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

N° d'ordre : 319

Série : 014

M E M O I R E

Pour obtenir le diplôme de :

Magister en Informatique

Option: Systèmes d'Information et Intelligence Artificielle Distribués

Thème

Expression et Vérification des Contraintes non Fonctionnelles d'une Architecture SADL

Présenté par :

M^r^ LATRECHE FATEH

Dirigé par :

Dr. F. BELALA Maître de Conférence – Université Mentouri Constantine

Devant le Jury composé de :

- Président : Pr. Z. BOUFAIDA Professeur – Université Mentouri Constantine
- Rapporteur : Dr. F. BELALA Maître de Conférence – Université Mentouri Constantine
- Examinateur : Dr. N. ZAROUR Maître de Conférence – Université Mentouri Constantine
- Examinateur : Dr. S. CHIKHI Maître de Conférence – Université Mentouri Constantine

Soutenu le : 10/12/2007

Remerciements

Je remercie tout d'abord Dr BELALA FAIZA, Maître de Conférence à l'université mentouri de Constantine pour m'avoir encadré, conseillé et soutenu pendant toute ma durée de travail.
Je tiens à exprimer mes plus sincères remerciements aux membres du jury qui m'ont fait l'honneur de participer à ma soutenance.
Je tiens également à remercier tous les membres du laboratoire LIRE.

Résumé

L’étape la plus importante dans la démarche de développement des systèmes logiciels est la conception architecturale, elle consiste à définir la structure du logiciel, c'est-à-dire déterminer précisément comment le système global est partitionné en composants et quelles sont les fonctions offertes par chaque composant. Dans ce contexte de nombreux langages de description d’architectures (ADLs) ont été proposés par divers groupes industriels et académiques, ces langages offrent un modèle de haut niveau d’abstraction indépendant des détails techniques d’implémentation. Il s’approche du modèle mental du développeur.

Une caractéristique importante des ADLs est qu’ils fournissent des capacités complémentaires pour le développement et l’analyse architecturale. Néanmoins, leur prise en compte des contraintes non fonctionnelles liées aux systèmes est incomplète ou mal traitée.

L’objectif de ce travail est d’exploiter l’expressivité et la puissance de la logique de réécriture via son langage Maude pour fournir au langage de description d’architecture SADL la possibilité de décrire et de vérifier les propriétés non fonctionnelles.

La logique de réécriture est reconnue comme cadre unificateur de plusieurs langages de programmations et modèles de concurrences (réseaux de Petri, logiques des processus, …etc). Dans ce contexte nous proposons un cadre sémantique extensible basé logique de réécriture pour décrire tous les concepts de base du langage SADL. Ensuite nous étendons ce cadre formel par l’intégration d’autres constructions permettant de spécifier les propriétés non fonctionnelles liées aux composants et aux connecteurs d’une architecture SADL.

Mots Clés: Langage de Description d’Architectures, propriétés non fonctionnelles, Logique de Réécriture, Le langage Maude.

Abstract

The most important step in the approach of software systems development is the architectural conception, it consists in software structure definition that is to determine exactly how the global system is divided in sub-components and which functions are offered by each sub-component.

In this context numerous architecture description languages (ADLs) were proposed by industrial and academic groups, these languages offer a high-level abstraction model independent of technical implementation details. It approaches the mental model of the developer.

An important ADLs characteristic is that they provide complementary capabilities for the architectural development and analysis. Nevertheless their taking into account of the nonfunctional constraints related to the systems incomplete or is badly treated.

The objective of this work is to exploit rewriting logic power and expressivity to allow the architecture description language SADL describing and checking the non-functional properties.

Rewriting logic is known as a unified framework of several programming languages and models of competitions (Petri nets, logics of processes, etc.).

In this context we propose an extensible semantic framework based on rewriting logic to describe all the basic concepts of the SADL language. Then we extend this formal framework by the integration of other constructions allowing the specification of non-functional properties related to components and connectors of SADL architecture.

Key words: Architecture Description Language, non-functional properties, Rewriting Logic, Maude language.

Table des Matières

INTRODUCTION GENERALE

1 CONTEXTE ET MOTIVATION 1

2 PROBLEMATIQUE 2

3 CONTRIBUTION 3

4 CONTENU DU MEMOIRE.....4

CHAPITRE 1 : ETAT DE L'ART SUR LES ADLS

1 INTRODUCTION 5

2 LES CONCEPTS CLES DES ADLS 7

2.1 LE COMPOSANT 7
2.2 LE CONNECTEUR 10
2.3 LA CONFIGURATION.....12

3 INTERETS DES ADLS.....13

4 DESCRIPTION DES PRINCIPAUX ADLS 14

4.1 UNICON.....14
4.2 RAPIDE.....17
4.3 WRIGHT.....19
4.4 OLAN 20
4.5 ACME 21

5 COMPARAISON ENTRE LES ADLS.....23

5.1 LE COMPOSANT 24
5.2 LE CONNECTEUR 25
5.3 LA CONFIGURATION.....27

6 CONCLUSION.....29

CHAPITRE 2 : LE LANGAGE SADL

1 INTRODUCTION 32

2 PRESENTATION DU LANGAGE SADL 33

2.1 L'ARCHITECTURE.....34
2.2 LES STYLES ARCHITECTURAUX.....35
2.3 LE SYSTEME DE TYPE DANS SADL 36

3 UN EXEMPLE D'UNE ARCHITECTURE SADL 37

3.1 LE PREMIER NIVEAU DE L'ARCHITECTURE DU COMPILATEUR 38
3.2 LE DEUXIEME NIVEAU DE L'ARCHITECTURE DU COMPILATEUR 41

4 LE MECANISME DE RAFFINEMENT DANS SADL 43

5 SEMANTIQUE FORMELLE DU LANGAGE SADL 46

6 CONCLUSION 47

CHAPITRE 3 : LA LOGIQUE DE REECRITURE ET LE LANGAGE MAUDE

1 INTRODUCTION 49

2 LA LOGIQUE DE REECRITURE 49

2.1 LA THEORIE EQUATIONNELLE 51
2.2 LA THEORIE DE REECRITURE 52

3 LE LANGAGE MAUDE 55

3.1 COREMAUDE 57
3.2 FULL MAUDE 64

4 CONCLUSION 66

CHAPITRE 4 : UN CADRE SEMANTIQUE FORMEL POUR LE LANGAGE SADL

1 INTRODUCTION 67

2 MODELE SEMANTIQUE D'UNE ARCHITECTURE SADL 68

2.1 LA FORMALISATION DES ELEMENTS ARCHITECTURAUX 69
2.2 LA FORMALISATION DE LA RELATION DU RAFFINEMENT 73
2.3 ANALYSE D'UNE ARCHITECTURE SADL 76

3 INTERET DU MODELE PROPOSE 81

4 CONCLUSION 81

CHAPITRE 5 : INTEGRATION DES PROPRIETES NON FONCTIONNELLES DANS LE LANGAGE SADL

1 INTRODUCTION 83

2 TRAVAUX ANTERIEURS 85

3 EXTENSION DU MODELE SEMANTIQUE AVEC L'ASPECT NON FONCTIONNEL 87

3.1 LA SPECIFICATION 88
3.2 LA VERIFICATION 90

4 CONCLUSION.....92

CONCLUSION GENERALE 93

BIBLIOGRAPHIE 95

Introduction Générale

1 Contexte et motivation

A partir des années 90 une nouvelle démarche de développement de logiciels a émergé, il s'agit de l'utilisation de la conception architecturale dans le processus de développement de logiciel. La notion de l'architecture d'un logiciel est de plus en plus largement adoptée par la communauté scientifique et industrielle de l'informatique, en effet son utilisation offre les avantages suivants :

- Ø Amélioration de la qualité du logiciel développé;
- Ø Réduction du coût de développement;
- Ø Minimisation du temps de développement;
- Ø Faciliter la maintenance et l'évolution;
- Ø Diminution de la charge de travail.

Vu l'importance de la description de l'architecture d'un logiciel, différents langages de description d'architectures (ADL) ont été développés par différents organismes et équipes de recherches, le but essentiel de ces ADLs est de fournir des notations permettant de définir et d'exprimer la structure haut niveau de l'architecture du logiciel en termes d'une collection de composants et de connecteurs et de leur structure d'interconnexion globale [17], parmi ces ADL nous retrouvons : SADL [26], Wright [29], ACME [27], Rapide [30], Aesop [33].

