf{end}_i$ $\rightarrow$ P) $\sqcap$ Entrée.fin_i $\rightarrow$ P)

end.

Adaptateur

L'adaptateur joue le rôle du connecteur dans cette modélisation. En effet, son comportement est totalement non-fonctionnel, son utilité est seulement de présenter une prise dans le protocole HTTP sur laquelle viendra se greffer GZip. C'est par lui que transitent toutes les données et qui fait le lien entre les deux processus Unix impliqués dans la conception du système.

Ce connecteur possède quatre rôles qui sont coordonnés par la **glue**. Le protocole de communication créé pour le Compressing Proxy est contenu dans la **glue** de ce connecteur et sera donc le centre du problème de bloquage. L'événement *prêt* signale la transition de l'adaptateur entre l'écriture des données dans GZip et la lecture des données provenant de GZip.

Les connecteurs dans le langage Wright servent en plus à identifier le nom des actions des composants qui seront mises en communication, c'est pourquoi on retrouve dans l'écriture de ce composant toutes les actions utilisées dans les spécifications précédentes. Afin de préciser le comportement du connecteur, il est possible de rajouter une spécification de traces précisant le lien entre les modes de communications utilisés par les composants et donc permet de différencier ces modes de communication. Une spécification de traces est un prédicat devant être vérifié par toutes les traces de la glue. Cette spécification doit exprimer le fait qu'entre la valeur reçue (y) par l'adaptateur provenant du filtre F_1 qui est un argument de fonction (donc transmis en une seule fois) est segmenté puis transmis en fragments à GZip par la valeur x (et inversement entre GZip et le filtre F_2). La transmission entre l'adaptateur et GZip se fait donc itérativement, ce qui peut s'exprimer au niveau de l'adaptateur par : *entre deux occurrences de l'événement écrit ?y, la concaténation des valeurs x des événements in !x doit être égale à y.*

Connecteur Adaptateur =
    role Un = écrit ?y → Lecture [] fin →§
    role Deux = in !x → Sortie [] **end**$_i$ → §
    role Trois = out ?y → Sortie [] **end**$_o$ → §
    role Quatre = lit !x → Écriture ⊓ fermeture →§
    glue =
        let Q = Trois.out ?y → Q [] **end**$_o$ → Quatre.lit !x → **glue**
        in let P = Deux.in !x → P ⊓ **end**$_i$ → prépare → Q
        in Un.écrit ?y → P
    spec ⋯
end.

Spécification Wright

Une fois les types de composants et les types de connecteurs présentés, la spécification du Compressing Proxy s'écrit :

System Compressing Proxy
    Composant Filtre *[spécification du type des filtres]*
    Composant Gzip *[spécification du type de GZip]*
    Connector Adaptateur *[spécification du type de l'adaptateur]*
Instances
    f1, f2 : Filtre ;
    A : Adaptateur ;
    G : Gzip.

Attachments

f1.Écriture **as** A.Lecture ;
A.Sortie **as** G.Entrée ;
G.Sortie **as** A.Entrée ;
A.Écriture **as** f2.Lecture.

end.

Les connexions entre les composants se font par une identification du nom des actions entre les composants et le connecteur. Cette identification a lieu dans le connecteur, comme nous l'avons vu ci-dessus. Dans la discussion ci-après, nous proposons une modification de la syntaxe du langage permettant de se dégager de cette contrainte.

Pour visualiser le comportement du système, il suffit d'enregistrer tous les événements qui ont été déclenchés au cours du déroulement de l'action. Cet enregistrement s'appelle la trace du système. La trace suivante exhibe un comportement possible du système menant à un bloquage.

Composant	**Événement**
Filtre 1	Écriture.écrit !y
Adaptateur	Un.écrit ?y, Deux.in !x
Gzip	Entrée.in ?x, Entrée.fin$_i$, Sortie.out !y

Après l'événement fin apparu dans le composant Gzip, le système se bloque puisque Gzip ne peut pas transmettre des données à l'adaptateur car celui-ci n'a pas envoyé son signal de fin de transmission (**end**$_i$).

La section suivante expose les possibilités de Wright quant aux points du cadre d'étude des ADL énoncés au début du chapitre. Certains de ces points seront illustrés par l'exemple que nous venons de traiter.

2.1.4 Discussion

Interface

La définition des interfaces est complète dans le sens de [?] : on a la possibilité de spécifier à la fois les événements que le composant va produire et ceux auxquels il va réagir. La construction de port permet de séparer entre ces deux types d'événements, la spécification du composant décrivant l'interaction entre les événements en précisant principalement un ordre d'apparition.

Cette définition d'interfaces par type permet de remplacer facilement un type de composant par un autre : il suffit que le nouveau type possède les mêmes événements et réagisse aux mêmes. En revanche le nouveau type pourra avoir une spécification différente. Dans le cas présent, il est possible de remplacer le type des filtres par tout autre type ayant un port d'entrée et un port de sortie, puisque le comportement des filtres n'est pas du tout spécifié.

Connecteurs

Le langage fournit un mécanisme d'abstraction pour les connecteurs. Le mode de communication autorisé par ces connecteurs est synchrone. Dans l'exemple du Compressing Proxy, les événements Écriture.écrit !y du Filtre 1 et Un.écrit ?y de l'adaptateur sont simultanés. Les connecteurs servent généralement à désynchroniser la communication entre les composants, mais entre un connecteur et un composant on aura toujours uniquement une communication synchrone. Puisque les connecteurs ont en plus la charge d'identifier les actions, il semble être difficile de construire des connecteurs génériques, i.e. dont l'écriture ne dépende pas du système. Un moyen de contourner ce problème serait d'effectuer les identifications de noms au niveau des *Attachments*, ce qui permettrait alors une véritable définition de *types* de connecteurs.

Configuration

La configuration d'un système spécifié en Wright est implicite puisqu'elle est définie dans la spécification des connecteurs. L'utilité des *Attachments* est d'identifier les ports et les rôles qui seront raccordés, cela pour mettre en valeur le principe de composition utilisé par ce langage.

Composition

Le mécanisme de composition du langage est du type «Composition correcte». Allen et Garlan utilisent le fait que la sémantique de CSP est totalement abstraite vis-à-vis du bloquage pour proposer une notion de composition plus forte que la simple «connectabilité». Ils appellent cette notion la compatibilité entre un port et un rôle.

La définition donnée dans [?] utilise le raffinement de CSP (si A et B sont deux processus, le raffinement est défini par : $traces(B) \subseteq traces(A)$), légèrement modifié pour permettre aux processus donnés pour les ports, P, d'avoir un alphabet plus ou moins grand que celui des processus donnés pour les rôles, R. Si on fait une analogie avec des prises, cette modification permet que les prises femelles soient plus large que les prises mâles ($\alpha P \subseteq \alpha R$) ou bien qu'une prise femelle n'utilise qu'une partie des possibilités de la prise mâle ($\alpha R \subseteq \alpha P$). C'est pourquoi dans la définition ci-dessous, on modifie les processus de telle sorte que leurs alphabets soient égaux (on a : $R_{+B} = R \parallel STOP_B$, où $STOP_B$ est le processus terminant correctement sur l'alphabet B). Une seconde modification plus importante concerne les comportements des ports : un composant peut se bloquer mais on voudrait être sûr que le connecteur auquel il est rattaché ne le permettra pas, c'est pourquoi on ne regarde le comportement du port que sur les traces décrites par le rôle (dans une version déterministe, c'est-à-dire que les choix non-déterministe $\sqcap$ sont transformés en choix déterministe $[]$). On obtient donc la définition suivante [?] :

Définition 5 *Un port P est compatible avec un rôle R (P* ***compat*** *R) si*
$R_{+(\alpha P \setminus \alpha R)} \sqsubseteq (P_{+(\alpha R \setminus \alpha P)} \parallel det(R))$

Exemple 1 *Dans l'exemple du Compressing Proxy, on voit aisément que le port :*

port $\textit{Écriture} = \textit{écrit}\,!y \rightarrow \textit{Écriture} \sqcap \textit{fin} \rightarrow \S$

est compatible avec le rôle :

role $Un = \textit{écrit}\,?y \rightarrow Lecture\ []\ \textit{fin} \rightarrow \S$

puisque :
$\alpha\textit{Écriture} = \{\textit{écrit}, \textit{fin}\}$; $\alpha Un = \{\textit{écrit}, \textit{fin}\}$; $det(Un) = \textit{écrit}\,?y \rightarrow Lecture \sqcap \textit{fin} \rightarrow \S$;
et donc : $Un \sqsubseteq (\textit{Écriture} \parallel det(Un))$.

Cette définition de compatibilité permet de déduire qu'une configuration de composants et de connecteurs compatibles est sans bloquage. Ceci est prouvé par le théorème [?] :

Théorème 1 *Si un connecteur C = (Glue $\parallel R_1 \parallel \ldots \parallel R_n$) est sans bloquage et conservatif et que $\forall i$ P_i **compat** R_i alors le système S = (Glue $\parallel P_1$ **as** $R_1 \parallel \ldots \parallel P_n$ **as** R_n) est sans bloquage.*

Il y a un algorithme permettant de décider si un connecteur est sans bloquage et la notion de conservativité précise juste que la trace de la glue est un sous ensemble de la mise en parallèle des traces des rôles.

La composition du système est donc autorisée si le système ne se bloquera pas : dans le cas du Compressing Proxy, la configuration que l'on a donnée n'est donc pas correcte puisqu'il y a un bloquage possible. Pour regarder cette propriété, il faut vérifier toutes les hypothèses du théorème et voir quelle est celle qui est en cause dans l'incompatibilité. Ce travail demanderait l'utilisation de l'outil FDR (*Failures Divergence Refinement*), qui permet d'automatiser la recherche de traces.

Modularité

Toutes les spécifications que l'on peut écrire dans Wright sont des spécifications plates. Cependant un mécanisme de style permet de déclarer séparément des types de composants et de connecteurs contraints par des axiomes. Ce style peut ensuite être instancié dans une spécification.

Raffinement

Il n'y a pas de concept de raffinement formel dans Wright. Il est cependant possible de remplacer un type de composant par un autre dès que le nouveau type de composants possède des ports *compatibles* avec les rôles auxquels étaient attachés les ports de l'ancien type de composants. Il est alors permis d'ajouter au nouveau type d'autres ports et d'avoir une spécification plus complexe.

Dynamisme

Une évolution du langage a été proposée dans [?] afin de permettre une reconfiguration du système durant son exécution. L'exemple traité est celui d'une architecture Client-Serveur dans laquelle l'éventualité de la panne du serveur est prévue. Afin de garantir au client un service même dans ce cas, on prévoit alors de rediriger les requêtes du client vers un serveur de secours.

Une nouvelle construction est ajoutée au langage : un *Configuror* gère des événements de contrôle décrivant quand les reconfigurations sont autorisées pour les composants. Il consiste en une spécification contenant des mots-clés (*new,del, attach, detach*) et ayant comme sémantique un processus CSP pur.

Afin de vérifier des propriétés, Allen, Garlan et Douence définissent une grammaire exprimant toutes les transformations possibles du *Configuror*, ce qui permet par exemple de vérifier qu'à tout moment le client est connecté à un serveur (par des preuves de réécriture). On peut toujours étudier les propriétés de compatibilité et de bloquage.

Analyse

Les propriétés que l'on peut vérifier sur une spécification Wright sont limitées par l'utilisation de CSP. En effet, des propriétés d'équité par exemple ne sont pas formalisables dans ce langage. Il y a la possibilité de rajouter des axiomes dans les déclarations de types ou de styles puis de vérifier ensuite sur les traces si ces propriétés sont vérifiées ou non, par exemple avec l'outil FDR.

La principale analyse que l'on peut effectuer est l'absence de bloquage, de par le mécanisme de composition.

2.2 COMMUNITY

Ce langage a été défini par José Luis Fiadeiro et Tom Maibaum [?] afin de valoriser la théorie des catégories en informatique et plus particulièrement dans le domaine de l'architecture logicielle. En effet il semble qu'une grande partie du vocabulaire utilisé par les ingénieurs (interconnexion, configuration, instantiation, composition, ...) puisse être formalisé assez simplement dans cette théorie, sans contrainte de syntaxe, seulement en imposant une structure de catégorie sur les spécifications [?].

COMMUNITY est un langage de design de programmes parallèles appliqué à l'architecture logicielle [?] : partant des mêmes concepts que WRIGHT, Fiadeiro et Maibaum considèrent qu'une architecture est un ensemble de composants communicant les uns avec les autres. Ce langage n'est pas cité dans le survey de N. Medvidovic [?] car plusieurs points importants du cadre d'étude ne sont pas traités par ce langage (la vérification de propriétés, les architectures dynamiques), mais plusieurs axes de recherche sont envisagés pour remédier à ces points faibles.

