République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université Mentouri de Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

THÈSE

De
Doctorat en Sciences Informatique

Présentée Par

Chafia Bouanaka

Conception d'un ADL pour les Applications Distribuées et Mobiles, basé Logique de Réécriture

Soutenue le 20/10/2010, devant le Jury :

Président		
	M. Boufaïda	Professeur à l'Université Mentouri, Constantine
Directeurs de thèse		
	F. Belala	MC. à l'Université Mentouri, Constantine
	M. Bettaz	Professeur à l'Ecole Nationale Supérieure d'Informatique, Alger
Examinateurs		
	A. Mokhtari Aissani	Professeur à l'USTHB, Alger
	R. Boudour	MC. à l'Université Badji Mokhtar, Annaba
	N. Zeghib	MC. à l'Université Mentouri, Constantine
Invité		
	K. Barkaoui	Professeur au CNAM, Paris

Remerciements

Enfin, ce travail arrive à sa fin et il ne reste plus qu'à remercier toutes les personnes ayant contribué à sa réalisation.

Je voudrais tout d'abord remercier les membres du jury. Merci à Mme Aissani Mokhtari Aicha, professeur à l'USTHB, Mme Nadia Zeghib, maître de conférence à l'université Mentouri de Constantine et Mr Boudour Rachid, maître de conférence à l'université Badji mokhtar de Annaba, d'avoir accepté d'examiner mon travail. Enfin, je remercie Mr Boufaida Mahmoud, professeur à l'université Mentouri de Constantine de m'avoir accordé l'honneur d'être le président de mon jury.

Je tiens à exprimer ma profonde reconnaissance à Mme Belala faiza, maître de conférence à l'université Mentouri de constantine, qui a été beaucoup plus une collaboratrice que directrice de thèse. Que sa qualité humaine (sa modestie plus particulièrement) et professionnelle, sa disponibilité et ses conseils si précieux, puissent retrouver ici un geste de remerciement le plus sincère.

J'adresse mes plus sincères remerciements à Mr Bettaz Mohamed, professeur à l'École Nationale Supérieure d'Informatique, qui a bien voulu participer comme co-directeur de ma thèse et ce malgré ses lourdes préoccupations et responsabilités. Je le remercie pour ses orientations, ses conseils et de me faire part à chaque fois de ses travaux et résultats.

Que Mr Barkaoui Kamel, professeur des universités au conservatoire nationale des arts et métiers de Paris, trouve ici l'expression de ma gratitude pour les sacrifices qu'il a consentis à l'élaboration et au suivi de tous les accords-programme établis entre les équipes VESPA et GL/SD. Je le remercie plus particulièrement pour son perpétuel soutien, ses encouragements et pour le temps précieux qu'il m'a consacré lors de mes visites au sein de son équipe.

Il m'est difficile de ne pas remercier individuellement chaque membre de l'équipe GL/SD du laboratoire LIRE pour le cadre de travail amical et stimulant qu'ils m'ont offert.

Le présent travail est le reflet des encouragements, du soutien moral et de la confiance de plusieurs personnes, notamment mon père pour tous les sacrifices qu'il a consentis à mon égard et par la suite à mes enfants, ma mère, mon mari, mes frères et soeurs. Je tiens à les en remercier vivement et à les prier de trouver ici mes sincères reconnaissances et gratitude.

Résumé

Afin d'adopter une démarche de construction de systèmes à code mobile pilotée par l'architecture, nous définissons un cadre sémantique formel pour la specification de l'évolution bidimensionnelle, topologique et comportementale, de tels systèmes. Du fait que les éventuels changements sur la topologie de l'architecture logicielle peuvent avoir des effets de bord sur les calculs futurs, une structure et base sémantique communes sont utilisées pour spécifier la reconfiguration dynamique et le comportement de ce type de systèmes : Les interfaces constituent la structure commune à travers laquelle les effets de bord de chaque type de dynamique sur l'autre sont interceptés. Une approche à base de règles est adoptée comme base sémantique pour la description des deux types de dynamique. Différentes facettes (topologie, comportement et reconfiguration) d'un système à code mobile peuvent être alors facilement spécifiées dans un même cadre sémantique. Les interactions entre ces différentes vues sont assurées grâce à la structure commune définie sur les interfaces.
Par ailleurs, la validation des constructions syntaxiques ainsi définies par un modèle mathématique est une tâche nécessaire. En effet, le modèle sémantique associé à une architecture logicielle mobile est une double catégorie dont les objets de base sont les interfaces. La catégorie horizontale modélise les configurations possibles, ayant les composants et leurs liens comme morphisms. La catégorie verticale modélise les actions observables de l'architecture ainsi que les opérations de reconfiguration sur les interfaces des composants.

Table des Matières

Liste des tableaux

Table des figures

Introduction

La mobilité du code est l'aptitude à reconfigurer dynamiquement les liens entre les composants logiciels d'une application et leur localisation physique dans un réseau d'ordinateurs. Cette définition met en avant la nécessité de pouvoir détacher dynamiquement un élément logiciel de son contexte d'exécution actuel et de pouvoir le rattacher par la suite au nouveau contexte, c.à.d, que les frontières de l'unité de mobilité doivent être bien délimitées et que ses points d'accès soient bien définis. Ainsi, les applications à code mobile s'intéressent beaucoup plus à la vue externe de l'unité logicielle et ses interactions. Elle favorise la vue globale de la dynamique sociale au détriment de la vue individuelle d'un composant particulier en se basant sur l'émergence de la structure dès les premières phases du cycle de développement du logiciel. De ce fait, il est nécessaire de voir le système comme un assemblage d'entités logicielles en interaction ; chacune pouvant être elle-même un système. Une fois les entités identifiées, la manière de les assembler constitue l'architecture ou la structure du système. En conséquence, l'architecture du système permet d'identifier ses composants, particulièrement leurs frontières et fonctionnalités, et leur moyen d'interaction. Elle sert aussi à orchestrer les interactions entre ses composants afin de faire émerger le comportement global.

La discipline des architectures logicielle a atteint un niveau de maturité acceptable à travers la standardisation de ses concepts et notations de base et le développement de plusieurs méthodes de conception basées architecture. La contribution principale de

cette discipline par rapport au paradigme objet est qu'elle a justement réussi à dissocier les aspects liés à la coordination de ceux liés au calcul en mettant en avant le concept de connecteurs(Allen et al., 1997). Cependant, la vue architecturale des systèmes mobiles nécessite des mécanismes supplémentaires, pour prendre en charge la dimension espace permettant d'identifier la localisation actuelle des entités logicielles mobiles.

0.1 Contexte

Un travail ardu est actuellement en cours pour la formalisation de la dimension espace propre aux applications à code mobile et ce dès la phase de description de l'architecture logicielle. Deux grandes familles de modèles emergent actuellement dans le domaine de la description des architectures logicielles mobiles : les langages de description d'architectures(ADL) et les graphes.

- Les ADL : Dans leur version originale, la plupart des ADL n'ont pas été conçus pour supporter la mobilité des composants logiciels ; ce qui a amené leurs concepteurs à les enrichir par de nouveaux concepts afin de permettre la définition de la notion de localisation que ce soit explicitement ou implicitement.
  - Définition explicite : Les ADL conscients de la localisation enrichissent la structure du composant par un élément localisation. Une variable entière est ajoutée à la structure du composant logiciel dans Darwin (Magee et al., 1995) pour indiquer la machine effective hébergeant actuellement le composant. De façon similaire, un type abstrait de données est défini dans CommUnity (Lopes et al., 2006). Cette formalisation est dédiée à la mobilité d'entités physiques qui naviguent à travers un réseau de localisations. Le processus handoff des téléphones cellulaires spécifié dans (Oliveira et al, 2008) en utilisant CommUnity et l'application d'assistance routière (Bruni et al., 2008) sont des exemples significatifs

de cette catégorie d'applications. Néanmoins, la restriction de la mobilité des composants à une simple mise à jour de l'élément localisation n'est pas toujours la solution souhaitée.

  - Définition implicite : La plupart des applications s'intéressent à la mobilité logique de composants logiciels faisant abstraction de leur distribution effective qui est généralement transparente grâce à la couche intergiciel ; la mission principale de celle-ci étant justement de cacher la distribution des entités logicielles. De ce fait, une autre catégorie de formalismes propose une notion de localisation implicite représentée par des composites qui hébergent les composants mobiles.

- Les graphes : équipés d'outils de transformation, ils émergent actuellement comme une approche naturelle pour la spécification des architectures logicielles et leur reconfiguration dynamique. Trois sous-classes de l'approche basée graphe ont été identifiées dans (Bradley, 2004).
  - La première sous-classe opte pour une séparation claire entre la dynamique comportementale d'un composant et la dynamique de l'architecture, comme c'est le cas pour les travaux présentés dans (Allen et al., 1998 ; Milner, 2008). Il faut souligner ici la nécessité d'utiliser au moins deux formalismes pour spécifier la même architecture logicielle en l'absence de moyens pour spécifier les effets de bord de l'évolution du comportement sur l'évolution de la topologie et vice-versa.
  - Une deuxième sous-classe de modèles utilise les graphes pour décrire les deux types de dynamique. A titre d'exemple, le travail de (Baresi et al., 2002) considère les diagrammes de classes comme outils de description de l'architecture logicielle d'une application et la transformation de graphe comme moyen de spécification de la dynamique de la topologie ; instanciation de classes et destruction d'objets. Cette façon de faire génère des constructions additionnelles qui sont parfois assez complexes et non naturelles.

– La dernière sous-classe tente de greffer différents modèles comportements aux modèles de transformation de graphes telle que l'approche de résolution des contraintes utilisée dans (Hirch et al., 1999) ou d'enrichir les constructions des langages existants (Magee et al., 1998) pour supporter les concepts basés graphes. Néanmoins, cela est aussi effectué de manière non naturelle et compliquée.

0.2 Contribution

Dans un contexte de mobilité, deux types de configurations sont manipulées : la configuration structurelle qui sert à déterminer les composants et les liens actuellement présents dans cette application et la configuration comportementale qui modélise le comportement de l'application en décrivant les évolutions des liens entre composants. Les deux types de configuration peuvent évoluer dynamiquement avec d'éventuels effets de bord d'une sur l'autre. L'objectif de ce travail est de définir un cadre sémantique formel commun pour la spécification de l'évolution bidimensionnelle, structurelle et comportementale des systèmes à code mobile. Une approche basée sur la réécriture est adoptée pour la spécification non seulement du comportement mais aussi la structure dynamique de cette classe de systèmes.

Dans un travail préliminaire, nous adopterons l'approche ADL en enrichissant CBabel(Rademaker et al., 2004), un langage de description d'architecture basé sur la logique de réécriture (Meseguer, 1992), par des primitives de mobilité afin de voir l'aptitude de la logique de réécriture à supporter la dimension espace des applications à code mobile. La solution la plus évidente consistera à définir explicitement la notion de localisation par un attribut dans le composant mobile. Un ensemble de règles de réécriture seront par la suite définies afin de gérer la mobilité des composants à travers un univers de

localisations via des opérations de mise à jour de cet attribut.

Ce travail permettra de mettre en évidence l'impuissance de la logique de réécriture à définir des localisations implicites à travers l'hiérarchisation des composants logiciels. Cela est dû au fait que les théories de réécriture manipulent des termes algébriques plats ; nécessitant la construction de l'état global de manière statique. Ce qui constitue une contrainte pour les architectures logicielles mobiles dans lesquelles non seulement les valeurs de l'état évoluent mais aussi sa structure. Grâce à l'introduction des interactions dans le processus de réécriture en décorant les règles non seulement par les effets de bord de la réécriture mais aussi par les conditions contextuelles nécessaires à leur application, la logique des tuiles ; une extension de la logique de réécriture, permet une construction modulaire et réactive de systèmes concurrents. C'est justement cette vision réactive qui sera exploitée pour définir un nouveau langage dédié à la description des architectures logicielles dynamiques en général et mobiles en particulier. La logique des tuiles constituera le cadre sémantique sous-jacent au langage proposé (MoSAL, pour Mobile Software Architectures Language) pour deux motivations principales :

- L'aptitude de la logique des tuiles à implémenter la caractéristique principale des architectures logicielles ; à savoir la séparation des préoccupations entre les dimensions calcul et coordination dans un système distribué.
- La sémantique des interfaces de composants logiciels peut être bien définie.

Les différents composants de l'architecture logicielle seront définis grâce à la généralisation de la notion d'interfaces ; servant initialement à guider la réécriture, à la spécification des composants et leurs interactions. Ces interfaces constituent les objets de base à partir desquels seront construites les deux types de configurations, structurelle et comportementale.

Au lieu de manipuler des termes algébriques sur les interfaces qui sont très dépendants de

la syntaxe pour définir la configuration structurelle exprimant la topologie, la distribution et la connectivité dans une architecture logicielle, nous utiliserons les hypergraphes. Étant donné que les interfaces des composants sont les objets de base subissant les transformations causées par le calcul ou la redistribution spatiale des composants mobiles, elles seront considérées comme noeuds du graphe. Les composants et leurs attachements par contre correspondront aux hyperarcs du graphe et exprimeront la distribution des interfaces. Les reconfigurations dynamiques de l'architecture logicielle seront alors définies par des opérations de transformation de graphes. Quand aux fonctionnalités observables associées aux différentes interfaces et fournies par les composants de l'architecture, elles seront spécifiées en attachant à chaque interface l'action correspondante. Chaque action sera définie par une règle de réécriture exprimant une évolution possible de cette interface.

Du fait que les éventuels changements sur la configuration structurelle peuvent avoir des effets de bord sur les calculs futurs et vice-versa, une structure et une base sémantique communes seront utilisée pour spécifier la reconfiguration dynamique et le comportement des architectures logicielles mobiles. Les interfaces constitueront la structure commune à travers laquelle les effets de bord de chacun des deux types de dynamique sont interceptés et une approche unifiée à base de règles sera adoptée pour décrire les deux types de dynamique.

La validation des constructions syntaxiques par un modèle mathématique est une tâche nécessaire. Ainsi, une grande partie de notre contribution sera consacrée à la construction d'un modèle mathématique pour les architectures logicielles mobiles décrites dans le langage proposé. En effet, le modèle sémantique associée à une architecture logicielle mobile est une double catégorie dont les objets de base sont les interfaces. La catégorie horizontale modélise les configurations possibles avec les composants et

leurs liens comme morphismes. La catégorie verticale modélise les actions observables de l'architecture ainsi que les opérations de reconfiguration sur les interfaces des composants. Les termes de preuves générés par les règles de déduction de la logique des tuiles définissent les scénarios de reconfiguration et les calculs valides dans l'architecture logicielle considérée.

0.3 Structure de la thèse

Cette thèse comporte six chapitres. Les éléments de base nécessaires à la prise en charge de la mobilité du code ainsi que la motivation d'une approche autour de l'architecture logicielle seront présentés dans le chapitre 1. Les concepts de base des architectures logicielles ainsi que deux modèles de leur description, les langages de description d'architecture et les approches à base de graphes, seront étudiés dans le chapitre 2. Une étude approfondie sur la description des architectures logicielles mobiles sera réalisée dans le chapitre 3. Une étude comparative entre les approches les modèles de description d'architectures logicielles mobiles sera dégagée dans la suite du chapitre, tout en justifiant judicieusement les critères considérés. La présentation du cadre sémantique formel adopté sera faite dans le chapitre 4. Le travail préliminaire d'enrichissement du langage CBabel par des primitives de mobilité physique fera l'objet du chapitre 5. Nous partirons du constat de l'incapacité de la logique de réécriture à supporter des structures hiérarchiques pour suggérer un nouveau modèle de description d'architectures logicielles mobiles, via le langage MoSAL qui tire sa sémantique de systèmes de tuiles particuliers. Une étude de cas de taille réaliste avec ses aspects d'implementation sera utilisée pour illustrer les concepts et les apports du langage MoSAL. Les éventuels travaux futurs et perspectifs de ce travail seront présentés à la fin de ce manuscrit.

Chapitre 1

Généralités sur la mobilité du code

1.1 Introduction

Les réseaux à grande échelle ont envahi tous les domaines de la vie et deviennent de plus en plus indispensables. Cela est dû à la dérision du matériel d'une part et à l'augmentation considérable de la performance des infrastructures de communication d'autre part. L'augmentation de la taille et de la performance des réseaux n'est pas seulement la cause mais aussi l'effet d'un changement fondamental qui est la vulgarisation et l'ubiquité des réseaux d'ordinateurs. Ainsi, une panoplie d'applications et de systèmes distribués sont accessibles au large public (commerce électronique, messagerie électronique, achat en ligne, etc.). Cependant, cette évolution n'est pas sans obstacles (Picco, 1998) :

1. Problème de passage à l'échelle : la plupart des résultats, qui sont significatifs pour les réseaux à petite échelle, ne sont plus applicables dans des réseaux larges tel qu'Internet.

2. Reconfiguration dynamique des réseaux actuels : avec l'avènement de la technologie de connexion sans fils, les nœuds d'un réseau quelconque peuvent se déplacer, se connecter et se déconnecter de manière had-hoc. La topologie du réseau n'est plus

alors définie de manière statique. Et de ce fait, certaines évidences sur les systèmes distribués classiques doivent être revues et réadaptées à ces réseaux dynamiques.

3. Problème de spécialisation et de filtrage des services : Le large éventail de services offerts ainsi que la variété des catégories d'utilisateurs du réseau Internet nécessitent un haut degré de spécialisation des services destinés à l'utilisateur final.

Plusieurs tentatives ont été menées pour résoudre ces problèmes à différents niveaux d'abstraction (Applicatif, technologique et méthodologique). La plupart des solutions adoptées tentent d'adapter des modèles et technologies existants aux nouveaux besoins ; adaptation de l'architecture client-serveur à titre d'exemple. Néanmoins, elles ne fournissent pas le degré de flexibilité, de spécialisation et de reconfiguration attendu. Une nouvelle approche, appelée mobilité du code, est alors proposée. Elle consiste à doter le système logiciel du potentiel de déplacer un code ou un fragment de code entre deux terminaux physiques.

1.2 Motivations des applications à code mobile

Le concept de code mobile n'est pas tout à fait nouveau, il existait déjà dans les systèmes d'exploitation distribués sous le nom de migration de code. Il a été utilisé principalement dans :

- les premiers utilitaires de soumission de travaux batch à distance (Boggs, 1973) et des contrôleurs Postscript d'imprimantes (ADOBE, 1985).
- la migration de processus actifs y compris leur état d'exécution et le code associé ; introduite dans les systèmes d'exploitation distribués afin de répartir la charge entre les nœuds d'un réseau de communication, présente une approche plus structurée de mobilité de code. Cette migration est à la charge du système d'exploitation

qui est le seul à décider du moment et de l'endroit vers lequel le processus sera redirigé. Elle est donc totalement transparente au programmeur qui n'a ni le contrôle ni la visibilité d'une telle opération.

– la migration d'objets offre une granularité plus fine des entités mobiles par rapport à la migration de processus entiers. A titre d'exemple, le système d'exploitation Emerald (Jul, 1988) autorise la migration de différents types d'objets allant d'une donnée simple et atomique à des objets complexes. Contrairement au système COOL dans lequel la migration d'objets est totalement transparente, Emerald permet au programmeur de demander explicitement la migration d'un objet vers un nœud particulier ou de déterminer la localisation d'un objet.

La migration de processus ou d'objets des systèmes d'exploitation s'attaque à des problèmes de mobilité de code et d'état d'exécution à travers les hôtes de petits systèmes distribués fortement couplés pour des fins de répartition de la charge entre les nœuds du réseau de communication sous-jacent. Ce mécanisme ne peut être directement appliqué aux réseaux à grande échelle où la mobilité du code est exploitée pour des fins d'adaptation et de spécialisation de services. Et donc ce n'est plus du ressort du système d'exploitation d'initier l'opération de mobilité mais plutôt l'application elle même. Ce qui nécessite la violation du principe de transparence à la programmation par la définition de primitives de gestion de la mobilité du code au niveau langage de programmation. Et par conséquent, la mise en œuvre d'applications distribuées sur des réseaux à grande échelle nécessite une remise en cause des concepts de programmation existants.

– Au niveau système d'exploitation : elle correspond à la migration d'un processus actif de la machine sur laquelle il est en cours d'exécution vers une autre machine. Les mécanismes de migration assurent au processus de restituer son état d'exécution dans l'environnement distant et ce en établissant des liens entre le

processus et son environnement d'exécution (les descripteurs de fichiers ouverts et les variables d'environnement). La migration des processus permet de réguler ou repartir la charge entre les nœuds du réseau de manière transparente à l'utilisateur et au programmeur aussi. Elle ne peut être mise en œuvre que si le code et l'état du processus sont déplacés ensemble à travers les hôtes d'un petit système distribué fortement couplé, généralement homogène et fiable, avec une large bande passante et une faible latence.

- Au niveau langage de programmation : la notion de migration de processus a été étendue en introduisant une couche supplémentaire, appelée système à code mobile offrant différentes formes de mobilité et ce pour les motivations suivantes (Romain et al., 2000) :
  - La mobilité du code est exploitée à une large échelle : les systèmes à code mobile apparaissent dans des réseaux à grande échelle constitués d'hôtes hétérogènes ; caractérisés par une bande passante variable fluctuant entre des connexions sans fil à faible débit jusqu'à des liens de fibre optique très rapides, gérés par différentes autorités présentant un niveau de fiabilité variable.
  - La programmation est consciente de la localisation (Location aware) : la localisation est une abstraction ayant une forte répercussion sur la conception ainsi que l'implémentation des applications distribuées. Les systèmes à code mobile ne s'intéressent pas à la publication de la localisation des composants de l'application, mais cette information peut influer sur les actions futures que l'application peut entreprendre.
  - La mobilité est sous le contrôle du programmeur : un ensemble de mécanismes et d'abstractions sont mis à la disposition du programmeur lui permettant d'injecter ou retirer des fragments de code ou des composants entiers dans des

nœuds distants. Le rôle de l'environnement d'exécution sous-jacent est maintenant restreint à fournir les fonctionnalités de base pour contrôler le transfert de données/code en assurant la sécurité de se transfert. Il n'a plus aucun contrôle sur la politique de migration du code.

– La mobilité n'est plus un mécanisme de répartition de la charge : alors que la migration de processus et d'objets vise à subvenir aux besoins de répartition de la charge et d'optimisation des performances, les systèmes à code mobile traitent de la personnalisation des services, l'extension dynamique des fonctionnalités de l'application, l'autonomie, la tolérance aux fautes, etc.

1.3 Approches, méthodes et technologies pour la mobilité du code

Les systèmes à code mobile proposent une nouvelle approche pour les applications distribuées. Ils s'intéressent à la spécification des déplacements d'éléments logiciels à travers un réseau de localisations tout en assurant un changement d'environnement d'exécution de ces éléments sans altération ni changement de leur structure interne ni leur comportement. Contrairement aux systèmes concurrents qui se concentrent sur la vue de l'intérieur vers l'extérieur (zoom out) des composants, les systèmes à code mobile mettent l'accent sur la vue de l'extérieur vers l'intérieur (zoom in) du composant. Cela est dû au fait que la mobilité favorise la vue globale de la dynamique sociale au détriment de la vue individuelle d'un élément donné. La dynamique de la topologie des réseaux de communication actuels ainsi que les problèmes de spécialisation et de filtrage des services offerts à l'utilisateur final nécessitent des mécanismes de gestion de la mobilité du code à un niveau applicatif et non plus au niveau du système d'exploitation. Plusieurs systèmes

à code mobile ont été proposés (Danianou, 2001 ; Mascolo et al., 1999 ; Jul et al, 1988 ; Fong et al. 1998) ; parfois de manière chaotique. Cela a généré une certaine confusion autour des concepts, abstractions, terminologie et sémantique de la mobilité du code. Par conséquent, les méthodologies, applications et technologies à code mobile ont besoin d'être revues.

1.3.1 Niveau méthodologique

D'après Carzaniga et al (Carzaniza, 1997), la mobilité du code est l'aptitude à reconfigurer dynamiquement les liens entre les composants logiciels d'une application et leurs localisations physiques dans un réseau d'ordinateurs. Un système à code mobile est alors vu comme un ensemble de composants qui se déplacent à travers un espace de localisations. Le composant peut être soit un fragment de code capable de parcourir un espace d'adressage d'un réseau de communication ou bien une entité physique qui se déplace dans un espace géographique. La définition précédente dégage les concepts clé devant être pris en charge par une approche de conception de systèmes à code mobile. Cette approche doit inclure les trois concepts de base suivants :

- L'unité de mobilité : représente le plus petit composant autorisé à se déplacer. Cette caractéristique permet d'identifier les entités du modèle qui sont capables de se déplacer.
- La mise en évidence de la coordination : d'un point de vue conceptuel et même pratique, il est nécessaire de découpler les fonctionnalités des composants individuels de la manière dont ils interagissent entre eux, afin de limiter leurs dépendances de leur environnement et donc faciliter leur mouvement. Particulièrement, en sachant qu'un composant mobile ne pourra pas prédire au préalable ses interactions encore moins leur lieu d'apparition. Ainsi, l'approche de conception doit favoriser la

séparation entre les connexions (liens) des composants logiciels et leurs propriétés fonctionnelles, c.à.d., séparer la dimension coordination de la dimension fonctionnelle. De ce fait, la spécification de la coordination se concentrera beaucoup plus sur les mécanismes nécessaires pour découvrir le monde extérieur, échanger des informations et synchroniser les actions à entreprendre. Ceux sont les raisons pour lesquelles, les mécanismes d'interaction entre composants deviennent des critères d'évaluation de systèmes à code mobile.

- La notion de localisation (location) : dans une optique de génie logiciel, le calcul mobile est l'étude de systèmes dans lesquels des unités de calcul peuvent changer de localisation dynamiquement. L'univers ou l'espace de localisations peut être de deux types : physique ou logique.
  - La mobilité physique implique le mouvement d'éléments physiques dans un espace géographique, généralement identifié par une adresse IP, des coordonnées GPS, etc. Ce type de localisation est mieux adapté pour les applications à dispositifs mobiles telles que le système GSM.
  - La mobilité logique concerne des unités mobiles (code et états) qui migrent d'un hôte à un autre. Typiquement, les hôtes sont stationnaires et donc l'espace de mobilité reflète la structure du réseau sous-jacent. Dans ce type de mobilité, tout l'interet est porté sur l'évolution de la structure (architecture) de l'application en termes de migration d'entités mobiles, plûtot que sur le déploiement physique de celles-ci. Ainsi, la localisation de l'entité mobile est identifiée par le composite auquel elle appartient actuellement.

L'unité de mobilité, l'espace de mobilité et la gestion du changement de contexte causé par la mobilité, sont des concepts pertinents dans la définition de modèles de spécification de systèmes à code mobile. En effet, le choix de l'unité de mobilité est capital pour tout modèle puisqu'il influe sur le choix des deux autres aspects. Alors que la définition

des localisations montre la perception du modèle de l'espace de mobilité, la gestion des changements de contexte reflète la perception du composant de son environnement à travers des mécanismes de coordination lui permettant de se rattacher au système destination.

1.3.2 Niveau conception

Une fois les concepts de base nécessaires aux systèmes à code mobile définis, leur mise en œuvre nécessite de nouvelles approches car les méthodes de conception traditionnelle ne sont plus suffisantes pour l'exploitation de la mobilité et la reconfiguration dynamique des composants logiciels. Parmi les approches proposées, nous pouvons citer le modèle Client/Serveur, le code à la demande, l'évaluation à distance et les agents mobiles (Picco, 1998 ; Fuggetta et al, 1998).

Le modèle Client/Serveur : Dans ce paradigme, un composant B (le Serveur), offrant un ensemble de services, est placé dans un site S_B. Les ressources et le code nécessaires pour chaque service résident aussi dans le site S_B. Le composant Client A, résidant dans le site S_A, demande l'exécution d'un service à l'aide d'une interaction (message) avec le composant B. Comme réponse, B réalise le service demandé, en exécutant le code correspondant en accédant aux ressources (résidants dans S_B aussi). En général, le service produit un résultat qui sera fourni au client à l'aide d'une autre interaction.

L'évaluation à distance : Dans ce cas, un composant A possède le code correspondant à la réalisation du service mais ne possède pas les ressources nécessaires, qui résident dans un site distant S_B. En conséquence, A envoie le code à un composant B résidant dans le site distant. B exécute le code en utilisant les ressources disponibles dans S_B et renvoie les résultats à A.

Le code à la demande : Concernant le code à la demande, le composant A est capable d'accéder aux ressources nécessaires ; le composant A et ses ressources résident dans le même site S_A. Cependant, A n'a pas le code nécessaire pour la réalisation du service demandé. A demande alors le code au composant B, se trouvant dans le site S_B. Cette fois-ci, c'est le composant B qui migre vers le site S_A en emportant avec lui le code et l'état de l'exécution.

Les agents mobiles : un agent constitué d'un code et d'un état d'exécution est muni de la capacité de se déplacer d'une machine hôte vers une autre pour collecter les données nécessaires à son exécution.

Le modèle client/serveur ne présente pas une mobilité de code effective car seules des données sont échangées entre le fournisseur du service et son demandeur. Dans l'évaluation à distance et le code à la demande, c'est le code qui migre d'une machine à une autre. Elles diffèrent dans le fait que c'est le demandeur du service qui détient le code dans la première approche, alors que dans la deuxième c'est le fournisseur du service qui le détient et c'est lui qui migre emportant avec lui le code demandé. Afin d'épargner le programmeur des détails techniques de ce transfert de code entre nœuds du réseau de communication, le niveau technologique aussi a besoin d'être revu.

1.3.3 Niveau technologique

Afin d'assurer la prise en charge de la mobilité du code, les technologies ont introduit une nouvelle couche au dessus de la couche système d'exploitation et réseau contre l'élimination de la couche vraie système d'exploitation distribué. Cette couche sert de conteneur des composants d'une application distribuée et permet de gérer la mobilité du code à un niveau supérieur à celui du niveau système d'exploitation. Elles ont aussi introduit de nouveaux concepts (Cugola et al., 1996 ; Picco, 1998) :

1. L'environnement de calcul (Localisation) :

   Chaque environnement de calcul retient l'identité de sa machine hôte afin de fournir aux applications la capacité de reloger dynamiquement leurs composants dans d'autres hôtes.

2. L'unité d'exécution (Exécution Unit) :

   Une unité d'exécution est composée d'un segment de code et de l'état d'exécution. Le segment de code fournit une description statique du calcul. L'état est lui aussi composé d'un espace de données et d'un état d'exécution :

   – L'espace de données représente toutes les ressources accessibles par les routines actives de l'unité d'exécution (ces ressources peuvent exister dans d'autres environnements de calcul).
   – L'état d'exécution contient les données privées de l'unité d'exécution ainsi que les informations de contrôle de l'exécution (la pile d'exécution, le compteur ordinal, etc.).

   Pour contrôler l'accès aux différentes ressources (données et code), les systèmes à code mobile définissent deux autres notions :

3. La clôture sur les données :

   représente un sous-ensemble de l'espace de données représentant toutes les ressources locales ou distantes accessibles par la routine en cours d'exécution ainsi que les routines qu'elle peut appeler.

4. La clôture sur le code :

   représente toutes les routines directement ou indirectement visibles à partir de la routine en cours d'exécution.

La séparation claire entre l'unité d'exécution d'une entité mobile et son environnement de calcul ou sa localisation(code et état d'exécution) constitue la caractéristique principale

des applications à code mobile sur laquelle se basera la définition des mécanismes de mouvement et de redéploiement de cette entité.

1.4 Mécanismes nécessaires à la mobilité du code

Selon le mode d'invocation du code mobile adopté (client/serveur, évaluation à distance ou le code à la demande), le segment de code, l'état d'exécution et l'espace de données d'une unité d'exécution peuvent résider dans différents environnements de calcul. Cela nécessite la détermination des éléments impliqués dans le mouvement de l'entité mobile ainsi que le mécanisme de rétablissement des liens entre celle-ci et son espace de données.

1.4.1 Types de mobilité

Le type de mobilité sert à déterminer les éléments de l'entité mobile qui sont concernés par le mouvement. Deux formes de mobilité existent :

La mobilité forte : dans ce cas de figure, le système à code mobile permet le mouvement du code et de l'état d'exécution vers un autre environnement de calcul. Ce type de mobilité est généralement utilisé dans les agents mobiles.

La mobilité faible : utilisée dans le code à la demande ou bien l'évaluation à distance. Dans ce type de mobilité, seul le code est autorisé à se déplacer d'un environnement de calcul vers un autre. A la réception de ce code, l'environnement de calcul a deux options :

- Soit créer une nouvelle unité d'exécution pour l'exécuter.
- Ou bien le rattacher à une unité d'exécution existante.

Dans le cas d'un code à la demande, le code est rattaché à une unité d'exécution par un lien dynamique.

1.4.2 Liens avec l'espace de données (ressources)

Une fois l'entité mobile reçue par son nouvel environnement de calcul, l'ensemble de ses liens avec les ressources accessibles doit être réarrangé. Cela peut provoquer la destruction de liens avec certaines ressources, l'établissement de nouveaux liens ou bien la migration de certaines ressources à l'environnement de calcul distant. Le choix dépend de la nature de la ressource impliquée, du type de liens à cette ressource et des contraintes imposées par l'application. De ce fait, deux approches existent, par réplication et par partage :

Stratégies de Réplication : elle aussi propose des réplications statique et dynamique.