Certains ADL existants permettent la spécification formelle et la vérification des descriptions architecturales. En se basant principalement sur des algèbres de processus comme le π-calcul, CSP ou FSP, ces ADL supportent l'expression et l'analyse du comportement des composants, des connecteurs et des topologies (configurations) d'architectures. En contrepartie ces ADL sont plus complexes à utiliser, ils font recours à plusieurs formalismes, et sont parfois jugés insuffisants pour décrire formellement le bon fonctionnement d'architectures de logiciels.

L'expérience a montré que les formalismes sémantiques utilisés actuellement présentent des limites concernant la formulation de certains concepts inhérents aux ADLs tels que la synchronisation et la connexion dynamique entre les composants architecturaux.

Le langage SADL (Structural Architecture Description Langauge) est un langage de description d'architectures proposé par le laboratoire SRI aux Etats-Unis [26]. Il est basé comme tous les autres ADL sur les concepts de composant, de connecteur et de configuration. La particularité de ce langage est qu'il est dédié à la description structurale des hiérarchies d'architecture à différents niveaux d'abstractions grâce à un mécanisme de raffinement explicite. Sa sémantique se base sur la ω-logic qui est équivalente en puissance à une faible logique de deuxième ordre.

Dans le cadre de ce travail, nous reprenons la sémantique de ce langage dans un formalisme plus adéquat qui est la logique de réécriture.

La logique de réécriture, proposée par José Meseguer [31,38], est considérée comme un formalisme unificateur de concurrence dans laquelle plusieurs modèles bien connus des systèmes concurrents sont intégrés (le lambda calcul, les systèmes de transitions étiquetés, les réseaux de Petri, la machine chimique abstraite, CCS, LOTOS, E-LOTOS [11]).

Elle constitue une conséquence des travaux de Meseguer sur les logiques générales pour décrire les systèmes informatiques et particulièrement les systèmes concurrents.

Les énoncés de base de cette logique sont appelés règles de réécriture et ont la forme : $t \rightarrow t'$ if C, où t et t' sont des termes algébriques décrivant un état partiel du système concurrent. Une règle de réécriture dans ce cas, décrit un changement d'un état partiel vers un autre si une certaine condition C est vérifiée.

La logique de réécriture bénéficie aussi de la présence de nombreux langages et environnements opérationnels, parmi eux nous citons Maude au USA, CafeOBJ au Japon et ELAN en France. Le langage Maude défini par J Meseguer [09] est une des implémentations les plus performantes de la logique de réécriture, c'est un langage déclaratif de haut niveau et un système de haute performance, il constitue la seule implémentation de la logique de réécriture qui emploie systématiquement et d'une manière efficace le mécanisme de réflexivité. En plus, Maude est multi paradigme, il supporte et combine la programmation fonctionnelle, système et orienté objet.

Un autre aspect favorisant l'utilisation de ce langage est la présence de nombreux outils permettant de faciliter la tâche de vérification des systèmes.

2 Problématique

L'expansion très large de l'informatique a conduit à l'apparition de nouvelles applications soumises à de nombreuses contraintes, la plupart de ces contraintes sont qualifiées d'être non

fonctionnelles comme par exemple la performance, la qualité de service, la sécurité, la robustesse, la portabilité, etc.

Dans ce contexte, les langages de descriptions d'architectures qui sont des notations servant à fournir aux concepteurs de logiciels un support de haut niveau d'abstraction permettant de décrire la structure des applications souffrent de limites concernant la prise en compte de l'aspect non fonctionnel des systèmes logiciels.

En effet, les travaux menés jusqu'à présent sur le développement centré architecture de logiciels portent essentiellement sur la spécification formelle des architectures. La vérification et l'analyse des architectures logicielles restent un point ouvert de recherche. Il est alors indispensable d'associer à une architecture logicielle des spécifications précises tant fonctionnelles que non fonctionnelles. Ces spécifications décrivent des services offerts par les différents composants de cette architecture, mais aussi ceux dont ils ont besoin pour fonctionner. L'environnement possède des propriétés qui ont une influence sur le fonctionnement du logiciel. Il peut s'agir de propriétés temporelles, des propriétés sur le mode de panne ou sur les attaques possibles en termes de sécurité, etc. Ces spécifications sont rarement complètes et formelles.

En général, le terme contrat est de plus en plus utilisé pour décrire les propriétés d'un composant architectural. Les propriétés non fonctionnelles, devant être exprimées à part, permettent de caractériser le degré de satisfaction d'un logiciel, c'est-à-dire comment ses fonctionnalités sont réalisées.

De par leur complexité et leur haut niveau d'abstraction, elles sont rarement prises en compte dans le processus de développement d'un logiciel, particulièrement sa conception architecturale. En effet, le concepteur doit pouvoir considérer ce type de propriétés dès le début du processus, et les estimer avant la phase d'implémentation du système.

3 Contribution

L'objectif principal de ce travail est de fournir au langage de description d'architecture SADL la possibilité de décrire et de vérifier formellement les propriétés non fonctionnelles liées aux architectures logicielles.

Notre contribution dans ce travail est alors double, d'une part nous proposons un cadre sémantique formel basé logique de réécriture pour décrire les concepts de base du langage SADL, ceci permettra de fournir une sémantique bien définie au plan architectural. De plus, l'analyse des architectures SADL par l'usage des critères liées aux respects de contraintes de

styles ou par l'usage des techniques de vérifications formelles devient possible et se fait d'une façon assez naturelle.

D'autre part, nous étendons ce cadre formel par l'intégration d'autres constructions permettant de spécifier les propriétés non fonctionnelles liées aux composants et aux connecteurs d'une architecture SADL.

4 Contenu du mémoire

Le présent mémoire est organisé comme suit :

- Ø Le premier chapitre de ce mémoire constitue un état de l'art sur les langages de description d'architectures, nous commençons ce chapitre par la définition des éléments de base d'un ADL. Puis nous présentons une brève synthèse des ADLs de référence développés par le milieu académique suivi par une étude comparative de ces derniers.
- Ø Dans le deuxième chapitre nous présentons le langage de description d'architectures SADL à travers ses éléments de base et les différents mécanismes offerts par ce langage en particulier le mécanisme de raffinement. Nous terminons ce chapitre en évoquant les limites de ce langage.
- Ø Le troisième chapitre est dédié à la présentation de la logique de réécriture et son langage Maude.
- Ø Dans le chapitre 4 nous proposons un cadre sémantique formel basé logique de réécriture pour le langage SADL permettant de décrire tous ses éléments de base, puis nous montrons comment ce cadre peut être étendu pour prendre en compte les aspects de raffinement et d'analyse des architectures logicielles.
- Ø Dans le dernier chapitre nous détaillons notre approche d'intégration des propriétés non fonctionnelles dans le langage SADL. Nous montrons tout d'abord comment élargir le modèle des architectures SADL présenté dans chapitre précédent pour inclure des constructions permettant de décrire l'aspect non fonctionnel des architectures logicielles, puis nous présentons l'approche de validation adoptée.
- Ø Enfin une conclusion générale conclut ce mémoire et présente les perspectives de notre travail.

Chapitre I

Etat de l'Art sur les ADLs

1 Introduction

Face à l'augmentation de la complexité des systèmes informatiques, de nouvelles approches et méthodes de développement ont été élaborées, parmi ces approches on trouve le développement de logiciel par composants qui consiste à concevoir un système logiciel en se basant sur un ensemble de composants reliés entre eux par des connecteurs, qui collaborent pour réaliser les fonctionnalités désirées du système global.

Dans ce contexte, il deviendra de plus en plus nécessaire d'utiliser des techniques centrées sur la description de l'architecture de ces systèmes pour mieux contrôler leur complexité et aussi pour garantir leur réutilisabilité.

Une architecture logicielle à base de composants décrit l'ensemble des composants qui la composent, donne la définition de leur assemblage et prend en compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant. On peut dire que la définition de l'architecture d'un système correspond à l'établissement du plan de construction du logiciel. Elle permet la conception d'applications en se détachant de détails techniques propres à l'environnement et en respectant les conditions fixées par les futurs utilisateurs.

En maîtrisant l'architecture conceptuelle, il est alors plus facile de gérer ses éventuelles évolutions. En effet la modification d'un plan est plus simple que la modification d'un système complet [13].

Il n'existe pas réellement une définition globale de l'architecture d'un logiciel mais nous trouvons dans la littérature quelques définitions proches les unes des autres, parmi ces dernières nous présentons :

Définition 1 : l'architecture d'un logiciel reflète sa structure de différentes manières, du texte aux graphes, et pour laquelle on ne précise que les types de composants par leur interface [18].