2.2.1 Le langage

Voici une description abstraite de programme en COMMUNITY :

program Exemple **is**

data Σ

var V

read R

init I

do $\underset{g\in\Gamma}{[]}$ g : [B(g) $\rightarrow \underset{a\in D(g)}{\|}$ a := F(g,a)]

Σ représente les types de données du programme, par une signature munie d'axiomes (Σ, Φ) ; R est l'ensemble des attributs que le programme lit dans son environnement ; V est l'ensemble des attributs locaux ; I est la condition initiale sur les attributs locaux ; Γ est l'ensemble des noms d'actions (un 2^V ensemble) qui serviront à la synchronisation par rendez-vous (et qui représentent comme pour WRIGHT les choix de modélisation et donc l'interface du composant) ; pour chaque action g, B(g) est la garde de cette action ; D(g) est le domaine de l'action g (l'ensemble des variables qu'elle peut modifier) ; F(g,a) est une expression (ou un ensemble d'expressions si on veut du non-déterminisme) de même sorte que a. F(g,a) désigne la modification que l'action g va faire subir à l'action a.

Un programme est composée de deux parties distinctes : une signature (Σ, V, R, Γ) et un corps (I, F, B). Les spécifications algébriques des types de données peuvent être locales à chaque programme, mais on considérera pour simplicité que les types de données seront fixes pour tout le système, ce qui permet de les sous-entendre. La particularité de ce langage est que chaque sous-partie d'un programme (signature et corps) peut être munie d'une structure de catégorie.

La catégorie (SIG) des signatures définie dans [?] est donnée par :

Définition 6 *Les objets de la catégorie SIG sont les signatures* (Σ, V, R, Γ)*. Un morphisme de signature est donné par une application*
$\sigma_\alpha : V_1 \cup R_1 \to V_2 \cup R_2$ *et une application* $\sigma_\gamma : \Gamma_1 \to \Gamma_2$ *vérifiant :*

1. $\forall a \in V_1 \cup R_1, sort(a) = sort(\sigma_\alpha(a))$
2. $\forall a \in V_1, \sigma_\alpha(a) \in V_2$
3. $\forall a \in V_1, \sigma_\gamma(D_1(a)) = D_2(\sigma_\alpha(a))$

Les conditions sur les morphismes sont motivées par des considérations sémantiques, en limitant les morphismes (qui représentent les connexions) aux morphismes raisonnables. La première assure qu'un morphisme préserve les sortes ; la seconde signifie qu'une variable locale (dans V) ne peut jamais devenir globale (dans R), l'inverse étant possible puisque lors d'une synchronisation, une variable R sera reliée à une variable V et sera donc locale au système créé ; la troisième dit qu'une action ne peut élargir son domaine sur des variables locales n'appartenant pas à son programme.

La catégorie (PROG) des programmes Community est définie par :

Définition 7 *Les objets de la catégorie PROG sont les programmes* Community *(signature et corps). Un morphisme de programme est un morphisme de signature vérifiant de plus :*

1. $\Phi \vdash (I_2 \Rightarrow \sigma(I_1))$
2. $\forall g_1 \in \Gamma_1, a_1 \in D_1(g_1)$,
   $\Phi \vdash B_2(\sigma(g_1)) \Rightarrow (F_2(\sigma(g_1), \sigma(a_1)) = \sigma(F_1(g_1, a_1)))$
3. $\forall g_1 \in \Gamma_1, \Phi \vdash (B_2(\sigma(g_1)) \Rightarrow \sigma(B_1(g_1)))$

La première condition assure que la condition initiale de chaque composant est vérifiée dans le système global ; la seconde signifie que les assignations sont respectées ; la troisième que les gardes ne peuvent pas être affaiblies.

2.2.2 Sémantique

La sémantique du langage est donnée au dessus d'un modèle pour la signature Σ (donné par une Σ-algèbre $\mathcal{U}$). Une structure d'interprétation pour une signature de Community est un triplet $(\mathcal{T}, \mathcal{A}, \mathcal{G})$ où :

- $\mathcal{T}$ est un système de transition $(\mathcal{W}, w_0, \mathcal{E}, \to)$: ensemble d'états $\mathcal{W}$, état initial w_0, ensemble d'événements $\mathcal{E}$, fonction de transition $\to$.
- $\mathcal{A}$ est une famille d'applications $\mathcal{A}_s : A_s \to (\mathcal{W} \to s_\mathcal{U})$ indexée par S (ensemble des sortes)
- $\mathcal{G}$ est une application de Γ vers $2^\mathcal{E}$

La sémantique d'un programme est une structure d'interprétation de la signature sous-jacente vérifiant :

- $(\mathcal{S}, w_0) \models I$
- $\forall g \in \Gamma, a \in D(g), e \in \mathcal{G}(g), w$ et w' tels que $w \xrightarrow{e} w'$,
  $\mathcal{A}(a)(w') = [\![F(g, a)]\!](w)$, où $[\![\ ]\!]$ est la fonction d'interprétation dans la Σ-algèbre.
- $\forall w \in \mathcal{W}, g \in \Gamma$,
  si $e \in \mathcal{G}(g)$ et $\exists\ w' \in \mathcal{W}$ tel que $w \xrightarrow{e} w'$, alors $(\mathcal{S}, w) \models B(g)$

On peut donc voir la sémantique d'un programme COMMUNITY comme un ensemble de systèmes de transition communiquant par synchronisation de leurs actions par rendez-vous. Cette idée sera mise à profit plus loin.

2.2.3 Compressing Proxy

D'après la description faite dans l'introduction du chapitre, on distingue trois sortes de composants : les filtres, l'adaptateur, et GZip. Chacun des composants est décrit par un programme COMMUNITY que l'on reliera ensuite pour former le système entier. On supposera, pour faciliter les descriptions, que les données transmises entre les composants sont d'un type *elem* qui pourra représenter des entiers, des caractères ou tout autre type de données. Cela permet de ne pas préciser quelle signature Σ on utilise.

Filtres

Les deux filtres sont modélisés de la même manière : ils ont une action pour calculer des valeurs (la *fonction* du filtre, qui n'est pas précisée), une autre pour écrire cette valeur, et une dernière pour lire des valeurs de l'extérieur. Dans le cadre qui nous intéresse, on aurait pu différencier les deux filtres, mais, afin de limiter le nombre de programmes à écrire, un seul suffit. On n'utilisera qu'un seule action de ce programme pour le filtre haut et une seule pour le filtre bas.

Les actions du filtre sont organisées comme suit : le filtre reçoit une valeur de l'extérieur (*inputval*) qu'il lit et stocke dans sa variable locale x, puis il effectue un calcul sur cette valeur ; le filtre envoie enfin cette valeur, placée dans la variable y, au filtre suivant. On assigne à la valeur y une valeur arbitraire (le nom du type) afin de ne pas donner de précisions sur la fonction du filtre. Un filtre a de plus la possibilité de s'arrêter dès que le flot est tari, ce qui est traduit par l'action *close* à qui on assigne le processus $\surd$. La signature Σ est implicite.

program Filtre **is**

var compteur : $\{1, 2, 3\}$; x, y : elem

read inputval : elem

init compteur = 1

do lecture : [compteur = 1 $\rightarrow$ x := inputval $\|$ compteur := 2]
    [] calcul : [compteur = 2 $\rightarrow$ compteur := 3]
    [] écriture : [compteur = 3 $\rightarrow$ y := elem $\|$ compteur := 1]
    [] fin : [compteur = 3 $\rightarrow$ $\surd$]

end.

GZip

Le composant GZip possède aussi des actions correspondant à l'entrée des valeurs et à la sortie. Il y a deux actions correspondant à la fin des transmissions en entrée, la première synchronisée avec l'adaptateur (end_i), la seconde autonome (fin_i) et deux actions signifiant la fin des sorties vers l'adaptateur, end_o et fin_o. Les actions fin seront activées dans le cas où un buffer interne sera plein ou bien pour d'autres raisons spécifiques au programme. Les actions $retour_i$ et $retour_o$ servent à spécifier que le mode de communication entre Gzip et l'adaptateur est incrémental : les valeurs passent par un *pipe* UNIX et donc sont segmentées et transmises en fragments.

Comme pour les filtres, il n'y a pas de relation entre les valeurs d'entrée et celles de sortie : on ne spécifie pas le programme.

program GZip **is**

var comp : {1,2,3,4,5} ; x, y : elem

read inputval : elem

init comp = 1

do entrée : [comp = 1 → x := inputval ∥ comp := 2]
[] fin_i : [comp = 2 → comp := 3]
[] end_i : [comp = 2 → comp := 3]
[] retour_i : [comp = 2 → comp := 1]
[] sortie : [comp = 3 → y := elem ∥ comp := 4]
[] fin_o : [comp = 4 → comp := 5]
[] end_o : [comp = 4 → comp := 5]
[] retour_o : [comp = 4 → comp := 3]
[] fin : [comp = 5 → comp := 1]

end.

Adaptateur

L'adaptateur possède deux entrées et deux sorties. On aura donc deux valeurs à lire dans des composants externes ($inputval_g$ et $inputval_f$). Comme ce composant n'a qu'un rôle de prise, les valeurs qu'il récupère sont simplement transmises au composant suivant dans le flot. Ce comportement passif est souligné par le fait qu'il a été modélisé par un connecteur en Wright. Cependant, les modes de communication entre les composants sont décrits de manière différente selon qu'il s'agit d'une communication entre l'adaptateur et les filtres (entrée/sortie) ou entre l'adaptateur et Gzip (*pipe*). On utilisera le même procédé que pour GZip (utilisation des actions *retour*), mais on ne décrit pas les relations entre $inputval_f$ et y, comme on peut le faire dans Wright.

Les variables x et y sont des variables locales : au cours du déroulement du programme on voit qu'elles prennent des valeurs différentes.

program Adaptateur **is**

var comp : {1,2,3,4,5,6,7,8} ; x, y : elem

read inputval_f, inputval_g : elem

init comp = 1

do lecture : [comp = 1 → x := inputval_f ∥ comp := 2]
[] sortie : [comp = 2 → y := x ∥ comp := 3]
[] retour_o : [comp = 3 → comp := 2]
[] end_o : [comp = 3 → comp := 4]
[] prépare : [comp = 4 → comp := 5]
[] entrée : [comp = 5 → x := inputval_g ∥ comp := 6]
[] retour_i : [comp = 6 → comp := 5]
[] end_i : [comp = 6 → comp := 7]

[] écriture : [comp = 7 → y := x ‖ comp := 8]
[] fin : [comp = 8 → comp := 1]
end.

Spécification COMMUNITY

Une spécification COMMUNITY consiste en un diagramme catégorique dans lequel sont exprimées les relations entre les composants (voir le diagramme de configuration, figure **??**). Ces relations sont en fait exprimées par un partage de signature qui permet d'identifier les actions synchronisées, ainsi que les valeurs transmises et reçues. Nous verrons plus en détail dans la section suivante ce mécanisme de composition. Les variables sont renommées (par le mécanisme lui-même) afin d'éviter les clashs. Les actions nommées par un $comm-$ sont celles résultants de la synchronisation entre des actions des composants, les autres actions sont indicées par le nom du composant dans lequel cette action a lieu.

Le programme résultant de la composition est exprimé à isomorphisme près par le programme suivant dans lequel on ne retrouve qu'une variable dans la catégorie R, puisque maintenant tous les composants sont reliés et donc lisent des valeurs provenant d'autres composants. La dernière valeur permet de garder le système ouvert (on peut le connecter à d'autres programmes).

program Compressing Proxy **is**
var comp_a : {1,2,3,4,5,6,7,8}, ; comp_g : {1,2,3,4,5} ; comp_f1, comp_f2 : {1, 2, 3} ;
x, y, x_f1, y_f1, x_f2, y_f2, x_g, y_g : elem ;
read inputval : elem
init comp_a = 1 ∧ comp_g = 1 ∧ comp_f1 = 1 ∧ comp_f2 = 1
do lecture_f1 : [comp_f1 = 1 → x_f1 := inputval ‖ comp_f1 := 2]
[] calcul_f1 : [comp_f1 = 2 → comp_f1 := 3]
[] comm-f1-ad : [comp_f1 = 3 ∧ comp_a = 1 →
y_f1 := elem ‖ x := y_f1 ‖ comp_f1 := 1 ‖ comp_a := 2]
[] comm-f1-ad-fin : [comp_f1 = 3 ∧ comp_a = 3 → comp_a := 4]
[] comm-ad-gz : [comp_a = 2 ∧ comp_g = 1 →
x_g := x ‖ comp_a := 3 ‖ comp_g := 2]
[] ···
end.

Le bloquage est bien présent dans cette modélisation puisque si le composant GZip effectue l'action fin_i, alors le compteur *comp_ a* ne passera pas à la valeur 4, ce qui empêchera toute transaction future car la valeur de ce compteur apparaît dans les gardes des actions suivantes.

2.2.4 Discussion

Interface

L'interface d'un programme COMMUNITY est définie par sa signature (Σ, V, R, Γ). Elle se conforme aussi à la notion d'interface complète de [?], car on peut préciser les besoins autant que les produits du composant. Cependant, il n'y a pas de spécification axiomatique de ces besoins et de ces produits, ni de spécification concernant leurs relations. On peut seulement déduire de la garde modifiant un attribut du programme quel attribut externe est nécessaire et dans quel état.

Il y a eu un travail important concernant la notion d'interface dans [?]. Cette notion est fondamentale dans le domaine des langages de coordination puisqu'elle permet de séparer clairement dans un programme ce qui sert à la communication de ce qui est calculatoire. Un des objectifs du travail présenté dans cet article est de trouver un mécanisme pour relier les interfaces des composants avec leur spécification, dans le paradigme catégorique (ie, comment faire pour configurer un système à partir des interfaces puis que son comportement satisfasse les comportements de chacun des sous-programmes ?).