La tratégie de réplication statique (Re-Binding) : utilisée pour les ressources banalisées (les variables du système et les bibliothèques). Les liens avec les ressources de l'unité d'exécution d'origine sont détruits et de nouveaux liens sont crées avec les ressources de l'environnement de calcul destinataire.

La stratégie de réplication dynamique : deux techniques sont possibles :

- Par mouvement : la ressource est transférée avec l'unité d'exécution à l'environnement de calcul destinataire. Cette technique est utilisée pour les ressources non partageables.
- Par clonage : une copie de la ressource est envoyée avec l'unité d'exécution

Stratégie de partage (référence réseau) : dans ce cas de figure, la ressource n'est pas transférée. Néanmoins, toute tentative d'accès à la ressource par l'unité d'exécution, dans son environnement de calcul destinataire, provoque une communication avec l'environnement de calcul source. La création de liens inter-environnements de

calcul n'est pas très désirable car elle est soumise aux problèmes liés au réseau de communication.

1.5 Conclusion

Les systèmes à code mobile proposent une nouvelle approche de conception d'applications distribuées. Ils s'intéressent à la spécification des mouvements d'éléments logiciels à travers un réseau de localisations tout en assurant un changement d'environnement d'exécution de ces composants sans altération ni changement de leur structure interne ni leur comportement. Cela signifie que les points d'interaction de l'entité avec son environnement sont bien définis. Ces points d'interaction représentent justement le moyen pour détacher/rattacher l'entité à son environnement. L'apport principal des modèles à base de composants et plus particulièrement les architectures logicielles est cette séparation entre le calcul et les interactions. Ces modèles ont les caractéristiques requises pour la prise en charge de la mobilité dès les premières phases du cycle de développement des systèmes logiciels. Avant d'étudier de plus prêt l'apptitude des architectures logicielles à prendre en charge la mobilité du code, nous présenterons dans le chapitre suivant leurs concepts de base et leurs modèles de description.

Chapitre 2

Architectures logicielles et leur description

2.1 Introduction

Le génie logiciel vise à définir des méthodes et proposer des outils de production et de maintenance du logiciel. Il est défini comme l'ensemble des activités de conception et de mise en œuvre de procédures tendant à rationaliser le développement du logiciel et son suivi. Cette discipline est née suite à la crise qu'a connue le logiciel dans les années 70(Djikstra, 72). Son objectif principal est de définir des approches de conception et des paradigmes de programmation permettant la maîtrise de la complexité croissante des systèmes logiciels et la proposition de métriques d'estimation de l'effort requis pour aboutir à un logiciel fiable. La décomposition ; à travers la définition de la structure des systèmes logiciels, et la réutilisation ; à travers la définition de l'unité de réutilisation, sont au centre des concepts clé proposés par le génie logiciel pour justement maîtriser la complexité et réduire l'effort et le coût de conception d'un logiciel.

Les concepts et paradigmes clé ayant contribué à l'évolution de la discipline du génie logicielle et la naissance de la branche architectures logicielles sont les suivants :

– Le concept de structuration : Djikstra (Djikstra, 1968) a été un des pionniers à

proposer une structuration impérative du système à concevoir avant de se lancer dans l'écriture de son code. Il a mis en évidence le besoin de la notion d'abstraction dans la conception de systèmes de taille réelle.

– La modularité : introduite par Parnas (Parnas, 72) au début des années 70, elle a constitué un concept clé pour la structuration d'une application puisqu'une importance majeure est consacrée aux décisions de conception ; qui doivent se faire avant d'initier les travaux de mise en œuvre. La modularité a apporté une nette amélioration quand à la souplesse et le contrôle conceptuel du logiciel tout en réduisant le temps de développement des systèmes.
– Le paradigme objet : le concept d'encapsulation en particulier a constitué une contribution considérable dans la discipline du génie logiciel en améliorant l'analyse, la conception et le développement des systèmes logiciels. Car une application n'est plus maintenant vue comme une suite d'instructions mais plutôt comme une collection d'objets encapsulant données et comportement et coopérant afin de réaliser l'objectif global de l'application. Néanmoins, la structure de l'application reste toujours peu visible du fait que les couplages entre objets ne sont pas explicites, mais plutôt enfouis dans l'implémentation de ceux-ci sous forme d'instructions d'invocation des méthodes des objets. Ce qui constitue un handicap pour la réutilisation de l'objet dans d'autres contextes, d'une part. Puisque cela nécessite une remise en cause de l'implémentation pour corriger ses liens avec l'environnement. D'autre part, cette façon de faire représente une contrainte pour la conception des applications actuelles qui sont de plus en plus concurrentes, distribuées, et pouvant être exploitables sur des plateformes hétérogènes grâce à l'ubiquité numérique(les réseaux domotiques, réseaux ad-hoc, réseaux de capteurs, etc.)

A la mise en évidence des lacunes du paradigme objet dans le domaine de la réutilisation vers la fin des années 90 (voir figure 2.1), la notion de composant logiciel a été proposée comme unité alternative de réutilisation. Le découplage entre les dimensions calcul (implémentation) et interaction (coordination) constitue l'apport principal du paradigme composant par rapport au paradigme objet. Il est mis en œuvre par la spécification, dans l'interface du composant, non seulement des services fournis mais aussi des services requis de l'environnement. La description de ces derniers dans l'interface met en évidence une description claire des contraintes de réutilisation d'un composant dans un contexte quelconque. Alors que le découplage entre l'implémentation et l'interface met en avant la description de l'assemblage (interactions) des composants. Les concepts d'interface de composant et d'assemblage constituent les éléments de base de définition de l'architecture logicielle d'une application. Le principe de base de l'architecture logicielle est la séparation entre l'aspect fonctionnel des composants et les interactions qu'ils entreprennent avec les autres composants et ce pour une meilleure maîtrise et une bonne gestion de la complexité des systèmes logiciels. Malgré que cette définition mette à l'écart d'autres aspects structurels et conceptuels des systèmes, elle met à pied d'égalité le calcul et l'interaction.

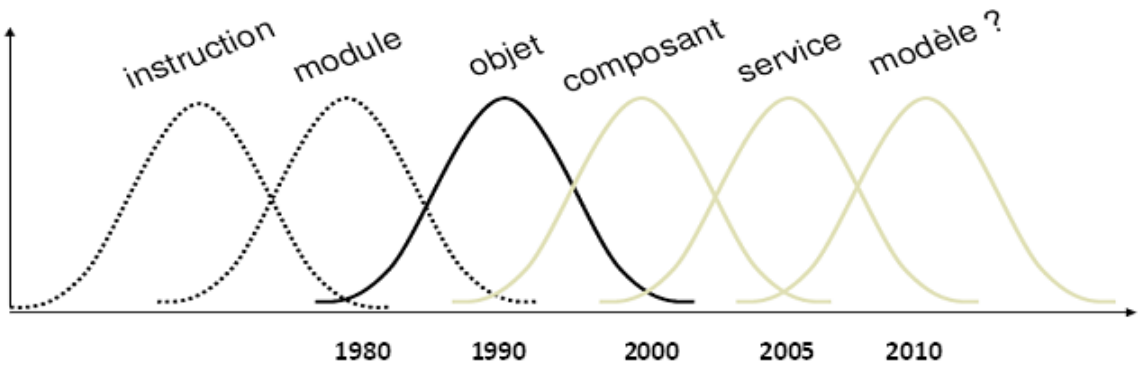


FIGURE 2.1 – Historique de la granularité des applications.

2.2 Concepts de base d'une architecture logicielle

L'architecture logicielle permet d'identifier des entités de connexion appelées connecteurs et de décrire de manière précise les protocoles de communication entre composants (Allen, 1996 ; Allen, 1997a) en plus des entités de structuration, les composants. Les idées de base telles que donner de l'importance aux décisions prises dans les premières phases de développement du système ainsi que de l'importance d'une définition correcte de la structure du système ont contribué à faire évoluer la discipline des architectures logicielles et l'introduire (Garlan, 1994) comme une activité de conception explicite dans le processus de développement du logiciel.

2.2.1 Définitions

De nombreuses définitions existent dans la littérature, chacune d'elles met en valeur certains aspects de l'architecture logicielle.

- Shaw et al. (Shaw, 1995) mettent l'accent sur les composants et leurs interactions : *L'architecture d'un système logiciel définit ce système en termes de composants et d'interactions à travers ces composants. En plus de la spécification de la structure et la topologie du système, l'architecture montre la correspondance entre les besoins du système et les éléments du système à construire. Elle peut aussi s'attaquer au niveau propriétés de ce système telles que la capacité, la consistance, la compatibilité des composants, etc.*
- Bass et Kazman (Bass et al 1997) favorisent la vision boite noire des composants d'une architecture logicielle en introduisant le concept de propriétés des composants. Ils définissent l'architecture logicielle d'un programme ou un système logiciel par la ou les structures du système, comprenant les composants logiciels, les

propriétés visibles de l'extérieur de ces composants et les relations entre ces composants. Ces propriétés expriment les principales caractéristiques d'un composant lui permettant d'offrir une vue assez complète aux autres composants (l'environnement) sans recourir à la description de la structure interne et l'implémentation de ce composant. Ces propriétés englobent les caractéristiques de performance, la gestion des fautes, l'utilisation des ressources partagées, etc. Cette définition vise à préciser que la description de l'architecture logicielle doit faire abstraction de certains détails du système et ne fournir que les informations nécessaires et suffisantes constituant un point de départ pour l'analyse, la prise de décisions et par conséquent la réduction des risques.

– Une autre définition (Dewayne et al 1992), aussi acceptée que celle de Bass et al., insiste sur les choix de conception pouvant être faits à ce niveau d'abstraction et influant sur la structure globale du système. Ces choix englobent l'organisation grossière, la structure globale de contrôle, les protocoles de communication et de synchronisation, l'attribution des fonctionnalités aux éléments de conception, la distribution physique des éléments de conception et la composition des éléments de conception.
– Kruchtenel et al. (Kruchtenet al., 2006) dans une définition assez récente précisent que *l'architecture logicielle implique la structure et l'organisation par lesquelles des composants interagissent pour créer de nouvelles applications, possédant la propriété de pouvoir être les mieux conçues et analysées au niveau système.*
– Des efforts de standardisation, de la définition d'une architecture logicielle, ont été notamment menés par l'IEEE sous la norme IEEE 1471-2000, IEEE(2000) qui propose la définition suivante : *une architecture définit la structure des composants d'un programme ou d'un système, leur interdépendances, et les principes et directives régissant leur conception et leur évolution.*

Toutefois, toutes ces définitions approuvent le fait qu'une architecture est considérée comme l'organisation nécessaire d'un système caractérisé par ses composants, leurs relations avec l'environnement, et les principes qui guident leur conception et évolution.

2.2.2 Rôles d'une architecture logicielle

La description de l'architecture logicielle d'un système possède quatre rôles principaux dans un projet informatique (Garlan et Shaw, 93).

1. Elle facilite la compréhension de la structure d'un système. Car, le principe d'abstraction des architectures logicielles préconise la mise à l'écart des détails inutiles. Ce qui procure à la description architecturale une meilleure lisibilité.
2. Elle permet l'analyse et la prévention des qualités architecturales avant l'implémentation du système.
3. Elle sert de base pour la génération de code et l'identification des possibilités de réutilisation.
4. Elle définit les invariants du système et sert de pivot pour son évolution.

2.2.3 Éléments d'une architecture logicielle

Le composant

Un composant est une unité de calcul ou de stockage à laquelle est associée une unité d'implémentation. La granularité du composant varie de la définition d'une simple procédure à la définition d'une application complète.

Le composant peut être simple ou composé et est constitué deux parties : interface et implémentation. L'interface comprend la description des services fournis et requis par le composant et définit les interactions du composant avec son environnement. Ces

interactions sont décrites en termes des règles et contraintes d'exploitation des services du composant. La seconde partie correspondant à la mise en oeuvre des services fournis par le composant et permet la description du fonctionnement interne de celui-ci.

Le connecteur

Un connecteur correspond à un élément architectural qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui régissent ces interactions. Il est considéré comme un médiateur dans la communication et la coordination entre composants. Un connecteur comprend également deux parties. La première correspond à son interface. Elle permet de décrire le rôle de chacun des participants à une interaction. La seconde partie correspond à la description de son implémentation. Il s'agit là de la définition du protocole associé à l'interaction.

Le connecteur n'est pas nécessairement considéré comme une entité de première classe dans tous les modèles où il peut être un simple composant particulier dédié à l'implémen-tation d'un protocole de communication.

La configuration

Représente un graphe d'instances de composants et de connecteurs servant à décrire un arrangement particulier de ceux-ci afin de définir un système donné. Une configuration définit la structure et le comportement d'une application formée d'un ensemble d'instances de composants et de connecteurs.

Style architectural

Un style architectural (Abowd et al., 1993 ; Abowd et al., 1995) caractérise une famille de systèmes qui partagent des propriétés structurelles et des éléments de sémantique.

Plus précisément, un style architectural permet de décrire un vocabulaire commun en définissant un ensemble de types de composants et de connecteurs, des contraintes de configuration déterminant les compositions autorisées (propriétés et contraintes partagées par toutes les configurations appartenant à ce style) d'éléments architecturaux qui sont des instances des types préalablement définis et un ensemble de propriétés fournissant des informations de sémantique et de comportement.

Un style architectural peut être défini comme un type que l'architecture doit avoir pendant l'exécution (Le Métayer, 1998). Les éléments du système ne peuvent interagir qu'à travers les liens spécifiés par le style. Les styles architecturaux permettent donc de décrire des patrons d'architecture pour la structuration des systèmes par analogie avec les patrons de conception (Gomma et al 1994) pour la programmation orientée objet.

L'utilisation des styles architecturaux a les avantages suivants :

- Au niveau modélisation, l'utilisation des styles architecturaux favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être encore appliquées à de nouvelles applications.
- Au niveau conception, elle facilite la compréhension de l'organisation de l'architecture de l'application. Par exemple, concevoir une application selon le style Client/Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants.
- Finalement, l'utilisation des styles architecturaux peut contribuer considérablement dans la réutilisation significative du code dans la phase implémentation.

Par ailleurs, Les systèmes construits selon un même style architectural sont plus compatibles que ceux qui mélangent plusieurs styles. Car cela facilite considérablement l'interopérabilité et la réutilisation de parties de la même famille de systèmes.

2.3 Description des architectures logicielles

Deux grandes classes de modèles de description d'architectures existent :

- les langages de description d'architecture dynamiques,
- les modèles de graphes.

2.3.1 Description des architectures logicielles par les ADL

Les langages de description d'architecture sont des langages généralement déclaratifs dans lesquels le couplage entre la spécification architecturale et l'implémentation est faible. Ces langages dédiés permettent de spécifier des configurations architecturales comme des assemblages de composants et éventuellement de connecteurs. Ils peuvent être classés en deux grandes familles (Medvidovic et al, 2000) :

- La famille des langages de description qui privilégie la description des éléments de l'architecture et leur assemblage structurel. On y trouve Wright (Allen, 1997) et Rapide (Lukman et al., 1995). Cette famille de langages a pour but la formalisation, la vérification et la validation d'architecture. Certains ADL comme Wright permettent aussi de spécifier le comportement dans la définition des composants. Ils associent à chaque port une description formelle dans l'algèbre de processus CSP (Communicating Sequential Processes) précisant ses interactions avec l'environnement. La partie calcul décrit le comportement interne du composant suite à des invocations externes.
- La famille de langages de configuration, tel que Darwin (Magee et al., 1995), se focalise sur la description de la configuration d'une architecture et sur la dynamique du système.

La totalité de ces ADL intègre la notion de composants qui sont définis selon l'approche boîte noire, c.à.d., leur structure interne et leur implémentation sont cachées et seule

leur interface d'interaction est visible aux autres composants. Néanmoins, deux visions existent pour les connexions :

- La déclaration implicite considère une connexion comme un simple lien entre exactement deux composants (c'est le cas notamment de Darwin et Rapide)
- La déclaration explicite introduit une entité spécifique appelée connecteur qui permet de relier un groupe de composants (c'est le cas notamment de Wright et C2SADL). Le connecteur offre en plus la possibilité de spécifier le lien de communication qu'il matérialise, de décrire le protocole d'interaction liant les composants qu'il connecte (le type de messages échangés et l'ordre causal de leur envoi et de leur réception).

La considération ou la non prise en compte de l'entité connecteur n'est cependant pas une caractéristique déterminante pour la puissance d'expression des ADL. Les connecteurs peuvent être simulés par des composants décrivant le protocole d'interaction qui leur est associé. Dans le cas d'une déclaration explicite du connecteur, il y a une forte similitude avec la définition du composant. L'interface du connecteur définit le rôle des différents participants à l'interaction. Ces interfaces ne décrivent pas des services fonctionnels, mais les mécanismes de connexion entre composants. La description de l'implantation du connecteur définit le protocole qu'il est nécessaire de mettre en œuvre pour réaliser l'interaction.

Le langage C2SADL

C2SADL (C2 style Software Architecture Description Language) est un exemple de langage de description d'architecture dynamique basée sur le style C2 qui organise les composants d'une architecture logicielle en couches reliées par des connecteurs (voir figure 2.2). La communication entre composants est réalisée par envoi de messages asynchrones via les connecteurs.

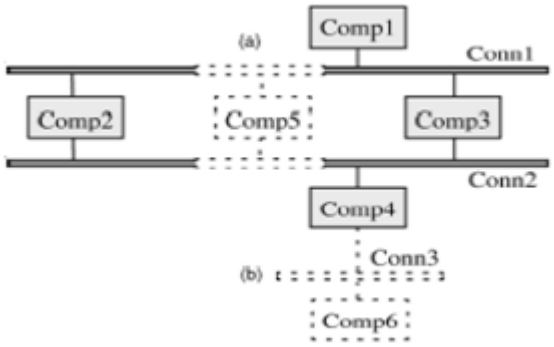


FIGURE 2.2 – Éléments d'une architecture logicielle dans C2SADL.

Le composant

Un composant dans C2SADL (Medvidovic, 1999) peut avoir un état et ses propres tâches de contrôle. Il possède aussi une structure externe et interne particulières. La structure externe est constituée des deux interfaces *top* et *bottom*. Son interface *top* définit l'ensemble des notifications auxquelles le composant répond et l'ensemble des requêtes qu'il envoie aux composants du niveau supérieur. L'interface *bottom* d'un composant définit l'ensemble des notifications qu'il émet vers le bas de l'architecture et l'ensemble des requêtes auxquelles il répond. En fait, les requêtes ne peuvent être envoyées que vers le haut de l'architecture à travers l'interface top, alors que les notifications ne seront envoyées que vers le bas à travers l'interface *buttom*. La structure interne définit le comportement du composant suite à une invocation externe sous forme d'un message. Un message dans le style C2 correspond soit à l'invocation d'une opération du composant ou une notification sur l'état d'exécution d'une opération. L'interface d'un composant est constituée de l'ensemble des messages pouvant être reçus et l'ensemble de messages pouvant être émis. Chaque composant C2 est défini par un nom, les interfaces top et down, un comportement et une éventuelle implémentation. La signature de chaque élément de

l'interface comporte les rubriques suivantes : un modificateur indiquant le sens de communication (Provided / Request), un nom, un ensemble de paramètres et un éventuel résultat. Par ailleurs, chaque paramètre a un nom et un type.

Le connecteur

Les connecteurs dans le style C2 jouent le rôle de médiateur d'interactions entre composants car ils contrôlent la transmission et la distribution des messages ; définis par un nom et un ensemble de paramètres typés. Ils permettent de lier plusieurs composants, d'une part par la diffusion des messages (demandes de requêtes ou événements) envoyés et destinés aux composants et d'autre part le filtrage de certains messages. Le connecteur suit l'une des politiques de filtrage suivantes :

- **aucun filtrage (no filtering) :** chaque événement est systématiquement diffusé à tous les composants reliés aux connecteurs.
- **notification filtrée (notification filtering) :** La notification n'est envoyée qu'aux composants qui se sont abonnés à cet événement.
- **filtrage conditionnel (conditional) :** le connecteur définit un ordre de priorité d'envoi du message aux composants qui y sont rattachés, basé sur des critères d'évaluation spécifiés par le concepteur lors de la construction de l'application. Le connecteur envoie par la suite un message de notification à chacun des composants selon l'ordre de priorité établit jusqu'à la satisfaction d'une condition de terminaison. Cette politique peut être utilisée pour sélectionner un composant parmi un ensemble de composants ayant des fonctions identiques mais implémentées différemment.
- **Puits d'évènement (Message Sink) :** le connecteur ignore tous les messages qu'il reçoit. Cette politique de filtrage est utile lors de l'isolation d'un sous-système de l'architecture ainsi que lors de l'ajout incrémental de composants à une architecture existante.

La Configuration

C2 permet de définir la configuration d'une architecture en spécifiant :

- la structure de l'application, c.à.d., les composants utilisés et les connecteurs assurant les interactions entre ces composants,
- la dynamique de l'application, c.à.d., les changements de l'architecture au cours de l'exécution comme par exemple l'ajout ou la suppression de composants.

Exemple 2.1. Spécification d'une architecture logicielle dans C2SADL

```
architecture CargoRouteSystem is {
component_types {
       component DeliveryPort is extern {Port.c2;}
       component GraphicsBinding is virtual {}
       ...
}
connector_types {
       connector FiltConn is {filter msg_filter;}
       connector RegConn is {filter no_filter;}
}
architectural_topology {
component_instances {
       Runway : DeliveryPort;
       Binding : GraphicsBinding;
       ...
}
connector_instances {
       UtilityConn : FiltConn;
       BindingConn : RegConn;
       ...
}
connections {
       connector UtilityConn {
       top SimClock, DistanceCalc;
       bottom Runway, Truck;
}
connector BindingConn {
       top LayoutArtist, RouteArt;
       bottom Binding;
}
...
}
}
}
```

FIGURE 2.3 – Description d'une architecture dans C2SADL.

Le système *CargoRouteSystem* est constitué de deux types de composants *DelivryPort* et *graphicsBinding* et de deux types de connecteur *FilConn* et *RegConn*. La configuration initiale, déclarée par la clause *Architectural_topology* (voir figure 2.3 ci-dessus), contient une instance de chaque composant et de chaque connecteur. Les interfaces *top* et *bottom* de chaque connecteur sont définies par la clause *connections*.

```
component DeliveryPort is
subtype CargoRouteEntity (int \and beh) {
state {
       cargo : \set Shipment;
       selected : Integer;
       ...
}
invariant {
       (cap >= 0) \and (cap <= max_cap);
}
interface {
       prov ip_selshp: Select(sel : Integer);
       req ir_clktck: ClockTick();
       ...
}
operations {
       prov op_selshp: {
              let num : Integer;
              pre num <= #cargo;
              post ~selected = num;
       }
       req or_clktck: {
              let time : STATE_VARIABLE;
              post ~time = time + 1;
       }
       ...
}
map {
       ip_selshp -> op_selshp (sel -> num);
       ir_clktck -> or_clktck ();
       ...
}
}
```

FIGURE 2.4 – Spécification d'un composant C2SADL.

Une spécification partielle du composant *DeliveryPort* est donnée dans la figure 2.4 ci-dessus. Elle définit la structure de l'état interne du composant par les variables *cargo* et *selected*, et un invariant exprimant le fait que la capacité du port doit être toujours comprise entre 0 et le maximum autorisé. Le comportement est défini par les

opérations à exécuter en réaction à une invocation externe. Les pré/post conditions de chaque opération sont exprimées dans la logique du premier ordre. La clause *map* fait correspondre à chaque interface d'entrée l'opération à exécuter.
Des exemples d'opération fournies et requises sont aussi définis. L'opération requise $ClockTick$ est fournie par un composant $SystemClock$ du système global. Cette opération permet de synchroniser l'horloge du composant $DelivryPort$ avec celles des autres composants.

CommUnity

CommUnity (Fiadeiro et al., 2003) se base sur les principes d'abstraction et de raffinement dans la spécification des composants de l'architecture d'un système. Les composants CommUnity ne sont pas nécessairement des programmes, mais plutôt des abstractions, appelés *designs*, qui peuvent être raffinés au fur et à mesure de la progression dans le processus de développement. Par ailleurs, la spécification de ces designs peut éventuellement être paramétrée par les types de données que les composants sont susceptibles de manipuler.

L'objectif visé par le support du concept d'abstraction n'est pas uniquement l'adoption d'une approche de construction incrémentale, mais aussi l'obtention d'une couche de conception architecturale la plus proche possible du domaine d'application considéré. Le principe d'abstraction est réalisé dans CommUnity à travers :

- le concept de sous-spécification (underspecification) en définissant des designs assez génériques dénotant non pas un programme unique mais des collections de programmes,
- les designs peuvent ne pas correspondre directement à ceux de la plate-forme d'implémentation finale. Ainsi, deux procédures de raffinements sont nécessaires

dans CommUnity : Compléter la spécification des designs abstraits par la définition des types de données choisis, ensuite réifier ces types de données afin d'intégrer des programmes dans l'environnement cible.

Le Composant

Les éléments d'une architecture logicielle dans CommUnity sont appelés *design* et sont décrit par un ensemble de canaux X (d'entrée, de sortie ou privés) et un ensemble d'actions Γ(partagées ou privées). Les canaux d'entrée sont utilisés pour lire des données en provenance de l'environnement, le design n'a aucun contrôle sur les valeurs reçues sur ces canaux. Les canaux de sortie et privés sont contrôlés localement par le design. Les canaux de sortie permettent de communiquer des valeurs à l'environnement. Les actions privées représentent des calculs internes, c.à.d., leur exécution est sous le contrôle total du design. Les actions partagées expriment des interactions avec l'environnement.
(X, Γ) constitue la signature du composant qui a la structure suivante :

component P **is**

out out(V)

in in(V)

prv prv(V)

do $[\,]g \in sh(\Gamma)\ \ g[D(g)]\colon L(g), U(g) \longrightarrow R(g)$

$[\,]g \in prv(\Gamma)g[D(g)]\colon L(g), U(g) \longrightarrow R(g)$

– V est un ensemble de variables qui peuvent être déclarées d'entrée (*in*), de sortie (*out*) ou internes (*private*). Les variables *in* permettent l'échange de données de l'environnement vers le composant, elles ne peuvent être que lues par celui-ci et non pas modifiées. Les variables *out* permettent de transmettre des résultats de calculs effectués par le composant à l'environnement. Quand aux variables internes, elles ne sont pas accessibles par l'environnement qui ne peut ni les lire ni les modifiées.

Elles interviennent dans les calculs internes du composant. Les variables in et out correspondent aux canaux du composant.

L'ensemble $loc(V) = prv(V) \cup out(V)$ est appelé ensemble de variables locales. Toutes les variables sont typées par $sort(v) \in S$.

- Γ est un ensemble de noms d'actions qui peuvent être déclarées privées ou partagées. Les actions privées représentent des calculs internes car elles sont sous le contrôle total du composant. Les actions partagées représentent des interactions possibles avec l'environnement. Elles correspondent à des points de rendez-vous car les interactions sont synchrones.
- Quatre attributs $D(g)$, $L(g)$, $U(g)$ et $R(g)$ sont associés à chaque action $g \in \Gamma$, avec :
  - $D(g)$ est un sous-ensemble de $loc(V)$. Il est constitué des variables (canaux) locales qui seront affectés par l'execution de g. S'il est possible d'inférer l'ensemble $D(g)$ à partir des assignations définies dans R(g), il peut être omis.
  - $L(g)$ et $U(g)$ sont deux conditions sur $\mathcal{X}$, tel que $U(g) \supset L(g)$, définissant l'intervalle de sensibilisation de toute action gardée qui implémente g. L(g) exprime la condition nécessaire pour sensibiliser g ; sa négation définit la condition de blocage. $U(g)$ est la condition suffisante pour la sensibilisation. Ainsi, la condition de sensibilisation de g est totalement définie seulement si $L(g)$ et $U(g)$ sont équivalents,
  - $R(g)$ définit les effets de bord de l'action g sur les variables locales. Si $D(g)$ est vide, $R(g)$ est une tautologie dénotée par skip.

Les différents attributs que CommUnity permet de rattacher aux actions (services) du composant représentent les mécanismes de mise en œuvre du principe de sous-spécification. Les actions peuvent être sous-spécifiées dans le sens où leurs conditions de sensibilisation peuvent ne pas être totalement définies. Un raffinement possible consiste

à réduire progressivement l'intervalle de sensibilisation délimité par L et U. Par ailleurs, les effets des actions peuvent aussi être partiellement définis. Le processus de raffinement s'arrête lorsque pour chaque action g, $L(g)$ et $U(g)$ coïncident et la relation $R(g)$ est totalement définie par des assignations conditionnelles. Le *design* ainsi obtenu est appelé programme.

La description d'un *design* peut être paramétrée par une collection de types de données utilisés pour structurer les données qui transitent sur ses canaux et définir ses actions.

Exemple 2.2. Déclaration d'un buffer borné

component buffer [t : sort, bound : t] **is**

in i : t

out o : t

prv rd : bool, b : list(t)

do put : $|\mathsf{b}| < bound$: b := b.i

[ ] **prv** next : $|\mathsf{b}| > 0 \wedge \neg$rd $\longrightarrow$ o := head(b)‖ b := tail(b)‖ rd := true [ ]

get : rd $\longrightarrow$ rd := false

Ce design modélise un buffer borné, sa capacité ainsi que la sorte des éléments stockés sont des paramètres, il utilise une politique FIFO. Il peut stocker des messages qu'il vient de recevoir dans la variable d'entrée i suite à une interaction avec l'environnement en exécutant l'action *put* ; partagée avec l'environnement, à condition que le buffer n'est pas plein. Le buffer peut aussi retirer des messages et les rendre disponibles à l'environnement sur la variable de sortie o en exécutant l'action *next*. Le dépôt de messages dans o n'est possible que s'il y'a encore des messages dans le buffer et que le message courant dans o a été déjà lu par l'environnement. Cette contrainte est modélisée par l'action *get* et la variable privée rd.

Le Connecteur

Les connecteurs ne sont pas des entités de première classe. CommUnity se contente de définir des liens entre les variables d'entrée et de sorties des composants. Il se base sur une communication par envoie de messages synchrones. Cette synchronisation requiert la définition de points de rendez-vous des émetteurs et récepteurs de messages. La présence des deux partenaires aux rendez-vous est sollicitée à travers l'exécution d'actions partagées et l'utilisation de variables partagées pour l'échange de messages.
Comme les noms des variables et des actions sont locaux aux composants, CommUnity définit le concept de canal d'interaction pour relier les composants qui participent à chaque interaction. Le rôle du canal consiste à établir des relations de correspondance entre les variables d'entrée d'un composant et les variables de sorties d'une part et les actions d'envoie et de réception d'autre part (actions partagées). Les liens entre variables *in/out* et actions partagées sont définis par des relations entre les signatures des composants qui interviennent dans chaque interaction.

La configuration

Une configuration dans CommUnity définit le schéma d'interaction entre composants par un diagramme comportant deux types de nœuds. Le premier type de nœuds est étiqueté par les signatures des composants faisant actuellement partie de cette configuration. Le deuxième type de nœuds permet de définir les canaux de communication entre composants, chacun est étiqueté par des ensembles d'interactions (liens entre des variables d'entrée/sortie et des actions partagées). Un arc d'un canal vers un composant montre le rôle joué par ce composant dans l'interaction. Il spécifie aussi les variables et les actions qui participent dans cette interaction. Le nœud étiqueté par *channel* représente l'ensemble des liens entre un composant de type $buffer$ et un composant de type *sender*. Le canal établi entre le $buffer$ et le *sender* est constitué d'une variable d'entrée pour

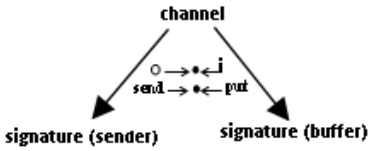


FIGURE 2.5 – Architecture logicielle dans CommUnity.

modéliser le médium sur lequel transitent les messages pour arriver au composant destinataire. Il comporte aussi une action partagée par les deux composants participants à l'interaction, le dépôt de la donnée du coté *sender* dans la variable de sortie o en exécutant l'action *send* et en même temps sa récupération par le $buffer$ en exécutant l'action *get* sur la variable d'entrée i.

π-ADL

Le langage π-ADL fournit les constructions de base pour la description de la structure et du comportement d'architectures logicielles statiques, dynamiques et mobiles. C'est un langage de spécification formelle supportant la vérification automatique de certaines propriétés. Comme la plupart des ADLs, π-ADL offre les éléments de base pour la description d'une architecture logicielle : les composants, les connecteurs et leur schéma de composition (configurations).

Le composant

Le rôle architectural d'un composant dans π-ADL est de spécifier les éléments de calcul d'un système logiciel. Un composant est défini par un ensemble de ports externes, représentant l'interface du composant, et un comportement interne. Les ports sont décrits par des connexions entre le composant et son environnement. Des canaux de communication entre éléments architecturaux ; appelés connexions, permettent de relier les ports

des composants aux rôles des connecteurs. Ces connexions représentent les points d'interaction de base. Un composant peut envoyer ou recevoir des données via ces connexions une fois établies.

Deux types de connexions sont offerts par le langage π-ADL, les connexions à sens unique véhiculent des données d'entrée, par contre les connexions bidirectionnelles peuvent véhiculer des données d'entrée et de sortie.

Le connecteur

Les connecteurs sont des composants particuliers dans le sens où ils ont des ports externes et un comportement interne. Cependant, leur rôle architectural est de relier les composants d'une architecture logicielle, ils spécifient les interactions entre composants.