Définition 2 : l'architecture d'un logiciel décrit un système logiciel comme une configuration de composants et de connecteurs, un connecteur met en relation des services

offerts à des services requis par les composants, cette configuration de composants et de connecteurs peut être utilisée comme un composant d'un autre système [19].

Définition 3 : L'architecture d'une application décrit la structure globale d'un système informatique comme une collection de composants qui interagissent entre eux [07].

D'autre part, dans [02] Anne-Marie et Sébastien avaient cité les arguments qui font apparaître clairement le rôle joué par la description de l'architecture au sein du processus de développement d'un système logiciel qui sont :

- Ø L'architecture constitue un cadre pour maîtriser la complexité d'un système, dont le but de répondre à toutes les exigences du cahier des charges, qu'elles soient fonctionnelles ou non;
- Ø L'architecture est une base pour la conception comme pour l'estimation des coûts et la gestion de projet;
- Ø L'architecture offre un support pour la prise en considération de l'évolutivité et de la réutilisation des systèmes logiciels, en plus elle constitue une base de raisonnement à des fins d'analyse et de validation.

Il existe essentiellement trois notations utilisées pour décrire les architectures des systèmes logiciels :

1) Les langages d'interconnexion de modules ou MIL (Module Interconnection Language) : ces langages permettent la description des modules logiciels d'une application et les dépendances entre ces modules [02]. L'idée de base derrière ces langages est que la programmation de systèmes se divise en deux activités différentes : la *programmation globale* ("in-the-large") et la *programmation détaillée* ("in-the-small").

La programmation globale se concentre sur la structuration de modules pour former un système tandis que la programmation détaillée concerne la construction des modules eux mêmes.

Ainsi, pour la programmation détaillée, on utilise les langages classiques. Cependant, en ce qui concerne la programmation globale, de nouveaux langages spécifiques sont crées : les langages d'interconnexion de modules (ou MILs) [07].

Alors en conclusion on peut dire que les langages d'interconnexion de modules permettent une description formelle de la structure des systèmes en termes d'un ensemble de modules et leurs interconnexions.

2) Les langages de configuration : l'objectif de cette catégorie de langages est la spécification de la configuration d'un système tout en assurant la possibilité de modification

c'est-à-dire l'ajout, la suppression et la modification de composants d'une manière statique ou dynamique [02].

3) Les langages de description d'architecture ou ADL (Architecture Description Language) : ces langages sont le résultat d'une orientation plus conceptuelle et plus formelle des langages de configuration, tant pour les composants que pour les connexions entre composants, ceci afin d'accroître les possibilités de raisonnement.

Ce chapitre présente un état de l'art sur les ADLs. Dans une première partie nous montrons c'est quoi un ADL, quels sont les éléments principaux dans un ADL, puis dans une deuxième partie nous étudions quelques ADLs les plus connus, et enfin nous terminons ce chapitre par une étude comparative entre ces ADLs.

2 Les concepts clés des ADLs

Dans cette partie nous décrivons d'une manière détaillée les éléments de base d'un ADL.

Les ADLs (architecture description language) ou encore langage de description d'architectures sont des notations formelles utilisées pour représenter et analyser les concepts architecturaux des systèmes. Ces langages fournissent une syntaxe concrète pour la caractérisation d'architectures logicielles d'une manière déclarative en termes d'assemblage de composants. Le but de ces langages est de fournir une vision de l'architecture de l'application en termes d'entités logicielles nécessaires au fonctionnement de l'application, et une description des interconnexions entre ces différentes entités [12].

Il n'existe pas encore de consensus sur le niveau d'abstraction et de description auquel devrait se situer un ADL mais un accord semble toutefois émerger sur un ensemble commun et minimal de concepts que l'on retrouve dans une grande partie des ADLs (le composant, le connecteur et la configuration).

2.1 Le composant

Un composant ("component") est le concept de base de tous les ADLs.

Un composant comme il a été défini dans [13] est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Il possède un état, il peut être simple ou composé, on parle alors dans ce dernier cas d'un composant composite. Sa taille peut aller de la fonction mathématique à une application complète. Il existe principalement deux parties dans un composant. Une première partie, dite extérieure, correspond à son interface. Elle comprend la description des services fournies et requises par le composant. Elle définit aussi

les interactions du composant avec son environnement. La seconde partie correspond à son implantation et permet la description du fonctionnement interne du composant.

Dans cette section, nous présentons les caractéristiques des composants selon six critères : l'interface, le type, la sémantique, les contraintes, l'évolutions et les propriétés non fonctionnelles [17].

La figure suivante présente un exemple qui contient la déclaration de deux composants d'une architecture client/serveur en utilisant l'ADL Wright.

```
Component Client
Port p = request!x -> reply?y -> p ∏ §
Computation = internalCompute -> p.request!x -> p.reply?y -> Computation ∏ §
Component Server
Port p = request?y -> reply!x -> p □ §
Computation = p.request?y -> internalCompute -> p.reply!x -> Computation □ §
Constrains
∃! S∈ Component, ∀C ∈ Component : TypeServer(S) Λ TypeClient(C)
⇒ Connected(C,S)
```

Figure 1.1 : Exemple de déclaration de composant

2.1.1 L'interface d'un composant

L'interface d'un composant est un ensemble de points d'interaction entre le composant lui-même et le monde extérieur, on peut dire que l'interface décrit les services fournis et les services requis par un composant. Ces services peuvent être définis par des signatures de méthodes, et de type d'objets envoyés et retournés.

Dans l'exemple précédant (figure 1.1), l'interface du composant Client est définie par un seul port p**,** dont le comportement se traduit par un processus indiquant qu'il est à l'origine de la requête (événement request!x) et à l'écoute de la réponse (événement reply?y) il se termine correctement (processus §).

Le signe ∏ indique que le processus choisi est déclenché par le processus qui l'englobe (on parle de choix non-déterministe) et le signe□ indique que le processus choisi est déclenché par l'environnement extérieur (on parle de choix déterministe).

Le signe § indique un processus de fin normale.

2.1.2 Le type d'un composant

Le type d'un composant est un concept qui représente l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. Ainsi, un type de composant permet la réutilisation de composants de même fonctionnalité soit dans la même architecture, soit dans une autre architecture.

En fournissant un moyen de décrire d'une manière explicite les propriétés communes à un ensemble de composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et sa conception [13].

A titre d'exemple dans la figure 1.1 on a deux types de composants : le type Client et le type Server.

2.1.3 La sémantique d'un composant

La sémantique d'un composant est la modélisation de son comportement en utilisant un formalisme de haut niveau, cette modélisation permet de garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements [16].

Dans l'exemple de la figure 1.1 le comportement du composant Client est décrit par une computation mettant en lien le comportement du port p et celui du composant indiqué par l'événement internalCompute.

2.1.4 Les contraintes d'un composant

Une contrainte est une propriété ou une assertion sur un système ou sur une de ses parties, où la violation de cette propriété va rendre le système incohérent et inacceptable.

On peut alors dire que les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra composants.

Dans l'exemple de la figure 1.1 il y a également une contrainte (mot clé constraints) spécifiant une propriété du système client/serveur. Cette contrainte indique qu'un client est connecté obligatoirement à un serveur et qu'un serveur peut avoir plusieurs clients.

2.1.5 L'évolution d'un composant

Comme un composant est un élément de conception, il doit alors avoir la possibilité d'évolution, ceci est possible à travers le sous typage ou le raffinement, mais cela sans perturber son intégration dans les applications déjà existantes.

2.1.6 Les propriétés non fonctionnelles

Les propriétés non fonctionnelles d'un composant (par exemple : la sécurité, la performance et la portabilité) ne peuvent pas être dérivées directement de la spécification de son comportement, elle doivent être spécifiées au niveau de l'architecture.