Connecteurs

Il n'y a pas construction explicite pour les connecteurs dans COMMUNITY. Cependant, dans [?], il est montré que les connecteurs de Wright peuvent être représentés par des programmes COMMUNITY.

Nous n'avons pas différencié les modes de communication dans la spécification de la section précédente. La principale différence entre un appel de fonctions et un *pipe* est que l'argument d'une fonction est passé en une seule fois, tandis qu'un *pipe* gère un flot de données segmentées. Une solution pour prendre en compte cette différence est de rajouter un composant modélisant un *Buffer* faisant un tampon entre GZip et l'adaptateur. Ce composant possède une mémoire lui permettant de stocker des valeurs et des action de lecture/écriture sur cette mémoire. Il suffit ensuite de connecter ce composant à GZip et à l'adaptateur par des *Channels* appropriés.

Composition et Configuration

On confond les deux paragraphes puisque la configuration d'un programme COMMUNITY est confondue avec sa composition. Celle-ci se base sur une idée de J. Goguen : la construction d'un système à partir de sous-parties consiste à prendre la colimite de ces parties.

La sémantique de la composition de programmes est identique à celle de Wright : c'est la composition parallèle des sous-programmes. Or dans un contexte catégorique, Fiadeiro et Maibaum ont montré [?] que cette composition parallèle correspond à des constructions connues (coproduit si les parties ne communiquent pas, pushout sinon). Une fois que la catégorie des programmes est prouvée être cocomplète (le pushout de deux programmes existe toujours, cf. [?]), il est possible de composer des programmes par ces constructions (la preuve de la propriété est constructive).

Dans notre exemple, les différentes parties communiquent, elles sont donc composées par des constructions de pushout. Les parties partagées correspondent aux interfaces communes utilisée par deux composants. Or on a vu dans un paragraphe précédent que ces interfaces sont définies par des signatures de COMMUNITY. Il faut donc, pour pouvoir réaliser le diagramme de pushout, plonger cette signature dans un programme. Comme les signatures et les programmes ont tous les deux une structure de catégorie, le plongement peut se définir par un foncteur. La relation entre les interfaces (coordination) et les programmes dépend alors des propriétés de ce foncteur [?, ?, ?].

Définition 8 *Soit F : SIG → PROG telle que pour toute signature (Σ, V, R, Γ) on ait :*

$$F(\Sigma, V, R, \Gamma) \equiv \begin{cases} data: & (\Sigma, \emptyset) \\ var: & V \\ read: & R \\ init: & true \\ do: & \underset{g\in\Gamma}{[]}\ g : [true \rightarrow \underset{a\in D(g)}{\|}\ a := s_a] \end{cases}$$

Cette application se prolonge en un foncteur.

Avec cette définition, on construit les diagrammes de configuration en utilisant les images du *Channels* (le nom d'une interface en COMMUNITY) par ce foncteur.

Exemple 2 *Ce Channel sert à la communication entre les filtres et l'adaptateur. Il possède une valeur (la valeur échangée) et deux actions (une pour la lecture/écriture, une autre pour le message de fin). Voici son image par le foncteur F :*

program *Channel1* ***is***

var *v :elem*

init *true*

do *a : [true →* $\surd$ *]*
[] *b : [true →* $\surd$ *]*

end.

Exemple 3 *Ce Channel sert à la communication entre l'adaptateur et GZip. Il possède deux valeur (les valeurs échangées) et quatre actions (deux pour la lecture/écriture, deux autres pour les messages de fin). Voici son image par le foncteur F :*

program *Channel2* ***is***

var *u, v :elem*

init *true*

do *a : [true →* $\surd$ *]* [] *b : [true →* $\surd$ *]*
[] *c : [true →* $\surd$ *]* [] *d : [true →* $\surd$ *]*

end.

Un *Channel* peut donc se voir comme un programme ayant un comportement passif. On obtient le diagramme de la figure **??**.

On remarquera que l'action fin du filtre F_1 est connectée à l'action end_o de l'adaptateur, elle-même reliée à l'action end_i de GZip. Cette série de connection permet de propager l'interruption du flot de données dans tout le système. On peut vérifier que les morphismes donnés dans cette figure sont bien des morphismes de programmes.

Ce mécanisme de composition des programmes suit le principe de «connectabilité». En effet, dans l'exemple précédent, la composition est possible bien qu'elle ne soit pas correcte dans le sens où un bloquage est possible. Contrairement à ce qui est dit dans [**?**], cette composition par colimites ne reflète pas celle de Wright, puisque dans le premier cas, la composition n'était pas possible. Pour améliorer ce mécanisme, il faudrait peut-être modifier la définition des morphismes car ce sont eux (à travers les *Channels*) qui permettent de relier les programmes.

$/F2psDict200dictdef$F2psDict begin $F2psDict/mtrxmatrixput/col-$
$10setgraybinddef/col00.0000.0000.000srgbbinddef/col10.0000.0001.000srgbbinddef/col20.0001.0000.000srgbbi$
end save -290.0 573.0 translate 1 -1 scale
/cp closepath bind def /ef eofill bind def /gr grestore bind def /gs gsave bind def /sa save bind def /rs restore bind def /l lineto bind def /m moveto bind def /rm rmoveto bind def /n newpath bind def /s stroke bind def /sh show bind def /slc setlinecap bind def /slj setlinejoin bind def /slw setlinewidth bind def /srgb setrgbcolor bind def /rot rotate bind def /sc scale bind def /sd setdash bind def /ff findfont bind def /sf setfont bind def /scf scalefont bind def /sw stringwidth bind def /tr translate bind def /tnt dup dup currentrgbcolor 4 -2 roll dup 1 exch sub 3 -1 roll mul add 4 -2 roll dup 1 exch sub 3 -1 roll mul add 4 -2 roll dup 1 exch sub 3 -1 roll mul add srgb bind def /shd dup dup currentrgbcolor 4 -2 roll mul 4 -2 roll mul 4 -2 roll mul srgb bind def $/F2psBegin$F2psDict begin
$/F2psEnteredStatesavedefdef$/F2psEnd $F2psEnteredStaterestoreenddef$
$F2psBegin10setmiterlimitn-100010536m-1000-1000l14804-$
$1000l1480410536lcpclip0.060000.06000sc7.500slwn48599524m137929524l137921975l48591975lcpgscol0sgrgsclip$
n 9341 2936 m 9368 2808 l 9395 2936 l 9368 2914 l 9341 2936 l cp gs 0.00 setgray ef gr col0 s gs clippath 5770 8611 m 5648 8657 l 5741 8566 l 5621 8643 l 5650 8687 l cp clip n 7112 7726 m 5648 8657 l gs col0 s gr gr
n 5770 8611 m 5648 8657 l 5741 8566 l 5738 8600 l 5770 8611 l cp gs 0.00 setgray ef gr col0 s gs clippath 9395 5870 m 9368 5998 l 9341 5870 l 9341 6013 l 9395 6013 l cp clip n 9368 4536 m 9368 5998 l gs col0 s gr gr
n 9395 5870 m 9368 5998 l 9341 5870 l 9368 5892 l 9395 5870 l cp gs 0.00 setgray ef gr col0 s gs clippath 9244 6439 m 9368 6398 l 9271 6485 l 9394 6413 l 9368 6367 l cp clip n 7773 7326 m 9368 6398 l gs col0 s gr gr
n 9244 6439 m 9368 6398 l 9271 6485 l 9276 6451 l 9244 6439 l cp gs 0.00 setgray ef gr col0 s gs clippath 9465 6485 m 9368 6398 l 9492 6439 l 9368 6367 l 9342 6413 l cp clip n 10962 7326 m 9368 6398 l gs col0 s gr gr
n 9465 6485 m 9368 6398 l 9492 6439 l 9460 6451 l 9465 6485 l cp gs 0.00 setgray ef gr col0 s gs clippath 12884 8536 m 12977 8627 l 12855 8581 l 12975 8657 l 13004 8613 l cp clip n 11514 7696 m 12977 8627 l gs col0 s gr gr
n 12884 8536 m 12977 8627 l 12855 8581 l 12887 8570 l 12884 8536 l cp gs 0.00 setgray ef gr col0 s /Times-Roman ff 150.00 scf sf 8401 3520 m gs 1 -1 sc (a
mapsto$entr$
'ee, b
mapstoend
$_i)col0shgr/Times-Romanff150.00scfsf83893739mgs1-1sc(c$
mapsto$sortie,d$
mapstoend
$_o)col0shgr/Times-Romanff150.00scfsf85575228mgs1-1sc(a$
mapsto$sortie,b$
mapstoend
$_o)col0shgr/Times-Romanff150.00scfsf85575439mgs1-1sc(c$
mapsto$entr$
'ee, d
mapstoend
$_i)col0shgr/Times-Romanff150.00scfsf77376961mgs1-1sc(a$
mapsto$lecture)col0shgr/Times-Romanff150.00scfsf77377173mgs1-1sc(b$
mapstoend
$_o)col0shgr/Times-Romanff150.00scfsf59328200mgs1-1sc(a$
mapsto
'ecriture) col0 sh gr /Times-Roman ff 150.00 scf sf 5932 8410 m gs 1 -1 sc (b
mapsto$fin)col0shgr/Times-Romanff150.00scfsf59167974mgs1-1sc(v$
mapsto$y)col0shgr/Times-Romanff150.00scfsf98346816mgs1-1sc(v$
mapsto$y)col0shgr/Times-Romanff150.00scfsf98347234mgs1-1sc(b$

Modularité

Il n'y a pas de mécanisme de structuration des programmes dans COMMUNITY. Cependant, vu que le mécanisme de configuration suit la construction de pushout, il est possible dans un tel diagramme de remplacer un objet par un diagramme entier. Cela permet une construction incrémentale du programme en rajoutant au fur et à mesure des programmes développés indépendamment du système (réutilisation de composants). Cette possibilité autorise de plus la correction de certain composant pouvant s'avérer défectueux au cours de la conception (cf. [?]).

L'utilisation de catégories en tant que contrainte syntaxique permet aussi d'envisager l'utilisation de plusieurs formalismes pour spécifier un système. Il suffit pour cela que ces formalismes soient munis d'une structure de catégorie et que les relations entre les différentes catégories soient vérifiées (propriétés du foncteur reliant ces catégories [?, ?]). Le choix de la catégorie utilisée peut faciliter la description de certains composants, ou même peut refléter un choix de style pour toute la spécification.

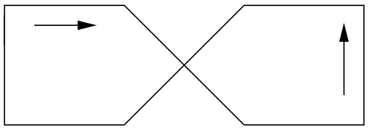
Raffinement

Le raffinement *local* se conçoit aisément en COMMUNITY puisque l'on peut définir ce raffinement par un morphisme entre le programme et un programme le raffinant. Les propriétés des morphismes de programme assurent que deux programmes reliés entre eux par un morphisme auront des actions communes et que sur ces actions, le morphisme ne permet qu'une fortification de la garde.

Par contre le second raffinement, le raffinement *global*, n'est pas traité. Pour cela il faudrait que l'on ait une relation entre deux diagrammes comportant des composants identiques mais reliés par des *Channels* différents. Cette relation pourra s'exprimer par un morphisme, mais pour savoir si ces deux configurations ont un lien (par exemple, elles ont le même résultat pour la même entrée), il faut des informations d'ordre sémantique qui ne sont pas données par ces diagrammes.

Dynamisme

Dans [?], une prolongation des connecteurs tels qu'ils ont été définis dans [?] est proposée afin de gérer les architectures mobiles. L'exemple traité pour illustrer le problème consiste en un circuit découpé en unités de longueur et ayant un point de croisement sur lequel circulent des wagons ayant la possibilité d'avancer de case en case en suivant le sens indiqué par les flèches.



La définition de cette prolongation est la suivante : un *connecteur transitoire* est un couple $< T, I >$ où T est un connecteur statique et I est une condition booléenne appelée condition d'interaction, portant sur les variables des rôles du connecteur T. Une autre possibilité aurait été de faire rentrer la condition d'interaction I dans l'écriture de la glue et des rôles (on aurait eu des connexions conditionnelles), mais cette version est moins satisfaisante puisqu'alors on alourdit la syntaxe des morphismes et des *Channels*. Les *connexions transitoires* permettent de spécifier les connexions uniquement lorsqu'il y a un problème de collision, ce qui réduit la taille du diagramme de configuration, qui comporterait $n^2 - n$ connections si le système possède n wagons. La visualisation du système à tout moment se fait par une règle d'application définie par la satisfaction de la condition I par les variables du composant P reliées aux variables de T intervenant dans la condition.

La sémantique d'un système doit maintenant capturer des informations dynamiques, c'est pourquoi dans [?] Fiadeiro et al. utilisent des règles de réécriture conditionnelles avec des diagrammes de configuration étendus. Un diagramme étendu est un diagramme dont les noeuds possèdent une valuation ν pour les attributs du programme qui labellent le noeud. Ces valuations doivent être compatibles avec les morphismes : si on a deux noeuds P, ν et P', ν' reliés par un morphisme f, alors on doit avoir pour tout a dans les attributs de P, $\nu(a) = \nu(f(a))$. Une première tentative avait été faite dans [?] en modélisant une architecture comme deux ensembles, l'un de connecteurs transitoires, l'autre de composants.