La configuration

L'assemblage des instances de composants et connecteurs est réalisé à travers leurs ports respectifs. Attacher un port d'un composant à un rôle d'un connecteur nécessite l'existence d'au moins une connexion de chaque côté. Cet attachement est effectué par unification des deux connexions ou par passage de valeur (qui peut être une donnée, une connexion ou un élément architectural entier).

π-ADL traite les trois éléments architecturaux composants, connecteurs et configuration (appelée architecture dans π-ADL) de manière uniforme et les considère comme des abstractions. Celles-ci comportent des définitions de types exprimant les données qui circulent entre éléments architecturaux, des ports ainsi que les différentes connexions qui leurs sont rattachés en précisant le sens de communication (in/out). La clause *behavior* permet de définir le comportement de l'abstraction. Elle décrit les transformations que doivent subir les données d'entrées pour obtenir les données de sortie. Des invariants sont utilisés pour définir des contraintes sur l'évolution de l'élément architectural à l'aide de la clause *assuming*. Les ports définissent les points d'interaction d'un composant avec

l'environnement. Pour un connecteur, ils décrivent les différents rôles de celui-ci dans les interactions auxquelles il participe. Le comportement du connecteur définit les liens entre les rôles d'entrée et les rôles de sortie.

Une architecture est considérée comme un composite, elle peut avoir des ports. Son comportement définit les instances des composants et connecteurs qui participent dans cette composition. Il définit aussi les attachements entre les ports des composants et les rôles des connecteurs.

Exemple 2.3. Système Client/Serveur

Soit une architecture constituée de deux types de composants, un element d'acquisition de données, des paires (Key, $data$), et un autre pour le tri de ces données (voir figure 2.6). La description boite noire ne spécifie que les ports de l'architecture, le sens de com-

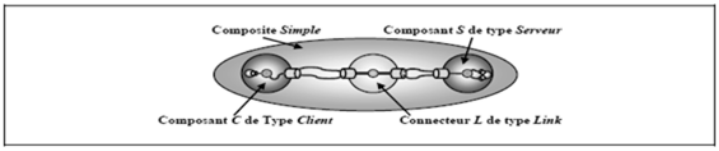


FIGURE 2.6 – Architecture du système client/serveur dans π-ADL.

munication (in/out) et le type de messages qui leurs sont associés. La vue transparente de cette architecture est obtenue par la description de la structure et le comportement de chaque composant. A titre d'exemple, le composant responsable du tri des données est décrit dans la figure 2.7 ci-dessous :

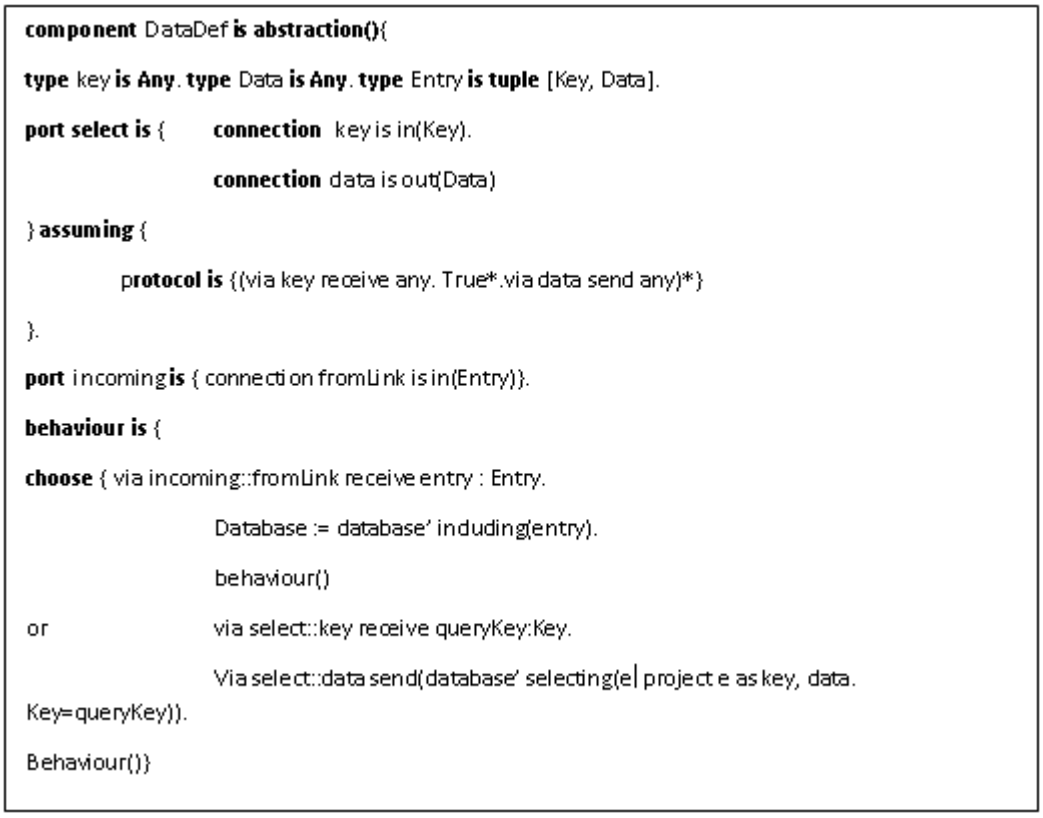


FIGURE 2.7 – Description d'un composant dans π-ADL.

2.3.2 Description des architectures logicielles par les graphes

Les grammaires de graphe à contexte libre offrent la possibilité de définir l'architecture logicielle d'un système par un graphe. Deux visions sont possibles dans la représentation de l'architecture logicielle :

- l'une considère les entités logicielles (composants) constituant l'architecture comme nœuds du graphe et les liens de communication entre ces entités (connexions) comme arcs (LeMetayer, 1998).
- l'autre considère les nœuds comme interfaces des composants logiciels et les arcs comme composants (Hirch et al., 1996).

Plusieurs modèles (LeMetayer, 1998 ; Hirsch, 1996 ; Bruni et al., 2007 ; Milner et al., 2005) de description d'architectures logicielles surtout dynamiques, existent dans la littérature. le choix des systèmes réactifs bigraphiques (Jensen et Milner, 2005) est motivé par le fait que c'est l'unique modèle à base de graphes dédié à la description de systèmes à code mobile.

Systèmes réactifs bigraphiques

La théorie des systèmes réactifs bigraphiques (BRSs)(Milner, est basée sur un modèle graphique pour la spécification d'applications distribuées à code mobile. Ce modèle offre les outils nécessaires pour la prise en charge des deux dimensions, interaction et distribution spatiale, de l'application à spécifier en fusionnant deux types de graphes d'où le nom de bigraphes.

- Le graphe des places spécifie l'hiérarchie des entités physiques ou logicielles, constituant l'application, et capables d'accueillir d'autres entités.
- Le graphe des liens montre les connectivités possibles entre les différentes entités.

Sur la base d'un ensemble commun de nœuds, correspondant aux entités physiques ou virtuelles d'une application distribuées, un bigraphe est construit à partir de la fusion de deux structures indépendantes : le graphe des places, ayant la structure d'une forêt, montre la distribution spatiale de l'application. Le graphe des liens est un hypergraphe établissant le schéma de connexion des différents nœuds. Alors qu'un arc dans le graphe des places montre la relation d'imbrication entre les éléments de l'application, un arc dans le graphe des liens établit une connexion entre les ports des ses éléments. Chaque arbre dans le graphe des places représente une région qui peut contenir des sites, correspondant aux feuilles de l'arbre, où d'autres bigraphes peuvent être insérés ou hébergés.

Définition 2.3.1. (Milner, 2004) Un bigraphe est défini par $G = (V, E, ctrl, G^P, G^L)$, tels que :

- V est un ensemble fini de nœuds,
- E est un ensemble fini d'hyper-arcs,
- $crtl\colon V \longrightarrow K$ est une transformation qui associe à chaque nœud v_i de V un contrôleur $k \in K$ indiquant le nombre de ports et son comportement dynamique.
- $G^P = (V, E, ctrl, prnt)\colon m \longrightarrow n$ est le graphe des places associé à G, où $prnt\colon m \cup V \longrightarrow V \cup n$ est une fonction de parenté associant à chaque nœud son parent hiérarchique. m et n représentent respectivement le nombre de sites et de régions dans le graphe des places.
- $G^L = (V, E, ctrl, link)\colon X \longrightarrow Y$ est le graphe des liens de G avec $link\colon X \cup P \longrightarrow E \cup Y$ est une transformation montrant le flux de données des noms internes X ou les ports P vers les noms externes Y ou les arcs E.
- $I = \langle m, X\rangle$ et $J = \langle n, Y\rangle$ sont les interfaces internes et externes du graphe respectivement, m étant le nombre de sites ou trous (emplacements dans le graphe susceptibles d'abriter de nouveaux éléments logiciels), X étant l'ensemble des noms internes. n est le nombre de régions (éléments du bigraphe aptes à s'intégrer dans d'autres bigraphes) et Y est l'ensemble des noms externes. Une interface $I = \langle 0, \emptyset\rangle$ peut être notée par $I = \epsilon$.

Les sites représentent des emplacements encore libres dans le bigraphe et susceptible d'accueillir de nouveaux éléments logiciels. Les régions par contre sont des éléments logiciels non encore hébergé.

Exemple 2.4. Exemple de bigraphe

Soit le bigraphe $G = (V, E\, ctrl, G^P, G^L)\colon \langle 0, x\rangle \longrightarrow \langle 3, \emptyset\rangle$ présenté dans la figure 2.8 dont :

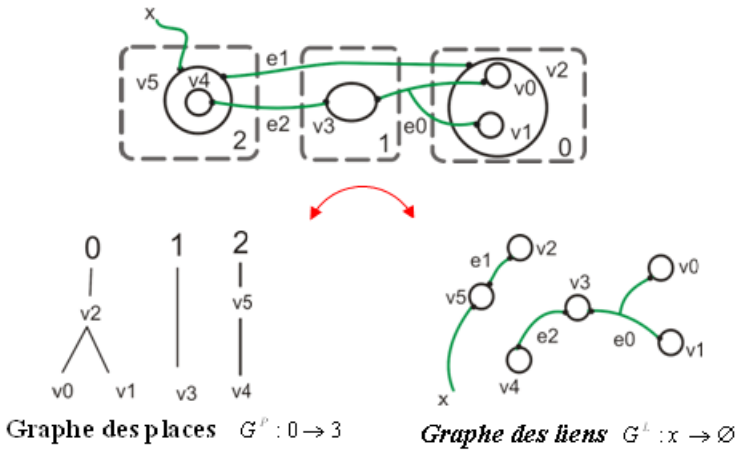


FIGURE 2.8 – Un bigraphe et ses deux structures.

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ est l'ensemble des nœuds ; représentés par des cercles et des ovales dans la figure 2.8
- $E = \{e_0, e_1, e_2\}$ est l'ensemble d'hyper-arcs ; représentés par des lignes et appelés liens,
- $ctrl = \{(v_0 : 1), (v_1 : 1), (v_2 : 1), (v_3 : 2), (v_4 : 1), (v_5 : 2)\}$ es une transformation,
- $prnt = \{(v_0 : v_2), (v_1 : v_2), (v_2 : \emptyset), (v_4 : v_5), (v_5 : \emptyset)\}$ est la fonction de parenté .

Le nombre de sites ici est nul ($m = 0$) , c.à.d., pas d'emplacement libre dans ce bigraphe de sites. Le bigraphe contient trois régions ($n = 3$) numérotées de 0..2 et représentées par des rectangles pointillés.

2.4 Conclusion

Ce chapitre présente un survol sur les concepts de base des architectures logicielles ainsi que leurs formalismes de description. Deux grandes familles de modèles de description d'architectures logicielles sont présentées. Le choix de ces trois langages parmi la

multitude d'ADL qui existent dans la littérature s'est basé sur la base formelle sous-jacente et l'aptitude du langage à décrire la reconfiguration dynamique des architectures logicielles. Le choix des BRS est justifié par le fait que c'est un modèle dédié à la description d'applications à code mobile. La description de la reconfiguration dynamique et plus particulièrement la mobilité dans ces modèles fera l'objet d'une étude détaillée dans le chapitre suivant.

Chapitre 3

Mobilité dans les architectures logicielles

3.1 Introduction

Les approches de conception traditionnelles ; basées sur la spécification des propriétés fonctionnelles et non fonctionnelles telles que l'interaction et la coordination dans la même entité logicielle, ne sont pas adéquates pour la conception d'applications distribuées qui exploitent la mobilité et la reconfiguration dynamique de composants logiciels. Cela est dû au fait que la mobilité du code favorise la vue globale de la dynamique sociale au détriment de la vue individuelle d'un composant particulier, alors que ces approches n'ont pas réussi à proposer des unités de construction autonomes capables de modifier leurs liens avec l'environnement de manière dynamique.

La réussite des architectures logicielles à découpler les propriétés fonctionnelles des composants de leurs interactions les rend un paradigme adéquat pour se concentrer sur la structure globale du système et ses éventuels changements dynamiques causés par la mobilité des composants. Cette dynamique structurelle entre dans un cadre plus large ; la reconfiguration dynamique d'architectures logicielles, dans lequel la mobilité ou bien la migration de code est considérée comme un scénario particulier de reconfiguration

dynamique. En effet, la migration du code signifie qu'une unité d'un certain type est supprimée d'une localisation et une autre unité du même type est crée dans une autre localisation. De ce fait, tout l'intérêt est porté sur le changement dynamique des composants et leurs connecteurs à un niveau architectural.

Bien que les langages de description d'architecture classiques présentent plusieurs avantages, tels que :

- La possibilité de décrire des composants hiérarchiques.
- La description explicite des connexions et interconnexions.
- L'existence d'outils d'analyse et de simulateurs de description d'architecture

La plupart de ces ADL présentent des points faibles :

- L'absence d'abstraction du fait que la plupart des ADL proposent des constructions syntaxiques proches de celles des langages de programmation.
- L'approche proposée par ces ADL est plutôt statique, la dynamique, que ce soit structurelle ou comportementale, est généralement non prise en charge.

Ainsi, la plupart des ADL s'intéressent à la modélisation de l'architecture logicielle à la phase de conception, alors que le calcul mobile nécessite une reconfiguration et une adaptabilité dynamique.

Les graphes et leurs règles de transformation émergent aussi ces dernières années comme modèle de description des architectures logicielles dynamiques.

Les mécanismes de prise en charge de la dynamique structurelle dans les architectures logicielles dans les deux catégories de modèles seront présentés dans la suite du chapitre. Un ensemble de critères et de contraintes sera aussi établi pour étudier l'aptitude de ces modèles à gérer la mobilité du code.

3.2 Architectures logicielles dynamiques

Nothing is as constant as the occurrence of changes, d'après Chris Salzman et al. (Salzman, 99). Les systèmes logiciels aussi n'échappent à cette règle car ils sont toujours amenés à évoluer pour diverses raisons : évolution de l'architecture, ajout de fonctionnalités, intégration de nouvelles technologies de développement ou de communication, modification de l'environnement d'exécution, personnalisation des services, etc.

Le besoin de doter le système de capacités à s'adapter à de nouveaux besoins est alors plus important que le besoin de créer le système lui même. Les changements deviennent des objectifs de première classe et nécessitent une architecture métier et technologique dont les composants peuvent être ajoutés, modifiés, remplacés et reconfigurés dynamiquement (Finger, 2000).

Afin que la dynamique structurelle et la mobilité de composants logiciels soient des opérations systématiques dans le processus du génie logiciel, une définition claire et précise du terme dynamique d'une architecture logicielle ou reconfiguration dynamique est nécessaire. Les architectures dynamiques sont définies dans (Canal et al., 1999) comme étant *celles qui décrivent comment les composants sont créés, interconnectés, et supprimés pendant l'exécution, et qui permettent la reconfiguration de leur topologie de communication pendant l'exécution.*

La dynamique de la configuration est une propriété qui introduit des difficultés supplémentaires dans la description des architectures logicielles. Contrairement à une architecture statique dont la description consiste à énumérer les différents composants et connexions qui la constituent, la configuration d'une architecture dynamique est soumise à des changements causés par l'ajout ou la suppression de composants et de connexions tout au long du cycle de vie de l'application.

Les modèles de description des architectures logicielles dynamiques doivent prendre en

considération les aspects suivants :

- Les aspects de type déclaratif, qui incluent la description de toutes les instances valides de l'architecture,
- l'ensemble des changements autorisés sur l'architecture ainsi que leurs contraintes structurelles ou fonctionnelles,
- Les aspects opérationnels spécifiant la gestion de l'évolution de l'architecture. Ils incluent la définition des événements qui provoquent la reconfiguration de l'architecture et la spécification des actions de transformation qu'elle doit subir en réaction à ces événements.

3.2.1 Dynamique des architectures logicielles dans les ADL

Les architectures logicielles dynamiques sont caractérisées par l'évolution de la structure par ajout/suppression de composants et de liens et de l'état du calcul par exécution d'actions fournies par les composants. Cependant, la plupart des ADL ont un regard statique sur les architectures logicielles et une vue boite noire sur les composants logiciels. Seuls quelques uns proposent des mécanismes de reconfiguration dynamique de l'architecture et/ou une vue transparente des composants. Nous présenterons les concepts et mécanismes proposés par trois ADL ; C2SADL, CommUnity et π-ADL pour la spécification de la dynamique structurelle et comportementale des architectures logicielles. Nous insisterons surtout sur le moyens offerts pour la spécification des effects de bords de chaque type d'evolution sur l'autre.

C2SADL

a. Description de la reconfiguration dynamique

C2SADL propose un ensemble d'opérations pour l'ajout, la suppression et la re-connexion des éléments de l'architecture pendant l'exécution du système. C2SADEL exploite aussi des connecteurs flexibles pouvant supporter la dynamique de la topologie.

Exemple 3.1. Reconfiguration d'un système Client/Serveur

Soit l'exemple du client/serveur de la figure 3.1 auquel nous souhaitons ajouter un client de manière dynamique. Le composant *Client*2 est ajouté à l'architecture *ClientServer*

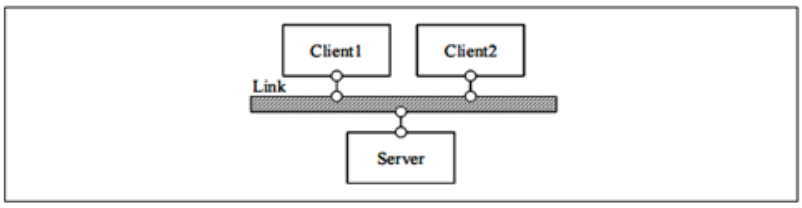


FIGURE 3.1 – Représentation graphique de système Client/Serveur dans C2SADL.

grâce à l'opérateur *addComponent*. Il est ensuite rattaché au connecteur *Link* grâce à l'opérateur *weld*. Ces deux opérations ne sont possibles qu'après construction du système.

ClientServer.**AddComponent**(Client2) ;

ClientServer.**Weld**(Link, Client2) ;

C2SADEL permet aussi de détacher un composant de l'architecture comme spécifier ci-dessous pour le composant *Client*1 qui sera momentanément détaché de l'architecture.

Link.**SetTopFilter**(Client2, **message.sink**) ;

De même que C2SADEL permet l'ajout de composants, il propose leur suppression, comme spécifier ci-dessous.

ClientServer.Unweld(Link, Client2) ;

ClientServer.RemoveComponent(Client2) ;

Le composant *Client*1 est d'abord définitivement détaché du connecteur *Link* grâce à l'opérateur *unweld*. Il est ensuite supprimé de l'architecture grâce à l'opérateur *Remove-Component*.

Par rapport aux trois aspects de la dynamique, C2SADEL ne prend en charge que l'aspect déclaratif en permettant de spécifier les attachements et les composants dynamiques. Par contre, il ne propose aucune primitive pour spécifier les contraintes à vérifier lors de la reconfiguration ni les effets de celle-ci. Ainsi les changements à apporter sur la configuration initiale doivent être planifiés avant la définition de l'architecture. De plus, ni l'origine de la création/suppression de ces composants, ni les moyens de gestion de ces composants une fois créés ne sont spécifiés .

b. Spécification du comportement

La clause *map* dans la déclaration d'un composant C2SADL fait correspondre à chaque interface fournie l'opération à exécuter. En effet, le comportement d'un composant est spécifié par un invariant et un ensemble d'opérations. L'invariant sert à décrire les propriétés à satisfaire par toutes les évolutions d'états autorisées du composant. Chaque opération a des pré-conditions et des post-conditions, exprimées dans logique du premier ordre, et un éventuel effect. Elle peut aussi porter la prescription (Provide/Require). Les pré/post conditions des opérations fournies expriment la sémantique qui leur est associée. Du fait que l'interface est complètement séparée du comportement, une fonction de correspondance (mapping) est établie entre les éléments de l'interface et les opérations associées. Cette fonction est totalement subjective, chaque élément de l'interface est associé à une et une seule opération. Le mapping n'est défini que s'il existe une relation de sous-type entre les paramètres définis dans la signature de l'élément de l'interface et ceux de l'opération. Pour le résultat, ce sera l'inverse. Cette notion de sous-typage

permet d'associer une même opération à de multiples interfaces. Bien que le comportement individuel des composants soit défini, C2SADL ne fournit aucun mécanisme de coordination afin de construire le comportement global de l'architecture.

CommUnity

a. Description de la reconfiguration dynamique

CommUnity a été étendu dans (Lopes et al., 2006) pour supporter la description des aspects liés à la distribution et la mobilité des systèmes d'une manière complètement indépendante des aspects calcul et interaction. Afin d'assurer la séparation des préoccupations entre le calcul, la coordination et la distribution, les designs ont été étendus pour être conscients de leurs localisation en associant à chaque élément d'un design (canal privé, canal de sortie, ou action) un conteneur capable de le déplacer vers différentes positions. Ainsi, CommUnity a opté pour une granularité fine des éléments mobiles en permettant à une variable ou une action de changer de localisation.

Cette extension est basée sur une représentation explicite de l'espace de mobilité des composants. En effet, CommUnity a défini un type de données spécial *Loc* décrivant l'ensemble de valeurs possibles représentant l'espace de mobilité. Il propose aussi un ensemble d'opérations pour mettre à jour l'hiérarchie et la taxonomie des localisations. Concrètement, les signatures des *designs* sont étendues par un ensemble de variables de localisation L_λ de type *Loc*, chaque canal de sortie, canal privé et action est assigné à un ensemble de variables de localisation :

– A chaque canal local x est associée une variable de localisation l en utilisant la notation $x@l$ dans la déclaration de x. Intuitivement, la valeur de l indique la position courante de x. Une modification de la valeur de l signifie le mouvement de x ainsi que tout les autres canaux et actions localisés dans l.

– A chaque nom d'action est associée un ensemble de variables de localisation $\Lambda(g)$ signifiant que l'execution de l'action g est distribuée à travers ces localisations. L'exécution de g consiste alors en une exécution synchrone d'une commande gardée dans chacune des localisations de $\Lambda(g)$.

Exemple 3.2. Description d'un composant mobile dans CommUnity

Soit un composant mobile *mobile_receiver* qui change de localisation suite à la reception d'un mot définissant sa nouvelle destination. Le *design* CommUnity qui modélise ce comportement est le suivant :

design mobile_receiver **is**
outloc l
in ibit : bool
out w@l : array(N, bit), k@l : nat
prv rd@l : bool, word@l : array(N, bit)
do rec@l : k<N⟶ word[k] :=ibit‖k :=k+1‖rd :=false
[ ] **prv** save-w@l : ¬ rd ∧k=N⟶rd :=true‖w :=word
[ ] new-w@l : rd∧k=N⟶rd :=false‖k :=0‖rd :=true‖l :=if(loc ?(w), loc(w), l)

La localisation du receveur est modélisée par un variable de sortie l car la mobilité est sous le contrôle du composant (mobilité subjective), si une nouvelle destination est fournie par l'environnement.
Le mouvement est réalisé par l'action $new - w$ si toutefois le mot nouvellement reçu représente une localisation. L'expression if(loc ?(w), loc(w), l) signifie que si le nouveau mot est une localisation alors l devient $loc(w)$, sinon elle garde son ancienne valeur.

b. Spécification du comportement

Le comportement d'un *design* (composant) dans CommUnity est décrit en associant à chaque action une expression de la forme :

$g[D(g)]$: $L(g)$, $U(g) \longrightarrow R(g)$

Les attributs de l'action g sont :

- $D(g)$ est un sous-ensemble de $loc(V)$ exprimant le sous-état du composant qui sera affecté par l'exécution de l'action libellée par g,
- $[L(g), U(g)]$ définit l'intervalle de sensibilisation de g,
- $R(g)$ définit les effets de bord de g sur les variables de V.

A chaque étape d'exécution d'un composant, chacune des actions dont la condition de sensibilisation (L(g)) est vérifiée, peut être exécutée. Dans ce cas, toutes ses assignations $R(g)$ seront effectuées automatiquement.

π-ADL

a. Description de la reconfiguration dynamique

π-ADL propose un opérateur *compose* pour construire un élément architectural par assemblage d'un ensemble de composants. Cet opérateur permet par ailleurs de créer les instances de composants cités dans l'assemblage, mais qui n'existent pas encore dans la configuration actuelle. Il crée aussi les liens de ces nouvelles instances. La syntaxe de l'opérateur *compose* est la suivante :

compose{

c1 ***is*** c1Type and c2 ***is*** c2Type ... cn **is** cnType}

where {p1 unifies p1′ and...pn ***unifies*** p′n

}

π-ADL permet de vérifier que l'instance n'a pas encore été créée à l'aide de la fonction booléenne $isConnectionUnified(C)$ qui teste la connexion de cette instance au reste de l'architecture logicielle.
π-ADL propose aussi l'opérateur **replicate** qui permet de définir un composite comme une réplication infinie de lui même.

b. Spécification du comportement

La sémantique opérationnelle de π-ADL est définie par un ensemble de règles de transition, chacune d'elles à la forme suivante :

$$r : \frac{P_1 \longrightarrow P_1' ... P_n \longrightarrow P_n'}{C \longrightarrow C'}$$

Les α_i sont appelées prémisses, alors que α est appelée conclusion. C et C' sont les états initial et final respectivement. Dans une règle de transition, les prémisses et les conclusions sont des relations transitives, c.à.d., si la transition étiquetée par $\alpha_1, ..., \alpha_n$ peut être sensibilisée, alors α peut l'être aussi. Des conditions et des effets de bord peuvent être exprimées à travers les prémisses et les conclusions respectivement.

Cette sémantique opérationnelle structurelle représente le comportement de π-ADL en termes d'un système de deduction. Ainsi, la sémantique formelle de π-ADL est définie par un système de transition dont les règles de transition formalisent la sémantique opérationnelle des constructions du langage.

3.2.2 Dynamique des architectures logicielles dans les modèles à base de graphes

L'avantage principal des grammaires de graphe est qu'elles permettent de décrire, en plus de l'aspect statique, l'aspect dynamique d'une architecture logicielle. En effet,

des règles de réécriture de graphes sont utilisées pour décrire la dynamique de l'architecture. Un système de réécriture de graphes est spécifié par un ensemble de règles de réécriture ; le rôle de chaque règle est de remplacer un sous-graphe (membre gauche de la règle) par un autre sous-graphe (membre droit de la règle). Ces règles permettent de décrire clairement les changements de la topologie d'une architecture en termes d'ajout et/ou de suppression de composants et/ou de connexions. Cette catégorie de modèles de description d'architectures logicielles dynamiques comprend principalement :

les travaux de Métayer : qui constituent probablement les premiers travaux de description basés sur les graphes (LeMetayer, 1998). Le modèle décrit est structuré en deux niveaux. Le premier décrit l'architecture sous forme d'un graphe et le second décrit les styles architecturaux par une grammaire de graphes. L'évolution de l'architecture est décrite par des règles de transformation de graphes. Le modèle définit une approche formelle basée sur un algorithme de vérification permettant de vérifier à priori et de manière statique la consistance des règles d'évolution de l'architecture. Cette approche permet de vérifier que les contraintes considérées par la description de l'architecture sont préservées par ces règles (J. Moo Mena, 2007).

les travaux de Hirsch et al. : les auteurs dans (Hirsch et al., 1996) utilisent un même modèle pour la description de l'architecture et de son évolution dynamique. Ils décomposent la description de l'architecture en trois parties. La première spécifie les règles de construction de la configuration initiale. La deuxième partie spécifie les règles régissant l'évolution dynamique de l'architecture. La troisième partie spécifie les règles régissant la communication. Ces travaux abordent aussi la problématique de la consistance du point de vue de la communication et des états de composants. Le cycle de vie des composants ; comprenant par exemple leur activation et leur désactivation, est décrit par affectation de labels aux nœuds des graphes et

des règles de réécriture. Ces labels correspondent à l'état courant des composants. La partie décrivant la communication est spécifiée via l'introduction d'événements (en notation Multi-formalismes CSP, Hoare) et leur prise en compte en étiquetant les arcs des graphes représentant les connexions entre composants (J. Moo Mena, 2007).

Les systèmes réactifs bigraphiques

a. Reconfiguration dynamique

La dynamique de la structure d'un bigraphe est spécifiée par des règles de transformation, appelées règles de réaction. Deux types de transformation sont possibles sur un bigraphe. La transformation sur les places exprime l'opération de mobilité dans le système considéré. Elle représente l'arrivée ou le départ d'une entité logicielle. De son côté, la transformation sur les liens exprime la connexion ou déconnexion d'un nœud du bigraphe à travers l'une de ses interfaces internes.

Définition 3.2.1. Une règle de réaction a la forme $(R\colon m \longrightarrow J,\ R'\colon m' \longrightarrow J,\ \eta)$ et stipule que le bigraphe R, appelé bigraphe *Redex*, peut être transformé en R', appelé bigraphe *Reactum*. $\eta\colon m' \longrightarrow m$ est une application sur les ordinaux qui désigne le nombre de sites. Les bigraphes R et R' sont sans noms internes alors que les noms externes sont identiques.

Exemple 2.3. Exemples de reconfiguration dynamique dans les bigraphes

La règle de réaction de la figure 3.2(Chang et al. 2007), exprime le fait qu'un agent, se trouvant dans la même région qu'une chambre, puisse y entrer. Par contre, la règle de réaction de la figure 3.3 (Milner, 2008) spécifie la livraison d'un message M à son destinataire R après avoir été authentifié via une clé K.

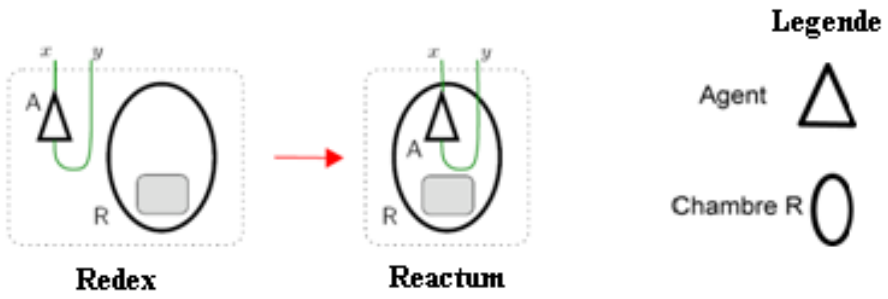


FIGURE 3.2 – Transformation sur les places.

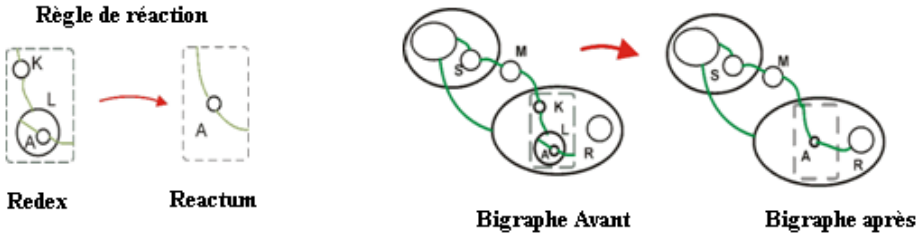


FIGURE 3.3 – Transformation sur les liens.

Spécification du Comportement

Ce modèle est dédié à la description d'applications à code. Il se concentre beaucoup plus sur la description de la structure et son évolution dynamique, mais ne propose aucune primitive pour raffiner cette description afin de spécifier le comportement des différents nœuds du graphe des places (les composants).

La sémantique associée à un BRS est définie par une catégorie mathématique (voir Définition 3.2.2) dont les objets sont des bigraphes et les morphismes de base sont des règles de réactions. Cette catégorie associe une interprétation sémantique à la dynamique de la structure (reconfiguration dynamique) du système considéré en termes de mobilité d'entités logicielles.

Définition 3.2.2. Etant donnée une signature K, un BRS noté généralement $BIG(K, R)$, est défini par la catégorie $'BIG(K)$ de bigraphes, équipée d'un ensemble de règles de réactions R.

Le tableau ci-dessous résume les mécanismes utilisés par quelques langages et modèles, y compris ceux présentés dans ce chapitre, pour la spécification de l'évolution structurelle et comportementale d'une architecture logicielle dynamique.

Modèle	Evolution de la structure	Comportement
C2SADL	Opérations de Reconfiguration	Opérations avec gardes
CommUnity	Mise à jour de la valeur d'un attribut	Actions gardées
Con Moto	Opérations de Reconfiguration	Non considéré
MobiS	Opérations de Reconfiguration	Non considéré
LAM	Opérations de Reconfiguration	Non considéré
π-ADL	Règles de transition	Règles de transitions
ADR	Règles de Transformation de graphes	Non considéré
Bigraphes	Règles de réactions	Non considéré

TABLE 3.1 – Description de la dynamique des architectures logicielles.