2.2 Le connecteur

Le connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants, ceci par la définition des règles qui gouvernent ces interactions. Par exemple, un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux [13].
Comme pour les composants, les connecteurs possèdent aussi deux parties, la première partie correspond à l'interface du connecteur c'est-à-dire la description des rôles des participants à une interaction. La seconde partie correspond à la description de son implantation, il s'agit là de la définition du protocole permettant la mise en oeuvre du protocole associé à l'interaction.
Dans [17] on trouve les six caractéristiques indispensables que nous devons prendre en compte pour une spécification exhaustive d'un connecteur, qui sont : l'interface, le type, la sémantique, les contraintes, l'évolution et les contraintes non fonctionnelles.
La figure 1.2 contient un exemple de déclaration d'un connecteur Link dans l'ADL Wright.

```
Connector Link
        Role c = request!x -> reply?y -> c ∏ §
        Role s = request?y -> reply!x -> s □ §
Glue = c.request!x -> s.request?y -> Glue
          □ s.reply!x -> c.reply?y -> Glue
          □ §
```

Figure 1.2 : Exemple de déclaration d'un connecteur

2.2.1 L'interface d'un connecteur

L'interface d'un connecteur est un ensemble de points d'interaction entre le connecteur lui-même et les différents composants attachés, son rôle essentiel est la définition des mécanismes de connexion entre composants.
Dans la figure 1.2 l'interface du connecteur Link est composée de deux rôles :

- le rôle client (c) dont le comportement est de déclencher une requête puis d'attendre une réponse,
- le rôle serveur (s) dont le comportement est d'attendre une requête et de déclencher une réponse.

2.2.2 Le type d'un connecteur

Le type d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants ou les mécanismes de décision de coordination et de médiation. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles [13]. Par exemple dans la figure 1.2 on trouve la spécification d'un connecteur de type Link.

2.2.3 La sémantique d'un connecteur

Une description d'architecture doit pouvoir modéliser la sémantique des connecteurs c'est-à-dire elle doit spécifier le protocole d'interaction.
Par exemple la sémantique du connecteur Link est décrite par la Glue qui lie les deux rôles.

2.2.4 Les contraintes d'un connecteur

Les contraintes d'un connecteur permettent la définition de ses limites d'utilisation, par exemple la contrainte qui défini un nombre maximum de composants interconnectés à travers le connecteur.

2.2.5 L'évolution d'un connecteur

Les interactions entre composants sont gouvernées par des protocoles complexes et changeables, alors les connecteurs doivent supporter l'évolution et le changement soit par le sous typage ou par le raffinement.

2.2.6 Les propriétés non fonctionnelles

Les propriétés non fonctionnelles des connecteurs ne sont pas entièrement dérivables de la spécification de leur sémantique, elles représentent des besoins supplémentaires pour une implémentation correcte du connecteur, la spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement du système à l'exécution.

2.3 La configuration

Une configuration (encore appelée l'architecture ou la topologie) définit la structure et le comportement d'une application formée de composants et de connecteurs. Une composition de composants, appelée dans certains contextes composite, est une configuration. La configuration structurelle de l'application correspond à un graphe connexe de composants et de connecteurs formant l'application. Elle détermine les composants et les connecteurs appropriés à l'application et vérifie la correspondance entre les interfaces des composants et des connecteurs. La configuration comportementale, quant à elle, modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles comme les propriétés spatio-temporelles ou la qualité de service. Elle définit également le schéma d'instanciation des composants au moment de l'initialisation de l'application, ainsi que le placement des composants sur les sites au moment du démarrage du système et leur évolution pendant la vie de l'application [13].
Dans [17] Medvidovic et Taylor ont précisé huit caractéristiques des configurations qui sont : la compréhensibilité de la spécification, la composition, le raffinement et la traçabilité, l'hétérogénéité, le passage à l'échelle, l'évolution de la configuration, le dynamisme et enfin les contraintes.

2.3.1 La compréhensibilité de la spécification

Le rôle majeur d'une configuration est de rendre compréhensible un système avec un haut niveau d'abstraction, par conséquent un ADL doit modéliser les informations structurelles avec une syntaxe simple et compréhensible.

2.3.2 La composition

La composition ou encore appelée la composition hiérarchique est un mécanisme qui permet aux concepteurs de décrire les systèmes logiciels à différents niveaux de détails, par exemple un système complexe peut être décrit par un seul composant dans un niveau d'abstraction élevé.

2.3.3 Le raffinement et la traçabilité

Une configuration doit permettre le raffinement de systèmes d'un niveau abstrait de description vers un niveau de description de plus en plus détaillé, et ceci à chaque étape du processus de développement (conception, implantation, déploiement).

2.3.4 L'hétérogénéité

Le but d'une architecture est de faciliter le développement de grands systèmes avec des composants et des connecteurs de taille variable, alors une configuration doit tenir compte de cet aspect et aussi elle doit être capable de spécifier une application indépendamment du langage de programmation, du système d'exploitation et du langage de modélisation.

2.3.5 Le passage à l'échelle (Scalability)

Une configuration doit supporter la spécification et le développement des systèmes qui peuvent grossir en taille, la seule méthode pour supporter le passage à l'échelle est à travers la composition hiérarchique.

2.3.6 L'évolution de la configuration

Une configuration doit être capable d'évoluer pour prendre en compte de nouvelles fonctionnalités et ceci par ajout ou retrait de composants et de connecteurs.

2.3.7 Le dynamisme

La configuration d'une application doit permettre la modification à l'exécution. Elle doit permettre la spécification du comportement dynamique de l'application, c'est-à-dire les changements de l'application qui peuvent arriver pendant son temps d'exécution comme par exemple la création ou la suppression d'instances de composants [13].

2.3.8 Les contraintes

Les contraintes liées à la configuration sont celles qui décrivent les dépendances voulues entre les composants et les connecteurs, ces contraintes sont importantes comme celles liées aux composants et aux connecteurs, la plupart de ces contraintes sont dérivées directement des contraintes locales.

2.3.9 Les propriétés non fonctionnelles

Un ADL doit permettre la spécification des propriétés non fonctionnelles au niveau de la configuration.

3 Intérêts des ADLs

Plusieurs difficultés rencontrées lors de la conception de logiciel ont été résolues par l'utilisation des ADLs, ils permettent la définition d'un vocabulaire précis et commun pour les

acteurs devant travailler autour la spécification liée à l'architecture (architectes, concepteurs, développeurs, intégrateurs et testeurs). Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans les détails d'implantation. Ils définissent de manière explicite les interactions entre les composants d'un système et fournissent un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments.

Les ADLs fournissent aussi une syntaxe concrète pour la caractérisation d'architectures logicielles d'une manière déclarative en termes d'assemblage de composants et de connecteurs. Le but de ces langages est de fournir une vision de l'architecture de l'application en termes d'entités logicielles nécessaires au fonctionnement de l'application, et une description des interconnexions entre ces différentes entités [12].

Une autre caractéristique importante inhérente aux ADLs est qu'ils sont basés sur des fondements formels permettant la vérification des configurations architecturales.

4 Description des principaux ADLs

Plusieurs langages de description d'architecture ont été proposés, le but global de ces langages est la fourniture d'une syntaxe concrète et un cadre conceptuel pour la modélisation des architectures des systèmes logiciels, mais nous pouvons dire que chacun de ces langages a des caractéristiques qui le rende mieux pour certains systèmes que pour d'autres.

Dans cette partie nous présentons une synthèse des ADLs de référence développés par le milieu académique, les autres nouveaux langages sont des extensions apportées à ces langages en intégrant la technologie XML et ses dérivés pour la spécification, la manipulation et l'utilisation de ces langages [06], par exemple :

- **xOlan** (L Bellissard & al, INRIA Rhône-Alpes, 2001), dérivé d'Olan.
- **xAcme** (B Schmerl , CMU, 2001) extension d'xArch, inspiré d'Acme.
- **xADL2.0** (E Dashofy, UCI, 2001) extension d'xArch, hiérarchisé et spécialisé "famille de produit", etc.

Enfin, nous donnons une étude comparative de ces ADLs par rapport à quelques propriétés.

4.1 UniCon

UniCon (qui signifie Universal Connector Support) est un langage de description d'architecture crée par l'université de Carnegie Mellon en Pennsylvanie, USA dont le concepteur principal est G.Zelesnik.

Ce langage est fondé sur les trois concepts de base des ADLs, à savoir le composant, le connecteur et la configuration.

Dans cette partie nous allons présenter une vue synthétique de ce langage.

4.1.1 Le composant

Dans UniCon le composant est le concept de base. C'est une entité d'encapsulation des fonctions d'un module logiciel ou de données. Un composant est caractérisé par une interface qui permet d'exhiber les fonctions et les données fournies et les fonctions et les données requises par le composant, il permet aussi de définir le mode d'accès de ces fonctions ou services.

Chaque composant dans UniCon a un type dont il dérive, ce concept de type permet de définir le mode de mise en œuvre ou le mécanisme qui permet de fournir les fonctions du composant, par exemple un composant peut être de type process si tout ce qu'il encapsule sera contenu dans un processus particulier à l'exécution, un autre type de composant est le type module si son implémentation correspond à des ensembles de fonctions contenues dans une bibliothèque, et il peut être de type shared data s'il correspond à une zone de données partagées.