L'utilisation de règles de réécriture permet de conserver la configuration sous forme de diagramme (qui évoluera au cours de l'exécution) et d'avoir la possibilité de faire appel à des techniques bien connues de réécriture de diagrammes pour formaliser les connecteurs, comme [?] ou la Rewriting Logic.

Analyse

L'analyse que l'on peut effectuer sur des programmes COMMUNITY est inexistante. Cependant il existe, par le biais de constructions catégoriques, un moyen pour relier des spécifications écrites dans une logique temporelle (LTL) à des programmes COMMUNITY. Pour cela, on définit une institution pour cette logique temporelle.

Définition 9 (Goguen/Burstall 83) *Une institution consiste en :*
- *une catégorie SIG de signatures*
- *un foncteur SEN : SIG $\to$ SET*
- *un foncteur MOD : SIG^{op} $\to$ CAT*
- *une relation de satisfaction $\models_\Sigma \subseteq$ MOD(Σ) $\times$ SEN(Σ)*

tels que pour tout morphisme de signatures $\sigma : \Sigma \to \Sigma'$, la traduction MOD($\sigma$) des modèles et SEN($\sigma$) des sentences préserve la relation de satisfaction, c'est-à-dire que pour tout $\varphi \in SEN(\Sigma)$ et $M' \in \mid MOD(\Sigma') \mid$, on ait (condition de satisfaction) :

$$M' \models_{\Sigma'} SEN(\sigma)(\varphi) \Leftrightarrow MOD(\sigma)(M') \models_\Sigma \varphi$$

On peut alors définir la catégorie SPEC des spécifications en LTL comme étant la catégorie des théories de cette institution [?, ?] :

Définition 10 *Les objets sont les paires $<\Sigma, \Phi>$, où Σ est une signature et $\Phi \subseteq SEN(\Sigma)$. Les morphismes sont des morphismes de signature $f : <\Sigma_1, \Phi_1> \to <\Sigma_2, \Phi_2>$ tel que pour tout $\varphi \in \Phi_1, \Phi_2 \models_{\Sigma_2} \underline{f}(\varphi)$, avec $\underline{f}$ = SEN(f).*

Cette catégorie est cocomplète [?]. On a vu dans les sections précédentes que pour pouvoir relier deux formalismes possédant une structure de catégorie, il suffit de définir un foncteur entre ces catégories. Si on veut analyser des programmes COMMUNITY, on transforme ces programmes dans des spécifications LTL par un foncteur Spec : PROG $\to$ SPEC, qui doit vérifier des propriétés particulières (conservation des diagrammes, ...) [?].

Cependant, il y a un problème non résolu sur l'émergence de propriétés lors de la composition des spécifications au niveau logique. En effet les morphismes de cette catégorie ne sont pas conservatifs : certaines propriétés du système qui sont écrites avec la signature d'un composant ne sont pas vérifiées au niveau de ce composant.

Par exemple si un système est composé des spécifications S_1 et S_2 et φ une propriété écrite avec la signature de S_1, alors on peut avoir : S_1 **and** $S_2 \vdash \underline{f}(\varphi)$, mais pas : $S_1 \vdash \varphi$.

Les premiers travaux entrepris dans l'article [?] montrent que des difficultés se présentent lorsque l'on veut vérifier des propriétés sur des programmes COMMUNITY. Un des remèdes que cet article apporte est de reprendre la base de la traduction par le foncteur *Spec* en y ajoutant des contraintes pour empêcher l'émergence des propriétés (sous forme de conditions sur les attributs des programmes). Mais pour le moment, ce point essentiel dans la définition d'un ADL n'est pas bien construit.

2.3 CHAM

Le langage CHAM (CHemical Abstract Machine) est un langage de spécification de processus concurrents, créé par Berry et Boudol [?] après reformulation de résultats démontrés par Banâtre et Le Metayer (langage Gamma [?]). Il est utilisé dans le domaine de l'AL par Inverardi et Wolf [?] car, comme pour Wright ou COMMUNITY, les auteurs appliquent les possibilités de leur langage pour modéliser des architectures d'interaction vues comme une synchronisation de processus concurrents.

La particularité du langage CHAM est que contrairement à CSP, il ne contient pas de modèle calculatoire prédéfini et donc n'influence pas la modélisation par un style particulier de concurrence. De plus, l'utilisation d'une sémantique opérationnelle permet d'utiliser les techniques de réécriture pour analyser le comportement de la modélisation.

2.3.1 Le langage

Une CHAM est déterminée par la donnée de :

1. molécules. Ce sont des termes construits sur une signature.
2. solutions. Une solution est un multi-ensemble de molécules (c'est le principe de base du langage Gamma) qui représente l'état de la CHAM. On peut combiner des solutions par des opérations :
   – Si S et S' sont des solutions, alors l'union des multi-ensembles $S \uplus S'$ est une solution.
   – Une solution peut être considérée comme une seule molécule par rapport aux autres solutions, par une construction d'encapsulation (*membrane*). Une solution dans une membrane peut évoluer indépendamment de son environnement. Si S est une solution, alors $\{\!|S|\!\}$ aussi.
   – Un opérateur réversible (*airlock*, $\triangleleft$) permet d'extraire des molécules d'une solution et de placer le reste de la solution dans une membrane : l'*airlock* est un constructeur de membranes. Si $m \in S$, alors $m \triangleleft S$ enlève une occurrence de la molécule m dans la solution S et place le reste de la solution S dans une membrane.

   Les solutions représentent le système à un moment donné en donnant l'état des termes et des variables décrits dans la spécification par des molécules. Ces constructions pour les solutions permettent de définir une abstraction de contexte $\mathcal{C}[.]$ utilisée pour faire référence à un ensemble arbitraire de molécules dans lequel on place une certaine molécule.

3. **règles.** Comme la sémantique d'une CHAM est une sémantique opérationnelle, ces règles définissent avec celles de la section suivante, l'ensemble des comportements possibles de la CHAM. La forme des règles est $S \to S'$ où S et S' sont des solutions pouvant contenir des méta-variables. À tout moment, une CHAM peut appliquer autant de règles que possible, et si plusieurs règles partent de la même solution, alors on obtient un comportement non-déterministe. Il faut remarquer qu'une machine ne calcule pas des transformations mais les cherche : il est donc possible d'obtenir des comportements cycliques ou bloqués (on dira alors que la solution est *inerte*).

L'utilisation de ces constructions pour modéliser des architectures de connexion est expliqué ci-dessous. Voici un exemple de spécification architecturale en CHAM représentant un composant ayant une entrée et une sortie :

syntaxe

M := P | C | M $\diamond$ M
P := ...
N := n_1 | n_2 |...
C := i(N) | o(N).

règles

$T_1 \equiv i(n_1) \diamond m, o(n_1) \diamond m' \to m \diamond i(n_1), m' \diamond o(n_1)$
$T_2 \equiv S_1 \longrightarrow S'_1$
...

solution initiale

S_i := ...

solution finale

S_f := ...

La partie syntaxique définit les molécules (M) représentant le composant. Il y a une distinction entre les éléments processeurs (P), ceux qui assurent la communication (C), et les noms de canaux assurant cette communication (N). Les éléments processeurs sont le nom du composant et une méta-variable se référant à la structure syntaxique inconnue des autres composants si le système n'est pas configuré, ou l'ensemble des noms des composants du système le cas échéant. Les éléments connectifs modélisent la communication entre les composants; dans cet exemple, la communication s'effectue par des opérations d'entrée/sortie sur un canal appartenant à l'ensemble N. L'opérateur «$\diamond$» donne l'état de l'élément processeur en précisant l'action qu'il peut accomplir ensuite et par quel canal (s'il y en a plusieurs). Par exemple la molécule i(n_1) $\diamond$ m signifie que la molécule m est prête à recevoir des données par le canal n_1.

Les règles caractérisent les comportements possibles du composant. Par exemple, si S est une solution contenant la molécule i(n_1) $\diamond$ P et la molécule o(n_1) $\diamond$ P' (avec P' élément processeur d'un autre composant), alors on peut appliquer la règle T_1 qui transformera les molécules responsables de la communication entre les éléments processeurs P et P'. En effet, pour pouvoir appliquer cette règle, il suffit d'instancier les méta-variables m et m' par P et P'.

Les solutions initiale et finale servent à reconnaître la fin d'une transmission cyclique ou à déterminer la fin d'un protocole de communication.

2.3.2 Sémantique

Le paradigme utilisé par ce langage (la programmation par transformation de multi-ensembles) implique une sémantique opérationnelle. En plus des règles particulières définies par chaque composant, le langage prévoit des règles de transformations générales qui seront prises en compte par toutes les spécifications.

Loi de Réaction Si les m_i et m'_j sont des instances des M_i et M'_j alors on peut appliquer la règle : $M_1, ..., M_n \to M'_1, ..., M'_p$ et obtenir la transformation suivante au niveau des solutions : $m_1, ..., m_n \to m'_1, ..., m'_p$

Loi Chimique

$$\frac{S \to S'}{S \uplus S'' \to S' \uplus S''}$$

Loi de la Membrane

$$\frac{S \to S'}{\{\!|\mathcal{C}[S]|\!\} \to \{\!|\mathcal{C}[S']|\!\}}$$

Loi de l'Airlock

$$m \uplus S \to m \triangleleft \{\!|S|\!\}$$

Ces lois permettent d'appliquer les règles de transformation définies dans les composants à tous les contextes $\mathcal{C}[.]$. On peut considérer que cette sémantique est proche de celle de WRIGHT puisqu'on peut imaginer que les transformations de multi-ensembles représentent des traces pour un certain type d'automates. On aura donc la possibilité de voir encore ces spécifications sous forme d'automates, ce qui permettra de comparer les approches.

2.3.3 Compressing Proxy

Les spécifications CHAM des composants intervenant dans le Compressing Proxy sont issues de [**?**].

Filtres

La syntaxe de ces composants se compose d'un élément processeur $\mathbf{CF}_u$ ou $\mathbf{CF}_d$ (les filtres F_1 et F_2) et d'une méta-variable Φ représentant les molécules des autres composants. Il y a une distinction entre le filtre situé avant l'adaptateur et le filtre situé après : chacun ne comporte qu'un seul canal, mais ce canal servira pour écrire à l'adaptateur pour le premier filtre et pour lire des données pour le second. Les éléments de l'ensemble N sont aussi des méta-variables qu'il faudra instancier lors de la configuration.

Les deux composants possèdent deux règles. La première est une instance de la règle générale de communication inter-composants décrivant une communication entrée/sortie. La seconde modélise le comportement itératif de cette communication. Les méta-variables apparaissant dans ces règles sont censées être instanciées par des molécules provenant d'autres composants, et dont la syntaxe est contenue dans Φ. On ne modélise pas dans cette spécification la différence entre la lecture/écriture à travers un flot et l'entrée/sortie à travers un *pipe*.

La molécule initiale des composants signifie que l'élément processeur attend avant de transmettre des données ou de les recevoir (il peut être au milieu d'un calcul par exemple), elle est identique à la molécule finale puisque ce composant intervient dans un protocole sans fin.

Filtre $\mathbf{CF}_u$

syntaxe

M := P | C | M $\diamond$ M
P := $\mathbf{CF}_u$ | Φ
N := n_1
C := $i(n_1)$ | $o(n_1)$.

règles

$T_1 \equiv i(n_1) \ \diamond \ m, o(n_1) \diamond \ \mathbf{CF}_u \rightarrow m \ \diamond \ i(n_1), \mathbf{CF}_u \ \diamond \ o(n_1)$
$T_2 \equiv \mathbf{CF}_u \ \diamond \ o(n_1) \rightarrow o(n_1) \diamond \ \mathbf{CF}_u$

solution initiale $S_i := \mathbf{CF}_u \ \diamond \ o(n_1)$

solution finale $S_f := \mathbf{CF}_u \ \diamond \ o(n_1)$

Filtre $\mathbf{CF}_d$

syntaxe

M := P | C | M $\diamond$ M
P := $\mathbf{CF}_d$ | Φ
N := n_1
C := $i(n_1)$ | $o(n_1)$.

règles

$T_1 \equiv i(n_1) \diamond \ \mathbf{CF}_d, o(n_1) \ \diamond \ m \rightarrow \mathbf{CF}_d \ \diamond \ i(n_1), m \ \diamond \ o(n_1)$
$T_2 \equiv \mathbf{CF}_d \ \diamond \ i(n_1) \rightarrow i(n_1) \diamond \ \mathbf{CF}_d$

solution initiale $S_i := \mathbf{CF}_d \ \diamond \ i(n_1)$

solution finale $S_f := \mathbf{CF}_d \ \diamond \ i(n_1)$

GZip

La syntaxe du composant GZip est presque identique à celle des filtres : on ajoute seulement un canal supplémentaire car ce composant peut émettre et recevoir sur deux canaux distincts ; on ajoute aussi un ensemble E de marqueurs indiquant à l'adaptateur qu'il est en position pour terminer sa transmission ($\mathbf{end}_i$ signifie fin d'entrée et $\mathbf{end}_o$ fin de sortie).