Dans la suite du chapitre, nous étudierons l'aptitude de chacun de ces modèles et langages à décrire des architectures logicielles mobiles.

3.3 Evaluation des modèles de description d'architectures logicielles dynamiques et mobiles

Afin de pouvoir dresser un bilan d'évaluation des différents formalismes de description d'architectures logicielles dynamiques et mobiles, nous avons besoin de dégager un ensemble de besoins et contraintes liées à la prise en charge de la mobilité.

3.3.1 Besoins et contraintes de la mobilité

Un ensemble de critères d'évaluation des modèles de spécification de systèmes à code mobile ont été établis par Roman et al.(Roman et al., 2000). Ces critères sont aussi valables pour les modèles de description des architectures logicielles :

La localisation : représente les différentes positions qu'un composant mobile est capable d'atteindre durant son exécution. Elle doit être explicitement considérée comme élément de première classes et identifiable des autres entités du modèle.

Conscience de la localisation actuelle : détermine si un composant est conscient de sa localisation courante ou non. Cette caractéristique est importante car elle permet de prendre des décisions qui dépendent de la localisation actuelle.

L'unité de mobilité : Un choix typique est de faire coïncider l'unité de mobilité avec l'unité de d'exécution, la plupart des modèles partagent ce choix. Certains paradigmes de conception de systèmes à code mobile favorisent une mobilité plus fine allant jusqu'à une variable ou une instruction, notamment le travail présenté dans (Mascolo et al., 1999). Dans ce travail, l'unité d'exécution n'est plus mobile mais plutôt c'est son comportement qui est dynamiquement augmenté par un code étranger qui ne sera rattaché à l'unité que si une certaine condition est vérifiée. CommUnity aussi propose une granularité fine allant jusqu'à une variable ou une action. Cette perception n'a pas eu un véritable écho dans la communauté des modèles formels. A un niveau architectural, l'unité de mobilité est un composant. La granularité du composant mobile peut varier d'un composant primitif contenant une instruction ou une variable jusqu'à un composant hiérarchique correspondant à une sous-architecture. Il suffit de changer le niveau d'abstraction.

Support de la mobilité : décrit les mécanismes offerts par le modèle pour supporter le mouvement d'une entité. Essentiellement, le support de la mobilité détermine les

conséquences du mouvement d'une entité sur le système ; provoquer la reconfiguration de l'architecture du système ou bien une simple modification de la valeur d'un attribut.

Décision de migration : cette caractéristique spécifie quand et qui provoque le mouvement de l'entité. S'il s'agit d'une décision prise par l'environnement, on parle de mobilité objective. Pour une décision prise par l'entité elle même, on parle de mobilité subjective.

Coordination : Les mécanismes de coordination supportés par un modèle déterminent le contexte que l'entité mobile est capable de percevoir. Ils doivent être spécifiés indépendamment de l'aspect fonctionnel de l'entité. Du fait que l'une des caractéristiques principales des ADL est la séparation entre la coordination et le calcul, ils supportent tous cette propriété. Ainsi, ce qui différencie les ADL du point de vue coordination est l'éventuel support de la notion de connecteur.

Formalisme : les modèles proposés doivent être suffisamment formels pour permettre une description précise de la sémantique des propriétés de distribution et de mobilité. Le formalisme choisi doit fournir les primitives requises pour la spécification des caractéristiques de la mobilité.

3.3.2 Bilan d'evaluation

Comme les critères de coordination et de la base sémantique formelle sont vérifiés dans les différents langages et modèles présentés dans ce chapitre, ils ne seront pas considérés dans cette étude.

1. Unité de mobilité : chaque modèle a défini sa propre unité de mobilité . Comm-Unity propose une granularité fine en autorisant le mouvement d'une variable ou une action. Dans π-ADL et LAM, les sous-composants sont mobiles. Une région

entière peut être déplacée d'un site vers un autre dans les bigraphes.

2. **Prise en charge de la notion de localisation :** la localisation est définie soit explicitement ou implicitement dans les ADL et autres formalismes dédiés à la spécification d'architectures logicielles mobiles :

   Déclaration explicite : dans ce cas de figure, la localisation est définie soit par enrichissement explicite de la structure du composant ou la définition de composants ou connecteurs particuliers représentant l'emplacement physique du composant mobile. Darwin enrichie la structure du composant par une variable entière pour indiquer la machine effective qui l'héberge. CommUnity définit un nouveau type abstrait pour spécifier la localisation des designs.

   Déclaration implicite : une autre catégorie de modèles a opté pour une déclaration implicite de la notion de localisation à travers la définition de composants composites. L'emplacement actuel d'un composant mobile est défini par le composant hiérarchique qui l'héberge. π-ADL (Oquendo et al., 2004) ne fait pas de distinction entre un composite jouant le rôle de conteneur de composants mobiles et un composite ordinaire ou hiérarchique. Mobis (Ciancarini et al., 1998) définit des primitives, appelées space, pour représenter des composants imbriqués, des localisations et des canaux. Les composites sont appelés locations dans LAM (Xu et al., 2003), chacun d'eux comporte un connecteur interne pour attacher les sous-composants à la localisation physique.

   D'autres formalismes font abstraction de la notion de localisation et proposent des types particuliers de connecteurs pour gérer la dynamique de l'architecture logicielle. Des connecteurs transitoires ont été utilisé par plusieurs modèles (fiaderio et al., 1999, Wermelinger, 1998) pour établir des liens dynamiques entre composants nomades qui désirent interagir. Des connecteurs frontières ont été

définis dans C2SADL (Medvidovic et al, 2001) afin d'établir des liens entre les composants mobiles et les hôtes du réseau.

Le tableau ci-dessous résume les différentes formes de déclaration de la notion de localisation ainsi que l'unité de mobilité.

Formalisme	Representation	Unité de mobilité
C2SADL	Connecteur frontière	Composants
CommUnity	Valeur d'un type abstrait de données	Composants et connecteurs
Con Moto	Composant physique	Composants logiques
Mobis	Composant composite	Composants, localisations, canaux
LAM	Composant composite	Composants
π-ADL	Composant composite	Composants
ADR	Graphes hiérarchiques	Des graphes
Bigraphes	Graphes des places	Des régions

TABLE 3.2 – Représentation de la localisation et de l'unité de mobilité.

3. Support de la mobilité

Les mécanismes (voir table 3.1 ci-dessous) proposés par les modèles de spécification d'architectures logicielles pour le support de la mobilité, varient d'une simple mise à jour de la valeur d'un attribut du composant logiciel jusqu'à la reconfiguration entière de l'architecture logicielle.

- Pour les langages ayant enrichi la structure du composant par un attribut, la mobilité consiste à mettre à jour la valeur de l'attribut.
- Pour les langages et modèles ayant défini des structures hiérarchiques, la mobilité constitue une reconfiguration de l'architecture logicielle, soit par modification de la structure du composant composite ou bien par transformation de graphes hiérarchiques.

3.4 Conclusion

La mobilité d'unités de calcul constitue un facteur additionnel de complexité dans les systèmes actuels. Elle nécessite l'émergence de la structure globale dès les premières phases du cycle de développent. Les architectures logicielles et leurs modèles de description ont le potentiel de prendre en charge cette propriété non fonctionnelle. Et de ce fait, plusieurs travaux ont été menés pour d'une part enrichir les ADL existants par la propriété de mobilité, et d'autre par définir de nouveaux modèles à base de graphes et leur règles de transformation pour la description des architectures logicielles dynamiques en générales. Néanmoins, l'utilisation de la plupart de ces modèles se restreint à la phase d'analyse du fait que le composant est considéré comme une boite noire. La spécification des propriétés fonctionnelles aura alors besoin d'utiliser d'autres formalismes avec tous les problèmes que cela peut engendrer, le problème majeur étant les éventuels effets de bord de l'evolution fonctionnelle sur la reconfiguration et vice-versa. Nous proposons dans les chapitres suivants, une approche de construction d'applications à code mobile centrée sur les interfaces, offrant les mécanismes nécessaires pour la spécification de la mobilité de composants et aussi leurs comportements.

Chapitre 4

La logique des tuiles comme cadre sémantique

4.1 Introduction

Ces dernières décennies ont été caractérisées par une large diffusion des systèmes de calcul à base de règles, particulièrement ceux supportant les propriétés de distribution et de concurrence. La sémantique opérationnelle de ces systèmes se base sur deux points essentiels une structure d'états et des règles de réduction. Afin de formaliser les étapes de calcul d'un système donné, la définition d'un univers d'états ou configurations globales est nécessaire. Chacun de ces états représente une étape intermédiaire dans l'évolution du système. Les règles de leur côté dénotent les évolutions locales possibles entre états du système. Dans la plupart des cas, certaines de ces règles peuvent s'exécuter simultanément, de manière indépendante ou bien interactive (synchrone). Un mécanisme d'identification (matching) est alors requis afin d'identifier l'occurrence d'une certaine configuration locale dans la configuration globale correspondant ; au membre gauche de l'une des règles. Cette identification permet de déterminer les règles potentiellement applicables à une configuration donnée et est comparable à la sémantique de sensibilisation dans les réseaux de Petri.

L'expansion des systèmes à base de règles a donné naissance à de nombreuses propositions de méthodologies ou de métaformalismes offrant un cadre de travail flexible pour la spécification de leur sémantique opérationnelle. L'idée de base de ces metaformalismes est d'offrir des directives générales pour spécifier de manière inductive la classe de toutes les étapes d'évolutions possibles d'un système, tout en faisant abstraction des détails inutiles et particuliers à ce système. Cela requiert généralement un système de déduction, construit de manière générique afin de différer le besoin de reconnaître les éventuels états du système considéré jusqu'à la phase de calcul. Le système de déduction permet de construire le calcul concurrent du système. Plusieurs modèles à base de règles existent dans la littérature, λ-calcul, la logique de réécriture, etc.

En résumé, un modèle de spécification de systèmes concurrents à base de règles de déduction a besoin d'offrir les mécanismes nécessaires pour définir :

- la structure de l'état,
- les règles de réduction décrivant les évolutions locales possibles des états,
- et un système de déduction muni d'un mécanisme de substitution, responsable de l'identification des règles applicables à chaque étape du calcul.

4.2 Structure de l'état

Afin d'éviter la représentation arborescente ordinaire des termes modélisant les états de systèmes logiciels, les nœuds correspondent aux noms de variables et de constantes et les arcs représentent les opérations, plusieurs structures mathématiques ont été adoptées pour formaliser la structure de ces termes. Les travaux de Hotz (Hotz, 1995) propose les $\mathcal{X}$-catégories, Michael Pfender(Pfender, 1974) propose les catégories $\mathcal{S}$-monoidales, etc. Un élément commun à toutes ces structures est qu'elles sont des enrichissements des catégories monoidales, qui constituent une base pour la description d'environnements

distribués en termes de diagrammes boites noires (wire and box). Cet enrichissement catégoriel est usuellement lié à l'ajout de transformations qui peuvent être ou non naturelles. L'utilisation de structures catégorielles à la place des structures arborescentes offre aux systèmes considérés des propriétés structurelles permettant de raisonner à un niveau d'abstraction indépendant des détails d'implementation propres aux structures manipulées. Par ailleurs, l'utilisation de structures catégorielles à la place de termes algébriques facilite la propagation de la distribution de l'état sur la génération du calcul, c.à.d., le comportement global sera défini de manière inductive à partir des comportements des éléments de base.

Définition 4.2.1. Une signature $\mathcal{F}$ est un ensemble d'opérateurs (symboles de fonction), chacun étant associé à un entier naturel par la fonction *arité*: $\mathcal{F} \longrightarrow \mathbb{N}$, associant à chaque opérateur son *arité*. $\mathcal{F}_n$ désigne le sous-ensemble d'opérateurs d'arité n.

On peut alors définir les termes construits sur ces opérateurs et un ensemble $\mathcal{X}$ de variables par clôture comme suit :

Définition 4.2.2. Les termes sur $\mathcal{F}$ sont définis par clôture comme le plus petit ensemble $T(\mathcal{F}, \mathrm{X})$ tels que :

- $\mathcal{X} \subset \mathcal{T}(\mathcal{F}, \mathcal{X})$, toute variable de $\mathcal{X}$ est un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$
- Pour tous $t_1, ..., t_n$ des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ et pour tout opérateur f d'arité n, $f(t_1, \ldots t_n)$ est un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

L'ensemble des termes clos (les termes dans lesquels il n'y a pas de variables) est noté $\mathcal{T}(\mathcal{F})$

Définition 4.2.3. Une substitution est une fonction $\sigma \colon \mathcal{X} \longrightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ pouvant être étendue à un endomorphisme sur l'algèbre des termes, et est définie inductivement par :

- $\sigma'(x) = \sigma(x)$, pour toute variable $x \in \mathcal{X}$,
- $\sigma'(f(t_1, ..., t_n)) = (f(\sigma'(t_1), ..., \sigma'(t_n))$, pour tout opérateur f d'arité n.

4.3 Logique de réécriture

Les systèmes de réécriture sont l'un des modèles de spécification de systèmes concurrents à base de règles. L'idée centrale de la réécriture est d'imposer une direction dans l'utilisation des axiomes, par la définition de règles de réécriture. Contrairement aux équations, qui ne sont pas orientées, une règle sur un ensemble de termes $\mathcal{T}$ est une paire ordonnée de termes $\langle l, r \rangle$, que l'on note $l \longrightarrow r$. La réécriture, ou le processus de remplacement d'une instance du membre gauche d'une règle par l'instance correspondante du membre droit de cette règle, est un paradigme de calcul intrinsèquement concurrent (Lincoln et al., 1994). En effet, l'application d'une règle de réécriture dépend uniquement de l'existence locale du terme gauche de la règle. Ainsi, une même ou plusieurs règles peuvent être appliquées simultanément à différentes positions. En particulier, les règles de réécriture ont été utilisées pour exprimer de manière déclarative le parallélisme implicite de programmes fonctionnels.

La logique de réécriture (Meseguer, 1992) est l'un des modèles basés sur les systèmes de réécriture, proposée par Meseguer comme outil fondamental pour la spécification de systèmes concurrents. La dynamique d'un système est exprimée à l'aide de règles de réécriture qui agissent sur son état, lui-même défini dans une logique équationnelle sous-jacente à la logique de réécriture. De nombreux auteurs ont mis en évidence l'adéquation de la logique de réécriture pour modéliser de nombreux types de systèmes dynamiques, comme par exemple les réseaux actifs (Meseguer et al., 2002), les systèmes biologiques (Eker et al., 2002), ou la sémantique des langages de programmation (Meseguer et al., 2007). Plusieurs systèmes implémentent des versions de cette logique : Maude (Clavel et al., 2007), Elan (Borovanský et al., 1998), et CAFE OBJ (Diaconescu et al., 2002).

4.3.1 Format des règles de réécriture

Les axiomes de base dans la logique de réécriture sont les règles de réécriture. Deux lectures complémentaires sont valables pour une règle de réécriture de la forme $t \longrightarrow t'$; t et t' étant des termes algébriques :

- D'un point de vue calcul, la règle de réécriture est interprétée comme une transition locale dans un système concurrent ; c.a.d., t et t' décrivent des patrons de sous-états pouvant apparaître dans l'état distribué d'un système concurrent. La règle sert à exprimer une transition locale pouvant avoir lieu dans un tel système et provoquant le changement d'un fragment local de l'état de l'instance du patron t vers l'instance correspondante du patron t'.
- D'un point de vue logique, la règle de réécriture $t \longrightarrow t'$ est interprétée comme une règle d'inférence permettant d'inférer des formules de la forme t' à partir de formules de la forme t.

4.3.2 Description de systèmes concurrents dans la logique de réécriture

La spécification d'un système concurrent requiert trois concepts de base : la définition de la structure des états, un ensemble de transitions locales, et un mécanisme de calcul. Pour qu'un système soit concurrent et permette une évolution parallèle ou coordonnée de fragments de l'état actuel, l'existence d'une structure algébrique générant des états distribués est nécessaire. Cette structure permet de capturer deux aspects importants :

- D'une part, l'état du système spécifié sera distribué selon une telle structure.
- D'autre part, les transitions sont elles aussi distribuées selon la même structure algébrique.

Par conséquent, ces transitions sont indépendantes les unes des autres et peuvent apparaître en concurrence.

Un système concurrent est décrit par une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$. La signature (Σ, E) d'une théorie de réécriture décrit la structure algébrique particulière des états du système considéré. Ces derniers sont distribués selon cette structure. Les règles de réécriture R définissent les transitions élémentaires et locales possibles applicables à l'état actuel du système concurrent. Chaque règle de réécriture correspond à une action pouvant survenir en concurrence avec d'autres actions.

Remarque 4.3.1. Les connecteurs logiques ou symboles de fonctions Σ et leurs propriétés structurelles E sont entièrement définis par l'utilisateur (Martio-Oliet et al., 1993). Ceci procure à la logique de réécriture une grande flexibilité dans la définition de la structure des états et offre ainsi la possibilité de représenter une variété de structures.

Définition 4.3.1. (Meseguer, 1992)

Une théorie de réécriture est un quadruplet $\mathcal{R} = (\Sigma, E, L, R)$ tels que :

- Σ est un ensemble de symboles de fonctions,
- E est un ensemble de Σ-équations,
- L est un ensemble d'étiquettes,
- R est un ensemble de paires $R \subseteq (L \times (T_{\Sigma, E}(X)^2)^+$, le premier composant est une étiquette et le second est une paire de classes d'équivalence de termes modulo les équations E avec X un ensemble infini et dénombrable de variables. Les éléments de R sont appelés règles de réécriture.

Une règle de réécriture de la forme $(r, ([t], [t']), ([u_1], [v_1]), ..., ([u_k], [v_k])$ est notée par :

$$r : [t] \longrightarrow [t'] \text{ if } [u_1] \longrightarrow [v_1] \wedge ... \wedge [u_k] \longrightarrow [v_k]$$

La partie $[u_1] \longrightarrow [v_1] \wedge ... \wedge [u_k] \longrightarrow [v_k]$ est appelée condition de la règle et peut être abrégée par la lettre C.

Une règle de réécriture peut être paramétrée par un ensemble de variables $\{x_1, ..., x_n\}$ qui apparaissent dans t, t' ou C, et nous écrivons :

$$r : [t(x_1, ..., x_n)] \longrightarrow t'[(x_1, ..., x_n)] \ \textit{if} \ C(x_1, ..., x_n)$$

Cette notion de théorie de réécriture est très générique et expressive. Elle permet une réécriture modulo des axiomes structurels E. Elle permet aussi de définir des règles conditionnelles génériques. Les conditions de celles-ci ne nécessitent pas une vérification des égalités, mais seulement l'existence de réécritures à travers des paires de termes de la condition.

4.3.3 Sémantique opérationnelle d'un système concurrent

Une théorie de réécriture est une description statique de ce qu'un système concurrent peut faire. La génération de phrases de la forme $t \longrightarrow t'$ par le langage associé à cette théorie constitue des calculs concurrents et correspond à des séquences de réécriture.
Étant donné une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$, le séquent $t \longrightarrow t'$ est prouvable dans $\mathcal{R}$ et on écrit $\mathcal{R} \vdash [t] \longrightarrow [t']$ ssi $[t] \longrightarrow [t']$ est obtenue par application finie des règles de déduction de la figure 4.1.
L'ensemble de séquents calculables à partir d'un état initial est obtenu par inférence sur les axiomes de base R. D'un point de vue calcul, ces séquents décrivent toutes les transitions concurrentes et complexes du système axiomatisé par $\mathcal{R}$ et d'un point de vue logique, ils décrivent toutes les déductions possibles aussi complexes qu'elles soient dans la logique axiomatisée par $\mathcal{R}$.
De manière générale, la déduction ou le calcul dans la logique de réécriture est une itération des étapes suivantes :

1. La règle de remplacement identifie toutes les règles de réécriture dont le membre gauche correspond à un sous-terme de l'état global courant. Les sous-termes non

1. Reflexivité : pour chaque $[t] \in T_{\Sigma(X)}$,

$$\frac{}{[t] \longrightarrow [t']}$$

2. Congruence : pour tout $f \in \Sigma_n$, $n \in \mathbb{N}$

$$\frac{[t_1] \longrightarrow [t'_1], ..., [t_n] \longrightarrow [t'_n]}{[f(t_1, ..., t_n)] \longrightarrow f[(t'_1, ..., t'_n)]}$$

3. Remplacement : pour chaque règle de réécriture

$$r : [t(x_1, ..., x_n)] \longrightarrow t'[(x_1, ..., x_n)] \; if u_1(\overline{x}) \longrightarrow v_1(\overline{x}) \wedge ... \wedge u_n(\overline{x}) \longrightarrow v_n(\overline{x}) \, \text{dans} \, R$$

$$\frac{[w_1] \longrightarrow [w'_1], ..., [w_n] \longrightarrow [w'_n]}{[[u_1(\overline{w_1}/\overline{x})], ..., [u_n(\overline{w_n}/\overline{x})]}$$
$$\overline{[[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w}/\overline{x})]]}$$

4. Transitivité :

$$\frac{[t_1] \longrightarrow [t_2] \;\; [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

FIGURE 4.1 – Règles de déduction de la logique de réécriture.

identifiés évoluent en eux même par application de la règle de réflexivité.

2. La règle de congruence permet de composer les effets locaux des différentes règles de base ; identifiées par la règle de remplacement, pour construire le nouvel état du système. Elle exprime l'évolution concurrente des différents sous-termes d'un état global.

   Ces étapes sont réitérées jusqu'à ce qu'il n'y ait plus de règle applicable.

3. Enfin, la règles de transitivité construit la séquence de réécritures faites en partant du terme initial jusqu'au terme final.

4.3.4 Modèle sémantique d'une théorie de réécriture

En plus de la sémantique opérationnelle associée à une théorie de réécriture et inférée par le système de déduction, la logique de réécriture définit aussi le modèle sémantique associé à cette théorie. L'idée consiste à décorer les séquents prouvables par les termes de preuves utilisés lors de la déduction.

Une sémantique concurrente est nécessaire pour mettre l'accent sur l'exploitation du parallélisme implicite dans la définition des systèmes de réécriture (Corradini et al, 1995). Généralement, cette sémantique peut être définie en imposant une relation d'équivalence entre les termes de preuves qui sont les mêmes par rapport à une exécution massivement parallèle, c.à.d., regrouper les différentes exécutions séquentielles possibles d'actions parallèles dans une même classe d'équivalence. Chaque classe d'équivalence représente un calcul abstrait correspondant à l'exécution du même ensemble de transitions causalement indépendantes.

Le modèle associé à une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$ est une 2-catégorie $\mathcal{T}_R(X)$ (voir Annexe A). Les objets de $\mathcal{T}_R(X)$ sont des classes d'équivalence de Σ-termes $[t] \in T_{\Sigma, E}(X)$ modulo les équations E, et les arcs sont des classes d'équivalence de termes de preuves. Les arcs identités correspondent naturellement aux termes de preuve générés par la règle de réflexivité et les arcs compositions sont générés par la règle de transitivité.

Les termes de preuves sont aussi dotés d'une structure algébrique identique à celle définie sur les objets de $\mathcal{T}_R(X)$. Cette structure est obtenue grâce à la règle de congruence de la logique de réécriture.

La logique de réécriture a été proposée comme une logique de changement dédiée à la spécification formelle de systèmes concurrents. Néanmoins, elle n'a pas de vision modulaire dans la spécification de ces systèmes, alors que la tendance actuelle pour maîtriser

la complexité des systèmes à spécifier est de les considérer comme un assemblage de sous-systèmes en interaction. Cette contrainte est due principalement au format ordinaire des règles de réécriture qui manque de moyens pour percevoir les interactions avec l'environnement ; propriété inhérente aux systèmes réactifs et modulaires. De ce fait, les règles de réécriture peuvent être librement instanciées par n'importe quel terme et dans n'importe quel contexte en faisant abstraction des informations contextuelles qui généralement servent à guider le calcul ou la déduction dans les systèmes réactifs. Par ailleurs, la vision plate de la structure de l'état a une répercussion directe sur le comportement global du système qui nécessite la manipulation de tout l'état global à chaque étape du calcul.

Pour surmonter ce problème, deux extensions majeures de la logique de réécriture ont été réalisées : la logique des tuiles (Bruni, 1999) et la logique de réécriture révisée (Bruni, 2006) qui a introduit les variables gelés et les théories de réécriture généralisées.

4.4 Logique des tuiles

Afin d'introduire les interactions dans le processus de réécriture, la logique des tuiles a étendue le format des règles de réécriture en enregistrant dans les étiquettes des transitions non seulement les effets de chaque changement, mais aussi les déclencheurs fournis par les sous-composants pour l'application de la transition à l'état global. Ces déclencheurs décrivent les conditions générales d'application de la règle. Elles correspondent aux stimulations contextuelles, formalisées par des interactions, nécessaires pour provoquer une réaction de la part d'un sous-composant du système. La conséquence directe qui découle de cette extension est la spécification modulaire de la structure d'un système concurrent et surtout la définition inductive du comportement global par composition des comportements locaux possibles de ses sous-composants.

Un des avantages majeurs de la sémantique compositionnelle est la possibilité de remplacer chaque sous-composant d'un état par un autre qui lui est équivalent sans affecter ni altérer le comportement global. Cela facile considérablement la reconfiguration dynamique de systèmes distribués.

4.4.1 Format des règles dans la logique des tuiles

L'objectif visé par la proposition du modèle de tuile est la spécification de systèmes ouverts et modulaires. Contrairement aux systèmes clos, les systèmes ouverts sont partiellement spécifiés ; c.à.d., contenant des variables, afin de leur procurer la capacité de s'exécuter dans différents contextes en agissant sur les valeurs de substitution des variables. Deux concepts sont alors nécessaires pour adapter les règles de réécriture à la spécification de systèmes ouverts :

- introduire la notion de contexte dans la phase de spécification pour obtenir des règles génériques applicables dans tout contexte,
- introduire la notion d'instanciation dans la phase de déduction ou calcul pour permettre l'application des règles à un contexte donné.

Notion de contexte

La capture de ces informations contextuelles est généralement réalisée par un mécanisme d'interaction entre les composants du système considéré. Comme le modèle des tuiles se base sur les systèmes de réduction pour la description de systèmes ouverts, le format des règles a besoin d'être enrichi par de nouveaux concepts permettant d'associer une sémantique interactive aux systèmes de réduction. Pour atteindre cet objectif, le modèle des tuiles a réalisé une mixture harmonieuse des idées proposées par les systèmes de réduction et les systèmes de transitions étiquetés (LTS). La première approche se base

sur l'observation et la manipulation de l'état global d'un système complexe. En particulier, l'état global courant peut être inspecté afin d'identifier un redex, un membre gauche d'une règle, qui sera un candidat potentiel pour l'application d'une étape de réduction. Généralement, les redexes servent à coordonner l'activité de plusieurs composants du système, et de ce fait le rôle d'une étape de réduction est justement de synchroniser leurs activités locales en une transition atomique de l'état global courant du système vers un nouvel état global. En d'autres termes, les systèmes de réduction nécessitent un aplatissement de la structure du système pour pouvoir spécifier le comportement global. Mais d'un point de vue composition, il serait plus judicieux de pouvoir dériver la sémantique opérationnelle de l'entité globale en termes de la sémantique de ses composants les plus basiques. Chose qui peut être une tâche difficile, puisque les réductions sont généralement des actions non locales nécessitant des interactions avec l'environnement. Le problème est dû au fait que la plupart des redexes dépendent non seulement du composant mais de son environnement externe aussi. Ainsi, pour pouvoir comprendre le comportement du composant à part, il est nécessaire de considérer ses interactions avec tous les environnements possibles. Et là interviennent les LTS et leur sémantique observationnelle. L'idée est d'utiliser des observations (les étiquettes des transitions) pour déduire les équivalences observationnelles sur les processus. Ces observations représentent les interactions. Par ailleurs, le format de spécification de la sémantique opérationnelle des LTS permet d'exploiter la structure complexe d'un état afin de garantir les propriétés de composition des comportements locaux pour construire le comportement global de manière inductive. Une idée émergente, pour définir une sémantique de réduction avec une vue interactive et observationnelle, consiste à utiliser les contextes d'observation (observing contexts) (Milner, 1992 ; Montanari et al., 2000). Le principe de base consiste à commencer à partir d'un système de réduction et d'essayer de définir la sémantique d'un composant local en l'intégrant dans tous les contextes possibles ; représentés par

des observations. Il est alors possible de prédire la sémantique du système résultat par inspection du comportement de chaque composant dans l'environnement constitué de tous les autres composants. Néanmoins, plusieurs questions opérationnelles restent ouvertes dans cette solution puisque la sémantique doit prendre en considération tous les contextes possibles. Plusieurs tentatives ont été menées pour définir une méthodologie générale de passage d'une sémantique de réduction vers une sémantique des LTS compositionnels (Sewell, 1998 ; Leifer et al., 2000 ; Cattani et al., 2000) afin d'introduire la notion de *contextualisation* dans le processus de réduction.

Instantiation dans les systèmes interactifs

Le problème dual à la contextualisation est le problème d'***instanciation***. Il apparaît lors de l'extension de la propriété de composition de processus clos(ground) vers des processus ouverts. Ainsi, l'équivalence sur des termes ouverts est usuellement définie via l'équivalence sur des termes clos, en considérant que deux contextes sont équivalents si leurs clôtures sur toutes les instanciations possibles sont équivalentes. Cependant, il est préférable d'éviter la clôture sur les instanciations et d'opter pour une approche plus compacte afin de forcer la modularité du cadre de travail. Cette approche a été principalement adopté dans les travaux de (Rensink, 2000) et (Bruni et al., 2000) pour fournir des formats de spécification généraux, garantissant la propriété de composition des systèmes ouverts. Ces travaux se basent sur l'idée d'enregistrer dans les étiquettes des transitions non seulement les effets de chaque changement, mais aussi les déclencheurs (contraintes) nécessaires pour l'application de la transition.

Modèle de tuiles

La logique des tuiles impose des contraintes dynamiques sur les termes auxquels une règle de réécriture peut être appliquée en la décorant par des observations qui assurent les synchronisations et décrivent les interactions entre composants d'un système concurrent. La règle de réécriture résultat est appelée tuile. Pour garantir une réécriture sensible au contexte, les règles de réécriture ont été enrichies par le concept d'interfaces. Les interfaces d'entrées expriment les conditions contextuelles déclenchant une règle et les interfaces de sortie définissent les effets de bords de la règle sur son environnement. De cette manière, le processus de réécriture est guidé par le système d'interaction avec son environnement, réduisant considérablement l'espace d'état de chaque étape de calcul.

Le format général d'une règle de réécriture est $\alpha : s \frac{a}{b} t$. Elle signifie que la configuration initiale s évolue vers une configuration finale t via la tuile α, produisant un effet b observable par le reste du système. Cette étape de réécriture n'est possible que si les sous-composants de s, c-à-d., les arguments auxquels s est connecté via son interface d'entrée, évoluent aux sous-composants de t en produisant un effet a qui agit comme déclencheur pour la tuile α. Une notation graphique est aussi possible (voir figure 4.2). La représentation graphique utilisée souvent dans la théorie des catégories montre et

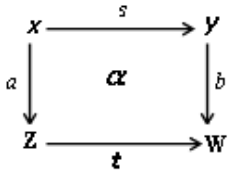


FIGURE 4.2 – Représentation graphique d'une tuile.

met en évidence les différents éléments constituant la tuile en termes d'objets (interfaces) et de morphismes (configurations et observations) et permet de voir le résultat double de la réécriture : nouvelle configuration et effet. Cette représentation permet aussi de bien comprendre les différentes opérations de composition des tuiles. Un effet de bord intéressant de la prise en considération des interactions dans le processus de déduction est la spécification modulaire et compositionnelle des systèmes considérés. La propriété de composition ne se restreint pas à la composition structurelle ; généralement présente dans tous les modèles actuels de la concurrence, mais s'étend à la construction du comportement global du système par composition inductive des comportements locaux possibles de ses composants.

4.4.2 Description d'un système concurrent dans la logique des tuiles

Étant donné que la logique des tuiles est mieux adaptée à la spécification de systèmes réactifs et ouverts, les interfaces constituent les objets de base manipulés dans cette logique. Différents schémas d'assemblage des interfaces peuvent être réalisés grâce aux opérateurs de composition parallèle et horizontale proposés par la logique des tuiles. Ces mêmes opérateurs expriment la distribution spatiale des états du système ; appelés dans ce cas configurations. Les informations contextuelles introduites via les interfaces correspondent à des invocations d'actions de base du système, qui elles aussi peuvent être composées horizontalement pour la synchronisation d'actions ou en parallèle pour construire des observations plus larges. Le comportement du système est défini par un ensemble de tuiles exprimant les transitions locales possibles de composants partiellement spécifiés. Le calcul dans la logique des tuiles correspond à une évolution coordonnée des configurations locales provoquée par des interactions avec l'environnement interne

et externe des composants. Formellement, un système concurrent est spécifié par un système de tuiles comme suit :

Définition 4.4.1. Un système de tuiles est un quadruplet $\mathcal{T} = (H, V, N, R)$ où H et$V$ sont des catégories monoïdales construites à partir du même ensemble d'objets (appelés interfaces) $O_H = O_V$, N étant un ensemble de noms de tuiles et $R\colon N \longrightarrow H \times V \times V \times H$ est une fonction associant à chaque $\alpha \in N$, un quadruplet (s, a, b, t) tels que $s\colon x \longrightarrow y$, $a\colon x \longrightarrow z$, $b\colon y \longrightarrow w$ et $t\colon z \longrightarrow w$ pour des objets appropriés x, y, z et w ; avec x et z les interfaces d'entrée et y et w les interfaces de sorties.