Une autre caractéristique importante de ce langage est que chaque type de composant impose un ensemble d'opérations possibles pour ce type de composant, ces opérations sont définis par des types de players, une liste complète des types de composant ainsi que leurs types de player est présenté dans [25].

La figure suivante présente un exemple de déclaration d'un composant en utilisant UniCon :

```
COMPONENT Client
   INTERFACE IS
        TYPE Process
         PLAYER Lookup_Annuaire IS RPCCall
               SIGNATURE ("char *", "char **")
         End Lookup_Annuaire
  End INTERFACE
        IMPLEMENTATION IS
              VARIANT client IS "client.c"
               IMPLTYPE(source)
       END IMPLEMENTATION
End Client
```

Figure 1.3 : Exemple de déclaration d'un composant dans UniCon

Dans cet exemple on définit un composant client de type Process avec un player Lookup_Annuaire de type RPCCall permettant l'accès aux fonctions à travers un mécanisme de RPC. Cet exemple contient aussi la déclaration de l'implémentation du composant qui permet de faire le lien avec le code contenu dans des fichiers source.

4.1.2 Le connecteur

Un connecteur dans UniCon permet de définir les règles d'interconnexion des composants, comme pour les composants les connecteurs ont aussi un type, ce type permet la définition du protocole associé au connecteur.

Par exemple un connecteur de type remote proccall permet de relier des composants de type process, ce protocole spécifie également des points de branchement autorisés pour les composants, ces points d'entrées ou de sorties sont définis par des rôles auxquels les players d'un composant peuvent être interconnectées dans [25] se trouve une description de tous les types de connecteurs ainsi que leurs rôles dans UniCon**.**

Un exemple de description d'un connecteur est le suivant :

```
CONNECTOR Remote-proc-call
        PROTOCOL IS
                TYPE RemoteProcCall
                ROLE definer IS definer
                ROLE caller IS caller
        END PROTOCOL
        IMPLEMENTATION IS
                BUILTIN
        END IMPLEMENTATION
END Remote-proc-call
```

Figure 1.4 : Exemple de déclaration d'un connecteur dans UniCon

Dans cet exemple on définit un connecteur *Remote-proc-call* de type *RemoteProcCall* avec deux rôles, le premier de type *definer* et l'autre de type *caller*, ce connecteur utilise comme implémentation celle prédéfinie par le système UniCon.

4.1.3 L'architecture

Une architecture dans UniCon correspond à un composant composite (ayant une interface de type general) dont l'implantation spécifie les interfaces de ses sous composants utilisés.

L'interconnexion dans une architecture correspond à la mise en relation des players des composants par un connecteur.
UniCon possède un formalisme graphique pour la représentation des architectures permettant de faciliter l'utilisation de ce langage.

4.2 Rapide

Rapide est un langage de description d'architecture, il fut proposé à l'origine au projet ARPA (Advanced Research Project Agency) en 1990 par l'université de Standford, USA et la société TRW inc [30].
Ce langage est fondé sur les événements concurrents spécifiquement conçus pour la simulation d'architecture logicielle distribuée [12], il possède un environnement textuel et graphique pour décrire la structure de l'architecture des systèmes. Cet environnement est crée par l'université de Standford.
En outre, le langage Rapide est fondé sur les concepts d'événement, d'architecture (configuration) et de connexion, que nous allons présenter dans ce qui suit.

4.2.1 L'événement

Une application dans Rapide est constituée d'un ensemble de modules ou de composants communiquant par envoie de messages ou événements.
Un événement dans Rapide est une information transmise entre composants (une demande de service) qui permet de construire des expressions appelées patrons d'événements (event patterns) caractérisant les événements circulant entre composants. La construction de ces expressions se fait avec l'utilisation d'opérateurs qui définissent les dépendances entre événements. Parmi ces opérateurs on trouve l'opérateur de dépendance causal (A→ B), l'opérateur d'indépendance (A || B), l'opérateur de différence (A~B) et l'opérateur de simultanéité (A and B). Ainsi, l'événement correspond à une information permettant de spécifier le comportement d'une application [13].

4.2.2 Le composant

Un composant dans Rapide est défini par son interface, qui peut être vue comme un ensemble de services fournis et un ensemble de service requis. Ces services sont de trois types :

- les services Provides fournis par le composant et appelés de manière synchrone par d'autres composants;
- les services Requires demandés par le composant et appelés de manière synchrone;

- Ø les « Actions » qui correspondent à des appels asynchrones entre composants, deux types d'actions existent : les actions in et out qui sont des événements acceptés et envoyés par un composant [13].

Un composant possède aussi une partie dédiée à la description de comportement (clause behaviour) c'est-à-dire le fonctionnement observable d'un composant, cette description peut être vue comme une contrainte sur le comportement du composant (ou module) à implanter, par exemple l'ordonnancement des événements ou des appels aux services.

4.2.3 L'architecture

Dans Rapide une architecture est la déclaration des instances de composants et les règles de connexions entre ces instances. Toutes les instances sont déclarées sous forme de variables. La règle d'interconnexion est composée de deux parties, la première est la partie gauche qui contient une expression d'événements qui doit être vérifiée, la seconde est la partie droite qui contient également une expression d'événements qui doivent être déclenchés après la vérification de l'expression de la partie de gauche [13].

La figure 1.5 présente un exemple de définition d'une architecture en Rapide

```
with Client, Serveur;
...
?s : Client; -- fait référence à une instance de Client
!r : Serveur; -- fait référence à toutes les instances de serveur
?d : Data; -- fait référence à un bloc de paramètre d'un certain type Data
...
?s.Send(?d) => !r.Receive(?d);;
 -- Si un client transmet un événement de type Send avec ce type de paramètres, alors l'événement est transmis à tous les serveurs de l'application avec ces paramètres.
```

Figure 1.5 : Exemple d'une architecture en Rapide

Cette définition d'architecture contient trois parties principales, la première partie correspond à l'importation des types de composants, la deuxième partie correspond à déclaration des instances de composants de l'application susceptible d'exister, et finalement la troisième partie correspond à la déclaration d'une règle d'interconnexion.

4.3 Wright

Wright est un langage de description d'architecture dédié à la vérification des protocoles entre les composants d'une architecture. Wright est basé sur les concepts de composant, de connecteur et de configuration qui sont définis par un calcul proche de CSP [29].
Une caractéristique importante de ce langage est qu'il permet une vérification formelle des architectures.

4.3.1 Le composant

Un composant dans Wright est une entité abstraite composée de deux parties, l'interface (interface) et la partie calcul (computation).
L'interface d'un composant est constituée de ports, où chaque port correspond à une interaction dans laquelle le composant participe, ces ports ont tous une description formelle par le langage CSP spécifiant son comportement par rapport à l'environnement.
Concernant la description du comportement elle est faite dans la partie computation qui reprend les interactions décrites au sein du port et montre les relations entre tous les ports ainsi que leurs lien avec les actions internes du composant [12].
La figure ci-dessous montre La description du composant Fourchette avec Wright.

```
Component Fourchette
Port Manche = pris → depose → Manche ∏ §
Computation = Manche.pris → Manche.depose → Computation ∏ §
```

Figure 1.6 : Exemple d'un composant Fourchette décrit avec WRIGHT

Ce composant possède un seul port Manche qui constitue son interface, le comportement de ce port est spécifié en utilisant un processus CSP, la spécification du comportement du composant est donnée à travers la clause Computation.

4.3.2 Le connecteur

Un connecteur dans Wright détermine les interactions entre les composants.
Chaque connecteur dans Wright possède un type et il spécifie un patron d'interaction d'une manière explicite et abstraite, d'autre part un connecteur est composé de deux parties : un ensemble de rôles et une spécification GLUE.
Les rôles définissent le comportement local souhaité ou attendu par les ports qui vont s'attacher aux extrémités du connecteur. La GLUE d'un connecteur décrit les participants interagissant au cours de la communication [12].

4.3.3 La configuration

La configuration dans Wright permet la description du regroupement de différentes instances de composants et de connecteurs, cette description est constituée de trois parties : une déclaration des composants et des connecteurs, et une déclaration d'instances des composants et des connecteurs et enfin la troisième partie qui correspond à la description des liens entre les instances de composants par les instances de connecteurs.