La règle T_1 est la règle générale pour la communication inter-composant via un pipe dont on a vu deux instances particulières dans la spécification des filtres. La règle T_2 gère le comportement itératif de la première règle en faisant intervenir les marqueurs, c'est elle qui modélise le mode de communication par *pipe*. La règle T_3 représente la synchronisation avec l'environnement pour terminer les transmissions. Deux règles supplémentaires capturent la possibilité qu'a GZip pour interrompre indépendamment les transmissions. La dernière règle impose une attente entre le début et la fin des transmissions, ceci pour modéliser le calcul qu'effectue ce composant.

syntaxe

M := P | C | E | M $\diamond$ M
P := **GZ** | Φ
C := $i(N)$ | $o(n)$
N := n_1 | n_2
E := $\mathbf{end}_i$ | $\mathbf{end}_o$.

règles

$T_1 \equiv i(n) \diamond m_1, o(n) \diamond m_2 \rightarrow m_1 \diamond i(n), m_2 \diamond o(n)$

$T_2 \equiv e \diamond m \diamond c \rightarrow c \diamond e \diamond m$

$T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n), \mathbf{end}_i \diamond m_2 \diamond i(n) \rightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o, m_2 \diamond i(n) \diamond \mathbf{end}_i$

$T_4 \equiv \mathbf{end}_i \diamond m_1 \diamond \mathbf{GZ} \diamond m_2 \rightarrow m_1 \diamond \mathbf{GZ} \diamond m_2 \diamond \mathbf{end}_i$

$T_5 \equiv \mathbf{end}_o \diamond \mathbf{GZ} \diamond m \rightarrow \mathbf{GZ} \diamond m \diamond \mathbf{end}_o$

$T_6 \equiv \mathbf{GZ} \diamond m \rightarrow m \diamond \mathbf{GZ}$

solution initiale $S_i := i(n_1) \diamond \mathbf{end}_i \diamond o(n_2) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$

solution finale $S_f := \mathbf{GZ} \diamond m$

Adaptateur

La seule modification à apporter à la syntaxe du composant GZip est le fait que l'adaptateur peut communiquer par quatre canaux différents : un pour le filtre haut, un pour la sortie vers GZip, un autre pour l'entrée de GZip et un dernier pour la sortie vers le filtre bas.

Les trois premières règles sont identiques à celles de Gzip. Les deux dernières servent à instaurer le comportement itératif de l'adaptateur, qui est plus complexe que ce que nous avons vu jusqu'à maintenant. Ce comportement est caractérisé par une phase pendant laquelle le composant commute à partir d'un mode dans lequel il accepte des données du filtre et les transmet à un autre composant (en fait, GZip) vers un mode dans lequel il accepte des données d'un composant (GZip) et les transmet vers le filtre suivant (cf. action prépare de la spécification Wright et Community).

syntaxe

M := P | C | E | M $\diamond$ M

P := **AD** | Φ

C := $i(N)$ | $o(n)$

N := n_1 | n_2 | n_3 | n_4

E := $\mathbf{end}_i$ | $\mathbf{end}_o$.

règles

$T_1 \equiv i(n) \diamond m_1, o(n) \diamond m_2 \rightarrow m_1 \diamond i(n), m_2 \diamond o(n)$

$T_2 \equiv e \diamond m \diamond c \rightarrow c \diamond e \diamond m$

$T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n), \mathbf{end}_i \diamond m_2 \diamond i(n) \rightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o, m_2 \diamond i(n) \diamond \mathbf{end}_i$

$T_4 \equiv \mathbf{AD} \diamond i(n_1) \diamond m \rightarrow i(n_3) \diamond \mathbf{end}_i \diamond o(n_4) \diamond \mathbf{AD}$

$T_5 \equiv \mathbf{AD} \diamond i(n_3) \diamond m \rightarrow i(n_1) \diamond o(n_2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$

solution initiale $S_i := i(n_1) \diamond o(n_2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$

solution finale $S_f := \mathbf{AD} \diamond i(n_3) \diamond m$

2.3.4 Discussion

Interface

L'interface d'un composant est défini par sa partie syntaxique. Cette partie détermine les modes de communication (ensemble C), les canaux intervenant dans la communication (ensemble N), ainsi que les actions à synchroniser avec l'environnement. Les règles définissent les besoins des composants quand leur membre gauche contient des éléments syntaxiques non définis par le composant (méta-variables), et spécifient des relations entre les parties proposées et requises. Les interfaces de CHAM satisfont donc le principe de «Communication Integrity» [**?**].

La communication entre les composants peut être synchrone ou asynchrone : l'exemple le plus simple est le composant GZip. Les marqueurs de fin $\mathbf{end}_o$ et $\mathbf{end}_i$ peuvent soit être synchronisés avec ceux correspondant dans l'adaptateur, soit être désynchronisés (règles T_4 et T_5).

Connecteurs

Il n'y a pas de mécanisme pour spécifier uniquement un composant connectif. Cependant, la déclaration d'un ensemble de type de connexions dans la syntaxe autorise l'utilisation de styles de communication variés. Pour caractériser ces types de communication, il suffirait de rajouter des règles précisant le fonctionnement de ces types de communication. Dans l'exemple traité, il n'y a qu'un seul type de communication, caractérisé par la règle T_1 des composants **AD** et **GZ**.

En effet, contrairement à la spécification Wright ou COMMUNITY, la spécification de l'adaptateur ne différencie pas les modes de communication entre les composants (entrée/sortie pour les filtres et l'adaptateur, *pipe* pour GZip et l'adaptateur). Ceci est dû au fait que CHAM ne gère pas les valeurs communiquées. En effet, la distinction entre ces deux modes est faite au niveau du passage de valeurs : en une seule fois pour le premier (en tant qu'argument d'une fonction), en fragments pour le second. Ceci est un point faible du langage, car un problème architectural pourrait provenir de l'incompatibilité entre certains modes de communication, ce qui ne serait pas pris en compte par la description selon les cas.

Configuration

La configuration de composants CHAM en un système CHAM est une opération semi-formelle (dans le sens où la configuration de COMMUNITY est formelle). Dans [**?**], les auteurs donnent la définition du système pour chacune des constructions.

- La syntaxe d'un système CHAM est construite en prenant l'union des syntaxes des composants ; les méta-variables disparaissent. Dans l'exemple ci-dessus, on aurait :
$$\mathrm{P} := \mathbf{CF}_u \mid \mathbf{CF}_d \mid \mathbf{AD} \mid \mathbf{GZ}.$$
- Les règles de transition du système sont composées de l'union des règles des composants. Il est possible d'éliminer les règles redondantes (celles concernant les principes de communication, par exemple).
- La solution initiale est l'union des solutions initiales.
- La solution finale, par contre n'est pas l'union des solutions finales, mais plutôt un sous-ensemble du produit des solutions finales, puisque tous les états finaux d'un composant (s'il y en a plusieurs) ne seront pas des états finaux pour le système. L'exemple du Compressing Proxy est un cas dégénéré car chaque composant ne possède qu'une seule molécule finale, la solution finale du système sera donc la solution de toutes ces molécules.

Le mécanisme de configuration n'est pas explicite. Si on pouvait mettre une structure de catégorie sur les spécifications CHAM, ce mécanisme ressemblerait à celui de COMMUNITY : un pushout dont la partie partagée serait un reflet des modes de communications utilisés dans la description (sous la forme d'une règle).

Composition

Le mécanisme de composition proposé est du type «Composition Correcte». En effet, les auteurs ont développé un algorithme [?] permettant de décider si une composition était correcte ou non, en prenant l'absence de bloquage comme définition de la correction.

Cet algorithme nécessite trois définitions préalables :

- graphe de comportement effectif (AC graph). Ce graphe s'obtient à partir d'une CHAM par procédure constructive. Le point intéressant est que toutes les actions qui ne font pas intervenir d'autres composants sont nommées de la même manière (actions silencieuses, τ).
- graphe d'hypothèses sur l'environnement (AS graph). Ce graphe s'obtient à partir du premier dans lequel on modifie les actions communicantes par leur réciproque (un input par un output, ...) et les actions silencieuses par α.
- relation d'équivalence. Cette relation est inspiré de la notion de bisimulation de Milner. La modification principale apportée est que l'on autorise une bisimulation partielle : un graphe de comportement peut bisimuler un graphe d'hypothèses mais seulement sur une partie des hypothèses de celui-ci. En effet comme on veut relier plusieurs composants à la fois, on conçoit qu'une partie des hypothèses d'un composant soit vérifiée par l'un et une partie par l'autre. Après s'être assuré qu'un graphe de comportement bisimulait une partie du graphe d'hypothèses, on modifie ce dernier pour ne plus avoir à vérifier cette partie.

L'algorithme de correction de la composition se déroule en cinq étapes, et permet de vérifier avec ces trois définitions précédentes que toutes les hypothèses de tous les AS graphs sont bien vérifiées par l'ensemble des AC graphs. À la fin de l'exécution de cet algorithme, l'ensemble des AS Graphs doit être vide.

Une propriété assure finalement la correction de cet algorithme.

L'utilisation des $ASgraphs$ et $ACgraphs$ permet également d'envisager des spécifications faisant intervenir des formalismes différents. Si on peut construire à partir d'une spécification ces deux graphes, alors il est possible au moins de vérifier que la composition des composants sera sans bloquage.

Modularité

Les constructions *membrane et airlock* permettent d'obtenir des spécifications structurées. Une solution enfermée dans une membrane se comporte comme une seule molécule : on peut instancier la molécule m dans une règle comme $T_2 \equiv e \diamond m \diamond c \rightarrow c \diamond e \diamond m$ par une telle solution. Si on fait une analogie avec des langages de programmation, on peut dire qu'une membrane joue le rôle d'un module.

Les interfaces de ces solutions membranées (ou modularisées) sont définies par les *airlock*. Ces constructions isolent des molécules d'une membrane qui entreront dans la communication avec d'autres modules. Afin de conserver les possibilités de communication, des règles particulières sont rajoutées pour décrire les relations entre l'opérateur d'airlock et « $\diamond$ ». Ces règles pourraient se compter dans les règles générales puisqu'elles décrivent un mécanisme appelé à se retrouver dès que les spécifications sont modularisées.

Raffinement

Il est possible de regarder formellement une notion de raffinement local ou global. Pour qu'une architecture raffine (simule) une autre architecture, il suffit qu'elles aient la même syntaxe et que si on regarde une séquence de transformations $S_1 \to ... \to S_n$ de la première, alors il existe une séquence $S'_1 \to ... \to S'_p$ de la seconde telle que $S_1 \subset S'_1, ..., S_i \subset S'_j, ..., S_n \subset S'_p$.

Dynamisme

Les architectures dynamiques sont traités implicitement dans ce langage. En effet, les règles ne précisent pas les liens entre les composants en les nommant et la configuration ne rajoute pas d'information quant à ces liens. Il est donc possible d'avoir une reconfiguration du système (comme dans l'exemple de [?]) sans ajouter de construction particulière.

Analyse

La sémantique opérationnelle du langage permet d'effectuer simplement des preuves sur les descriptions d'architectures en se basant sur la syntaxe. Voici quelques exemples de propriété que l'on peut vérifier :
- on peut vérifier que toute séquence de transformations partant de la solution initiale termine (on arrive sur une solution *inerte*). Autrement dit, que toute séquence de transformations est infinie si et seulement si les données en entrée sont infinies.
- on peut vérifier syntaxiquement que la modularisation proposée est correcte, c'est à dire que les interfaces permettent bien de communiquer avec le système et qu'il n'y aura pas de bloquage. Cette preuve diffère de l'algorithme donné dans une section précédente.

2.4 Comparaison

Cette section va mettre en valeur le fait que les concepts utilisés dans la formalisation d'architectures de connexion par ces trois ADL (Wright, COMMUNITY et CHAM) sont similaires. En effet, comme on l'a fait remarquer lors des paragraphes s'intéressant à la sémantique de ces langages, il est possible de voir ces spécifications comme étant une écriture mathématique d'automates que l'on composerait par un produit synchronisé. Cette notion est naturelle, car la vision par automates d'un graphe «boîtes et lignes» défini à partir du cahier des charges d'un système est la plus simple du point de vue mathématique.

Il y a cependant quelques points de différences avec les idées provenant des techniques de Model-Checking. Dans un premier temps, nous donnerons une idée de la traduction de ces spécifications vers des automates, puis nous discuterons du comportement de chacun des langages vis-à-vis des points du cadre. Cette deuxième partie permettra de montrer les points de différence avec la vision du Model-Checking.