Les morphismes de la catégorie horizontale et ceux de la catégorie verticale sont appelées respectivement configurations et observations. Des structures algébriques peuvent être définies sur les configurations et les observations dans la logique des tuiles. En variant la structure des configurations et des observations, les tuiles peuvent modéliser différents aspects des systèmes dynamiques, allant de la synchronisation des réseaux de transitions jusqu'aux dépendances causales des calculs et les systèmes d'acteurs.

Les doubles catégories monoïdales constituent un cadre sémantique naturel pour les systèmes de tuiles du fait que les structures mathématiques décrivant les états du système et les actions qui les synchronisent, appelées configurations et effets respectivement, sont des catégories monoïdales ayant le même ensemble d'objets, les interfaces.

Ce qui différencie la logique des tuiles de la logique de réécriture est que chaque règle vise à décrire le comportement possible d'un système ouvert ; une sorte de processus modulaires dont l'évolution est dynamiquement dépendante de la synchronisation de ses sous-composants.

4.4.3 Sémantique opérationnelle d'un système de tuiles

Comme c'est la cas pour la logique de réécriture, le calcul correspond à une déduction dans la logique des tuiles. A partir de l'ensemble de règles R défini dans le système de tuiles T, des séquents plus larges peuvent être générés par application de l'ensemble de règles de déduction de la figure 4.3.

La première règle de déduction considère les règles R d'un système de tuiles comme ses séquents de base. Les règles de base ainsi que les arcs identité constituent les générateurs de séquents plus larges. Intuitivement :

- Un arc identité verticale signifie qu'un élément de la configuration peut rester inactif en évoluant vers lui-même durant une étape de réécriture, pas d'effet ni de déclencheur.
- Un arc identité horizontale permet de propager l'effet d'une étape de calcul à une autre à travers une substitution identité.
- La construction d'étapes de calcul plus larges est réalisée via les règles de composition qui montrent la manière de combiner, séquentiellement, en parallèle ou bien par imbrication (emboîtement) des tuiles de base.

Règles générant les tuiles de base, les identités horizontale et verticale :

$$\frac{R(\alpha)=\langle s,a,b,t\rangle}{\alpha\colon s\frac{a}{b}t} \qquad \frac{a\colon x\to z\in V}{id_H\colon id_x\frac{a}{a}id_x} \qquad \frac{t\colon x\to y\in H}{id_V\colon t\frac{id_x}{id_x}t}$$

Compositions horizontale, verticale et parallèle :

$$\frac{\alpha\colon s\frac{a}{b}t\ \ \beta\colon h\frac{b}{c}f}{\alpha*\beta\colon s;h\frac{a}{c}t;f} \qquad \frac{\alpha\colon s\frac{a}{b}t\ \ \beta\colon t\frac{c}{d}h}{\alpha;\beta\colon s\frac{a;c}{b;d}h} \qquad \frac{\alpha\colon s\frac{a}{b}t\ \ \beta\colon s\frac{c}{d}t}{\alpha\otimes\beta\colon s\frac{a\otimes c}{b\otimes d}t}$$

FIGURE 4.3 – Règles de déduction de la logique des tuiles.

Le modèle de tuiles définit une composition spatiale et temporelle (voir figure 4.4) des règles afin de construire le comportement global à partir des comportements locaux des

sous-composants et ce de manière inductive par différentes formes de composition des tuiles de base et d'identité :

La composition parallèle correspond à la construction d'étapes de calcul concurrentes, dans lesquelles au moins deux configurations disjointes évoluent en concurrence. L'état obtenu d'une étape concurrente correspond à la composition parallèle des différents éléments de cette étape.

La composition horizontale se base sur le partage d'actions pour assurer la coordination de l'évolution des composants. Elle correspond à une réécriture synchrone, l'effet de la première tuile fournit le déclencheur de la seconde tuile. La tuile résultat exprime le comportement synchrone des deux tuiles.

La composition verticale modélise l'exécution d'une séquence d'étapes déduite à partir d'une configuration initiale. Elle correspond à la composition séquentielle de calculs.

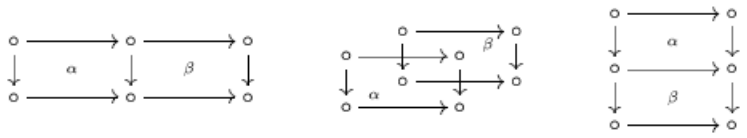


FIGURE 4.4 – Composition horizontale, parallèle et verticale des tuiles.

La composition spatiale permet de définir le comportement des états structurés (complexes) comme une évolution coordonnée (composition horizontale) ou parallèle des activités de leurs sous-composants, tandis que la composition verticale (dans le temps) offre un cadre puissant pour modéliser la composition des calculs.

Comme c'est le cas pour les théories de réécriture, une correspondance peut être établit entre les déductions et les calculs concurrents dans les systèmes de tuile, en décorant

chaque séquent par un terme de preuve qui encode les causes du calcul. D'autre part, des déductions (computationally) "calculablement" équivalents peuvent être identifiés via des ensembles d'axiomes sur les termes de preuve.

Exemple 4.1 Système d'envoi de message asynchrone

Cet exemple, inspiré de (Bruni et al., 2002), spécifie un système de communication par envoi de message asynchrone par un système de tuiles $\mathcal{R}_{AC} = (H_{AC}, V_{AC}, N_{AC}, R_{AC})$. Ce système a un seul type d'interfaces de connexion avec son environnement qui est le canal de communication.

Les configurations de base décrivent les différents états de ce système :

- communication initialisée par génération d'un port vide ; $proc \colon \lambda \longrightarrow \circ$,
- existence d'un message sur le port ; $msg_a \colon \lambda \longrightarrow a$,
- message envoyé sur le canal a par le biais du port ; $send \colon \circ \longrightarrow a$,
- message reçu sur le canal ; $receive \colon \circ \longrightarrow a$,
- identification du sens de communication (in/out) par duplication du port existant.

Les actions possibles sur ce canal sont :

- envoyer un signal sur le canal pour annoncer l'intention d'envoi d'une donnée ; $signal \colon \circ \longrightarrow \circ$,
- poster un message sur le canal ; $create_a \colon a \longrightarrow a \otimes a$,
- retirer un message du canal ; $consume \colon a \otimes a \longrightarrow a$.

Le comportement est décrit par les règles suivantes :

$$comm \colon proc \xrightarrow[signal]{} proc$$

$$put_a \colon send_a \xrightarrow[create_a]{signal_a} msg_a \otimes send_a$$

$$put_a \colon receive_a \otimes msg_a \xrightarrow[consume_a]{signal_a} receive_a$$

4.4.4 Modèle sémantique d'un système de tuiles

Le modèle sémantique associé à un système de tuiles est une double catégorie (voir Annexe A) :

- La catégorie horizontale associe une interprétation sémantique à la structure des états d'un système concurrent, les objets de base étant les interfaces. Les morphismes de base de cette catégorie servent à montrer les liens qui existent entre les interfaces d'entrée et de sortie. Chaque morphisme montre les interfaces de sortie qui seront affectées par l'évolution d'une ou plusieurs interfaces d'entrée. Des morphismes plus complexes peuvent être générés donnant une interprétation sémantique aux configurations ou la structure des états possibles du système concurrent.
- La catégorie verticale associe une interprétation sémantique aux actions observables du système concurrent, les objets de base étant toujours les interfaces. Chaque morphisme de base dans cette catégorie définit une évolution possible de la valeur d'une interface, c.à.d., une action possible sur l'interface. Des actions plus complexes sont générés à l'aide de deux types de composition : la composition parallèle définit les tuples d'actions indépendantes pouvant s'exécuter en parallèle. La composition verticale montre les séquences d'actions possibles.
- Les transformations naturelles définies par les tuiles de base montrent les actions autorisées sur chaque configuration. Les termes de preuves générés par la déduction dans la logique des tuiles correspondent aux calculs possibles dans le système concurrent.

4.5 Conclusion

Ce chapitre a été consacré à la présentation du cadre sémantique formel adopté pour la définition d'un modèle de spécification des architectures logicielles mobiles. La notion d'interface, définie dans la logique des tuiles, contribuera considérablement dans la définition de ce modèle. Les différentes facettes d'une application ; structure, comportement et dynamique de la structure, seront décrites dans un cadre sémantique commun sur la base des interfaces qui joueront aussi le rôle de moyen d'interaction entre la dynamique comportementale et structurelle. En effet, les effets de bords de l'évolution du comportement sur la structure et vice-versa seront perceptibles à travers les interfaces. Le détail du modèle proposé sera présenté dans les chapitres suivants.

Chapitre 5

Enrichissement d'un ADL par des primitives de mobilité

5.1 Introduction

Une application à code mobile est caractérisée par l'aptitude de ses entités logicielles à migrer d'un endroit vers un autre à travers un réseau d'ordinateurs pour diverses raisons ; la recherche de données, de ressources, etc. Cette migration provoque une reconfiguration dynamique des liens entre les entités logicielles et leurs localisations physiques. Néanmoins, il est toujours nécessaire d'assurer le maintien et l'acheminement des interactions entre les entités mobiles et les autres entités logicielles en permettant l'identification de leurs emplacements actuels à tout moment. A partir des ces exigences et d'un point de vue méthodologique, tout modèle de conception d'applications à code mobile doit supporter la spécification des trois concepts de base (voir chapitre 1) caractérisant ce type d'applications : l'unité de mobilité, la mise en évidence de la coordination et la notion de localisation.

Étant donné que l'objectif du présent travail est la conception d'un ADL basé sur la logique de réécriture pour la spécification d'applications à code mobile, une étape incontournable consistera à vérifier l'aptitude de ce cadre sémantique à supporter les concepts

cités.

- Concernant l'unité de mobilité, il est possible de considerer une mobilité à gros grains en faisant correspondre l'unité de conception, la théorie de réécriture objet ou le module orienté objet Maude, de la logique de réécriture à l'entité mobile.
- Deux grandes approches émergent pour la modélisation de la localisation. La représentation explicite de cette notion signifie que la structure de l'unité mobile sera enrichie par la déclaration explicite d'un attribut identifiant sa localisation actuelle. Dans ce cas de figure, les primitives de gestion de la mobilité se restreignent à une simple mise à jour de la valeur de cet attribut. La deuxième approche se base sur la notion de conteneur responsable de l'hébergement de l'entité mobile à travers la définition d'une structure hiérarchique dans laquelle les composites représentent une localisation pour les sous-composants. Dans ce cas, la mobilité d'une unité remet en cause la composition et l'assemblage du composite, provoquant une reconfiguration de ce dernier. La première approche est directe et évidente, reste à vérifier l'aptitude de la logique de réécriture à définir des structures hiérarchiques.
- Du fait que la logique de réécriture se base sur le partage de variables pour assurer la coordination entre les éléments d'un système concurrent, l'état global a une structure plate. Cela peut constituer une contrainte pour une spécification modulaire et réactive, mettant en évidence les interactions entre les différents éléments du système.

Afin de confirmer ou informer ce qui vient d'être avancé, nous avons pensé à enrichir d'abord un ADL basé sur la logique de réécriture, CBabel en l'occurrence (Rademaker et al, 2005), par des primitives de mobilité (Bouanaka et al., 2008b).
Dans ce chapitre, nous introduisons la notion de localisation ainsi que les primitives nécessaires à sa mise à jour sur le langage CBabel.

5.2 Langage CBabel

Le langage de description d'architecture CBabel(**B**uilding **A**pplications **b**y **E**volution with **C**onnectors) est un élément du projet $R - RIO$ (Reflective-Reconfigurable Interconnectable Objects)(Braga et al., 2003). Ce projet intègre les principaux concepts de l'approche de programmation des architectures logicielles et la programmation méta-niveau. CBabel est un langage déclaratif dont la sémantique est définie dans la logique de réécriture.

En plus de la description des aspects fonctionnels des architectures logicielles par composition d'un ensemble de modules et de connecteurs via une topologie d'interconnexion, CBabel offre aussi les outils nécessaires pour décrire les propriétés non fonctionnelles telles que la qualité de service, la sécurité, etc, sous forme de contrats (Cerqueira et al, 2003).

5.2.1 Concepts de base

CBabel a une syntaxe simple. Seuls les concepts de base nécessaires à l'ajout de la propriété de mobilité et leur translation dans la logique de réécriture seront présentés.

- Le module : décrivant un composant ou connecteur, est défini dans CBabel par :

  Connector | module ⟨class⟩ [(parameter-list)] ⟨internal variables⟩ port-list [instances ;]

- Le port : un port dans CBabel définit une interface à travers laquelle un composant peut fournir un service à son environnement (in port) ou demande un service (out port). Il peut être défini par :

  in | out port ⟨return-type⟩ ⟨port-type⟩ (parameter-list) [instances] ;

- L'application : une application dans CBabel décrit une topologie particulière du

système considéré. Elle est constituée d'un ensemble d'instances de composants, d'instances de connecteurs et de liens entre eux.

– Les connexions : les assemblages entre composants et connecteurs sont définis grâce à la déclaration de liens dans le module application, en utilisant le mot clé *link*. Un lien relit un port in (respectivement out) d'un composant à un rôle out(respectivement in) d'un connecteur.

    link _.outport to _.inport

Ces différents concepts sont illustrés à travers l'exemple du Producteur/consommateur ci-dessous.

Exemple 5.1 Producteur/Consommateur dans CBabel

Cette architecture est composée de trois éléments qui sont : PRODUCER, CONSUMER, BUFFER. Le composant PRODUCER accède au BUFFER pour déposer un élément (item) qu'il vient de produire.

```
module PRODUCER {
    out port int (String item) put ;
}
```

Le CONSUMER accède au BUFFER pour consommer un item déposé.

```
module CONSUMER {
    in port String (void) get ;
}
```

La taille de la mémoire tampon est limitée, le Producer ne peut ajouter un nouvel item que si le buffer n'a pas atteint sa taille maximale. Le Consumer ne peut retirer un item à partir d'une mémoire tampon vide.

```
connector BUFFER{
    in port (int item) put ;
```

in port int (void) get ;

}

La topologie initiale de ce système est constituée d'une instance du producer, une instance du Consumer et une instance du Buffer.

module Producer-Consumer

instantiate BUFFER as buff ;

instantiate PRODUCER as Prod ;

instantiate CONSUMER as Cons ;

link prod.put to buff.put ;

link Con.get to buff.get ;

}

5.2.2 Sémantique de CBabel

La sémantique basée logique de réécriture des différentes constructions de CBabel a été définie dans (Rademaker et al., 2005) en associant au concept de module et connecteur des modules objets, aux ports des messages, aux interactions des paramètres de messages et à la topologie d'une architecture logicielle une configuration dans le sens orienté objet ; un ensemble d'instances d'objets et des messages.

Le composant

Un composant CBabel est défini par une théorie de réécriture ou un module objet dans le langage Maude, langage d'implementation de la logique de réécriture, contenant la déclaration d'une classe ***class***.

- Le nom de cette classe correspond au nom du composant à modéliser.
- Les attributs de la classe définissent les variables locales du composant ; constituant

sa structure interne.

– Une instance de la classe correspond à une instance du composant modélisé.

La théorie de réécriture associée à un composant est notée par (Σ, E , R) dont :

– La signature Σ décrit la structure des états du composant, c'est-à-dire la spécification de l'ensemble des sortes, des opérations, des messages utilisés pour décrire l'aspect statique d'un composant. Cette signature comporte aussi la déclaration d'une classe :

$C : Class \mid A$

où A est l'ensemble des attributs de la classe C. Σ comporte aussi la déclaration de l'opération d'instantiation de la classe :

$op\ instantiate : Oid\ Cid \rightarrow Object.$

– L'ensemble E contient, en plus des équations décrivant la sémantique des opérations définies dans le composant, une équation définissant la sémantique de l'opération d'instantiation :

$eq\ instantiate(o, C) =< o : C|a_1 : v_1, ..., a_n : v_n > .$

où o est l'identificateur de l'objet, C est l'identificateur de la classe, les a_i représentent les identificateurs des attributs de l'objet, les v_i représentent leurs valeurs initiales. Cette équation d'instantiation permet de créer une instance o d'un composant C avec des attributs initialisés à $v_1, v_2, \ldots, v_n$.

– Enfin, R représente l'ensemble des règles de réécriture spécifiant le comportement d'un composant architectural.

Le connecteur

Comme le connecteur est considéré dans la plupart des ADL comme un composant particulier servant à implémenter les protocoles de communication entre composants, son

comportement se restreint à recevoir des messages sur les rôles d'entrée et les faire parvenir aux composants destinataires sur l'un des rôles de sortie. Par ailleurs, il peut réagir aux messages selon les contrats de coordination définis. Par conséquent, un connecteur est également modélisé par un module orienté objet. La différence entre la modélisation d'un composant et d'un connecteur consiste dans le fait que dans le premier, les règles de réécriture décrivent le comportement du composant, alors que dans le connecteur, elles spécifient les correspondences (binds) entre les rôles d'entrée et les rôles de sortie.

Le port

Tout composant est muni de ports d'entrés (in) pour fournir des services (services offerts) et des ports de sorties (out) pour demander des services (services requis) que d'autres composants doivent fournir pour le bon fonctionnement du système global. Les ports dans la sémantique associée à CBabel sont déclarés par des opérations dans la signature Σ de la théorie de réécriture (Σ, E, R) décrivant un composant. La déclaration des ports d'entrées et de sorties d'un composant est la suivante :

$op\ \_in : Oid \rightarrow IPortId\ .$

$op\ \_out : Oid \rightarrow OPortId\ .$

Chaque nom de port d'entrée (respectivement de sortie) de sorte $IPortId$ (respectivement de sorte $OPortId$) est rattaché à l'identificateur de l'objet qui le définit. Cette manière de déclarer des ports permet d'ajouter de nouvelles instances de composants avec leurs propres ports.

Chaque objet peut envoyer des messages sur ses ports de sortie. Le premier argument de ce message comporte l'identificateur de l'objet, le deuxième correspond au nom de son port et le dernier à la donnée (data). La déclaration d'un message est la suivante :

$op\ msg : Oid\ PortId\ Item \rightarrow Msg$

$OPortId$ et $IPortId$ sont des sous-sortes de $PortId$.

La configuration

La configuration sert à créer les instances des composants et connecteurs. Elle sert aussi à construire les assemblages nécessaires entre ces éléments en définissant des attachements entre les ports des composants et rôles des connecteurs. La description de la configuration initiale est composée deux parties : la déclaration des instances des composants et des connecteurs intervenant dans la topologie initiale et la déclaration des liens entre ses instances.

La notion de configuration (topologie) correspond à un module système dans Maude ayant la syntaxe suivante :

mod $module - name$ **is**

(Σ, E, R)

endm

Σ contient la déclaration des composants et des connecteurs qui seront importés par la clause "***including module-name***". Elle contient aussi les opérations nécessaires pour construire une configuration qui a la structure d'un multi-ensemble constitué d'objets (instance de composant et /ou connecteur) et de messages.

$$eq\ Configuration = instantiate(o1, C1) \otimes \ ... \otimes instantiate(on, Cn)$$

Une connexion (lien) entre deux instances de composants $o1$, $o2$ est définit par une équation qui identifie le service demandé par le composant $o1$ et le service fourni par $o2$ à travers le remplacement du message que $o1$ envoie par le message que $o2$ reçoit :

$$eq\ M(o1,\ O) = M'\ (o2, I)$$

M et M' sont les noms des messages ; paramétrés par l'identificateur de l'instance $o1$ (respectivement $o2$), un port de sortie O (respectivement port d'entrée I).

La table ci-dessous résume la sémantique associée aux éléments d'une architecture logicielle dans CBabel.

Concept CBabel	Sémantique dans Maude
composant/connecteur	module orienté objet
instance de composant/connecteur	objet
variables locales	attributs de la classe
port	opération
événements de communication	messages échangés
comportement du composant	règles de réécriture
liens entre ports d'un connecteur (bind)	règles de réécriture
configuration (topologie)	module système
Connexion composant/connecteur (link)	équation

TABLE 5.1 – Sémantique basée Logique de réécriture des concepts CBabel.

Exemple 5.2. Producteur/Consommateur dans Maude

Le composant PRODUCER est déclaré dans un module objet qui importe les modules DECLARATION (voir **Annexe B** pour plus de détail sur ce module),

(**omod** PRODUCER **is**

including DECLARATION .

class PRODUCER | tableItemP : ItList, data : Item .

op _out : : Oid ⟶ OPortId .

var prod : Oid .

var D : Item .

var T : ItList .

/*** créer une instance PRODUCER avec it1 et it2 les données produites***/.

eq instantiate(prod, PRODUCER) = ⟨prod : PRODUCER | tableItemP : nil

'it2 'it1, data : nil⟩ .

/**générer un item à partir de la table'tableItemP'**/.

crl [prod-generer] : generer(prod, T D)⊗⟨ prod : PRODUCER | tableItemP : T D, data : nil ⟩ ⇒ ⟨prod : PRODUCER | tableItemP : T, data : D ⟩ generer(prod, T) **if**(D =/= nil) .

/*** déposer l'item sur le port de sortie 'prod.out'***/.

crl [prod-deposer] :⟨prod : PRODUCER | tableItemP : T, data : D⟩ ⇒ deposer(prod, prod.out, D)⟨prod : PRODUCER | tableItemP : T, data : nil⟩if(D =/= nil) .

endom)

Les composants $BUFFER$ et $CONSUMER$ seront déclarés de la même manière (voir **Annexe B** pour des détails supplémentaires sur le code de ces deux modules).

La topologie initiale du système PRODUCER-CONSUMER est déclarée dans le module système TOPOLOGIE qui crée les instances des différents composants grâce à l'opération *instantiate* et établit les liens entre les deux composants et le connecteur $BUFFER$ par la déclaration d'équations entre messages des composants :

(**mod** TOPOLOGIE **is**

protecting INT .

extending PRODUCER .

extending BUFFER .

extending CONSUMER .

op initialConf : → configuration .

vars prod buff cons : Oid .

vars N M : Int .

var e : status .

vars I T : ItList .

var D : Item .

/*****Liens entre composants et connecteur*****/

ceq deposer(prod, prod.put, D) ⊗⟨buff : BUFFER | buff-items : N, buff-maxitems : M, cont : I, etat : e ⟩= deposer(buff, buff.put, D)⊗⟨buff : BUFFER | buff-items : N, buff-maxitems : M, cont : I, etat : e ⟩if(e =/= Full) .

eq retirer(cons , cons.get)⊗⟨ buff : BUFFER | buff-items : N, buff-maxitems : M, cont : I, etat : e ⟩ = retirer(buff , buff.get)⊗⟨buff : BUFFER | buff-items : N, buff-maxitems : M, cont : I, etat : e ⟩ .

/*****Configuration initiale*****/

eq initialConf = generer('Prod1, nil'it2 'it1) ⊗ instantiate('Prod1, PRODUCER)⊗ instantiate('Buff1, BUFFER)⊗ instantiate('Cons1, CONSUMER) ⊗demande('Cons1).

endmd)

5.3 Mobile CBabel

Avant d'aborder les détails techniques d'enrichissement de CBabel par des primitives de mobilité, une mise au point sur l'aptitude de CBabel et plus précisément sa base sémantique à implémenter les trois concepts de base caractérisant les applications à code.

- Le concept de mise en évidence de la coordination découle directement du principe de séparation des préoccupations entre les aspects fonctionnels et de coordination des architectures logicielles, puisque CBabel est un ADL.
- Le choix de l'unité de mobilité dépend du niveau de détail et de granularité que le concepteur désir atteindre. Mais dans une perspective de généricité de la solution, nous préférerons considérer l'unité de conception (le composant) comme unité de

mobilité. La granularité de ce composant peut toujours varier d'une simple instruction à un fragment de code assez conséquent selon le niveau de détail choisi.

– Le concept de localisation et sa prise en charge par contre nécessite plus de labeur et seront détaillées dans la section suivante.

5.3.1 Concept de localisation

La notion de localisation permet d'identifier l'emplacement d'entités mobiles à travers un espace d'adressage. Ce dernier peut varier d'un ensemble d'adresses IP ou de coordonnées GPS dans le cas de dispositif physiques mobiles à des localisations abstraites représentant les entités composites les contenant.

Du fait que la sémantique associée à un composant CBabel est définie par un module objet dans Maude, comportant la déclaration d'une classe, la structure de ce composant est enrichie par un attribut *location* servant à déterminer sa localisation actuelle. Deux opérations : $get-loc$ et $change-loc$ sont aussi définies pour permettre l'accès et la mise à jour de cet attribut. Du fait qu'on ne s'intéresse qu'aux hôtes source et destination d'un composant mobile, l'espace de mobilité sera alors discret. Par ailleurs, le composant mobile sera dans un état transitoire entre son ancienne et sa nouvelle localisation durant la période de migration et ne sera pas en mesure de s'exécuter. Afin d'exprimer l'état du composant, transitoire ou non et suspendu ou non, deux autres attributs *state* et *moving* ainsi que les opérations de mise à jour correspondantes sont rajoutés à la structure du composant mobile. Concrètement, la spécification d'un composant ordinaire CBabel sera importée et étendue par la clause **sous-classe** comme suit :

(**omod** mobile-component **is**

including component .

including MO-State .

including BOOL ./la sorte Booleans definie dans Maude/

including LOCATION .

/*****Décration de la classe*****/

class mobile-component|container : Pid, state : MO-State, moving : BOOL, loc : Location .

subclass mobile-component ¡ component .

/*****Déclaration des operations *****/

op get-state : mobile-component → MO-State .

op get-moving : mobile-component → BOOL .

op get-loc : mobile-component → Location .

/*****Déclarations des messages*****/

msg modif-state : mobile-component → msg .

msg modif-moving : mobile-component → msg .

msg modif-loc : mobile-component Location → msg .

var l : Location .

eq instantiate('mobComp, mobile-component)=⟨MC : mobile-comonent |container : p, state : s, moving : m, loc : l⟩ .

eq get-state(⟨x : Oid |container : p, state : s, moving : m, loc : l⟩) = s .

eq get-moving(⟨x : Oid|container : p, state : s, moving : m, loc : l⟩) = m .

eq get-loc(⟨x : Oid |container : p, state : s, moving : m, loc : l⟩) = l .

crl [modif-state] : modif-state(x)⟨x : Oid |container : p, state : s, moving : m, loc : l⟩ ⇒ ⟨x : Oid |container : p, state : idle, moving : m, loc : l⟩ **if** state = active **else** ⟨x : Oid | container : p, state : active, moving : m, loc : l⟩ .

rl[modif-moving] : modif-moving(x)⟨x : Oid | container : p, state : s, moving : m, loc : l⟩ ⇒ ⟨x : Oid | container : p, state : s, moving : not(m), loc : l⟩ .

rl[modif-loc] : modif-loc(x, l')⟨x : Oid | container : p, state : s, moving : m, loc :

l⟩ ⇒ ⟨x : Oid | container : p, state : s, moving : m, loc : l'⟩ .

endom)

L'espace de mobilité des composants est défini par une théorie fonctionnelle :

(**fmod** LOCATION **is**

sort Location.

endfmod)

$MO-State$ est une théorie de réécriture fonctionnelle définissant les état possibles (*idle* et *active*) d'un composant :

(**fmod** MO-State **is**

sort MO-State . **op** idle : → MO-state .

op active : → MO-state .

endfmod)

5.3.2 Primitives de mobilité

L'ensemble de primitives de mobilité est défini dans une théorie de réécriture appelée Mobility-Primitives, qui sera importée dans les théories de réécriture du système à spécifier. Deux types de mobilité sont possibles :

Mobilité physique

Elle apparaît dans les réseaux sans fils et concerne le déplacement d'un composant matériel (PDA, téléphone cellulaire, etc.) à travers des zones de couverture du réseau. Le déplacement du composant matériel implique le mouvement de composants logiciels aussi. Par conséquent, la position des composants est mise à jour sans affecter l'environnement d'exécution (le conteneur du composant et son environnement). Ce scénario de mobilité est exprimé par deux règles de réécriture *change-loc* et *activate*. Dans une

configuration composée d'un message *change-loc(O,l')* et un objet ⟨O : MO, container : P, state : Active, moving : False, loc : l⟩ , les opérations suivantes sont exécutées :

- l'exécution de l'objet est tout d'abord suspendu ($state = idle$),
- l'attribut *moving* devient *true* pour indiquer que le composant est dans un état transitoire,
- La nouvelle position l' est assignée à l'attribut *loc*.

A l'arrivée, le composant reprend son état d'exécution grâce à la règle *activate* qui modifie les valeurs des attributs ($state = active$ et $moving = false$)

Mobilité logique

Elle consiste en une migration du composant d'un environnement d'exécution vers un autre à travers un réseau d'ordinateurs. Pour réaliser ce changement de localisation, il est nécessaire de suspendre l'exécution du composant, l'extraire et l'envoyer à sa nouvelle destination. A l'arrivée, le composant sera intégré dans sa nouvelle configuration afin de reprendre son exécution. Deux autres règles sont alors définies :

- *initiate-mov* est chargée d'initier la migration du composant. Elle suspend son exécution, l'extrait de la configuration actuelle et le met dans un message pour l'envoyer à son destinataire.
- *receive-component* est chargée par contre de l'intégration du composant reçu dans la configuration destination.

(**omod** Mobility-Primitives **is**

including Mobile-Component .

msg change-loc : Mobile-Component Location → msg .

msg activate : Mobile-Component → msg .

msg initiate-mov : Mobile-Component Location →msg .

msg send-component :Pid Location Mobile-Component→ msg .

msg receive-component : Pid Location Mobile-Component →msg .

var o : Oid .

var l l' : Location .

crl[change-loc] : change-loc(O,l')⟨O : Oid | container : p, state : s, moving : m, loc : l⟩ ⇒ ⟨O : Oid | container : p, state : Idle, moving : True, loc : l'⟩ **if** get-state(O) = Active and not get-moving(O) .

crl[activate] : activate(O)⟨O : Oid | container : p, state : s, moving : m, loc : l⟩ ⇒ ⟨O : Oid | container : p, state : Active, moving : False, loc : l⟩ **if** get-state(O)=idle and get-moving(O) .

crl[initiate-mov] : initiate-mov(O,l') ⟨O : Oid | container : p, State : s, moving : m, loc : l ⟩ ⇒ send-component(P',l',⟨O : Oid | container : p, State : Idle, moving : True, loc : l ⟩) **if** get-state(O)=Activate and not get-moving(O).

crl[receive-comp] : receive-component(P',l',⟨O : Oid | container : p, State : s, moving : m, loc : l ⟩)⇒ ⟨PI : P', cf : C ⟨O : Oid | container : p', State : Active, moving : False, loc : l'⟩ **if** get-state(O)=idle and get-moving(O).

endom)

Nous avons tenté d'exploiter cette théorie de réécriture dans la spécification du système de revue des conférences **(Bouanaka et al., 2008a)**. Ce système est constitué d'un président de la conférence, d'un président du programme, de relecteurs et des auteurs désirant soumettre des articles. Les unités mobiles représentent les différents formulaires échangés entre ces quatres types de composants.

Le problème majeur que nous avons rencontré a consisté en l'incapacité de décrire la structure hiérarchique des différents composants pour pouvoir montrer la localisation actuelle de chaque formulaire. Seuls le transfert de ces formulaires est exprimé par la

théorie de réécriture ci-dessus à travers l'envoi et la reception de messages contenant des composants mobiles. Ce problème est dû à l'incapacité de définir des structures hiérarchiques dans lesquelles la mobilité d'un composant n'est pas un simple envoi de message, mais une reconfiguration du composite.
Dans la section suivante, nous exploiterons la théorie de réécriture gérant la mobilité physique de composants pour la spécification du processus handover du système GSM, **publié dans (Bouanaka et al., 2008b).**

5.4 Etude de cas : Le processus handover du système GSM

Le réseau GSM (Global System for Mobile communication) est un exemple concret de systèmes mobiles dans lequel des utilisateurs de téléphones cellulaires sont continuellement en mouvement. C'est un exemple type de mobilité de terminaux physiques. Notre intérêt se portera plus particulièrement au processus "Handover" qui illustre parfaitement le passage du contrôle de l'unité mobile entre deux stations de base.