4.4 Olan

Olan est un langage élaboré par l'INRIA Rhône-Alpes dans le cadre du projet SIRAC, son but principal et de fournir un environnement complet pour la construction, la configuration et le déploiement d'applications réparties [13].

Ce langage vise aussi à l'intégration du logiciel existant en l'encapsulant dans un composant Olan, Olan permet de fournir un moyen permettant de définir une architecture de manière abstraite grâce au langage *OCL* (*Olan Configuration Language*) en établissant une hiérarchie de composants connectés entre eux .

Dans le langage Olan il existe deux concepts de base qui sont le composant et le connecteur.

4.4.1 Le composant

Un composant dans Olan peut être un composant primitif ou un composant composite.

Un composant primitif de Olan est l'unité de base d'une application, il est caractérisé par une interface et une implantation.

L'interface définit les services fournis et requis par le composant, les services sont décrits par le langage *OIL* (Olan *Interface Language*) qui est une extension du langage *IDL* de l'*OMG*.

La figure 1.7 présente l'interface d'un composant Annuaire dans Olan.

```
Typedef sequence <*,char> statStruct;
Interface AnnuaireItf {
Readonly attribute string Pays ;
provide init() ;
// service pour rechercher une adresse à partir d'une clé
provide Lookup (in string cle, out string email) ;
// notification générée lorsqu'un rapport est envoyé
notify SendStats (in statStruct statReport) ;
} ;
```

Figure 1.7 : Exemple d'une interface d'un composant dans Olan

Cet exemple décrit l'interface d'une application d'annuaires répartis dans laquelle un client a un annuaire par pays qui fournit les services suivants :

- Un service permettant de récupérer l'adresse d'une personne dans le pays en question;
- Un service qui envoie de manière asynchrone un rapport des statistiques d'utilisation de l'annuaire.

L'implantation d'un composant primitif se fait par la mise en œuvre d'un composant qui correspond à un ensemble d'entités logicielles de divers type, cela en utilisant le langage *OCL* qui permet la description de cet ensemble d'entités, à chaque entité correspond un module, donc l'implantation d'un composant primitif correspond à la liste de modules utilisés ainsi que la liaison explicite entre les services déclarés dans l'interface et ceux définis dans les modules.

Un composant composite dans Olan permet la définition de la structuration d'une application de composants coopérants, ce concept de composant composite dans Olan définit une hiérarchie de composants qui peuvent être partiellement ou totalement réutilisés dans d'autres applications.

4.4.2 Le connecteur

Le connecteur permet la définition de la manière de communication entre composants pendant l'exécution, un connecteur met en jeux deux parties : la partie qui contient le service du composant initiateur de la communication et la partie des services des composants destinataires.

4.5 Acme

Acme [27] est un langage de description d'architecture établi par la communauté scientifique dans le domaine des architectures logicielles. Il a pour but principal de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADLs, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADLs [13].

Alors nous pouvons dire que Acme est langage de description d'architecture fédérateur des langages de description d'architecture existants, car la plupart des ADLs ont des notions similaires comme : le composant, le connecteur, la configuration.

Dans ce qui suit nous citons les caractéristiques essentielles de ce langage :

- Acme est basé sur sept concepts fondamentaux qui sont: le composant, le connecteur, le système, le port, le rôle, la représentation et la carte de représentation (repmap);

- Ø Acme dispose d'un mécanisme d'annotation pour l'intégration d'information spécifique ne concernant pas la structure;
- Ø un canevas favorisant l'intégration d'outils facilitant la spécification formelle de la sémantique d'une architecture.

La figure ci-dessous montre les éléments fondamentaux d'une description Acme.

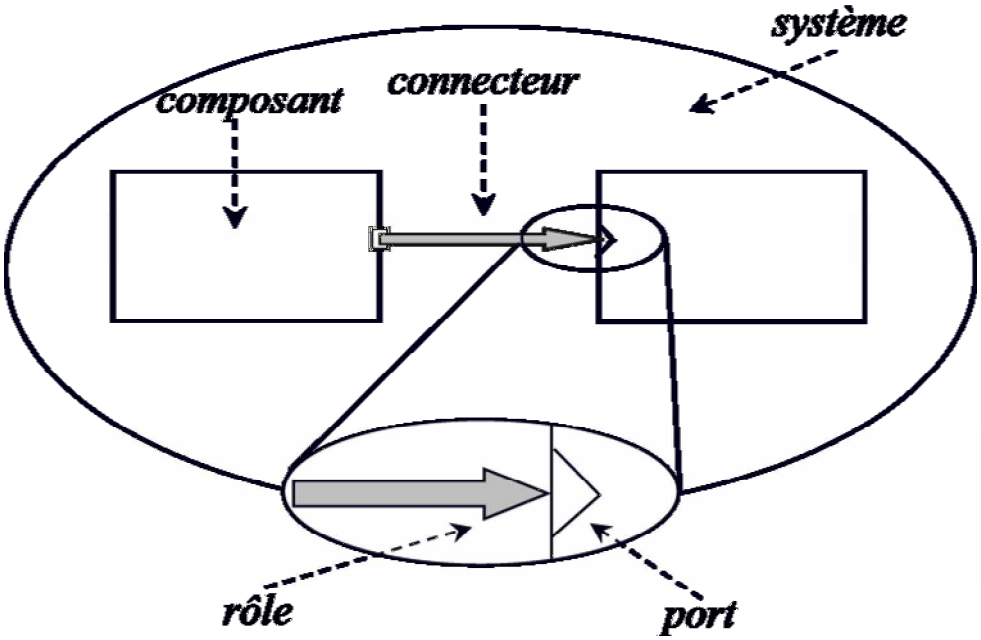


Figure 1.8 : Eléments d'une description Acme

4.5.1 Le composant

Un composant dans Acme est une entité de traitement ou de sauvegarde de données d'une application. Chaque composant est doté d'une interface constituée de plusieurs types de ports, chaque type de port identifie un point d'interaction entre le composant et le monde extérieur.

4.5.2 Le connecteur

C'est le médiateur de communication qui coordonne la connexion entre composants, un connecteur dans Acme possède aussi une interface qui inclut un ensemble de types de rôles.

4.5.3 Le système

Le système correspond à la configuration c'est-à-dire l'assemblage de composants et de connecteurs.

4.5.4 La représentation et la carte de représentation

Ces deux concepts permettent à Acme de supporter la description hiérarchique d'une architecture. Ainsi, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé et peut donc être raffiné.

Chaque nouvelle description (sous élément) d'un élément est appelée une représentation. La correspondance entre l'élément et ses représentations est spécifiée grâce à la carte de représentation. Ainsi, la carte de représentation permet d'établir la correspondance entre les ports de l'interface d'un composant et ceux définis dans les interfaces de ses sous composants [13].

4.5.5 Exemple d'une architecture Acme

L'exemple suivant tiré de [13] montre une description d'un système System composé de deux composants, le composant client et le composant server ainsi qu'un connecteur rpc.

```
System simple_cs = {
Component client = { Port send-request }
Component server = { Port receive-request }
Connector rpc = { Roles {caller,callee}}
Attachments : {
client.send-request to rpc.caller ;
server.receive-request to rpc.callee}
```

Figure 1.9 : Exemple d'une architecture décrit avec ACME

Les composants client et server sont spécifiés par un seul port send-request et receive-request respectivement, ces deux ports sont les éléments responsables à toutes les interactions entre ces deux composants, la clause Connector permet d'introduire le connecteur rpc qui a deux rôles, le rôle caller et le rôle callee, enfin la clause Attachments qui coordonne les connexions entre les deux composants.

5 Comparaison entre les ADLs

Dans cette partie nous présentons une étude comparative entre les ADLs déjà présentés dans la section précédente, nous adoptons le gabarit de comparaison défini par Nenad Medvidovic et Richard N. Taylor dans [17] qui consiste à comparer les ADLs selon les propriétés des composants, des connecteurs et des configurations.

5.1 Le Composant

La plupart des ADLs partagent dans leurs vocabulaire le mot "composant" (component), un composant comme il a été déjà défini correspond à une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Un composant peut être de petite taille comme par exemple un simple procédure ou une application complète. Un composant peut avoir son propre espace mémoire comme il peut le partager avec d'autres composants.
Dans cette partie nous discutons sur le support fournis par les ADLs pour modéliser les différents aspects des composants.