2.4.1 Traduction

Avant de donner une idée de la traduction, nous rappelons la définition du produit synchronisé de n automates :

Définition 11 *Soit une famille d'automates $\mathcal{A}_i =< Q_i, E_i, T_i, q_{0_i} >$, $i = 1..n$ (ensemble d'états, ensemble des étiquettes, fonction de transition, état initial) auquel on rajoute l'étiquette*

'-' signifiant «ne rien faire» pour tout automate qui resterait inactif lors d'une transition globale de l'ensemble des composants. Le produit cartésien $\mathcal{A}_1 \times ... \times \mathcal{A}_n$ *est l'automate* $\mathcal{A} =< Q, E, T, q_o >$ *avec :*
- $Q = Q_1 \times ... \times Q_n$
- $E = \prod_{1 \leq i \leq n}(E_i \cup \{-\})$
- $T = \left\{ \begin{array}{c} ((q_1, ..., q_n), (e_1, ..., e_n), (q'_1, ..., q'_n))| \text{ pour tout } i \\ e_i = \ '-' \text{ et } q'_i = q_i \text{ ou } e_i \neq \ '-' \text{ et } (q_i, e_i, q'_i) \in T_i \end{array} \right\}$
- $q_0 = (q_{0_1}, ..., q_{0_n})$

On peut alors donner des intuitions sur cette traduction :
- Wright. La sémantique d'un composant Wright est le processus correspondant à la spécification (**spec**) de ce composant. Il est possible de définir une application f entre la modélisation d'un composant par un automate et celle par un processus algébrique CSP, puisque les automates sont un modèle d'exécution d'un processus CSP. On ne peut pas déterminer un unique automate à partir d'un composant Wright puisque la spécification ne comporte pas d'état initial. La sémantique d'un système est la mise en parallèle de processus CSP (la *glue* et les ports), qui, dans cette application f, se projette vers une forme de produit synchronisé (soit P et Q deux processus, on a $f(P \parallel Q) = f(P) \times f(Q)$). L'utilisation de la déclaration d'*Attachments*, qui consiste à relier des ports à des rôles, permet juste d'assurer que la *glue* contraint bien les échanges entre les composants à l'intérieur du système.
- Community. La sémantique des programmes COMMUNITY est directement basée sur les automates, la traduction consiste donc à prendre la fonction sémantique du langage. Le mécanisme de composition des programmes suit le même principe que la composition parallèle de processus concurrents [?], il est donc aussi semblable à un produit synchronisé.
- CHAM. La construction d'un automate à partir d'une spécification CHAM est décrite par Inverardi et Wolf dans [?] (les *AC Graphs*). Le mécanisme de composition décrit par l'algorithme donné dans la section **??** est basé sur le matching des AC Graphs par des AS Graphs, qui se fait sur les noms de molécules observables (celles qui ne sont pas silencieuses). Ce mécanisme est semblable à un produit dans lequel on synchronise ces molécules observées.

2.4.2 Cadre

On cherche dans cette section à comparer chacun de ces formalismes sur les points du cadre énonçés dans l'introduction du chapitre. Le résultat se trouve dans le tableau synoptique suivant.

	Wright	COMMUNITY	CHAM
Interfaces	complète	complète	complète
Connecteurs	oui	non	non
Configuration	explicite (texte)	explicite (diagramme)	implicite
Composition	opérateur ‖ CSP	pushout	name-matching
Modularité	imports de styles	oui (construction progressive)	oui (membrane)
Raffinement	non	oui (local)	oui
Dynamisme	oui	oui	oui
Analyse	propriétés des traces	indirecte (dans LTL)	oui (réécriture)

2.4.3 Discussion et Conclusions

L'un des objectifs fixés par ces langages, la description d'architectures de connexion par formalisation du diagramme «boîtes et lignes», est partiellement réalisé. En effet, cela implique de donner une véritable construction pour les connexions, ce que ne fait que le langage Wright. Le fait que les deux autres ADL ne possèdent pas une telle construction n'est pas prohibitif puisque cela n'empêche pas la spécification du problème du Compressing Proxy. Cependant, le langage CHAM n'a pas la même puissance que les deux autres puisque les modes de communication ne sont pas différenciés, ce qui peut poser des limites dans la spécification d'autres exemples.

La formalisation de cette notion de connecteur apporte surtout une justification au fait que ces descriptions architecturales ne sont pas exactement celles qu'on aurait fait avec une technique de Model-Checking. En effet, les connecteurs de Wright permettent d'ajouter à la modélisation un automate qui n'apparaîtra pas dans le système final, mais qui sert à vérifier une sorte de compatibilité avant la composition par produit. Cette vérification préliminaire peut se voir comme une vérification de typage avant la compilation au niveau des langages de programmation.

Un point commun avec les techniques du Model-Checking est que la spécification d'un système est facilitée par le fait que l'on ne s'occupe pas des fonctionnalités des «boîtes» : la description d'un système complexe peut se faire rapidement, avec cependant les mêmes limitations que pour le Model-Checking (finitude des états des automates).

Les trois langages permettent de décrire des architectures dynamiques, aussi bien dans le cas du calcul mobile (principalement le langage CHAM), que de la modification interactive des composants. Dans les trois cas, la formalisation de cette dernière notion de dynamisme passe par l'utilisation de techniques de réécriture. On peut comprendre la motivation pour l'utilisation de ces techniques : une modification dynamique des connexions peut se voir comme une transformation du diagramme, dont la plus simple formalisation est une *réécriture* de ce diagramme (cf. [?]).

L'analyse des descriptions d'architectures n'est pour le moment pas très développée dans ces langages. Ce point, essentiel, n'a pas fait l'objet de résultats intéressants permettant de donner du poids à ces descriptions par rapport à ce que l'on peut faire dans le cas du Model-Checking.

Ces trois langages ne sont pour le moment qu'au stade du prototypage (même si Wright a été utilisé dans un exemple industriel), donc plusieurs améliorations peuvent y être apportées afin de les rendre plus maniables et plus performants quant à l'analyse que l'on peut effectuer sur les spécifications. Chaque langage a apporté un concept intéressant : la formalisation des connexions pour Wright, la formalisation de notions d'ingénierie dans les catégories pour COMMUNITY, le mécanisme de composition pour CHAM. Il serait peut-être fructueux de les rassembler en un seul formalisme.

Chapitre 3

Spécification d'Architectures dans CASL

3.1 Introduction

Le langage CASL (*Common Algebraic Specification Language*) [?] est un formalisme de spécifications algébriques pour décrire des structures CASL : des algèbres multi-sortées avec des relations de sous-sortage, des opérations partielles et des prédicats.

Une spécification de base en CASL comporte une déclaration de signature Σ, c'est-à-dire un ensemble de sortes munies de leur relation de sous-sortage ainsi que des noms d'opérations et des noms de prédicats avec leur profil, et un ensemble d'axiomes Φ pour donner des propriétés sur les structures que l'on considérera comme les modèles de la spécification. La logique utilisée pour écrire les axiomes est essentiellement la logique équationnelle du premier ordre. Pour chaque signature Σ, la classe de toutes les Σ-structures est notée $\mathbf{Mod}[\Sigma]$. Toute spécification SP de base dénote une classe $[\![SP]\!] \subseteq \mathbf{Mod}[\Sigma]$ constituée des Σ-structures vérifiant tous les axiomes de Φ : $[\![SP]\!] = \{A \in \mathbf{Mod}[\Sigma] \mid A \models \Phi\}$.

Une première série de constructions est proposée afin de structurer la spécification : translation, union, dissimulation, extensions *free* et *loose*. Les spécifications génériques ainsi que leur instantiation par un style de pushout sont aussi proposées. Les spécifications structurées par ces constructions ont une sémantique compositionnelle dans laquelle chaque spécification SP détermine une signature $Sig[SP]$ et une classe $[\![SP]\!] \subseteq \mathbf{Mod}[Sig[SP]]$ de modèles. On dira que SP est consistante si $[\![SP]\!]$ est non vide.

Une seconde série de constructions, qui a pour objectif de décrire l'architecture modulaire du système en cours de développement, est proposée dans [?]. L'introduction de constructions différentes est motivée par des expériences pratiques et des travaux théoriques qui montrent la nécessité de distinguer les deux types de constructions. Le modèle d'architecture sous-jacent est celui d'«architecture d'implémentation» qui vise à relier des modules de spécifications à des modules d'un langage de programmation (comme ceux de Ada ou de ML). Dans ce modèle, les types de connexions entre les composants, les modes de communication, les aspects dynamiques, ne sont pas pris en compte.

Nous détaillons dans un premier temps les motivations qui ont conduit à la définition de cette deuxième série de constructions de spécifications, puis nous donnons une idée de leur syntaxe ainsi que de leur sémantique. Dans une seconde partie, nous montrerons comment il est possible de décrire aussi des architectures de connexion. Cette partie est basée sur la conclusion du chapitre précédent, en particulier sur le fait que les idées sous-jacentes à la description d'architectures de connexion sont celles d'automates et de synchronisation. Cette possibilité de décrire des architectures de connexion dans un formalisme à priori dédié à la spécification d'architectures d'implémentation montre que ces deux concepts ne sont pas disjoints.

3.2 Les constructions d'architectures

3.2.1 Motivations

Les motivations qui ont conduit à définir ces constructions proviennent du processus de développement d'un programme à partir de spécifications. Ce processus peut se voir comme une spécification des besoins SP_0 que l'on raffine successivement jusqu'à obtenir le programme P lui-même : $SP_0 \rightsquigarrow SP_1 ... \rightsquigarrow P$. Cependant, CASL n'est pas dédié à l'accomplissement complet de ce processus (ce n'est pas un *wide-spectrum language*) : il présuppose la détermination d'une signature $Sig[P]$ et d'un modèle $[\![P]\!]$ de cette signature reflétant le programme [1]. Ensuite, CASL permet de vérifier que la chaîne de raffinements successifs aboutit à une spécification SP_{fin} dont les modèles contiennent le modèle du programme ($[\![P]\!] \in [\![SP_{fin}]\!]$).

On peut imaginer un processus plus efficace permettant un découpage de la spécification en modules comme : $SP \rightsquigarrow OP(SP_1, ..., SP_n)$, qui seront ensuite raffinés indépendamment puis recomposés, par exemple à l'aide de constructions propres à un langage de programmation ($LINK_{OP}$). Le problème vient du fait que l'opérateur de découpage OP ne peut pas être un opérateur de construction des spécifications puisqu'une condition nécessaire est que cet opérateur doit être monotone par rapport au raffinement :

$$\frac{SP_1 \rightsquigarrow SP'_1 \; ... \; SP_n \rightsquigarrow SP'_n}{OP(SP_1, ..., SP_n) \rightsquigarrow OP(SP'_1, ..., SP'_n)}$$

Il faut pour cela que l'opérateur OP soit *constructif*, c'est-à-dire que pour chaque réalisation $P_1, ..., P_n$ de $SP_1, ..., SP_n$, on doit être capable de construire une réalisation $LINK_{OP}(P_1, ..., P_n)$ de $OP(SP_1, ..., SP_n)$. Or la plupart des opérateurs de construction des spécifications ne vérifient pas cette propriété (voir [?] pour des exemples), il faut donc en introduire des nouveaux.

D'autres motivations moins techniques sont engendrées par le fait que les opérations sur les spécifications ne sont pas censées refléter la structure finale du programme. On doit pouvoir, au cours d'un raffinement, modifier cette structure et la faire évoluer car sinon cela voudrait dire que l'on connaît cette structure dès le départ.

3.2.2 Spécifications Architecturales

Syntaxe

Une spécification architecturale consiste en une liste de déclaration d'unités, indiquant les modules requis avec une spécification pour chacun d'entre eux, et une expression de modules (ou *unit term*) décrivant la manière dont ces modules seront combinés :

[1] Cette étape n'est encore réalisée pour aucun véritable langage de programmation

units $U_1 : SP_1$;
...
$U_n : SP_n$
result $LINK_{OP}(U_1, ..., U_n)$

Les déclarations et les définitions d'unités peuvent se faire par plusieurs opérateurs :
- généricité implicite : la spécification d'un module d'une unité nécessite une autre unité développée indépendamment ($U_2 : SP_1$ **given** U_1) ;
- généricité explicite : on veut pouvoir utiliser dans le *result term* plusieurs fois la même unité dont la spécification est paramétrée, en modifiant son paramètre. On déclare pour cela une *generic unit* de la forme $F : SP_1 \to SP_2$;
- amalgame : mettre en commun des unités déjà développées (U_1 **and** U_2) ;
- réduction et renommage : permettre de cacher des constructions auxiliaires de la spécification ou de les renommer ;
- instantiation : pouvoir instancier une *generic unit* par une unité dont la spécification est un raffinement du paramètre ;
- construction d'unité générique : soit la description suivante

  units $F : SP_0 \to SP_1$;
  $G : SP_1 \to SP$
  result $\lambda X : SP_0 \bullet G[F[X]]$

  alors cette description construit une unité réalisant $SP_0 \to SP$.

Sémantique

Pour donner la sémantique des *unit term*, il faut au préalable définir la sémantique des *generic unit*. Si on considère une spécification de la forme $SP' \to SP$, alors SP est implicitement vue comme une extension de SP' : si Σ est la signature de SP et Σ' celle de SP', alors on a $\Sigma' \subseteq \Sigma$ et $[\![SP]\!]|_{\Sigma'} \subseteq [\![SP']\!]$. Un problème émerge lors de la formalisation de la sémantique : comment gérer le fait que plusieurs arguments d'une *generic unit* peuvent posséder des éléments syntaxiques communs, puisqu'ils peuvent à priori être développés séparément ? Cette question est soulevée dans [?] et débouche sur la notion de *sharing* (ou partage).

La sémantique des *unit term* doit prendre en compte le mécanisme de composition des modules dans un langage de programmation : on ne peux pas combiner n'importe quels modules entre eux. Il faut que la sémantique du *result term* dénote une structure, c'est-à-dire qu'il soit consistant. Deux types de propriétés sont utilisées :
- syntaxiques : on vérifie la bonne formation des unités à partir d'informations sur les signatures et sur le partage qu'elles doivent accepter ;
- sémantiques : pour savoir si on peut remplacer une unité par une autre dans une définition, on construit un environnement ρ associant les noms d'unités à des structures puis un contexte γ dans lequel les noms d'unités sont associés à des spécifications. On dira qu'un environnement convient à un contexte si pour tout nom d'unité U, $\rho(U) \in [\![\gamma(U)]\!]$.