5.4.1 Présentation informelle

Le réseau GSM (voir figure 5.1 ci-dessous)est un standard de communication sans fils permettant à des utilisateurs mobiles ; appelés stations mobiles), de communiquer avec d'autres utilisateurs fixes ou mobiles aussi. Afin de rendre cette communication possible, le réseau GSM est constitué d'un ensemble de zones géographiques subdivisées elles aussi en cellules, chacune correspondant à une zone de couverture assurée par un BTS (base station transceiver). L'ensemble des cellules appartenant à la même zone sont contrôlées par un même contrôleur de state de base (BSC :Base State Controller). Le processus

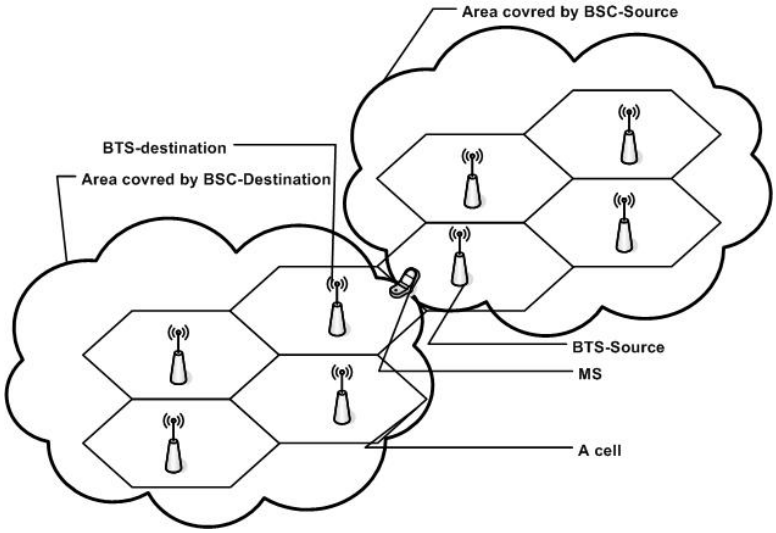


FIGURE 5.1 – Processus Handover du réseau GSM.

handover du réseau GSM permet d'assurer la continuation de l'appel d'un usager mobile à la frontière de deux zones de couverture, c.à.d., lors du passage du contrôle d'un élément de l'organisation vers le voisin que ce soit entre cellules ou entre zones. Il existe trois types de handover dépendant des BTS impliqués dans cette passation du contrôle. Nous nous intéresserons au handover touchant deux BTS appartenant à des BCS différentes. Le détail des étapes de réalisation du handover est décrit dans (GSM, 1998). Nous les résumons dans :

Demande du Handover :

La BSC actual détecte la nécessite d'un handover en analysant les dernières mesures reçues de la station mobile (MS). Elle suspend la transmission et envoie une requête de handover à la station de contrôle mobile (MSC). Cette dernière la redirige vers la BSC adéquate.

Commande du handover :

La nouvelle BSC choisit l'une de ses antennes BTS et la prépare pour la réception de la nouvelle MS. Le processus de passage du contrôle est alors initier par la transmission du message de commande du handover à la MS à travers l'ancienne BSC. Ce message permet à la MS de localiser le canal radio de la nouvelle BTS/BSC.

Terminaison du handover (Burst) :

À la réception de la commande de handover, la MS tente d'établir les connexions des couches basses avec les nouveaux BTS/BSC en envoyant un message burst à la nouvelle BSC. En cas de succès, la transmission est rétablie entre la MS et la nouvelle BSC à travers son BTS.

5.4.2 Architeture CBabel du système GSM

Sur la base de la description du processus handover, les composants MS BSC-source, BSC-destination et la MSC sont identifies. Ces composants sont interconnectés selon la figure 5.2.

Les rectangles et cercles de la figure représentent les composants et les connecteurs respectivement. Un arc d'un composant vers un connecteur représente un port de sortie et un arc d'un connecteur vers un composant représente un port d'entrée. Les arcs épais montrent des liens qui seront détruits lors du handover et les arcs discontinus montrent les liens à créer. Les stations *BSC* source et destination ont les mêmes ports d'entrée et de sortie. Pour des raisons de lisibilité, seuls les ports concernés par le processus handover sont représentés.

Seuls les composants concernés, *MS*, *BSC-source* et *BSC-Dest*, par le processus Handover seront étudiés dans ce qui suit. Le composant *MS* est composé d'un port d'entrée pour la reception de la notification de terminaison du processus, un port pour l'envoi des

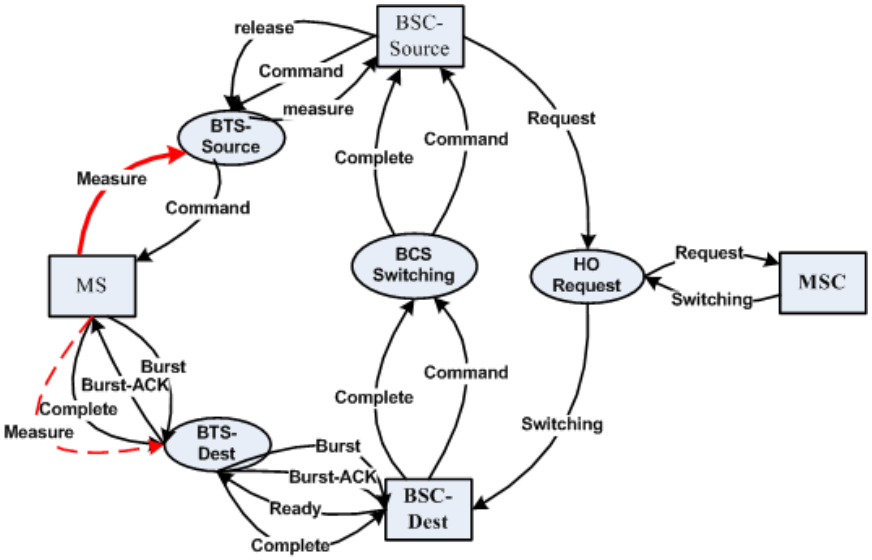


FIGURE 5.2 – Architecture du système GSM.

mesures, un autre port pour le signal burst et un port pour l'envoi du signal de fin du transfert de contrôle.

module MS{

in port command ;

in port burst-Ack ;

out port measure ;

out port burst ;

out port complete ;

}

La description CBabel du composant $BSC - Source$ est :

module BSC-Source{

in port measure ;

in port complete ;

in port command-in ;

out port request ;

out port command-out ;

out port release ;

}

Ces deux composants sont reliés par le connecteur HO-request suivant :

Connector HO-request

var int request = int(0) ;

in port request-in ;

in port switch-in ;

out port request-out ;

out port switch-out ;

interaction {

request-in >

guard (true){

before {

request = int(1) ;

}

}> request-out ;

switch-in >

guard (request==int(1)){

before {

request = int(0) ;

}

}> switch-out ;

}

}

Les interactions entre les ports *switch-in* et *switch-out* ont une garde $request == int(1)$ puisque une requête *switch* ne peut être envoyée à MSC que si le connecteur HO-request a déjà reçu un message requête de la BSC-Source.

5.4.3 Mobilité des stations de base dans mobile CBabel

La description précédente donne une idée sur la structure de chacun des composants participant aux handover en termes de ports d'entrée/sortie, mais elle ne définit pas le comportement de chacun d'eux lors de la commutation du contrôle entre les deux BSC (source et destination). Cette description ne définit pas les opérations nécessaires pour mettre à jour la localisation de la station mobile(MS) de (BTS-Source/BSC-Source) vers (BTS-Dest/BSC-Dest). Par ailleurs, l'établissement dynamique de la connexion entre MS et la nouvelle BSC n'est pas aussi défini.

Du fait que CBabel manque d'opérateurs pour exprimer ce comportement, le passage à Maude n'est pas direct et doit être étendu par les primitives de mobilité en utilisant la théorie de réécriture $Mobility - Primitives$. Les composants MS et BSC-source sont transcripts en des theories de réécriture orientées objets. BSC-source est l'initiateur de la mobilité de MS.

(**omod** MS **is**

including CBABEL-CONFIGURATION .

including Mobility-Primitives .

class MS | Loc : Location, moving : Boolean, state : status .

op Command : Oid Location → PortInId [ctor] .

op Brust-Ack : Oid Oid→PortInId [ctor] .

op Measure : Oid int →PortOutId [ctor] .

op Burst : Oid Oid → PortOutId [ctor] .

op Complete : Oid Oid →PortOutId [ctor] .

eq instantiate('O, 'MS)=⟨O : MS, Loc :(BSC-source,BTS-source), moving : False, state : Active⟩ .

rl [movement] : Command(MS,(BSC-dest, BTS-dest))⟨O : MS,Loc : (BSC-source,BTS-source), moving : False, state : Active⟩ ⇒ change-location((BSC-dest,BTS-dest)) send(Burst(MS,BSC-dest)) ⟨O : MS, Loc : (BSC-source,BTS-source), moving : False, state : Active⟩ .

rl [Movement-end] : Burst-Ack(MS,BSC-dest)⟨O : MS,Loc : (BSC-dest,BTS-dest), moving : True, state : Idle⟩ ⇒ send(create-link(MS,BTS-dest)) send(Complete(MS,BTS-dest)) activate(MS)⟨O : MS, Loc : (BSC-dest,BTS-dest), moving : True, state : Idle⟩. **rl** [measure] : ⟨O : MS, Loc : (BSC-source,BTS-source), moving : False, state : active⟩ ⇒ send(measure(BTS-source, ms)) **If** period = expired.

endom)

PortInId et *PortOutId* sont des sortes d'identificateurs de ports définis dans CBABEL-CONFIGURATION (Braga et al. 2003). *send* est un message générique défini aussi dans CBABEL-CONFIGURATION.

(**omod** BSC-Source **is**

including CBABEL-CONFIGURATION .

including Mobility-Primitives .

class BSC-source | s :state .

op measure : Oid int →PortInId [ctor] .

op command-in : Oid Location → PortInId [ctor] .

op complete : Oid Oid → PortInId [ctor] .

op request : Oid Oid Oid → PortOutId [ctor] .

op command-out : Oid Location → PortOutId [ctor] .

eq instantiate(BSC-source, BSC) = ⟨BSC-source, s :state ⟩.

rl [measure] : measure(MS, ms) ⟩BSC-source, s⟨ ⇒ send(request(MSC, BSC-source , MS)) ⟨BSC-source, s⟩ **if** ms ¿ sill **else** ⟨ BSC-source, s⟩ .

rl [command] : command-in(MS,(BSC-dest, BTS-dest)) ⟨BSC-source, s⟩ ⇒ send (command-out(MS,(BSC-dest, BTS-dest))) ⟨BSC-source, s⟩ .

rl [complete] : complete(MS,BSC-dest) ⟨BSC-source, s⟩ ⇒ send (destroy-link(MS,BTS-source)) ⟨BSC-source, s⟩ .

endom)

5.5 Conclusion

Une extension du langage CBabel par les primitives de mobilité a été réalisée. Elle s'est basée sur la définition de la notion de localisation comme attribut dans la modélisation d'un composant mobile. Des règles de réécriture ont aussi été définies pour décrire le scénario de mobilité de sa localisation actuelle vers sa nouvelle destination.

Cette extension a mis en évidence l'incapacité de la logique de réécriture à définir des structures hiérarchiques et modulaires nécessaires pour la définition implicite de la notion de localisation. Et par conséquent, les primitives de mobilité se restreignent à une mise à jour de la valeur d'un attribut. Ce qui rend la reconfiguration dynamique d'une architecture par la redistribution des composants mobiles difficile, voir impossible. Ce problème sera résolu dans le chapitre suivant grâce à la logique des tuiles.

Chapitre 6

Un langage de description d'architectures logicielles mobiles

6.1 Introduction

L'objectif de ce travail est de définir une démarche de construction d'applications à code mobile en partant de la vue la plus abstraite, qui est l'architecture logicielle. Cette dernière constitue un contrat établissant les correspondances attendues entre les besoins du système et les fonctionnalités attribuées aux composants et ce à travers les différents raffinements qu'ils subiront tout au long de leur cycle de développement. Par ailleurs, l'architecture d'un système doit refléter sa structure d'assemblage à différents niveaux de détails, c-à-d., une composition hiérarchique des composants de l'architecture. Ce concept permet de spécifier un système logiciel en adoptant une approche descendante avec différents niveaux de raffinement, le point de départ étant la vue la plus générale formée des principaux composants et connecteurs. Chaque itération sur la définition de la structure des composants constitue un raffinement de cette description initiale. Le nombre d'itérations dépend du niveau de granularité désiré.

Une première question s'impose : est t-il possible de générer le comportement global du système de manière incrémentale et hiérarchique comme c'est le cas pour la définition

de sa structure. Cette question se complique encore en présence d'une reconfiguration dynamique due à la mobilité, l'ajout ou la suppression d'un composant causant une réorganisation de la structure et de la hiérarchie du système. Un problème majeur est de pouvoir déterminer les répercussions directes de cette restructuration dynamique sur le comportement global du système.

La plupart des modèles de description d'architecture que ce soient les ADL ou ceux basés sur les graphes se focalisent sur la description de la structure en considérant les composants comme des boites noires laissant ainsi la sémantique des interfaces non définie. Les quelques ADL, qui en plus de la structure permettent de spécifier le comportement des composants, laissent le problème de la construction compositionnelle et hiérarchique du calcul global ouvert ou bien utilisent des modèles sémantiques nécessitant l'aplatissement de la structure hiérarchique.

Afin de permettre la conception incrémentale et hiérarchique de la structure ainsi que le comportement d'applications à code mobile, nous définissons un modèle centré sur les interfaces et basé sur la logique des tuiles comme cadre sémantique sous-jacent.

Une première ébauche des concepts syntaxiques de ce langage a été publiée dans(Bouanaka et al., 2008c), appelé MoSAL (pour **M**obile **S**oftware **A**pplications **L**anguage). La présen-tation détaillée des concepts syntaxiques et la définition du modèle sémantique associé à MoSAL ont été publiés dans (Bouanaka et al. 2010). Le modèle offre trois niveaux de description. Le détail du modèle proposé sera présenté dans le reste du chapitre. Les trois couches, structurelle, comportementale et de reconfiguration nécessaires à la spécification des architectures logicielles dynamiques seront par la suite détaillées.

6.2 Syntaxe du langage MoSAL

Dans un contexte de mobilité, deux types de configurations sont manipulées :

La configuration structurelle : appelée aussi topologie de l'application, détermine les composants et les liens qui forment cette application. Cette configuration sert à vérifier la compatibilité de l'assemblage des interfaces des différents composants actuellement présents.

La configuration comportementale : modélise le comportement de l'application en décrivant les évolutions des liens entre composants, c-à-d., survenue d'interactions provoquant des réactions de la part des composants fournisseurs de services.

Deux principaux critères d'évaluation sont à considérer lors de la définition de la configuration d'une architecture logicielle dans un environnement de calcul mobile :

- La dynamique de l'application : la définition de la configuration structurelle doit faciliter la spécification du comportement dynamique de l'application, c-à-d., elle doit permettre aux éléments architecturaux d'être dynamiquement modifiables.
- L'évolution de la configuration : l'application peut être amenée à réaliser de nouvelles fonctionnalités à cause d'une modification ou une évolution de sa structure. Il est alors nécessaire que la configuration permette la modification du comportement et la structure de l'architecture en ajoutant ou en supprimant des composants ou connecteurs.

Ces deux critères se résument en la capacité de chaque type de configuration à percevoir les effets de bord de l'évolution de l'autre type de configuration. Pour faciliter cette perception des effets de bord de l'évolution de la configuration comportementale sur la configuration structurelle de l'architecture et vice-versa, le modèle proposé considère les interfaces comme objets de base pour la définition des deux types de configuration :

- La distribution des interfaces entre composants représente la configuration structurelle, les valeurs des différentes interfaces constituent la configuration comportementale.
- La dynamique de la configuration structurelle correspond à une redistribution des interfaces à cause de la mobilité d'un composant. La dynamique de la configuration comportementale consiste à modifier les valeurs ces interfaces causées par une ou plusieurs interactions entre composants de l'architecture.
- Les effets de bord de l'évolution d'un type de configuration sur l'autre sont aussi interceptés à travers les modifications de valeurs ou de distribution des interfaces.

La spécification bidimentionnelle de la dynamique des applications à code mobile ainsi que ses effets de bord sera réalisée à travers les couches, structure, comportement et reconfiguration, définies par MoSAL. Un système Client/serveur, inspiré de (Baros, 2007), sera utilisé tout au long de ce chapitre pour illustrer les différents concepts introduits par MoSAL.

Exemple 6.1. Système Client/serveur

Le système est composé de

- un serveur primaire responsable d'attribuer un temps d'exécution aux différents clients qui y transitent,
- un serveur secondaire, un compétiteur du serveur primaire mais plus puissant en offrant des services avec un temps d'exécution cours, permettant de fournir le service aux clients immédiatement. Chaque client est responsable de juger si le temps d'attente est excessif et donc quitter le serveur primaire pour aller chercher le service chez le serveur delay.
- un générateur qui génère des clients et les envoie au serveur primaire,
- un dernier composant Stats responsable de faire différentes formes de statistiques

sur les clients : tempts d'attente, temps d'exécution, taux d'arrivée, etc.

6.2.1 Niveau Architectural

A un niveau architectural, les composants sont considérés comme des boites noires, connectées à leur environnement à travers les interfaces. En effet, la vue architecturale est définie par un ensemble d'interfaces représentées ici par leur noms pour des raisons de simplicité ; des structures de données plus complexes peuvent être facilement définies. La configuration structurelle ou topologie de l'architecture logicielle est définie par un hypergraphe dont :

- les noeuds correspondent aux interfaces,
- les hyperarcs correspondent aux composants
- les liens de connexion entre composants sont représentés par la superposition d'une interface de sortie d'un composant sur une interface d'entrée d'un autre. Ils servent à attacher les interfaces et expriment les flux de données ou de contrôle (invocation de service) entre ces composants.

Remarque 6.2.1. Les liens entre composants sont définis par des arcs, chacun étant spécifiant une correspondence entre un type d'interface de sortie d'un composant et un type d'interface de sortie d'un autre. Un arc entre deux interfaces signifie que le service demandé par le premier composant est fourni par le second ou bien dans le cas d'une interaction que le message envoyé par l'un sera reçu par l'autre.

Remarque 6.2.2. Les connecteurs ne sont plus considérés comme entités de première classe dans le modèle défini. Ce dernier se base sur des connecteurs dynamiques (voir Bouanaka et al. 2007 pour plus de détails) qui ne seront créés qu'au moment de l'interaction entre deux composants.

Définition 6.2.1. Une architecture logicielle est définie par un hypergraphe $SA =$

$(I_{SA}, C_{SA}, s_{SA}, d_{SA})$, tels que :

- I_{SA} est un ensemble fini de noeuds associées aux interfaces visibles de l'architecture logicielle,
- C_{SA} est un ensemble fini d'hyperarcs correspondant aux composants de l'architecture considérée,
- s_{SA}, d_{SA}: $S \longrightarrow \wp(I_{SA})$ sont des fonctions totales retournant les nœuds source et destination d'un hyperarc fourni en paramètre.

Deux fonctions I_{SA} et C_{SA} sont aussi associées à chaque hypergraphe afin de déterminer ses interfaces et composants.

Exemple 6.2. Architecture du Système Client/Serveur Client/serveur

L'architecture logicielle du système Client/Serveur ; présenté dans l'exemple 1, est définie par l'hypergraphe $SA = (I_{SA}, C_{SA}, s_{SA}, d_{SA})$, tels que :

- $I_{C/S} = \{ClientIn, Tick, ClientOutPS, GiveUp, ClientOutDS\}$ est l'ensemble des interfaces visibles de cette architecture,
- $C_{S/C} = \{G, PS, DS, Stats\}$ est l'ensemble des composants,

Les noeuds sources et destinations de l'arc PS sont $s_{C/S}(PS) = \{ClientIn, Tick\}$ et $d_{C/S}(PS) = \{ClientOutPS, GiveUp\}$ respectivement. Les noeuds sources et destinations des autres hyperarcs sont définis de manière similaire.

Graphiquement, cette architecture est présentée par l'hypergraphe de la figure 6.1, dont les carrés représentent les composants et les cercles pleins représentent les interfaces. Un lien de connexion entre deux composants est représenté par la superposition d'une interface d'entrée/sortie d'un composant et une interface de sortie/entrée d'un autre composant.

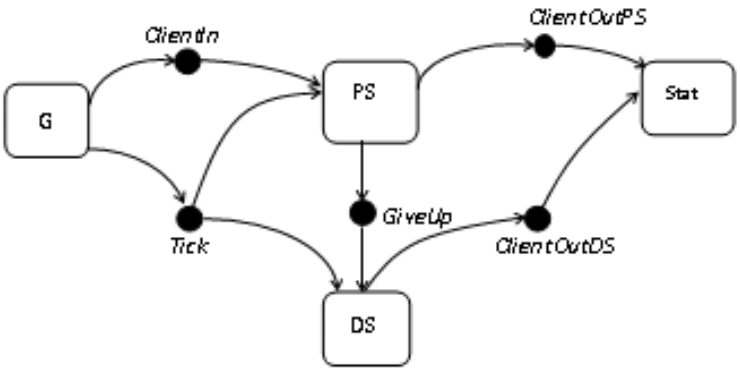


FIGURE 6.1 – Architecture du système client/serveur

Chaque composant est considéré comme une transformation entre ses interfaces d'entrée et de sortie. A titre d'exemple, le serveur primaire PS est une transformation de l'ensemble de ses interfaces d'entrée $\{ClientIn, Tick\}$ vers l'ensemble de ses interfaces de sortie $\{ClientOutPS, GiveUp\}$.

$\{GiveUpPS\} \rightarrow \{GiveUpDS\}$ représente une connexion entre les composants PS et DS qui partagent le nom de cette interface.

A ce niveau de description, le serveur primaire PS de la figure 6.1 est défini par ses interfaces visibles uniquement : $ClientIn$, $Tick$, $ClientOut$ et $GiveUp$. Si toutefois la structure détaillée de ce composant ou tout autre composant composite est pertinente, elle peut toujours être décrite par une autre vue offerte par le langage MoSAL, appelée *vue hiérarchique*.

6.2.2 Niveau hiérarchique

Partant du principe de la nécessité d'une description itérative et incrémentale de tout type d'applications, des itérations peuvent être réalisées sur la description de l'architecture globale par des raffinements successifs de celle-ci. Ainsi, les structures hiérarchiques sont autorisées dans MoSAL en permettant l'imbrication de graphes dans les hyperarcs

correspondant aux composants, afin de décrire leurs structures internes. Un graphe *body*, ayant la même structure que celle de l'architecture, peut être utilisé pour décrire les interfaces internes d'un composite ainsi que leurs interconnexions via ses sous composants.

Définition 6.2.2. Un composant hiérarchique HC est un triplet $HC = (I_{HC},\, B_{IC},\, m_{HC})$ tels que :

- I_{HC} est un ensemble d'interfaces ayant HC comme hyperarc
- $B_{IC} = (I_{B_{HC}},\, CB_{HC},\, sB_{HC},\, dB_{HC})$ est un hypergraphe ayant la structure que SA
- $m_{HC} \colon I_{HC} \longrightarrow I_{B_{HC}}$ est une fonction partielle établissant une correspondance entre les noeuds internes (noeuds de B_{HC}) et les interfaces visibles (I_{HC}) du composant hiérarchique.

Étant donné que les composants hiérarchiques cachent leur structure interne, les fonctionnalités des sous-composants ne sont accessibles qu'à travers les interfaces du composant hiérar-chique. C'est pourquoi le rôle des interfaces visibles est de recevoir les données qui parviennent d'une direction, interne ou externe, et de les faire suivre à l'autre direction. Une fonction de correspondance m_{HC} est alors définie entre les interfaces visibles I_{HC} et les interfaces internes $I(B_{HC})$.

Remarque 6.2.3. Un composant primitif $SC = (I_{SC},\, \{SC\},\, \emptyset)$ est un composant hiérarchique particulier ayant un hypergraphe *body* vide, il est constitué d'un seul hyperarcs qui est SC lui-même et des interfaces d'entrée et de sortie visibles I_{SC}.

Exemple 6.3. Structure interne de PS

Une vue de plus près de l'hyperarc PS de la figure 6.1, montre un composant hiérarchique dont le graphe *body* est présenté dans la figure 6.2. Il est obtenu par assemblage d'une queue $FIFO$ et un ensemble de clients Ci actuellement en attente d'un temps d'exécution. Formellement, le composant hiérarchique PS est défini par $(I_{PS},\, B_{PS},\, m_{PS})$, tels que :

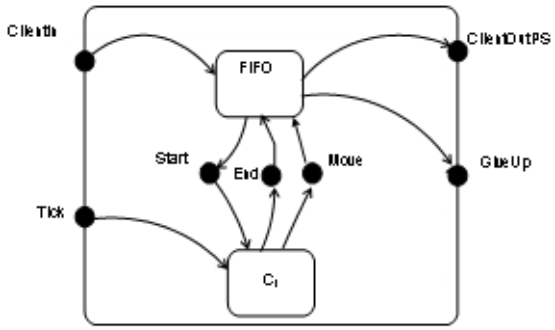


FIGURE 6.2 – Structure interne du composant hiérarchique PS

- $I_{PS} = s_{C/S}(PS) \cup d_{C/S}(PS) = \{ClientIn, Tick, ClientOutPS, GiveUp\}$
- B_{PS} correspond à l'hypergraphe de la figure 6.2.
- La relation de correspondance m_{PS} entre les interfaces visibles I_{PS} et les interfaces $I(B_{PS})$ des sous-composants de PS, établit des liens d'interaction avec l'environnement externe. Elle est représentée ici par les quatre relations suivantes :
  - $\{ClientIn\} \rightarrow \{ClientInFIFO\}$,
  - $\{Tick\} \rightarrow \{Tick_{CI}\}$,
  - $\{ClientOut_{FIFO}\} \rightarrow \{ClientOutPS\}$,
  - $\{GiveUp_{FIFO}\} \rightarrow \{GiveUp\}$.

Remarque 6.2.4. Les relations de correspondence entre les interfaces visibles et internes de PS sont représentées par la superposition directe de l'interface d'un sous-composant sur une interface visible de PS. $ClientIn$ du composant $FIFO$ est superposée sur l'interface $ClientIn$, l'interface $Tick$ de Ci sur l'interface $Tick$ et ainsi de suite.
Les interfaces internes $Start$, End et $Move$ permettent des interactions internes entre sous-composants.

Cette description hiérarchique facilite la prise en charge de la mobilité d'un composant logiciel d'un composite hiérarchique HC_1 vers un autre HC_2 en basculant entre les

niveaux de la hiérarchie, c-à-d., considérer la structure interne de HC_1 et HC_2 et voir par contre le composant mobile comme unité atomique.
Une fois la struture détaillée de l'application établie par des raffinements hiérarchiques de la description de l'architecture globale, elle peut encore détaillée par la spécification du comportement associé à chaque composant ainsi que la dynamique da la structure et ce grâce au niveau de description comportemental.

6.2.3 Niveau dynamique

Cette vue constitue une des principales contributions de MoSAL car la plupart des approches de spécification des architectures logicielles négligent l'aspect comportemental induisant à une sémantique non bien définie des interfaces des composants ou bien nécessitent des formalismes additionnels pour décrire le comportement associé à l'architecture.
Dans le cadre du langage MoSAL, les vues architecturale et hiérarchique concernent la description statique de la structure. Un raffinement de ces deux vues est possible à travers la spécification des fonctionnalités observables et les changements dynamiques de la structure et l'état des composants logiciels.

Evolution de la configuration comportementale

Elle concerne la spécification des changements locaux possibles sur la configuration comportementale d'un composant. Elle sert à rattacher à chacune des interfaces les actions qui lui sont associées.
Les actions observables associées aux interfaces visibles d'un composant logiciel, que ce soit primitif ou composite, sont définies par un ensemble de règles de réécriture R_{SC} exprimant leurs évolutions locales possibles. Ces actions ne sont autorisées que si le

système se trouve dans des états ou des configurations particulières. Un ensemble de tuiles T_{SC} est alors défini pour spécifier les configurations nécessaires à l'application des différentes actions ainsi que leurs effets de bord sur les autres composants du système global.

Définition 6.2.3. Le comportement d'un composant logiciel SC est défini par la paire $Behav_{SC} = (R_{SC}, T_{SC})$, tels que :

- R_{SC} est un ensemble fini de règles de réécriture définissant les actions possibles sur les interfaces I_{SC},
- T_{SC} est un ensemble fini de tuiles définissant les changements locaux possibles sur le composant SC.

R_{SC} et T_{SC} manipulent des termes algébriques générés à partir de l'ensemble des interfaces visibles I_{SC}, muni d'opérateurs de composition particuliers dont la sémantique sera définie dans la section suivante. Les règles de réécriture R_{SC} spécifient les actions possibles sur les interfaces visibles de SC, c-à-d., les évolutions de possibles des valeurs de ces interfaces. Par contre, les tuiles T_{SC} définissent les états observables (configurations initiales et finales des tuiles) auxquels les actions R_{SC} sont applicables. Elles spécifient aussi les données contextuelles nécessaires à l'exécution de ces changements ou actions sur les interfaces visibles de SC, c-à-d., les tuiles définissent les conditions et les effects de bords des actions.

Le comportement du composant logiciel est construit par propagation des effets locaux des tuiles à sa configuration globale grâce aux règles de déduction de la logique des tuiles.

Exemple 6.4. Comportement du serveur primaire

Toujours dans le cadre de l'exemple du système Client/Serveur, des exemples d'actions R_{PS} sur les interfaces visibles du serveur primaire. sont :

- $Receive(msg(c))$: $(ClientIn = null) \longrightarrow (ClientIn = msg(c))$ pour la reception d'un nouveau client sur l'interface $ClientIn$,
- $Empty(I)$: $(I \neq null) \longrightarrow (I = null)$ sert à vider l'interface spécifiée comme paramètre de la règle,
- $Full(ClientOut)$: $(ClientOut = null) \longrightarrow (ClientOut \neq null)$
- $Signal$: $(Tick = null) \longrightarrow (Tick = 1)$ indique l'arrivée d'un signal d'horloge du composant générateur.

Ces règles de réécriture constituent un sous-ensemble d'actions possibles sur les interfaces du serveur primaire. De façon similaire, d'autres actions peuvent être aussi définies pour envoyer un client soit sur l'interface $ClientOutPS$, s'il a terminé sa tâche, ou bien sur l'interface $GiveUp$ s'il préfère aller chercher le service ailleurs. Les actions associées aux interfaces $Start$, End et $Move$ sont locales au serveur primaire.

Comme nous le verrons par la suite, le rôle principal du composant hiérarchique PS est de se reconfigurer à cause de l'arrivée ou du départ d'un client. De ce fait, il n'a pas de fonctionnalités propres qu'il fournit à son environnement et les clients ne sont que de passage pour obtenir un temps d'exécution.

Un exemple de tuile, $Forward$ permet le transfert d'un client de l'interface $ClientIn$ à l'interface correspondante du composant $FIFO$ afin de l'insérer dans la file d'attente.

$$Forward(msg(c))\text{: } (ClientIn = null,\ ClientIn_{FIFO} = null)$$
$$\frac{Receive(msg(c)) \otimes id_{ClientInFIFO}}{Empty(ClientIn);\ Full(ClientIn_{FIFO})} (ClientIn = null,\ ClientIn_{FIFO} = msg(c))$$

La réception d'un message sur l'interface $ClientIn$ de PS déclenche la tuile $Forward$. Le composant PS passe d'abord par un état non observable correspondant à l'évolution de l'interface $ClientIn$ de $null$ à $msg(c)$. Ensuite, le message est retiré de cette interface et est déposé sur l'interface $ClientInFIFO$.

De part l'évolution de sa configuration comportementale, en considérant les changements

de valeurs des interfaces visibles des différents composants, une architecture logicielle mobile est soumise à autre type d'évolution ; l'évolution de la configuration structurelle causée par la migration d'un composant mobile d'un composant hiérarchique vers un autre.

Evolution de la configuration structurelle

Un composant hiérarchique est soumis à un autre type d'évolution, la reconfiguration dynamique, qui définit ses restructurations possibles déclenchées par la migration d'un sous-composant vers un autre composant hiérarchique. Ce type d'évolution agit sur la configuration structurelle des composants hiérarchiques en appliquant une redistribution des interfaces ; par suppression des interfaces du composant mobile du composant hiérarchique source et leur injection dans le composant hiérarchique destination. Cela provoque une transformation du graphe *body* du composite actuel et destinataire. Cette redistribution affecte aussi les liens de correspondance entre les interfaces visibles du composite et le composant mobile. Le comportement futur du composant est aussi affecté par cette réorganisation. Car l'ajout ou la suppression de certains liens de correspondance définit ou inhibe certaines actions visibles. Par conséquent, le comportement du composite est soit augmenté ou bien diminué des fonctionnalités du composant mobile reçu ou partant. Néanmoins, il ne sera pas altéré par les réarrangements locaux de la structure du composant causés par une redistribution locale de ses interfaces internes ; mise à jour des liens seulement. Une telle redistribution locale ne sera pas perçue par l'environnement externe puisque les interfaces visibles du composant restent inchangées. Le scénario de migration d'un composant d'un composant hiérarchique vers un autre est générique et est indépendant du comportement de l'application. Il consiste à isoler le composant mobile par la destruction de ses liens avec les autres composants du côté composite source et la création de nouveaux liens de connexion du côté destination. Ainsi,

un ensemble de tuiles génériques est défini pour prendre en charge la restructuration d'un composant hiérarchique suite au départ ou l'arrivée d'un composant mobile.