5.1.1 L'Interface

La plupart des ADLs supportent la modélisation de l'interface de composant, mais ils n'utilisent pas la même terminologie pour ce concept, par exemple un point d'interface dans SADL ou Wright est un port, et dans UniCon est un player.
Tous Les ADLs font une distinction entre les points d'interface dédiés aux services fournis et ceux dédiés aux services requis, mais Rapide impose une distinction en plus entre les interfaces synchrone et asynchrone.
Les points d'interface sont typés dans les ADLs : Acme, SADL et UniCon**,** par exemple UniCon supporte un ensemble de player prédéfinis, incluons : RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, ReadFile, WriteFile, RPCDef, et RPCCall.

5.1.2 Le type

Tous les ADLs font une distinction entre le type d'un composant et les instances, Rapide par exemple fait cette distinction à l'aide d'un langage de typage séparé.
En effet tous les ADLs fournissent un système de typage extensible, mais UniCon ne supporte qu'un ensemble de type de composants prédéfinis : Module, Computation, SharedData, SeqFile, Filter, Process, ScharedProcess et General.
D'autre part Acme, Rapide, SADL, et Wright permettent la paramétrisation des interfaces de composants, cela se fait comme dans les langages de programmation tels que ADA et C++.

5.1.3 La sémantique

Tous les ADLs supportent la spécification de la sémantique des composants, le modèle sémantique sous-jacent des ADLs varie de l'expression de la sémantique dans la liste des propriétés (UniCon) vers des modèles de comportement dynamique de composants (Rapide et Wright).

5.1.4 Les contraintes

Tous les ADLs contraignent l'utilisation de composants par la spécification des interactions autorisées.

5.1.5 L'évolution

Un certain nombre d'ADLs modélisent les composants comme des entités statiques, par exemple UniCon définie le type d'un composant par énumération en interdisant le sous typage et par conséquent il n'offre aucun support d'évolution, mais il existe quelques ADLs qui permettent l'évolution de composants en utilisant le sous typage, par exemple Acme supporte un sous typage structural stricte, Rapide permet l'évolution des composants via l'héritage de l'OO.

5.1.6 Les propriétés non fonctionnelles

Malgré le besoin de la spécification des propriétés non fonctionnelles, il existe un manque considérable de supports pour ce type de spécification dans les ADLs existants.

5.2 Le Connecteur

Les ADLs existants modélisent les connecteurs sous différentes formes et sous différents noms par exemple : Acme, SADL, UniCon et Wright modélisent les connecteurs explicitement et ils les appellent "connecteur" (connector), Rapide les appelle "connexion", dans cette section nous ferons une comparaison entre les connecteurs de différents ADLs présentés précédemment.

5.2.1 L'Interface

D'une manière générale, uniquement les ADLs qui modélisent les connecteurs comme des entités de première classe supportent explicitement la spécification d'interface des connecteurs. La plupart des ADLs modélisent les interfaces de composants et de connecteurs de la même manière mais ils font référence à ces interfaces différemment, de cette façon ce sont des points d'interface de connecteurs dans Acme, UniCon**,** et dans Wright se sont des rôles typés et nommés.

5.2.2 Le type

Seulement les ADLs qui modélisent les connecteurs comme des entités de première classe font la distinction entre les types de connecteurs et les instances, par exemple UniCon utilise

des types de connecteurs prédéfinis (Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcCall et RTScheduler).
D'autre part, SADL et Acme offrent une facilité de paramétrisation qui permet une spécification flexible de la signature d'un connecteur.

5.2.3 La sémantique

Un ADL englobe généralement une théorie formelle permettant la modélisation de la sémantique des connecteurs, par exemple Wright est basé sur CSP pour modéliser les connecteurs par des protocoles d'interactions, Rapide est basé sur la théorie des ensembles d'événements partiellement ordonnés (POSET) qui fournit une sémantique opérationnelle [19].
Enfin, il est important de noter que certains ADLs qui modélisent les connecteurs d'une manière explicite, comme Acme par exemple, n'offrent pas toujours des outils pour la définition de la sémantique des connecteurs .

5.2.4 Les contraintes

Les ADLs qui modélisent les connecteurs comme des entités de première classe contraignent leurs utilisation par les interfaces, par exemple Rapide ne place aucune contrainte sur ces connecteurs, par contre UniCon fait une restriction sur le nombre de players de composants attachés à un rôle de connecteur par l'utilisation des attributs MinConns et MaxConns. D'autre part dans Wright la spécification du protocole d'interaction du le rôle permet de poser des contraintes sur le connecteur.

5.2.5 L'évolution

Plusieurs ADLs emploient des mécanismes d'évolution de connecteurs identiques par rapport aux mécanismes d'évolution employés pour les composants, par exemple Acme emploie le sous typage structural et SADL emploie le sous typage de connecteurs et leurs raffinement en utilisant les styles architecturaux, Wright supporte aussi le l'évolution de connecteurs mais par l'utilisation de la paramétrisation.

5.2.6 Les propriétés non fonctionnelles

La plupart des ADLs ne supportent pas la spécification des propriétés non fonctionnelles à part UniCon**.**

5.3 La Configuration

5.3.1 Compréhensibilité de la spécification

Il existe deux types de description de configuration, les descriptions de configuration en ligne (par exemple Rapide) qui tendent vers l'encombrement de la configuration par les détails de connecteurs, dans un autre coté il existe les configurations explicites (tel que dans Wright) qui permettent une meilleure compréhension de la structure de l'architecture.

La compréhensibilité de la spécification peut être améliorée aussi par l'adoption d'une syntaxe claire dans un ADL, enfin la présence d'une notation graphique pour un ADL (tel que UniCon**)** permet d'atteindre la compréhensibilité.

5.3.2 La composition

La plupart des ADLs offrent des moyens explicites pour supporter la composition hiérarchique des composants où la syntaxe utilisée pour les composants est similaire à celle utilisée pour la configuration, par exemple Wright permet la spécification des composants et des connecteurs composites.

Cependant UniCon n'a aucune construction explicite pour la modélisation de l'architecture, mais il modélise l'architecture par un composant composite.

5.3.3 Le raffinement et la traçabilité

Les ADLs existants supportent de manière limitée cette propriété, parmi l'ensemble des ADLs existants on trouve deux qui offrent des construction dédiées au raffinement qui sont SADL et Rapide. SADL permet le raffinement de l'architecture du logiciel afin d'obtenir une description plus « concrète » du système, c'est-à-dire une architecture de plus bas niveau pouvant être proche d'une implémentation en utilisant la notion de "mapping" la figure 1.12 montre un exemple de mapping. Rapide utilise des patrons d'événements qui servent à définir la mise en correspondance entre les architectures de différents niveaux d'abstraction.

```
arch_map  MAPPING FROM arch_L1  TO arch_L2
BEGIN
    comp  -- >  (new_comp )
    conn  -- >   (new_comp!subcomp)
    port  -- >  ( )
          . . .
```

Figure 1.10 : Exemple d'un *mapping* en SADL

5.3.4 L'hétérogénéité

Tous les ADLs qui supportent l'implémentation des architectures sont liés à un langage de programmation particulier, par exemple UniCon supporte uniquement le développement avec des composants implémentés en langage C.

Plusieurs ADLs mettent en place des restrictions qui limitent le nombre et le type de composants et de connecteurs pouvant être supportés, par exemple UniCon ne permet que certains types de composants et de connecteurs (par exemple : PIPES, FILTERS et SEQUENTIAL FILES).

Finalement, un autre aspect d'hétérogénieté est la granularité des composants, la plupart des ADLs supportent la modélisation des composants de petite et de grande taille, par exemple un composant peut modéliser une seule opération, tel que un composant de type computation dans UniCon, comme il peut modéliser un composant de grande taille en utilisant la composition hiérarchique.

5.3.5 Le passage à l'échelle

Pour supporter le passage à l'échelle il existe deux façons, la première consiste à ajouter les nouveaux éléments à l'intérieur de la configuration, la deuxième permet d'ajouter les nouveaux éléments aux bornes de la configuration. Pour supporter ces deux façons un ADL doit possède au minimum l'aspect de composition.

D'autre part les ADLs qui permettent l'attachement d'un nombre quelconque de composants à un connecteur adaptent mieux le passage à l'échelle.

5.3.6 L'évolution de la configuration

L'évolution d'une configuration architecturale peut être considérée sous différents points de vue. Soit par l'ajout de nouveaux composants ou par la permission des descriptions architecturales incomplètes ce qui est plus fréquent durant la phase de conception des systèmes, dans ce contexte Rapide et UniCon ne supportent pas les descriptions architecturales incomplètes.