La formalisation de ces propriétés dans [?] permet d'assurer qu'une fois le processus de raffinement terminé et qu'on possède une réalisation de chaque unité déclarée, la composition des unités dans le *result term* dénote une structure (cf. [?], théorème 1). L'avantage de cette méthode est de pouvoir déterminer à l'avance si un *result term* est correct, avant même de fournir les réalisations des unités.

3.3 Description d'architectures de connexion

Comme nous l'avons vu dans la partie précédente, le mécanisme de spécifications architecturales développé dans [?] se projette facilement sur les mécanismes de modularisation des langages de programmation. En ce sens, ce mécanisme modélise des architectures d'implémentation (voir Chapitre 1, déf. 3). Cette section montre comment il est possible de décrire également des architectures de connexion, c'est-à-dire de modéliser dans CASL des concepts tels que : connexion, dynamisme, communication.

Cette modélisation a pour objectif de mieux comprendre les modélisations de ces concepts qui ont été montrées dans le chapitre précédent, dans des formalismes à priori mieux adaptés. Tous les concepts n'ont pas trouvé d'échos dans la modélisation de CASL, mais un travail futur serait d'étendre et de compléter cette modélisation afin de se rapprocher des autres formalismes.

3.3.1 Méthode

La méthode employée se base sur la modélisation d'une architecture en termes d'automates et dont la composition se fera par une opération de synchronisation sur ces automates. Nous cherchons à mettre en valeur les spécifications qui correspondent aux modes de communication, c'est-à-dire savoir quelles sont les spécifications qui ne se projettent pas exactement sur un module de programme. La structure de spécification décrite dans la section précédente est conservé : un «composant» d'architecture va s'écrire comme une unité et toutes les unités sont combinées dans la *result unit* en utilisant des unités génériques représentant les modes de communication.

Il est plus intuitif de considérer ces unités génériques comme de nouvelles opérations de définitions d'unités. En effet, elles correspondent à la définition formelle d'opérations sur les automates (synchronisation, concaténation, ...), qui construisent un automate à partir de plusieurs automates donnés en arguments. Concevoir ces unités génériques en terme de modules n'a alors plus de sens, ce qui se rapproche du concept de connecteurs dans Wright.

Composants

Les composants sont décrits par les spécifications des unités en tant qu'automates. Pour que les modèles de ces spécifications contiennent des automates, il suffit de partir de la définition d'un automate comme un quintuplet $< A, E, T, I, F >$ (alphabet, ensemble d'états, fonction de transition, ensemble d'états initiaux, ensemble d'états finaux) puis de se donner des sortes pour représenter les ensembles et des opérations ou des prédicats munis d'un profil pour représenter les fonctions et les constantes. On obtient une spécification de la forme :

spec AUTOMATE_A =

sort $Actions[A], Etats[A], Elem$

op $init[A], final[A] : Etats[A];$
  $a : Elem \rightarrow Actions[A]$

pred $is\_step[A] : Etats[A] \times Actions[A] \times Etats[A]$

axioms ...

end

Les axiomes permettent de décrire les étapes de transitions autorisées par l'automate par des relations booléennes puisque la fonction de transition est spécifiée par un prédicat. L'automate décrit ci-dessus ne possède qu'un état initial et qu'un état final ; dans un cas plus général, il suffit de rajouter des prédicats $is\_initial$ et is_final avec les axiomes correspondants.

Les modèles de cette spécification sont tous les automates possédant les transitions données par les axiomes sous la forme $is\_step[A](etat, action, etat')$.

Opérations

La spécification de l'opération de synchronisation suit le même principe : étant donné deux automates, on peut définir formellement le produit synchronisé de ces deux automates (voir la définition **??**, page 43). Cette définition construit un automate particulier à partir des automates donnés en arguments. Pour exprimer cette opération dans CASL, on donnera au-dessus des spécifications des arguments la signature de l'automate produit à laquelle on rajoutera les axiomes limitant les modèles à un unique automate. La spécification AUTOMATE[B] comporte au moins les mêmes sortes que celle de l'AUTOMATE[A] en les indiquant par la lettre B. On obtient la spécification suivante de la synchronisation sur une action de deux automates :

spec PRODUIT =
  AUTOMATE_A **and** AUTOMATE_B
then
  free { **type** $Actions[C] ::=$ **sort** $Actions[A]$ | **sort** $Actions[B]$
    axiom $a(e) = b(e') \Leftrightarrow e = e'$ **}**
then
  free type
    $Etats[C] ::= init[C] \mid \_\_ \times \_\_(Etats[A]; Etats[B])$
  pred $is\_step[C] : Etats[C] \times Actions[C] \times Etats[C]$
  vars $x : Etats[C]; z : Elem \to Actions[C];$
    $u1, u2 : Etats[A]; v1, v2 : Etats[B];$
    $e, e' : elem$
  axioms
  - $z(e) \in Actions[A] \wedge z(e) \neq a(e') \Rightarrow (is\_step[C](u1 \times v1, z(e), u2 \times v2)) \Leftrightarrow is\_step[A](u1, z(e) :: Actions[A], u2))$
  - $z(e) \in Actions[B] \wedge z(e) \neq b(e') \Rightarrow (is\_step[C](u1 \times v1, z(e), u2 \times v2) \Leftrightarrow is\_step[B](v1, z(e) :: Actions[B], v2))$
  - $z(e) \in Actions[A] \wedge z(e) \in Actions[B] \Rightarrow (is\_step[C](u1 \times v1, z(e), u2 \times v2) \Leftrightarrow is\_step[A](u1, z(e) :: Actions[A], u2) \wedge is\_step[B](v1, z(e) :: Actions[B], v2))$

end

Cette spécification ne suit pas exactement la définition 11. En effet, l'alphabet de l'automate produit n'est pas composé de couples car on renomme les actions du type (a, -), ou (-, b), par a ou b. De plus la construction libre utilisée pour construire l'ensemble $Actions[C]$ sert à identifier les actions de l'automate produit avec l'union des actions des automates A et B. Or la synchronisation sous-entend qu'une action de chacun des automates est identifiée dans le produit, c'est pourquoi on quotiente cette union.

Les trois axiomes décrivent totalement la fonction de transition (sous forme de prédicat) en fonction de l'appartenance de l'étiquette à l'une des catégories de $Actions[C]$:

- si l'action z(e) est de la forme a, alors la fonction de transition a la même valeur que la fonction de transition de l'automate A sur la première composante du couple représentant l'état de l'automate produit ;
- si l'action z(e) est de la forme b, alors la fonction de transition a la même valeur que la fonction de transition de l'automate B ;
- si l'action z(e) est l'action synchronisée, alors la fonction de transition modifie chacune des composantes du couple d'état en fonction de la fonction de transition des deux automates.

Une généralisation possible de cette spécification est d'augmenter le nombre d'automates à synchroniser et/ou le nombre d'actions à synchroniser. Cependant, il n'est pas possible de paramétrer cette spécification par un entier n représentant un nombre arbitraire d'automates et/ou d'actions. Cette limitation empêche pour le moment la spécification d'architectures dynamiques (par exemple en autorisant la connexion de n clients à un serveur).

Lorsqu'on va vouloir utiliser cette spécification en tant que spécification d'une unité générique, il va falloir la transformer en un foncteur. Cette transformation se fait assez simplement, puisque les arguments de ce foncteur sont les arguments de l'opération de synchronisation : les automates A et B. Une autre vérification à apporter est de montrer que ce foncteur est persistant : cela signifie qu'étant donné un automate C et un automate A, on doit pouvoir dire si C est le résultat d'une opération ayant A comme argument. Plus précisément, si C est construit avec A, alors on doit pouvoir reconstruire A à partir des informations contenues dans C.

Le foncteur ne rajoute pas de nouvelles valeurs dans les sortes [A], puisque l'on prend la construction libre au-dessus de ces sortes. Il suffit alors de regarder si le prédicat de transition ne crée pas de nouvelles transitions dans A. Le premier axiome ne fait que reprendre les transitions de A différentes de l'action a, le second reprend les transitions de B différentes de b. Le dernier axiome concerne l'action synchronisée : la transition est faite sur les deux automates simultanément et ne peut pas modifier les prédicats is_step de A ou B.

On peut continuer la spécification des opérations sur les automates par la concaténation, ce qui se fait exactement de la même manière. On peut définir ainsi une sorte d'algèbre de connexions à partir desquelles on pourrait décrire les différents types décrits dans [**?**]. Afin de tester ces constructions, on va traiter l'exemple du Compressing Proxy du chapitre précédent dans CASL.

3.3.2 Compressing Proxy

La spécification de cet exemple est déduite des automates que l'on peut construire à partir des spécifications écrites dans les trois ADL du chapitre précédent. On distingue trois types d'éléments : filtres, GZip et adaptateur. Nous ferons la même simplification que celle faite pour la spécification COMMUNITY : on ne distingue pas le filtre haut du filtre bas, il y aura une seule spécification pour ces deux composants avec les deux actions d'entrée/sortie possible dont on n'en synchronisera qu'une seule. On supposera, pour faciliter les descriptions, que les données transmises entre les composants sont d'un type *Elem* qui pourra représenter des entiers, des caractères ou tout autre type de données.

Filtres

Un filtre est donc un automate à quatre actions : lire des valeurs, préparer des valeurs (autrement dit l'action qui représente le *calcul* du filtre), écrire la valeur calculée, puis une dernière action pour fermer le flot de données. La fermeture du flot place le filtre dans un état final ($état_4$) que l'on assimilera à une fin avec succès du protocole.

spec FILTRE =

sort $Actions[F], Etats[F], Elem$

op $état\_1, état\_2, état\_3, état\_4 : Etats[F];$
$lecture, écriture : Elem \to Actions[F]$
$calcul, fin : Action[F]$

pred $is\_step[F] : Etats[F] \times Actions[F] \times Etats[F]$

var $e, e' : Elem$

axioms

- $is\_step[F](état\_1, lecture(e), état\_2)$
- $is\_step[F](état\_2, calcul, état\_3)$
- $is\_step[F](état\_3, écriture(e'), état\_1)$
- $is\_step[F](état\_3, fin, état\_4)$

end

GZip

Ce composant possède normalement deux actions d'entrée/sortie et deux actions pour préciser la fin des transmissions (*end*), plus des actions auxiliaires pour spécifier le comportement itératif du protocole (*retour*). Cependant, comme pour la modélisation en COMMUNITY, nous distinguons les actions de fin qui seront synchronisées de celles qui ne le seront pas (fin). On obtient la spécification suivante :

spec GZIP =

sort $Actions[G], Etats[G], Elem$

op $état\_1, état\_2, état\_3, état\_4, état\_5 : Etats[G];$
$entrée, sortie : Elem \to Actions[G]$
$fin_i, fin_o, end_i, end_o, retour_i, retour_o, fin : Action[G]$

pred $is\_step[G] : Etats[G] \times Actions[G] \times Etats[G]$

var $e, e' : Elem$

axioms

- $is\_step[G](état\_1, entrée(e), état\_2)$
- $is\_step[G](état\_2, retour_i, état\_1)$
- $is\_step[G](état\_2, fin_i, état\_3)$
- $is\_step[G](état\_1, end_i, état\_3)$
- $is\_step[G](état\_3, sortie(e'), état\_4)$

- $is\_step[G](état\_4, retour_o, état\_3)$
- $is\_step[G](état\_4, fin_o, état\_5)$
- $is\_step[G](état\_4, end_o, état\_5)$
- $is\_step[G](état\_5, fin, état_1)$

end

Adaptateur

La spécification de ce composant suit exactement celle qui a été faite en COMMUNITY : on aura 8 états distincts et 10 actions permettant de modéliser les deux types de communication. Un point intéressant par rapport à la spécification COMMUNITY est qu'il est possible de rajouter un axiome similaire à celui qui peut être écrit dans la spécification Wright : la valeur reçue du filtre haut est fragmentée puis transmise à GZip incrémentalement. Cet axiome ne figure pas dans la spécification car il demande l'introduction de fonctions auxiliaires qui gêneraient la compréhension de l'exemple.

Comme on n'a pas à notre disposition la différence entre une variable locale et une variable globale, on utilise quatre noms de variables distincts représentant les valeurs présentes dans le flot de données.

spec ADAPTATEUR =

sort $Actions[A], Etats[A], Elem$

op $état\_1, état\_2, état\_3, état\_4, état\_5, état\_6, état\_7, état\_8 : Etats[A];$
$lecture, écriture : Elem \rightarrow Actions[A]$
$entrée, sortie : Elem \rightarrow Actions[A]$
$end_i, end_o, retour_i, retour_o, fin, prépare : Action[A]$

pred $is\_step[A] : Etats[A] \times Actions[A] \times Etats[A]$

var $x\_1, y\_1, x\_2, y\_2 : Elem$

axioms

- $is\_step[A](état\_1, lecture(x\_1), état\_2)$
- $is\_step[A](état\_2, sortie(y\_1), état\_3)$
- $is\_step[A](état\_3, retour_o, état\_2)$
- $is\_step[A](état\_3, end_o, état\_4)$
- $is\_step[A](état\_4, prépare, état\_5)$
- $is\_step[A](état\_5, entrée(x\_2), état\_6)$
- $is\_step[A](état\_6, retour_i, état\_5)$
- $is\_step[A](état\_6, end_i, état\_7)$
- $is\_step[A](état\_7, écriture(y\_2), état\_8)$
- $is\_step[A](état\_8, fin, état\_1)$

end

Spécification Casl

La spécification CASL du Compressing Proxy utilise les descriptions architecturales définies dans la section précédente. La seule opération présente dans cette spécification est l'opération de synchronisation, qui va identifier les actions simultanées. Comme les échanges entre le filtre et l'adaptateur sont constitués de deux actions distinctes (lecture/écriture de valeurs, la notification de fin fin), on préférera définir une seconde opération de produit synchronisé prenant deux actions ($Prod2$). Il faut pour cela modifier la spécification des automates A et B de telle sorte que suffisement d'actions apparaissent.