Définition 6.2.4. La reconfiguration d'un composant $HC = (I_{HC},\ B_{HC},\ m_{HC})$ ayant un comportement $Behav_{HC} = (R_{HC},\ T_{HC})$ est définie par une tuile générique :
$\Re\colon [HC,\ Behav_H C] \frac{move(SC, HC', m)}{\Re_B(B_{HC}, (I_{SC}, \{SC\}, \emptyset)) \otimes \Re_m(m_{HC}, m) \otimes \Re_R(R_{HC}, R_{SC}) \otimes \Re_T(T_{HC}, T_{SC})} [HC',\ Behav'_H C]$

avec :

- $[HC,\ Behav_{HC}] = [(I_{HC},\ B_{HC},\ m_{HC}),\ (R_{HC},\ T_{HC})]$
- $[HC',\ Behav'_{HC}] = [(I_{HC},\ B'_{HC},\ m'_{HC}),\ (R'_{HC},\ T'_{HC})]$

Cette tuile est déclenchée par la réception d'une requête de migration $move(SC,\ HC',\ m)$, précisant le composant concerné par la migration, sa nouvelle destination, et les correspondances à détruire. L'arrivée ou le départ d'un composant mobile, défini par une structure $(ISC,\ \{SC\}, \emptyset)$ et un comportement $Behav_{SC} = (R_{SC}, T_{SC})$, provoque une révolution dans l'architecture et le comportement d'un composant hiérarchique. La structure de ce denier évoluera de $(I_{HC},\ B_{HC},\ m_{HC})$ vers une nouvelle structure $(I_{HC},\ B'_{HC},\ m'_{HC})$. Son comportement aussi sera affecté en évoluant de (R_{HC}, T_{HC}) vers (R'_{HC}, T'_{HC}). Ces transformations est réalisée par l'effet de bord de la tuile.
Au niveau d'un composant hiérarchique, seule l'arrivée ou le départ d'un composant mobile est visible :

Arrivée d'un composant mobile

Dans le cas de la réception de SC par un composant hiérarchique, la tuile générique sera instanciée par la tuile *Connect* qui aura pour rôle d'intégrer SC dans le graphe de HC en mettant à jour ses interfaces internes $I(B_{HC})$ et systématiquement leurs liens.
L'ajout de nouvelles interfaces internes a une conséquence directe sur les fonctionnalités offertes par le composant hiérarchique, elles seront augmentés par celles du composant

mobile. Les correspondances entre les interfaces visibles et internes de HC sont enrichies par de nouvelles connections reliant certaines interfaces de SC. En effet, certaines actions non encore définies le seront désormais en rattachant certaines actions du composant mobiles aux interfaces visibles correspondantes. Cela induit à l'augmentation de R_{SC} et T_{SC} par de nouvelles actions et leurs implementations correspondantes.

Afin de reconfigurer la structure du composant hiérarchique et par conséquent enrichir ses fonctionnalités, certaines actions seront définies dans le déclencheur et l'effet de la tuile de reconfiguration et viendront se rajouter à $Behav_{HC}$. Nous pouvons alors écrire $\Re(HC, Behav_{HC}) = (I_{HC}, \Re_B(B_{HC}), \Re_m(m_{HC}), \Re_R(R_{HC}), \Re_T(T_{HC}))$ dont le composant hiérarchique résultat est obtenu par :

- $B'_{HC} = \Re_B(B_{HC}, (I_{SC}, \{SC\}, \emptyset)) = ((I(B_{HC}) \cup I_{SC}), C(B_{HC}) \cup \{SC\}, s', d')$,
- $m'_{HC} = \mathrm{Re}_m(m_{HC}, m) = m_{HC} \cup m$
- $R'_{HC} = \mathrm{Re}_R(R_{HC}, R_{SC}) = R_{HC} \cup R_{SC}$,
- $T'_{HC} = \mathrm{Re}_T(T_{HC}, T_{SC}) = T_{HC} \cup T_{SC}$.

Départ d'un composant mobile

Le départ d'un composant mobile est une opération de reconfiguration qui sera réalisée par l'instanciation de la tuile générique par *Disconnect*. Le déclencheur est dans ce cas une demande de migration qu'elle soit subjective, formulée par le composant mobile lui-même, ou objective, formulée par l'environnement. En réaction, la tuile *Disconnect* supprime SC du graphe *body* du composant hiérarchique, provoquant ainsi la destruction de certaines interfaces internes, de liens et de relations de correspondance. Du fait que certaines relations de correspondance seront détruites, des restrictions seront faites sur le comportement du composant hiérarchique en inhibant les actions fournies par SC ainsi que les tuiles correspondantes. Les reconfigurations nécessaires sur la structure et le comportement de HC seront faites de duale à l'ajout d'un composant mobile.

Exemple 6.5. Reconfiguration du serveur primaire

Soit l'architecture du composant PS de la figure 6.2. Un exemple de reconfiguration dynamique du serveur primaire est l'expiration du temps d'exécution d'un client qui génère une demande de migration $move(Ci, PS, m)$ sur son interface End_{Ci}.

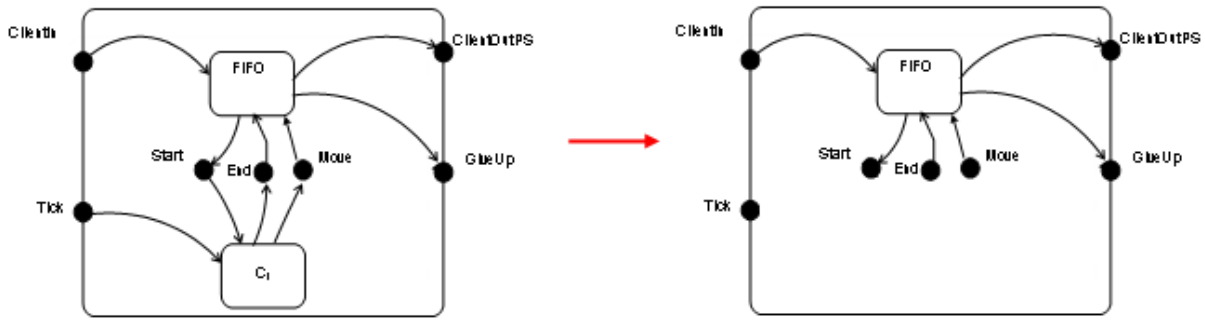


FIGURE 6.3 – Reconfiguration de PS

La tuile *Disconnect* est déclenchée. Elle extrait l'hyperarc du client, un graphe définit par $Ci = (I_{Ci}, Ci, \emptyset)$, de l'hypergraphe de PS en supprimant les interfaces visibles du client ($I_{Ci} = \{Start_{Ci}, End_{Ci}, GiveUp_{Ci}\}$) et leur liens avec le composant FIFO (voir figure 6.3). Étant donné que le client n'offre aucun service, c-à-d., $(m(HC, SC) = \emptyset)$, la suppression de ses interfaces n'aura pas de répercussion sur son comportement futur PS.

D'autres types de reconfiguration sont aussi possibles à différents niveaux du système Client / Serveur comme la mobilité d'un client du générateur vers le serveur primaire ou bien de ce dernier vers le serveur secondaire à la recherche du service s'il juge qu'il a trop attendu. Cette migration implique deux composants hiérarchique, source et destinataire, et est obtenue par une composition des tuiles *Connect* et *Disconnect*. Cela met en évidence l'importance des structures hiérarchiques définies dans MoSAL.

Contrairement à la plupart des ADL qui font intervenir plus d'un formalisme pour

la spécification de la dynamique d'une architecture logicielle et son comportement, un cadre sémantique commun est adopté dans ce travail. Il est important de préciser que les effets de bord des deux types de dynamique sont facilement recouverts grâce aux interfaces, qui constituent le concept de base à partir duquel les vues architecturale et comportementale sont construites. En effet, une demande de migration formulée à travers l'une des interfaces d'un composant hiérarchique provoque une reconfiguration de son graphe *body*. Cette reconfiguration affecte le comportement futur en inhibant les fonctionnalités du composant mobile. De manière similaire, la réception d'un composant mobile sur l'interface d'un composant hiérarchique, incite le composant à reconfigurer son graphe *body* et augmenter ses fonctionnalités par celles du composant nouvellement intégré.

6.3 Sémantique de MoSAL

La sémantique sert à associer un modèle mathématique aux programmes écrits selon la syntaxe d'un langage donné. Le modèle mathématique ainsi obtenu permet de vérifier la conformité d'un programme aux besoins du système à spécifier. Dans le cas du langage MoSAL, les programmes sont des systèmes de tuile et le modèle associé est une double catégorie. Le choix d'une double catégorie comme modèle sémantique trouve sa justification dans le fait que les applications à code mobile manipulent deux types de configurations : les configurations structurelle et comportementale ; chacune avec son espace de valeurs et ses évolutions possibles.

Le système de tuiles générant la double catégorie est défini comme suit :

Définition 6.3.1. A chaque composant hiérarchique $\langle HC, Behav_{HC}\rangle = [(I_{HC}, B_{HC}, m_{HC}), (R_{HC}, T_{HC})]$ est associé un système de tuiles définie par $TS_{HC} = (H_{HC}, V_{HC}, N_{HC}, R_{HC})$ tels que :

- Les objets de base des deux catégories horizontale et verticale correspondent aux d'interfaces visibles et internes, les interfaces des sous-composants,

$$O_H(HC) = O_V(HC) = I_{HC} \cup I(B_{HC})$$

- Les morphismes de base de la catégorie horizontale H_{HC} correspondent aux composants $C(B_{HC})$, ayant pour role la transformation des valeurs des interfaces d'entrées pour obtenir les valeurs des interfaces de sortie, plus ceux générés par m_{HC}, ayant pour role le transfert de données de et vers les sous-composants,
- Les morphismes de base de la catégorie verticale V_{HC} correspondent aux actions visibles et internes du composant HC,

$$R_{HC} \cup (\bigcup_{SCi \in C(B_{HC})} R_{SCi})$$

- L'ensemble de tuiles de base regroupe les tuiles de base de HC ainsi que ceux de ses sous-composants ,

$$T_{HC} \cup (\bigcup_{SCi \in C(B_{HC})} T_{SCi})$$

- N_{HC} est l'ensemble des étiquettes des tuiles.

L'ensemble d'objets, communs aux deux catégories, définit l'interprétation sémantique de toutes les interfaces, que ce soit les interfaces visibles de HC ou les interfaces internes représentant les nœuds du graphe *body* de HC. L'ensemble des morphismes de base de la catégorie horizontale formalise les hyperarcs (sous-composants) et les relations de correspondance entre interfaces visibles de HC et interfaces internes. L'originalité de ce modèle est que des morphismes plus larges sont systématiquement générés grâce aux opérations de composition des morphismes. Ces morphismes représentent les configurations globales du composant hiérarchique, par composition des interfaces des sous-composants. Les morphismes de base de la catégorie verticale correspondent aux actions associées aux différentes interfaces du composant hiérarchique (internes et visibles).

La table ci-dessous établie une correspondance entre les différents concepts définis par le langage MoSAL et les systèmes de tuiles.

Architecture Logicielle	Système de tuiles
• Composant Hiérarchique HC	• Système de tuile $ST(HC) = (H_{HC}, V_{HC}, N_{HC}, T_{HC})$
◦ Interfaces	◦ Objets de base des catégories H_{HC}et V_{HC}
◦ Configurations structurelles	◦ Morphismes de la catégorie H_{HC}
◦ Configurations comportementales	◦ Morphismes de la catégorie V_{HC}
◦ Changements locaux	◦ Tuiles $T_{HC} \cup T_R$
◦ Sémantique opérationnelle	◦ Calcul de déduction
• Composant primitif $SC = [(I_{SC}, \{SC\}, \emptyset), (R_{SC}, T_{SC})]$	• Morphisme $\in Mor(H_{HC})$ et défini par $ST(SC)(H_{SC}, V_{SC}, N_{SC}, R_{SC})$
◦ Interfaces I_{SC}	◦ Objets des catégories H_{SC} et V_{SC} $(I_{SC} \subseteq O(H_{HC}))$
◦ Configurations Comportementales	◦ Morphismes de la catégorie H_{SC}
◦ Observations R_{SC}	◦ Morphismes de la catégorie V_{SC}
◦ Changements locaux T_{SC}	◦ Tuiles T_{SC}
◦ Sémantique opérationnelle	◦ Calcul de déduction

TABLE 6.1 – Sémantique basée Logique des tuiles du langage MoSAL.

La catégorie horizontale

constitue le modèle sémantique associé aux configurations structurelles (topologies) de l'architecture logicielle/composant hiérarchique. Cette catégorie montre la distribution spatiale des interfaces et leurs schémas d'assemblages. Elle est construite de la manière suivante :

- Les objets de cette catégorie sont les interfaces visibles et internes du composant.
- Les morphismes de base de cette catégorie sont les composants / sous-composants de l'architecture/composant hiérarchique respectivement.

- Cette catégorie est munie de deux opérateurs de composition de morphismes. La composition parallèle reflète la distribution des interfaces. La composition horizontale montre les liens d'interactions possibles entre composants à travers le partage d'interfaces. Ce dernier opérateur est associatif et admet une identité horizontale.

Les termes de preuve générés par composition parallèle et horizontale des interfaces constituent les configurations possibles de l'architecture logicielle ou du composant hiérarchique.

La catégorie verticale

Elle modélise les actions observables sur les interfaces de l'architecture/ composant hiérar-chique et est construite comme suit :

- Les objets de cette catégorie sont toujours les interfaces visibles et internes du composant hiérarchique/l'architecture,
- Les morphismes de base de cette catégorie correspondent aux actions observables sur les interfaces,
- Cette catégorie aussi est munie de deux opérateurs de composition. La composition parallèle exprime l'evolution parallèle d'actions indépendantes. La composition verticale construit les traces d'exécution des actions. La composition verticale est associative et admet des morphismes identité.

Les termes de preuve de cette catégorie correspondent aux compositions d'actions possibles et représentent les actions globales de l'architecture ou le composant hiérarchique.

La double catégorie

Elle est construite par superposition de la catégorie verticale sur la catégorie horizontale via des transformations naturelles, interprétation sémantique des tuiles du composant hiérarchique/architecture, montrant les actions autorisées sur chaque configuration structurelle. Les termes de preuves de cette double catégorie constituent une

interprétation sémantique du comportement globale d'un composant hiérarchique par composition des comportements locaux de ses sous-composants. Ils correspondent au calcul global déduit par les règles de déduction de la logique des tuiles. Ces termes de preuves sont générés comme suit :

- Les objets de base sont toujours les interfaces du composant,
- Les morphismes de base correspondent aux tuiles de base du composant,
- Trois opérations de composition de morphismes sont définies sur cette catégorie. La composition parallèle montre l'evolution parallèle de termes de preuve indépendants. La composition horizontale des morphismes montre les interactions et les coordinations possibles entre sous-composants. Enfin, la composition verticale décrit des étapes de calcul de termes de preuve.

Une propriété pertinente de ce modèle est que la construction incrémentale et hiérarchique de la structure qui se répercute directement sur la génération du calcul global, qui lui aussi est généré par composition des comportements locaux des différents sous-composants appartenant à la structure actuelle du composant hiérarchique.

Exemple 6.6. Modèle sémantique associé au serveur primaire

Le modèle sémantique associé au composant hiérarchique serveur primaire est une double catégorie générée par le système de tuiles $TS_{PS} = (H_{PS}, V_{PS}, N_{PS}, T_{PS})$:

- L'ensemble d'objets communs aux deux catégories H_{PS} et V_{PS} contient les interfaces suivantes :

  $O_{PS} = \{ClientIn, ClientOutPS, Tick, GiveUp\} \cup \{Start(Ci), End(Ci), Move(Ci)\}$

  Pour des raisons de lisibilité, la duplication des interfaces $ClientInPS$, $ClientInFIFO$, $ClientOutPS$ a été omise,
- Les morphismes de base de H_{PS} sont : $FIFO$: $\{ClientIn, End, GiveUp\} \longrightarrow \{Start, ClientOut, GiveUp\}$ et Ci: $\{Tick, Start\} \longrightarrow \{End, Move\}$.

Une catégorie est alors générées par différentes formes d'interconnexions : parallèle pour la distribution spatiale et horizontale pour les liens d'interaction entre interfaces,

- Les morphismes de base de V_{PS} englobent toutes les règles de réécriture rattachées aux interfaces de PS et étiquetées par les noms des actions associées,
- L'interprétation sémantique de l'évolution de la configuration de PS suite à l'exécution d'une action est une transformation naturelle étiquetée par le nom de la tuile correspondante.

Il est clair que les concepts introduits dans toutes les couches du modèle (architecturale, hiérarchique et dynamique) trouvent leur interprétation sémantique dans la double catégorie ainsi définie :

- Cette sémantique a bien réussi à interpréter les aspects structurels de modélisation. En effet, la catégorie horizontale à travers ses termes de preuve associe une interprétation sémantique aux configuration structurelles(topologies) d'un composant hiérarchique/architecture,
- Elle aussi réussi à interpréter les configurations comportementales à travers les termes de preuve générés par la catégorie verticale, montrant les valuations possibles sur les interfaces,
- Elle a réussi à interpreter la dynamique, structurelle et comportementale, des composants à travers le calcul de preuves dans la double catégorie. Plus particulièrement, cette sémantique a fidèlement interprété les effets de bord de la dynamique d'une dimension sur l'autre dimension. En effet, l'interception de ces effets de bord est réalisée à travers la superposition de la catégorie verticale sur la catégorie horizontale, spécifiant ainsi les actions autorisées sur chaque configuration structurelle et vice-versa.

Par ailleurs, l'interprétation hiérarchique, dans laquelle le morphisme associé à un composant hiérarchique est une double catégorie, préserve la propriété d'abstraction des systèmes considérés. L'effet des deux types de reconfiguration, rearrangements locaux et migration de composants, sur le modèle sémantique dépend du niveau d'abstraction choisi. Un rearrangement local sur les liens entre sous-composants d'un composite n'affecte pas la double catégorie du fait que les objets de base, les interfaces, restent inchangés. Par contre, la migration d'un composant primitif met à jour les objets de base des composants source et destination par restriction des interfaces du premier et augmentation des interfaces du second. Néanmoins, les objets de base et donc la double catégorie associée à l'architecture logicielle ou le composant de niveau supérieur reste inchangé. Ainsi, il suffit de remonter vers le niveau hiérarchique supérieur pour constater que les interfaces d'un composant mobile, supprimées du composant hiérarchique source et réintégrées dans le composant destination, appartiennent toujours au composant hiérarchique de niveau supérieur. Néanmoins, la migration d'un composant altère le comportement des composants hiérarchiques source et destination puisque leur interfaces sont mis à jour. Fort heureusement, la propriété d'abstraction permet de basculer entre différents niveaux de détails selon le besoin.

Exemple 6.7. Reconfiguration dynamique de PS et son interprétation

La sémantique associée au composant hiérarchique serveur primaire dans l'exemple 6 peut être exploitée pour modéliser des opérations de reconfiguration à différents niveaux d'abstraction.

Différents scénarios de reconfiguration locales peuvent avoir lieu dans le composant hiérarchique PS par redirection de certains liens sans modification de la distribution de ses interfaces. Il est évident de noter que cette opération préserve le modèle associé du fait que l'ensemble d'objets communs O_{PS} et les morphismes de base dans V_{PS} (les

actions correspondantes) restent inchangés. Seuls les morphismes de H_{PS} seront altérés, exprimant le fait que l'espace des états a changé et par conséquent certaines branches d'exécution peuvent ou non être explorées dans la situation actuelle.
Si l'on considère maintenant l'interprétation sémantique de la migration d'un client de PS vers Delay ou Stats, nous pouvons noter qu'il suffit de raisonner au niveau système Client/Serveur afin de la traiter comme une reconfiguration locale des liens des interfaces.

6.4 Conclusion

Grâce à l'introduction des interactions dans le processus de réécriture en décorant les règles non seulement par les effets de bord de la réécriture mais aussi par les conditions contextuelles nécessaires à leur application, la logique des tuiles nous a permis de définir un cadre sémantique formel pour une construction modulaire et réactive de systèmes à code mobile. Cette construction modulaire et hiérarchique va au delà de la structure pour définir le comportement global comme une composition des comportements locaux des composants et ce grâce à la notion d'interfaces et l'opérateur de composition horizontale. Ce dernier a une importance particulière car il a un double rôle : D'une part, il permet de définir des schémas d'assemblage dynamiques entre les interfaces des composants dans la vue structurelle et d'autre part, il sert à propager les effets de bord des changements locaux sur l'architecture entière dans la vue comportementale. Cette caractéristique a été exploité pour définir une approche commune, à base de règles, pour décrire les deux types de dynamique. Les interfaces constituent la structure de base à travers laquelle les effets de bord de chacun des deux types de dynamique sont interceptés.

Conclusion générale

Avec la vulgarisation des réseaux de communication, la mobilité du code est devenue une propriété inhérente aux systèmes logiciels actuels. La mise en oeuvre de cette propriété nécessite l'emergence de la structure du système à spécifier dès les premières phases de son cycle de développement, tout en assurant la séparation des préoccupations entre le calcul et l'interaction. Les architectures logicielles sont un paradigme qui répond bien à ces exigences. La plupart des langages de description des architectures logicielles ont été étendus pour supporter la reconfiguration dynamique en général, seul Comm-Unity propose des primitives propres à la mobilité de composants logiciels. Les graphes et leur règles de transformation ont aussi été utilisés pour la description d'architectures logicielles dynamiques. Et là aussi seuls les bigraphes ont été définis pour les applications à code mobile. Les modèles à base de graphes ignorent généralement les aspects liés au comportement des composants de l'architecture, ce qui restreint l'utilisation de ces modèles à la première phase de conception des systèmes logiciels.

Un cadre sémantique formel, basé logique des tuiles, a été défini dans ce travail pour la spécification d'architectures logicielles mobiles. Une approche hiérarchique et compositionnelle a été adoptée pour la définition des vues structurelle et comportementale des architectures logicielles. Différentes facettes (topologie, comportement et reconfiguration) sont facilement spécifiées dans ce cadre sémantique commun.

La caractéristique principale du modèle est qu'il se base sur la notion d'interfaces de

composants pour définir les trois niveaux de spécification d'une architecture logicielle. Les niveaux architectural et hiérarchique définissent la configuration structurelle de l'architecture logicielle par un graphe montrant la répartition spatiale des interfaces entre composants primitifs ou hiérarchique. Le niveau dynamique de l'architecture définit les changements possibles sur les deux types de configuration ; comportementale et structurelle. Des actions observables sont tout d'abord rattachées aux interfaces précisant les évolutions possibles des valeurs de ces interfaces. Ensuite, les changements locaux possibles sur les composants sont définis par un ensemble de tuiles exprimant les effets des actions sur les états locaux de ceux-ci, ainsi que les éventuels effets de bord sur leur environnement. Les transitions entre les différentes configurations structurelles d'une architecture logicielle ou un composant hiérarchique sont contrôlées par des tuiles génériques qui montrent les conditions et les effets de bord des opérations de reconfiguration. Les interactions entre les différentes vues d'une architecture logicielle sont réalisées à travers la structure commune construite autour des interfaces.

Le modèle sémantique associé à une architecture logicielle reconfigurable et mobile est une double catégorie générée sur la base d'un système de tuiles résultant de la translation des différents concepts de base de MoSAL en logique des tuiles. La catégorie horizontale associe une interprétation sémantique aux configurations structurelles d'une architecture logicielle et la catégorie verticale décrit les évolutions possibles des interfaces appartenant à ces configurations .

Notre travail future se focalisera sur l'exploitation de la projection des concepts de la logique des tuiles vers la logique de réécriture, définie dans (Meseguer et al., 1997)et brièvement présenté dans l'annexe A, pour traduire les concepts syntaxiques et sémantiques de MoSAL dans la logique de réécriture. Des stratégies de contrôle seront aussi utilisées pour guider la réécriture et restreindre l'espace d'état d'une étape de réécriture afin

d'interpréter la notion de contexte et effets de bord de la logique des tuiles.

Cette projection n'est pas une finalité en soit, mais elle sera exploitée pour vérifier certaines propriétés inhérentes à la mobilité dans l'environnement Maude et son modèle checker.

Annexe A

Rappels sur la théorie des catégories

A.1 Introduction

La première publication sur la théorie des catégories a été réalisée par Eilenberg et Mac Lane en 1945 (Eilenberg et al., 1945), son objectif était de décrire (i)les interactions et les comparaisons possibles dans un contexte donné(espaces topologiques, les groupes et d'autres structures algébriques) et (ii) les interactions entre différents contextes dans le domaine des mathématiques pures. Contrairement à la théorie des ensembles qui repose sur le seul concept d'objet (élément), la théorie des catégories rajoute la notion d'arcs ou morphismes décrivant les liens entre ces objets. Et de ce fait, elle est rapidement devenue un langage fournissant les outils nécessaires pour décrire des structures communes entre différents contextes.

L'utilisation de la théorie des catégories dans l'informatique est justifiée par l'importance des liens entre éléments d'un système ou programme par rapport aux détails d'implementation d'un objet particulier. L'accent est mis sur les relations et connexions de l'objet avec son environnement. Cette abstraction facilite l'analyse et la vérification de propriétés. En particulier :

1. Certains foncteurs entre catégories offrent une définition naturelle des morphismes de simulation entre systèmes corresponds.
2. Dans les catégories aux sens restreint(les objets sont des états et les arcs sont des calculs), le diagramme de commutation définit des comportements équivalents.
3. Dans catégories au sens large (les objets sont des catégories et les arcs sont foncteurs de simulation), un isomorphisme peut définir une équivalence entre modèles.
4. Les constructions universelles (adjonction, reflection, etc.) dans une catégorie peuvent être utilisées pour définir la notion de modèle optimal.
5. Les constructions limites et colimites (produit, coproduit, pushout, pullback, etc.) dans une catégorie peuvent donner une interprétation théorique aux différentes compositions possibles.

A.2 Définitions

Les catégories peuvent être considérées, dans le domaine de l'informatique, comme une généralisation des systèmes de transitions, dont les états sont des objets et les transitions sont arcs. Ces derniers sont dotés d'une opération de composition partielle $\_;\_$, associative et admet des identités. Cette opération correspond intuitivement à la composition des calculs, alors que les identités représentent des composants oisifs dans le système à spécifier.

A.2.1 Catégorie

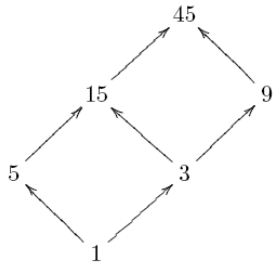
Avant de présenter les définitions formelles et abstraites sur les catégories, un exemple simple mais illustratif semble nécessaire. L'exemple, inspiré de (Brown et al., 2007), représente les diviseurs d'un nombre par une structure catégorielle.

Exemple 1.

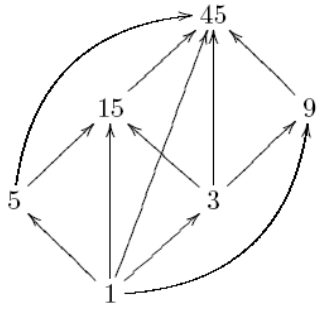
Les diviseurs du nombre 45 constituent un ensemble :

$$Div(45) = \{1, 3, 5, 9, 15, 45\}$$

Si la relation de diviseurs entre éléments de l'ensemble $Div(45)$ est significative, alors la représentation basée sur les ensembles n'est pas suffisante. Un diagramme contenant les diviseurs de 45 et leurs interrelations (i est relié à j si i divise j), sera plus représentatif.



Il faut noter que seuls les arcs générateurs sont représentés sur ce diagramme, des arcs résultants de la transitivité de la relation *diviseur* et construits par composition des arcs générateurs, peuvent être rajoutés pour obtenir un diagramme plus complexe.



En rajoutant une boucle sur chaque noeud i, du fait que chaque nombre est diviseur de lui même, une catégorie est alors obtenue.

Une catégorie est alors constituée d'une collection d'objets, une collection d'arcs orientés ou liens ou morphismes selon le contexte dans lequel cette structure est utilisée. les arcs sont munis d'une loi de composition qui est associative et admet un arc identité pour chaque objet. formellement, une catégorie est définie par

Définition A.2.1. Une catégorie $\mathcal{C}$ est constituée de :

- une collection d'objets O_c ; notés $(a, b, c, ...)$;
- une collection d'arcs $\mathcal{A}_\mathcal{C}$; notés $(f,\ g, ...)$ et munis de la structure suivante :
  - Chaque arc f a un domaine de définition $dom(a)$ constitué de l'ensemble des objets source de l'arc et un co-domaine $cod(a)$ constitué de l'ensemble des objets destination de celui-ci. On écrit :

    $$f : a \longrightarrow b \,\text{ou bien}\, a \xrightarrow{f} b$$

    Si a est le domaine de f et b est le co-domaine de f,
  - composition d'arcs : Étant donné deux arcs f et g, tel que $cod(f) = dom(g)$, la composition de f et g ; notée $f; g$, est un arc composite avec $dom(f; g) = dom(f)$ et $cod(f; g) = cod(g)$.

    $$a \xrightarrow{f} b \xrightarrow{g} c = a \xrightarrow{f;g} c$$

    ,
  - associativité : l'opérateur de composition est associatif

    $$f; (g; h) = (f; g); h$$

    si toutefois f, g, h vérifient la condition de composition d'arcs,
  - identié : pour chaque objet a, il existe un arc identité $id_a : a \longrightarrow a$, tel que pour tout $f : a \longrightarrow b$, alors

    $$1_a; f = f = f; 1_b$$

    .

Exemple 2 Quelques exemples de catégories

- La catégorie des ensembles (**Set**) : les objets de **Set** sont des ensembles et les morphismes sont des fonctions munies de l'opération de composition des fonctions et la fonction identité.

- La catégorie des graphes (**Graph**) avec les objets representant des graphes, notés $(V,\, E,\, s,\, d)$, et les morphismes sont des transformations $f\colon V \longrightarrow V'$, $g\colon E \longrightarrow E'$, telle que :

$$\forall a \in E, f(s(a)) = s'(g(a))$$

- La catégorie des monoides (**Mon**) : les objets sont des monoides (ensemble doté d'une opération binaire, associative et ayant un élément neutre e) et les morphismes sont les homomorphismes des monoides.

Définition A.2.2. La collection de morphismes d'un objet A vers un objet B dans une catégorie $\mathcal{C}$ est dite *Homset* et est notée $\mathcal{C}[A,\, B]$.

Définition A.2.3. Une catégorie $\mathcal{A}$ est une sous-catégorie de $\mathcal{C}$, si :

- $O_{\mathcal{A}} \subseteq O_{\mathcal{C}}$,
- $\forall A,\, B \in O_{\mathcal{A}}$, $\mathbf{A}[A,\, B] \sqsubseteq \mathbf{C}[A,\, B]$,
- la composition et les identités dans $\mathcal{A}$ coincident avec celles de $\mathcal{C}$.

A.2.2 Foncteur

Étant donné que les catégories sont elles mêmes des entités mathématiques, les morphismes entre de telles structures peuvent jouer un rôle important. En considérant les catégories comme objets, un morphisme entre deux catégories ; appelé foncteur, est alors une transformation préservant les objets et les morphismes entre ces objets.

Définition A.2.4. Soient $\mathcal{C}$ et $\mathcal{D}$ deux catégories, un foncteur $\mathrm{F}\colon \mathcal{C} \longrightarrow \mathcal{D}$ est une paire d'opérations $(F_O,\, F_A)$, tels que :

- $F_O\colon O_{\mathcal{C}} \longrightarrow O_{\mathcal{D}})$; $\mathrm{F_O}$ associe à chaque objet de $\mathcal{C}$ un objet de $\mathcal{D}$,
- $F_A\colon A_{\mathcal{C}} \longrightarrow A_{\mathcal{D}}$; $\mathrm{F_A}$ associe à tout arc de $\mathcal{C}$ un arc de $\mathcal{D}$,
- pour chaque $f\colon a \longrightarrow b \in \mathcal{C}$, alors $F_A(f)\colon F_O(a) \longrightarrow F_O(b)$,

– pour chaque $f\colon a \longrightarrow b$, $g\colon b \longrightarrow c \in \mathcal{D}$, alors $F(f;g) = F(f);F(g)$,

– pour chaque objet $a \in \mathcal{C}$, then $F(id_a) = id_{F(a)}$

Exemple 3

1. Dans la catétorie **Cat** de toutes les catégories, les objets sont des catégories et les morphismes représentent les différents foncteurs entre ces catégories.

2. Le foncteur **Hom** défini par $C[-,-]\colon C^{op} \times C \longrightarrow Set$ qui affecte à chaque paire d'objets $(A\,B)$ l'ensemble $C[A,\,B]$ et à chaque paire de morphismes $f\colon A \longrightarrow A'$ et $g\colon B \longrightarrow B'$ la fonction $C[f,\,g]\colon C[A,\,B] \longrightarrow C[A',\,B']$ transformant chaque arc $h\colon A \longrightarrow B$ en $f \circ h \circ g\colon A' \longrightarrow B'$.

A.2.3 Transformation Naturelle

Étant donné deux catégories $\mathcal{A}$ et $\mathcal{B}$, leur Homset dans **Cat**, c-à-d., **Cat**[**A**, **C**] noté aussi $\mathbf{C^A}$ est aussi équipé d'une structure de catégorie dont les objets sont des foncteurs et les morphismes sont dits **transformations naturelles**. Formellement,

Définition A.2.5. Une transformation naturelle entre deux foncteurs $\mathbf{F}, \mathbf{G}\colon A \longrightarrow B$ est formée d'une famille $\eta = \{\eta_A\colon \mathbf{F}(A) \longrightarrow \mathrm{G}(A)\}_{A \in O_A}$ de morphismes dans B indexée par les objets de $\mathcal{A}$

A.2.4 Catégorie monoidale

Définition A.2.6. Une catégorie monoidale stricte est une triplet $(\mathcal{C},\ \_ \otimes \_,\ e)$ dont :

– $\mathcal{C}$ est une catégorie dite de base (munie d'une composition $\_;\_$ et une identité id_a pour chaque objet a),

– $\_ \otimes \_\colon \mathcal{C} \times \mathcal{C} \longrightarrow C$ est un foncteur appelé produit tensor,

– e est un objet de $\mathcal{C}$ appelé élément neutre de $\_ \otimes \_$,

– l'opérateur $\_ \otimes \_$ est associatif sur les objets et les arcs.

A.2.5 2-catégorie

Les catégories constituent un cadre de travail pour la définition de toutes les situations ayant des objets, représentant des points dans l'espace, et des 1-morphismes entre ces objets, décrivant les chemins entre les points de l'espace. Il est alors possible de généraliser cette définition aux n-catégories en élargissant la notion de morphisme au 2-morphismes (chemins entre chemins), 3-morphismes, etc.
Formellement, les 2-catégories sont des cas particuliers des catégories enrichies, basées sur l'idée suivante :

> Étant donné une catégorie monoidale **V**, une catégorie enrichie à partir de V et notée **V**-catégrie est une catégorie contenant les objets de **V**. Pour chaque paire d'objets A et B de **V**, le homset **C**[**A**, **B**] constitue un objet de cette catégorie en ayant toujours des morphismes qui vérifient les axiomes d'une catégorie.

Définition A.2.7. Une 2-catégorie $\mathcal{C}$ est définie par :

– un ensemble d'objets $O_{\mathbf{C}}$ ou **0-cellules**,
– une catégorie $Mor_{\mathbf{C}}(A, B)$ est associée à chaque couple d'objets A et B avec :
  – les objets sont des morphismes de A vers B ou **1-cellules** ; notés $(f, g, ...)$,
  – les arcs sont des 2-morphismes ou 2-cellules ; notés $(\alpha, \beta, ...)$,
– des foncteurs de composition $\circ_-$: $Mor_{\mathcal{C}}(A, B) \times Mor_{\mathcal{C}}(B, C) \longrightarrow Mor_{\mathcal{C}}(A, C)$,
– un morphisme identité $1_A \in Mor_C(A, A)$ est associé à chaque objet A.

La 2-catégorie vérifie les propriétés usuelles d'une catégorie.

Remarque A.2.1. 1. Les fonctions source et puits appliquées aux objets 1-cellules donnent des 0-cellules et les fonctions source et puits appliquées aux objets 2-cellules donnent des 1-cellules.

2. Pour définir une 3-catégorie, il suffit de prévoir des structures 3-cellules (ou 3-morphismes $\psi\colon \alpha \longrightarrow \beta$).

A.2.6 Double catégorie

Définition A.2.8. Une double catégorie **D** est constituée de :

- un ensemble d'objets,
- un ensemble de morphismes horizontaux,
- un ensemble de morphismes verticaux,
- et une classe de doubles cellules ; graphes carrés (voir Figure A.1) dont les fonctions source et puits sont définies comme suit :

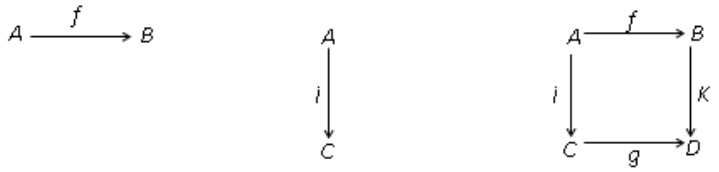


FIGURE A.1 – Cellule dans une double catégorie

avec les fonctions de composition, horizontale et verticale, et les identités associées vérifient les axiomes usuels d'une catégorie, c.à.d. :

- les objets et les morphismes horizontaux forment une 1-catégorie **H** dont l'opération de composition est $\_ * \_$,
- les objets et les morphismes verticaux forment une 1-catégorie **V** dont l'opération de composition est $\_ \cdot \_$,
- les doubles cellules peuvent être composées horizontalement grâce à l'opérateur $\_ * \_$ et verticalement grâce à l'opérateur $\_ \cdot \_$. Ces cellules forment une catégorie

horizontale $\mathbf{D}^*$ et une catégorie verticale $D^{\cdot}$

A.3 Translation de la logique des tuiles dans la logique de réécriture

Les 2-catégories sont un cas particulier des doubles catégories, en considérant les morphismes verticaux comme des identités sur les objets. Du fait que le modèle sémantique associé à une théorie de réécriture est une 2-catégorie et que le modèle sémantique associé à un système de tuiles est une double catégorie, alors la logique de réécriture peut être facilement intégrée dans la logique des tuiles.

Bien que la logique des tuiles est assez expressive pour spécifier des systèmes ouverts et réactifs, elle souffre du manque d'outils et de langages d'implémentation pour exécuter et vérifier les spécifications basées systèmes de tuiles.

Or la logique de réécriture constitue une base sémantique pour plusieurs langages d'implémentation (maude, cafeOBJ, etc.), malgré qu'elle est quelque peu restrictive du fait qu'elle ne permet que la spécification de systèmes clos. Le langage Maude en particulier offre un environnement complet pour l'edition, l'execution et la vérification des théories de réécriture.

Il est alors plus intéressant d'établir un lien dans ce sens c.à.d., relier plutôt la logique des tuiles à la logique de réécriture pour obtenir un cadre exécutable des spécifications écrites dans cette logique. Et de ce fait, l'exploitation du pouvoir d'expression de la logique des tuiles se fera à la phase de spécification de systèmes concurrents, alors que l'environnement d'exécution de la logique de réécriture sera exploité à la phase d'implementation et de vérification.

J. Meseguer et U. Montanari ont établi une translation de la logique des tuiles à la

logique de réécriture (Meseguer et al., 1997). En réalité, les auteurs ont intégré les deux modèles dans un cadre sémantique commun qui est la logique équationnelle d'appartenance partielle (Meseguer, 1996). Les propriétés intrinsèques à cette logique (partialité, sous-sorts et les assertions d'appartenance) offrent un cadre sémantique naturel pour la spécification de structures catégorielles, telles que les doubles catégories pour la logique des tuiles et les 2-catégories pour la logique de réécriture.

Meseguer et al., associent des théorie équationnelle d'appartenance aux concepts propres aux doubles catégories (objets, morphismes horizontaux, morphismes verticaux) et aux concepts propres aux 2-catégories (objets, 1-cellules, 2-cellules). L'idée de base est de ramener les double-cellules à des 2-cellules ordinaires ayant les mêmes domaines et co-domaines, en étant capable toujours de distinguer entre les configurations(morphismes horizontaux) et les observations ou effets (morphismes verticaux). Cette séparation est réalisée par la définition de deux types d'arcs : horizontaux et verticaux. Les opérations de composition verticale et horizontale entre tuiles sont dérivées de celles des 2-cellules. afin d'éviter la composition de 2-cellules provoquant des calculs hors ceux définis par les tuiles, (Bruni, 1999) a rajouté une stratégie interne à Maude permettant de dériver des calculs en ne parcourant que les chemins corrects.

Annexe B

Spécification de composants CBabel dans maude

Module Déclaration

Le module DECLARATION contient les déclarations des sortes (OPortId, IPortId, PortId, ItList, Item,...), des opérations (instantiate, $\otimes$,...)et de messages permettant de générer, de déposer, de retirer et de consommer un item. Ce module sera importé et utilisé par chaque module orienté objet décrivant un composant quelconque.

(**fmod** DECLARATION **is**

protecting CONFIGURATION .

including ITEM-LIST .

subsort Qid $<$ Oid .

sorts OPortId IPortId PortId .

*** ensemble des ports d'entrées et de sorties.

subsorts OPortId IPortId $<$ PortId .

sort status .

*** Etat de la mémoire tampon .

ops empty Full notFull : $\longrightarrow$ status .

op retirer : Oid PortId $\longrightarrow$ Msg .

*** messages

msg generer : Oid ItList ⟶ Msg .

*** Messages échangés entre les composants

ops deposer consommer envoyer-data : Oid PortId Item ⟶ Msg .

msg demande : Oid ⟶ Msg .

sort configuration .

subsorts Object Msg < configuration .

op vide : ⟶ configuration .

op _ ⊗ _ : configuration configuration ⟶ configuration [ctor assoc comm id : vide] .

op instantiate : Oid Cid ⟶ Object .

endfm)

Module de déclaration d'une liste d'objets

Le module fonctionnel ITEM-LIST déclare une structure de données ITEM-LIST permettant la manipulation d'une liste d'items. Il est défini par le code suivant :

(**fmod** ITEM-LIST **is**

protecting NAT .

protecting QID .

sorts ItList Item .

subsort Item < ItList .

subsort Qid < Item .

*** Opération pour construire une liste d'items.

op nil : ⟶ ItList [ctor] .

op _._ : ItList ItList ⟶ ItList [assoc id : nil ctor] .

op _._ : ItList Item ⟶ ItList [assoc id : nil ctor] .

op length : ItList ⟶ Nat .

*** sémantique de l'opération length.

eq length(E :Qid L :ItList) = 1 + length(L :ItList) .

endfm)

Spécification du composant buffer

Le module buffer de l'exemple Producteur/Consommateur définit une classe buffer comportant :

- un attribut buff-items, indiquant le nombre de données stockés dans la mémoire de ce buffer,
- un attribut buff-maxitems indiquant la taille maximale de la mémoire tampon du buffer,
- un attribut cont constituant la liste des données envoyées par le producteur et non encore récupérées par le consommateur

Ce module comporte aussi la déclaration d'un port d'entrée inP, un port de sortie outC, et un ensemble de règles de réécriture spécifiant le comportement du buffer suite à la reception d'une données sur inP et l'envoi d'une donnée sur le port outC.

omod BUFFER **is**

including DECLARATION .

protecting INT .

class BUFFER | buff-items : Int, buff-maxitems : Int, cont : ItList .

ops _.inP _.inC : Oid → IPortId .

op _.outC : Oid → OPortId .

var buff : Oid .

vars N M : Int .

var D : Item .

var I : ItList .

*** créer une instance ' buff' de composant BUFFER.

eq instantiate(buff, BUFFER) = <buff : BUFFER|buff-items : 0, buff-maxitems : 3, cont : nil> .

*** Dépôt d'un item D reçu sur le port d'entrée 'buff @inP' dans la mémoire tampon du buffer .

crl [buff-deposer] : deposer(buff, buff@inP, D) ⊗ <buff : BUFFER|buff-items : N, buff-maxitems : M, cont : I>⇒<buff : BUFFER | buff-items : N + 1, buff-maxitems : M, cont : D I > if(N < M and length(I) == N) .

*** Retrait d'un item D de la mémoire tampon et son dépôt sur le port de sortie buff@outC.

crl [buff-retirer] : retirer(buff, buff@inC) ⊗ <buff : BUFFER | buff-items : N, buff-maxitems : M, cont : I D>⇒<buff : BUFFER — buff-items : N - 1, buff-maxitems : M, cont : I> ⊗ envoyer-data(buff, buff@outC , D) if(N > 0 and length(I D) == N) .

endom).

Spécification du composant consommateur

Ce module objet déclare une classe CONSUMER contenant un attribut tableItemC, une mémoire tampon pour le stockage des données récupérées du buffer. Il comporte aussi la déclaration d'un port d'entrée cons.in pour la reception des données et un port de sortie con.out pour l'envoi d'une demande de transfert d'une donnée du buffer vers le consommateur. Il spécifie aussi le comportement associé à l'évolution des valeurs de ports du consommateur par les règles $cons - retirer$ et $cons - consomer$.

(**omod** CONSUMER **is**

including DECLARATION .

class CONSUMER | tableItemC : ItList .

op _.in : Oid ⟶ IPortId .

op _.out : Oid ⟶ OPortId .

var cons : Oid .

var D : Item .

var T : ItList .

*** créer une instance ' cons' de composant CONSUMER .

eq instantiate(cons, CONSUMER)=<cons : CONSUMER|tableItemC : nil> .

***Envoie d'une demande de retrait d'une donnée sur le port de sortie out du composant CONSUMER.

rl [cons-retirer] : demande(cons)⊗ <cons : CONSUMER | tableItemC :T>⇒retirer(cons, cons.out) ⊗ <cons : CONSUMER | tableItemC : T> .

***Consommation de la donnée reçue sur le port d'entrée cons.in .

rl [cons-consommer] : consommer(cons, cons.in, D) ⊗ <cons : CONSUMER | tableItemC : T>⇒<cons : CONSUMER | data : D T> ⊗ demande(cons) .

endom)

Annexe C

Publications de l'auteur

C.1 Revues

- **C. Bouanaka**, F. Belala, " MOSAL : A Tile Logic Based Language for Mobile Software Applications", paru dans International Review on Computers and Software, Vol 3(6), pp. 672-680.

C.2 Conferences internationales spécialisées avec actes et comité de lecture

- **C. Bouanaka**, F. Belala and K. Barkaoui, "A Tile Logic Based Semantics for Mobile Software Architectures".In Proc of the 4^{th} International Workshop on verification and Evaluation of Computer and Communication Systems (VECOS 2010), July 1-2, 2010, Paris-France. Papier sélectionné pour être publié dans la revue "International Journal for Critical Computer based Systems (IJCCBS".
- N. Benlahreche, F. Belala, **C. Bouanaka**, M. Benamar, "Vers un Modèle de Déploiement à base de Bigraphes ", Dans les actes de la 4ieme Conférence francophone sur les Architectures Logicielles (CAL 2010), pp. 141-155, 9-12 Mars 2010, Pau - France.

– **C. Bouanaka**, A. Choutri and F. Belala, "On Generating Tile System for a Software Architecture : Case of a Collaborative Application Session", In Proc. of the Second Conference on Software and Data Technologies (ICSOFT'2007), pp. 123-128, July 22-25, 2007, Barcelona, Spain.

C.3 Conferences internationales avec actes et comité de lecture

– K. Barkaoui, **C. Bouanaka**, José Martín Molina Espinosa : An Event Structure based Coordination Model for Collaborative Sessions. In Proc of the 11th International Conference on Enterprise Information Systems (ICEIS 2009), pp. 137-143, May 6-10, 2009, Milan, Italy.

– S. Boufenara, F. Belala, **C. Bouanaka**, " Les Zero-Safe Nets pour la préservation de la TTC dans les diagrammes d'activités UML ", Paru dans les actes de la 15ieme Conférence Francophone sur les Langages et Modèles à Objets (LMO'2009), pp. 91-106, 27-29 Mars 2009, Nancy, France.

– S. Boufenara, F. Belala, **C. Bouanaka**, "Synchronisation Schema in Activity Diagrams via Zero-Safe Nets", In Proc of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'2009), pp.161-167, May 10-13, 2008, Rabat - Morocco.

– **C. Bouanaka**, Faiza Belala, " Tile logic as an architectural model for mobility". In Proc. of the IEEE Symposium on Computers and Communications (ISCC'08), pp. 525-530, July 6 - 9, 2008, Marrakech, Morocco.

– **C. Bouanaka**, Faiza Belala, " Towards a mobile architecture description language ", In Proc of the 6th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA' 2008), pp. 743-748, March 31 - April 4, 2008, Doha,

Qatar.

– **C. Bouanaka**, F. Belala, A. Choutri, " A Tile logic based Model for a Collaborative Session Application ". In Proc. of the 4th IEEE GCC 2007, November 12-14, 2007, Manama - Bahrain.

– **C. Bouanaka**, F. Belala, "On Adding Some Mobility Primitives to an Architecture Description Language ". In Proc of the 4th International Multiconference on Computer Science and Information Technology (CSIT'2006), April 5-7, 2006, Amman, Jordan .

Bibliographie

(AADL, 2006) AADL. Avionics Architecture Description Language. http ://www.sae.org/technical/ standards/AS5506, Last Consultation : October 2006.

(Abowd et al. 95) G. D. Abowd, R. Allen, and D. Garlan. (1995). Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering Methodologies, 4(4), pp. 319-364.

(Abowd, et al 93) Abwod G.D., Allen R, and Garlan D. (1993). Using style to understand descriptions of software architecture. In Proc. of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 9-20, New York, NY, USA, ACM.

(ADOBE, 1985), ADOBE Systems Incorporated. PostScript Language Reference Manual. Addison-Wesly, 1985.

(Allen et al., 1997) Allen R. and Garlan D. (1997), A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, July 1997.

(Allen et al., 1996) Allen R. and Garlan D. (1996). The wright architectural specification langage. Technical Report CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pillsburgh, PA.

(Barais, 2002) Barais O. (2002), Approche statique, dynamique et globale de l'architecture d'applications reparties. Master report, Ecole Mine de Douai, Laboratoire d'Informatique fondamentale de Lille.

(Baros, 2007)Barros F. J. (2007). Representing hierarchical mobility in software architectures. In Proc. of the International Workshop on Software Engineering for Adaptive and Self Managing Systems (SEAMS'07), IEEE, pp. 5-10.

(Bass et al, 1997) Bass, and Kazman (1997). Software Architecture in Practice, Addison-Wesley.

(Ben-Shaul et al., 2001) Ben-Shaul I., Holder O., and Lava B. (2001). Dynamic adaptation and deployment of distributed components in hadas. IEEE Transactions on Software Engineering, Volume 27(9), pp. 769-787.

(Boggs, 1973), Boggs J. (1973). IBM remote job entry facility: Generalize subsystem remote job entry facility. IBM technical disclosure bulletin 752, IBM, Aug.

(Borovanský , 1998) Borovanský P., Kirchner C, Kirchner H, Moreau P.E., and Ringeissen C.(1998). An overview of ELAN. Electr. Notes in Theoretical Computer Science 15.

(Bouanaka et al., 2010) Bouanaka C., Belala F. and Barkaoui K. (2010). A Tile Logic Based Semantics for Mobile Software Architectures. In Proc of the 4th International Workshop on Verification on Verification and Evaluation of Computer and Communication Systems (VECOS 2010), July 1-2, 2010, Paris, France.

(Bouanaka et al, 2008 a) Bouanaka C., Belala F. (2008). Tile logic as an architectural model for mobility. In Proc. of the IEEE Symposium on Computers and Communications (ISCC'08), pp. 525-530, July 6 - 9, 2008, Marrakech, Morocco.

(Bouanaka et al., 2008 b) Bouanaka C., Belala F. (2008). Towards a mobile architecture description language, In Proc. of the 6th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA' 2008), pp. 743-748, March 31 - April 4, 2008, Doha, Qatar.

(Bouanaka et al. 2008c) Bouanaka C, Belala F. (2008), MOSAL: A Tile Logic Based Language for Mobile Software Applications, International Review on Computers and Software, Vol 3(6), pp. 672-680.

(Bouanaka et al., 2007) Bouanaka C., Choutri A. and Belala F. (2007). On Generating Tile System for a Software Architecture: Case of a Collaborative Application Session, In Proc. of the 2nd Conference on Software and Data Technologies (ICSOFT'2007), pp. 123-128, July 22-25, 2007, Barcelona, Spain.

(Brown et al., 2007) Brown R., Paton R., Porter T. (2007). Categirical language and hierarchical models for cell systems.

(Bruni, et al., 2007) Bruni R, Lluch A., Lafuente J., Montanari U., and Tuosto E. (2007). Style-Based Architectural Reconfigurations, Bulletin of the European Association for Theoretical Computer Science, EATCS. 94, pp. 161-180.

(Bruni et al., 2002) Bruni R., Meseguer J., and Montanari U. (2002). Symmetric and Cartesian Double Categories as a Semantic Framework for Tile Logic. Mathematical Structures in Computer Science, Volume 12(1), pp 53-90.

(Bruni et al., 2000) Bruni R., and Montanari U. (2000). Zero-safe nets: Comparing the collective and individual token approaches. Information and computation, Volume 156, pp. 46–89.

(Canal et al, 1999) Canal C., Pimentel E., and Troya J. M. (1999). Specification and refinement of dynamic software architectures. Software Architecture, Kluwer Academia Publishers, pp. 107-126.

(Carzania et al. 1997) Carzania J., Picco G. P., and Vigna G. (1997), Designing distributed applications with mobile code paradigms, In Proc. Of the 19th Int. Conf. on Software Engineering (ICSE'97), R. Taylor, Ed.,ACM Press, pp 22-32.

(Cattani et al., 2000) Cattani, G.L., Leifer, J.J., and Milner, R. (2000), Contexts and embeddings for a class of action graphs. Tech. rept. 496. Computer Laboratory, University of Cambridge.

(Cerqueira et al., 2003) Cerqueira R., Ansaloni S., loques O., Sztajnberg A.(2003). Deploying non-functional aspects by contract. In the 2nd Workshop on Reflective and Adaptative Midleware, Midleware Companion, Rio de Janeiro, Brazil, pp. 90-94, June 2003.

(Clavel et al., 2007) Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N., Meseguer J., and Talcott C. L. (2007). All About Maude, A High-Performance Logical Framework, Lecture Notes in Computer Science, Springer Volume 4350.

(Clavel et al., 2001) Clavel M., and Meseguer J. (2001). Reflection in conditional rewriting logic. Theoretical Computer Science. Volume

(Clavel, 2000) Clavel M.. Reflection in Rewriting Logic (2000). Metalogical Foundations and Metaprogramming Applications. CSLI Publications.

(Corradini et al., 1995) Corradini A., Gadducci F., and Montanari U. (1995). Relating two categorical models of term rewriting. In Proc. of Rewriting Techniques and Applications, Kaisers lautern, pp. 225-240.

(Cugola et al., 1996) Cugola G., Ghezzi C., Picco G. P., and Vigna G (1996). A characterization of mobility and state distribution in mobile code languages , 2nd ECOOP workshop on Mobile Object Systems, pp. 19-46, Australia.

(Danianou et al., 2001) Danianou N., Dulay N., Lupu E., and Sloman M.(2001). The Ponder Policy Specification Langage. In Proc. of Policy 2001 : Workshop on Policies for Distributed Systems and Networks , Bristol, UK, 29-31 January 2001 Springer-Verlag LNCS 1995, pp. 18-39.

(Dewayne, 1992) Dewayne E. P. (1992). Foundations for the study of software architecture. Software Engineering Notes, 17(4):40.

(Diaconescu et al., 2002) Diaconescu R., and Futatsugi K.(2002). Logical foundations of CafeOBJ. TCS, 285(2), pp. 289-318.

(Djikstra, 1968) Dijkstra, E. (1968). The structure of the multiprogramming system. Communications of the ACM, 11(5), pp. 341–346.

(Eker et al., 2002) Eker S., Knapp M., Laderoute K., Lincoln P., Meseguer J., and Sönmez M. K. (2002). Pathway logic : Symbolic analysis of biological signalling. In Pacific Symposium on Biocomputing, pp. 400-412.

(Eilenberg et al., 1945), Eilenberg S., and Mac Lane S. (1945). The general theory of natural equivalences, Trans. American Mathematics Society 58, pp. 231-294.

(Fiadeiro et al., 2003) Fiadeiro J.L., Lopes A., and Wermelinger M.. (2003), A Mathematical semantics for architectural connectors. In: Backhouse, R and Gibbons, J eds. Generic programming: advanced lectures. Lecture notes in computer science (2793). Berlin, Germany: Springer-Verlag, pp. 178–221.

(Finger, 2000), Finger P. (2000). Component-Based Frameworks for E-Commerce", Communications of the ACM 43(10), pp. 61-66.

(Fong et al, 1998) Fong P.W.L, and Cameron R.D (1998). Proof-linking: An architecture for modular verification of dynamically-linked mobile code. In Proc. of the 6th ACM SIG-Soft International Symposium on the Foundations of software Engineering, pp. 222-230 Orlando, Florida.

(Fuggetta et al., 1998) Fuggetta A., Picco G.P., and Vigna G. (1998). Understanding code mobility, IEEE Transactions on Software Engeneering, Vol 24, pp..

(Garlan , 2001) Garlan D. (2001). Software architecture. Scholl of computer science Carnegie Mellon University, Encyclopedia of software engineering.

(Garlan,1994) Garlan D., and Shaw M. (1994). An Introduction to Software Architecture. CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

(Gemma et al., 1994) Gamma E., Helm R., Johnson R., and Vlissides J. (1994). Design Patterns. Professional Computing Series. Addison-Wesley, October 1994.

(Guesta, 2002) Cuesta, C. (2002). Dynamic Software Architecture based on Reflection. PhD. Thesis, University of Valladolid. (In Spanish). Coordination Development Environment download www.atxsoftware.com/CDE

(Hirsch , 1999) Hirsch D., Inverardi P., and Montanari U., (1999). Modeling software architectures and styles with graph grammars and constraint solving. In WICSA'99 : Proceedings of the 1st Working IFIP Conference on Software Architecture, pp. 127–142, San Antonio, Texas. Kluwer Academic Publishers.

(Jul et al., 1988) Jul E., Levy H., Hutchinson N., and Black A., (1988). A fine-grained mobility in the Emerald System. ACM Trans. On computer Systems 6(2), pp. 109-133.

(Kruchten, 2006) Kruchten P., Obbink H, and Stafford J., (2006). The past, present, and future for software architecture. Software, IEEE, 23(2), pp. 22–30.

(Leifer et al., 2000) Leifer J.J., and Milner R. (2000). Deriving bisimulation congruences for reactive systems.:Palamidessi, C. (ed), Proceedings CONCUR 2000, 11th International Conference on Concurrency Theory. Lecture Notes in Computer Science, Springer-Verlag, Volume 1877, pp. 243–258.

(Le Metayer, 1998 Le Métayer D. (1998). Describing Software Architecture Styles Using Graph Grammars, IEEE Transactions on Software Engineering, Volume 24(7).

(Leymonerie, 2004) Leymonerie F. (2004). ASL : un langage et des outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques. Thèse de doctorat, Université de Savoie.

(Lincoln at al., 1994) Lincoln P., Narciso M. O., and Meseguer J.. Specification, transformation and programming of concurrent systems in rewriting logic. Technical Report SRICSL- 94-11, SRI.

(Lopes et al. 2004) Lopes A., and Fiadeiro J.L. (2004). Adding mobility to software architectures. In Proc.2nd Workshop on Foundations of Coordination Languages and Software Architectures, FOCLASA'03, Electronic Notes in Computer Science, Volume 97.

(Luckman, 1995) Luckham D.C., Kenney J.J., Augustin L.M., Vera L.M., Bryan J., and Mann W. (1995), Specification and analysis of system architecture using rapide. IEEE Transactions On Software Engineering, Volume 21(4). pp. 336-354.

(Macdufie, 2001). McDufie J.H. (2001), Using the architecture description language MetaH for designing and prototyping an embedded spacecraft attitude control system. In The 20th Conference on Digital Avionics Systems, volume 2, pp. 8E3/18E3/9, Daytona Beach, Florida,USA.

(Magee et al., 1995) Magee J., Dulay N., Eisenbach S., and Kramer J., (1995). Specifying distributed software architectures. Lecture Notes in Computer Science, 989 (2), pp. 31-35.

(Mascolo et al., 1999) Mascolo C., Picco G. P., Roman G. C. (1999). A Fine Grained Model for Code Mobility. In Proc. of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 39–56, Toulouse, France.

(Martio-Olietet al., 1993) Martio-Oliet N., and Meseguer J. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory.

(Medvidovic et al., 2000). Medvidovic, N., and Taylor, R.. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, Volume 28(1).

(Medvidovic et al, 1999) Medvidovic N., Rosenblum D.S., and Taylor R.N. (1999), A language and environment for architecture-based software development and evolution. In the 21st International Conference on Software Engineering, pages 44_53, Los Angeles, California,USA.

(Meseguer, et al., 2007) Meseguer J., and Rosu G. (2007). The rewriting logic semantics project. Theroretical Computer Science, 373(3), pp. 213-237.

(Meseguer et al, 2002) Meseguer J., Ölveczky P. C., Stehr M. O., and Talcott C. L.. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In DANCE, pp. 494-510. IEEE Comp. Soc.

(Meseguer et al., 1997) Meseguer J., and Montanari U. (1997), Mapping tile logic into rewriting logic, Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques, LNCS-1376, pp. 62-91.

(Meseguer, 1996), Meseguer J. (1996), Membership algebra. Lecture and abstract at the Dagstuhl Seminar on “Specification and Semantics”, July 9, 1996.

(Milner, 2008) Milmer R. (2008). Bigraphs: a space for interaction, available at http://www.cl.cam.ac.uk.

(Milner, 1992) Milner R. (1992). The polyadic pi-calculus (abstract). Page 1 of Cleaveland, R. (ed), Proceedings CONCUR '92, 3rd International Conference on Concurrency Theory. Lecture Notes in Computer Science, volume 630. Springer Verlag.

(Montanari et al., 1992) Montanari U., and Sassone V. (1992). Dynamic congruence vs. progressing bisimulation for CCS. Fundamenta informaticae,, volume 16, pp. 171–196.

(MOO MENA , 2007) MOO MENA F. J. (2007). Modélisation Des Architectures Logicielles Dynamiques, Application à la gestion de la qualité de service des applications à base de services Web. Thèse de Doctorat, Institut National Polytechnique de Toulouse.

(Moriconi et al., 1997) Moriconi M., and Riemenschneider R. (1997). Introduction to SADL 1.0. Technical Report SRI-CSL-97-01, SRI Computer Science Laboratory.

(Parnas, 1972) Parnas,D.L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), pp. 1053–1058.

(Pfender , 1974) Pfender M. (1974). Universal algebra in s-monoidal categories. Technical Report 95-22, University of Munich, Departement of Mathematics.

(Picco, 1998) Picco G.P. Understanding, Evaluating, Formalizing, and Exploiting code mobility. PhD thesis.

(Rademaker et al., 2005) Rademaker A., Braga C., Sztajnberg A. (2005), A Rewriting Semantics for a Software Architecture Description Language. Electronic Notes in Theoretical Computer Science, Volume 130, pp. 345-377.

(Ren et al., 2003) Ren Y., Bakken D., Courtney T., Cukier M., Karr A., Rubel P., Sabnis C., Sanders W., Schantz R., and Seri M. (2003). AQUA : An adaptative architecture that provides dependable distributed objects. IEEE Transactions on Computers, Volume 52(1), pp. 31-50.

(Rensink, 2000) Rensink A. (2000). Bisimilarity of open terms. Information and computation, Volume 156, pp. 345–385.

(Roman et al., 2000) Roman G., Picco, G., Murphy, A. (2000). Software Engineering for Mobility: A Roadmap. In Proc. of the International Conference on Software Engineering – Future of SE Track (Limerick, Ireland), pp. 241-258.

(Salzman et al., 1996) Salzman C., and Schoenmakers M., (1996). Dynamics and Mobility in Software Architecture.

(Sewell, 1998) Sewell P. (1998). From rewrite rules to bisimulation congruences. Sangiorgi D., & DeSimone R. (eds), Proceedings CONCUR'98. Lecture Notes in Computer. Science, Volume 1466, pp. 269–284, Springer Verlag.

(Shaw et al., 1996) Shaw, M., and Garlan, D. (1996), Software Architecture: Perspective on an Emerging Discipline. Prentice Hall, 1996.

(Shaw et al., 1995) Shaw M., DeLine R., Klein D., Ross T., Toung D., and Zelesnik G., (1995). Abstraction for Software Architecture and Tools to Support Them AA. IEEE Transactions in Software Engineering, SE-21(4), pp.314–335.

(Taylor et al., 1996) Taylor R., Medvidovic N., Anderson K., Whitehead E. Jason Jr., and Robbins E.. (1996). A Component and Message-Based Architectural Style for GUI Software, Department of Information and Computer Science, University of California, Irvine.

(Vestal, 1993a)Vestal S. (1993). A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center.

(Vestal et al., 1993b) Vestal S., and Binns P.. (1993), Scheduling and communication in MetaH. In Real-Time Systems Symposium, pp. 194-200, Raleigh Durham, North Carolina, USA.

(Wolf et al., 2002) Wolf A., Heimbigner D., Knight J., Devanbu P., Getz M., and Carzaniga A.(2002). Bend, Don't Break : Using reconfiguration to achieve survivabibility. In The 3rd Information Survivability workshop, Boston,Ma,USA. Kluwer Academic Publishers, ISBN : 1-4020-7043-8.

Abstract

In order to adopt a software architecture driven approach to specify mobile systems, we define a unified semantic framework for specifying the bidirectional, steady state and topology, evolution of such type of systems. Since eventual changes on architecture topology may have side effects on the ongoing computations, a unified structure and semantic basis are adopted to specify the two types of dynamism. Interfaces constitute the common structure on which side effects are perceptible. A rule based approach is used to define possible changes on the structure and state. Different facets (topology, behaviour, and reconfiguration) of mobile systems are then easily defined within the same semantic framework. Interactions between the various views are captured via interfaces.

Validating syntactic construction defined by the model is a necessary task. The semantic model associated to software architectures specified in MoSAL is also constructed. It consists of a double category with visible interfaces as basic objects. Horizontal category models software architecture possible configurations with components and links as basic morphisms, vertical category models both computation evolution and reconfiguration actions on components interfaces.

ملخص

من اجل انتهاج مسلك موحد لإنتاج أنظمة برامج متحركة متمحورة حول هندسته, نقترح إطار عمل موحد لوصف التطور الثنائيّ اتّجاه الوارد حدوثه في طوبولوجيا و حالة النظام. بما أنّ التغيرات الطوبولوجية قد تؤثر علي الحساب و العكس فان بنية و أساس دلاليّ مشترك تم استعمالهما لتعريف النوعين من الحركية. الواجهات تشكل البنية الأساسية المشتركة التي يتم من خلالها تبادل المعلومات حول التغييرات الحادثة في كلتي النوعين من التغيير. ثلاث مستويات من الوصف يمكن استعمالها الطبولوجيا و الحساب و تغير بنية النظام البرنامجي. هناك عمل إضافي لإرفاق معني لكل المصطلحات اللغوية المقترحة.