De même pour les échanges entre l'adaptateur et GZip, qui sont constitués de quatre actions ($Prod4$), il faudra ajouter des actions aux spécifications des automates A et B.

spec PRODUIT_2 =
  AUTOMATE_A **and** AUTOMATE_B
then
  free { **type** $Actions[C]$ ::= **sort** $Actions[A]$ | **sort** $Actions[B]$
    axiom $a_1(e) = b_1(e') \Leftrightarrow e = e'$
      $a_2 = b_2$ **}**
then
  ...
end

On transforme ces spécifications en foncteurs, ce qui se fait syntaxiquement, on obtient alors la spécification architecturale suivante :

arch spec COMPRESSING PROXY =
units $F1$: FILTRE
  $F2$: FILTRE
  G : GZIP
  A : ADAPTATEUR
  $Prod2$: AUTOMATE_A × AUTOMATE_B → PRODUIT_2
  $Prod4$: AUTOMATE_A × AUTOMATE_B → PRODUIT_4
  $U1 = Prod2[F1$ **fit** $f1, A$ **fit** $f2]$
  $U2 = Prod4[U1$ **fit** $f3, G$ **fit** $f4]$
  $U3 = Prod2[U2$ **fit** $f5, F2$ **fit** $f6]$
result $U3$
end

Les morphismes f_i servent à appliquer les *generic unit* à des arguments plus complexes que les spécifications AUTOMATE_A ou AUTOMATE_B, en l'occurrence les spécifications des composants. Cette utilisation de morphismes correspond à ce qui a été fait dans COMMUNITY avec le diagramme de configuration **??** (page 33), l'intermédiaire des *Channels* est intégré dans la définition de l'opération de produit synchronisé. On remarque que les différences de produit synchronisé correspond à la différence dans les Channels.

On ne donne les valeurs de ces morphismes que sur les éléments changeant de valeur, les notations $*[X]$ désignent toutes les sortes et les opérations des spécifications indicées par la lettre $[X]$.

- $f1 : *[F] \mapsto *[A], écriture \mapsto a_1, fin \mapsto a_2$
- $f2 : *[A] \mapsto *[B], lecture \mapsto b_1, end_i \mapsto b_2$
- $f3 : *[C] \mapsto *[A], sortie \mapsto a_1, end_o \mapsto a_2, entrée \mapsto a_3, end_i \mapsto a_4$
- $f4 : *[G] \mapsto *[B], entrée \mapsto b_1, end_i \mapsto b_2, sortie \mapsto b_3, end_o \mapsto b_4$
- $f5 : *[C] \mapsto *[A], ériture \mapsto a_1, end_o \mapsto a_2$
- $f6 : *[F] \mapsto *[A], lecture \mapsto b_1, fin \mapsto b_2$

Cette spécification contient le bloquage décrit par Inverardi et Wolf, en effet si l'automate GZip effectue la transition fin_i, alors l'automate adaptateur ne passera pas de l'état 3 à l'état 4, ce qui empêchera par la suite la réception des données provenant de GZip.

3.3.3 Discussion

Interface

Si on considère que les unités génériques formalisent les modes de communication entre les composants, les interfaces sont alors les arguments de ces unités génériques. Par exemple, dans le cas de la synchronisation, les spécifications des automates A et B précisent la signature minimale que les composants doivent posséder, en laissant toute transition possible, tous les ensembles d'états et d'actions non précisés. Une interface dans une spécification CASL est donc une spécification dont la classe des modèles est très vaste. Cette définition des interfaces permet la réutilisation des spécifications pour un grand nombre d'applications. Si on peut définir des types de composants spéciaux, leur spécification en CASL permettrait de construire des librairies.

La notion d'interface complète est présente dans la spécification architecturale CASL. En effet, l'utilisation que nous avons faite des unités génériques ne reflète pas la motivation principale de leur construction. Une unité générique sert justement à déclarer toutes les informations nécessaires à la construction d'une unité dans l'argument du foncteur. Une déclaration de la forme $U_1 : SP \to SP'$ signifie que pour construire une réalisation de SP', il est nécessaire de donner d'abord une réalisation de SP. On a donc la déclaration des fonctionnalités nécessaires (dans SP) aussi bien que la déclaration des fonctionnalités proposées (dans SP'). La relation entre ces deux types de fonctionnalités se fait dans la spécification SP'. On retrouve donc toute la définition 2.

Connecteurs

Il n'y a pas de mécanisme explicite pour spécifier les connecteurs dans CASL. On possède néanmoins deux possibilités pour spécifier des types de communication :

- identifier des composants qui n'effectuent pas de calcul, c'est-à-dire qui seraient spécifiés par des connecteurs en Wright, et décrire dans cette spécification les axiomes caractérisant les modes de communication. Dans l'exemple traité, il s'agit plutôt de décrire les *relations* entre les modes de communication ;
- rajouter dans la spécification du système un composant gérant les échanges de données, comme cela est fait en COMMUNITY. On peut utiliser les résultats de [?] dans CASL puisque la modélisation du comportement des connecteurs par pushout peut se traduire par une modélisation en termes de synchronisation. Dans le cas de l'exemple, on peut rajouter un composant $Buffer$ qui servirait de tampon entre l'adaptateur et GZip, modélisant ainsi la communication via un *pipe* Unix.

Configuration

La configuration est explicite : le *result term* donne la manière dont sont connectées les différentes unités. Aux opérations fournies dans [?], il est possible d'ajouter d'autres constructions permettant d'étendre la classe des architectures descriptibles par ce formalisme, comme nous l'avons fait avec le produit synchronisé.

Composition

Le mécanisme de composition des unités dans un *result term* est esquissé dans la partie décrivant la sémantique des constructions architecturales. La bonne formation de ce terme nécessite plusieurs niveaux de vérification (syntaxique et sémantique), pourtant on ne pourrait pas classer ce mécanisme comme étant de type «Composition Correcte». L'exemple du Compressing Proxy montre qu'il est possible de modéliser des composants et de les composer bien qu'il existe un bloquage possible. On a donc à faire plutôt à un principe de «Connectabilité», proche de celui de COMMUNITY.

Les considérations sémantiques apparaissant dans la correction du *result term* ne sont pas celles de Wright ou CHAM. Ces deux formalismes estiment qu'une composition est possible si elle ne comporte pas de bloquages. Une évolution possible de la méthode de spécification d'architectures de connexion serait de rajouter un mécanisme permettant de se rapprocher de ce qui est fait dans ces deux langages. On peut aussi estimer qu'il existe peut-être des systèmes pour qui un bloquage ne se présente pas comme un problème et dans ce cas laisser cette vérification au développeur de la spécification.

Modularité

Les mécanismes de structuration des spécifications déjà présents dans la langage CASL sont encore utilisables dans une spécification architecturale. Une fois la structure modulaire (ou les informations de connexion) énoncée dans les déclarations d'unités, il est possible d'utiliser des constructions classiques pour développer les spécifications d'unités.

De par la motivation qui a engendré la création de constructions particulières (pouvoir modifier la structure de la spécification au cours du développement), on voit que ce formalisme met en avant un processus progressif, ce qui est important pour les systèmes de grande taille.

Raffinement

Le premier type de raffinement (local) est possible dans CASL, puisque c'est un des mécanismes principaux de construction des spécifications. Même dans une description architecturale, les spécifications des unités peuvent être raffinées, et donc redémarrer un processus qui peut aboutir à l'introduction d'une nouvelle spécification architecturale.

Le raffinement global, dans la mesure où il implique une modification de la structure, n'a pas de sens pour le moment. En effet, l'utilisation d'une spécification architecturale intervient dès que la structure du système est fixée, afin de modulariser la réalisation des spécifications. Le raffinement global implique une modification de cette structure, en ajoutant ou en découlant des unités. Cette forme de raffinement est possible dans CASL tant qu'on n'a pas fait intervenir de spécification architecturale (sinon, cela signifierait que l'on connaît la structure du système dès le départ). On ne peut donc pas parler de raffinement global pour des spécifications architecturales.

Dynamisme

La spécification d'architectures dynamiques n'est pas envisagée dans le langage CASL. Un problème important provient des spécifications génériques qui modélisent les modes de communication. Pour pouvoir modifier la configuration, il faudrait paramétrer ces spécifications par le nombre d'actions à synchroniser, ou le nombre d'automates. Il faudrait aussi que le *result term* prévoie ces modifications. Or pour qu'un *result term* soit correct, il faut vérifier des propriétés sémantiques qui ne sont pas décidables, ce qui empêche les configurations arbitraires.

Analyse

Dans une spécification architecturale en CASL, le *result term* représente le système entier : dans le cas d'une architecture de connexions, on construit l'automate composé à partir des spécifications des composants. Il est donc possible de vérifier des propriétés globales sur le système, en énonçant des axiomes sur la signature de tous les composants.

Par exemple, il est possible de vérifier qu'il n'existe pas de bloquage dans un système en vérifiant que pour chaque état non-final, il existe une transition partant de cet état :

$$\forall q \in Etat[C],\ \exists q' \in Etat[C],\ \exists a \in Action[C] \bullet q \neq final[C] \Rightarrow (is\_step[C](q, a, q')$$

On pourrait de même exprimer des propriétés concernant l'atteignabilité ou l'équité. Contrairement au techniques de Model-Checking, la vérification de ces propriétés ne serait pas limité par la finitude des états, ce qui permettrait de spécifier des systèmes plus complexes. La démonstration de ces propriétés demanderait, pour simplicité, de développer un système de preuves structurées comme dans [?], qui prendrait en compte les opérateurs spécifiques des spécifications architecturales.

3.4 Conclusion

On a montré dans ce chapitre qu'il était possible, dans un formalisme dédié à la spécification d'architectures d'implémentation, de décrire aussi des architectures connectives. Cela montre que la différence qui existe entre ces deux concepts se réduit à une différence syntaxique pour des exemples simples, excepté en ce qui concerne le mécanisme de composition (de Wright ou CHAM) qui comporte des informations spécifiques à l'architecture logicielle.

Le travail effectué dans CASL a été principalement de refléter des descriptions faites dans les ADL décrits dans le chapitre 2. Afin de mieux comprendre les mécanismes de connexion dans CASL, il serait intéressant de traiter un exemple de taille plus importante.

Conclusion et Perspectives

La taille des systèmes informatiques a engendré des problèmes spécifiques, qui nécessitent l'intervention de méthodes et de techniques particulières. L'architecture logicielle est le domaine du génie logiciel s'intéressant à ces problèmes. Dans le cadre de systèmes critiques, les méthodes formelles sont de plus en plus utilisées afin de prévenir d'éventuelles erreurs de conception. L'intégration, dans ces méthodes formelles, des méthodes et des techniques architecturales est donc une évolution naturelle.

Il existe deux moyens pour spécifier des architectures : soit on se place dans une optique de développement de la spécification jusqu'à un programme exécutable, soit on décide qu'un prototypage rapide et abstrait du système est plus efficace qu'une spécification complète. On parlera dans le premier cas d'architectures d'*implémentation*, dans le second d'architectures de *connexion*. Nous avons donné dans le chapitre 1 une intuition sur ce que sont ces deux types d'architectures.

Nous avons montré dans le chapitre 2 que les trois langages de description d'architectures de connexion étudiés (Wright, COMMUNITY, CHAM) se basent sur un principe commun pour écrire leurs spécifications. Ce principe, utilisé dans le domaine du Model-Checking, a servi à la construction d'architectures de connexion dans un langage de spécification dédié à la description d'architectures d'implémentation (CASL, chapitre 3). L'opposition qui se présente entre les deux types de description d'architectures se trouve ainsi presque réduite à une différence syntaxique.

Il serait intéressant de mieux comprendre ce qui sépare encore ces deux types de descriptions, en particulier comment on pourrait construire, dans CASL, un mécanisme de composition suivant le principe de «Composition correcte». Ce mécanisme permet aussi de distinguer les techniques de description d'architectures de connexion du Model-Checking.

Il serait aussi intéressant de définir un ADL comportant les avantages de chacun des langages étudiés dans ce mémoire. Il faudrait que ce langage permette la définition d'interfaces des composants séparément de leur implémentation, possède un mécanisme de composition proche de celui de Wright ou CHAM et sur lequel on puisse effectuer une analyse complexe. Ce dernier point pourrait se faire par des techniques de démonstration, en définissant au préalable un système de preuves structuré.