5.3.7 L'aspect dynamique de l'application

La majorité des ADLs existants considèrent la configuration d'une manière statique, mais Rapide par exemple supporte la manipulation dynamique d'une configuration où tous les changements durant la phase d'exécution doivent être connus à l'avance, en plus il permet la spécification des configurations conditionnelles : la clause where de ce langage permet une

réécriture architecturale en phase d'exécution en utilisant les opérateurs link et unlink, Wright a adopté aussi une approche similaire pour le changement dynamique d'une architecture.

5.3.8 Les contraintes

La plupart des ADL imposent des contraintes qui doivent être validées sur la configuration, par exemple UniCon exige toujours qu'un rôle de connecteur doit être attaché à un player d'un composant. D'autre part plusieurs ADLs permettent la spécification des contraintes globales sur la configuration de la même manière que pour les composants, SADL offre des contraintes sur le raffinement des configurations, Wright permet aussi la spécification des invariants structuraux.

5.3.9 Les propriétés non fonctionnelles

Pour pouvoir spécifier les propriétés non fonctionnelles d'une configuration on doit la traiter comme un composant composite, Rapide par exemple supporte la modélisation de quelques propriétés non fonctionnelles des architectures.

6 Conclusion

Plusieurs difficultés rencontrées lors de développement des applications modernes ont trouvées une solution par l'adoption d'une approche de conception en termes d'architectures. Dans ce contexte, de nombreuses recherches ont permis d'introduire plusieurs langages de description d'architecture (ADL) : UniCon, Aesop, Rapide, Darwin, Wright, etc.

Dans une première partie de ce chapitre et après avoir présenter les aspects essentiels et les concepts fondamentaux des ADLs, nous avons présenté quelques exemples de ces langages les plus connus, c'est-à-dire UniCon, Rapide, Wright, ACME, Olan.

Chacun de ces ADLs prend en compte un certain nombre de préoccupations :

Rapide (D. C. Luckham, 1995) : Modélisation, simulation et analyse du comportement dynamique d'un système à l'exécution. Rapide se fonde sur le concept de POSET (*Partial Ordered event Sets*).

Acme (D Garlan, R Monroe, D Wile, CMU, 1997) : a été conçu comme langage passerelle pour permettre aux différents ADLs d'intéropérer. Cette vocation l'a conduit à se définir comme langage d'architecture générique.

Olan (L Bellissard, M Riveill, INRIA Rhône Alpes, 1995) : Développement, configuration, déploiement et administration d'applications réparties.

UniCon (M. Shaw, CMU, 1995) : Construction et vérification d'architectures à partir d'éléments architecturaux prédéfinis.

Wright (R Allen, D Garlan, 1997) : Vérification formelle, absence de blocage. Il décrit formellement le comportement d'une architecture à l'aide de CSP (*Communicating Sequential Process*).

Nous avons ensuite présenté une étude comparative entre ces ADLs selon les propriétés du composant du connecteur et de la configuration, et nous avons remarqué le manque notable concernant la description et la vérification des propriétés non fonctionnelles dans les ADLs étudiés.

Enfin, nous concluons ce premier chapitre par un tableau récapitulatif contenant les différents concepts et mécanismes offerts par les ADLs présentés au cours de ce chapitre.

		UniCon	**Rapide**	**Wright**	**Olan**	**Acme**
composant	**sémantique**	Les traces d'évènements	POSETS	CSP	aucun	utilise les modèles sémantiques des autres ADLs
	évolution	aucun	Le sous typage structural	instanciation des paramètres	instanciation dynamique et paresseuse	le sous typage structural
	propriétés non fonctionnelles	Des attributs pour le *scheduling* des applications	aucun	aucun	quelques attributs spécifiant l'utilisateur habilité	des annotations arbitraires dans la liste des propriétés
connecteur	**sémantique**	implicite dans liste de propriétés	POSETS, les connexions conditionn--elles	CSP	aucun	utilise les modèles sémantiques des autres ADLs
	évolution	aucun	aucun	instanciation des paramètres	instanciation dynamique et paresseuse	le sous typage structural
	propriétés non fonctionnelles	Des attributs pour le *scheduling* des applications	aucun	aucun	aucun	des annotations arbitraires dans la liste des propriétés

Configuration	**raffinement**	supporte la génération de systèmes	plans de raffinement	aucun	aucun	rep-maps
	propriétés non fonctionnelles	aucun	modélisation des informations temporelles	aucun	quelques attributs spécifiant l'utilisateur habilité	des annotations arbitraires dans la liste des propriétés
	composition	à travers les composants et les connecteurs composites	les mappings entre architectures et interfaces	à travers la computaion et la glue	composant composite	templates et rep-maps

Tableau 1.1 : Les concepts et mécanismes offerts par les ADLs

Chapitre II

Le Langage SADL

1 Introduction

La plupart du temps, l'architecture d'un logiciel est spécifiée d'une manière informelle et intuitive par des diagrammes de type Box-and-Line, ces modèles de description peuvent fonctionner avec des systèmes de structure simple, mais avec des systèmes de structure complexe ces modèles ne sont pas fonctionnels.

Pour cette raison plusieurs modèles centrés sur la description de l'architecture des systèmes logiciels ont été introduits par la communauté de l'ingénierie du logiciel, parmi ces modèles on trouve les langages de description d'architectures (ADLs).

Les ADLs se focalisent sur la structure de l'application globale à un haut niveau d'abstraction plutôt que sur les détails d'implémentation, pour cela ils fournissent une syntaxe bien définie et un cadre conceptuel pour la modélisation d'une architecture conceptuelle du système logiciel [08].

Le but global de ces ADLs est de surmonter les difficultés et les obstacles rencontrés lors de la conception de logiciel telle que la gestion de complexité, de la conformité ou encore de l'interopérabilité. Dans ce contexte, chacun de ces langages offre et utilise des mécanismes et des techniques qui le rend mieux adaptable pour quelques type de systèmes que pour d'autres par exemple :

- Aesop [32] fourni un mécanisme d'instanciation dynamique des composants qui le rende préférable pour la description des systèmes dynamiques;
- Rapide et basé sur le concept de posets (*Partial Ordered event Sets*), il est alors préférable pour la simulation et analyse du comportement dynamique des systèmes à l'exécution;

- Olan offre la notion d'instanciation dynamique et paresseuse et la notion de collection et de désignation associative, ce qui le rend mieux pour le développement, la configuration, le déploiement et l'administration d'applications réparties.

Dans ce chapitre, nous allons présenter le langage de description d'architecture SADL destiné à l'expression des hiérarchies d'architectures grâce à un mécanisme de mise en correspondance explicite entre architectures.

Nous allons tout d'abord présenter les concepts fournis par le langage SADL. Ensuite, dans une autre partie nous présentons le mécanisme de raffinement fourni par ce langage à travers l'exemple du compilateur.

Enfin nous terminons ce chapitre par une brève présentation des limites de ce langage SADL.

2 Présentation du langage SADL

SADL (Structural Architecture Description Language) est un langage de description d'architectures proposé par le laboratoire SRI [26]. Il est basé comme tous les autres ADLs sur les concepts de composant, de connecteur et de configuration. La particularité de ce langage est qu'il est dédié à la description structurale des hiérarchies d'architecture à différents niveaux d'abstractions grâce à un mécanisme de raffinement explicite.

SADL a été conçu pour décrire la structure architecturale (la composition) et de raffinement de styles. Les caractéristiques principales de SADL sont la mise en correspondance explicite entre les architectures, la définition d'architectures génériques, les styles architecturaux et les patrons de raffinement des architectures. Un patron de raffinement est présenté dans une table contenant deux schémas d'architectures, une association des éléments abstraits et concrets, et si nécessaire des contraintes sur un ou les deux schémas [08].

Ce langage offre aussi une notation textuelle précise pour la description de l'architecture logicielle tout en conservant le modèle Box-and-Line intuitif. En effet, il met une distinction forte entre les différents types d'éléments architecturaux (par exemple les composants et les connecteurs) et il définit clairement leur rôle dans la description des architectures. De même le langage SADL offre des facilités non seulement pour la spécification des architectures logicielles, mais aussi la spécification des contraintes sur ces architectures [26].

Alors, le modèle traditionnel de type Box-and-Line est représenté dans SADL comme suit :

les `Boxes` correspondent aux composants, et les `Lines` correspondent aux connecteurs, où chaque composant a un ensemble de points d'interaction avec son environnement. Ces points sont appelés ports. Ce modèle de base a été étendu pour inclure les styles architecturaux, les architectures hétérogènes, les contraintes sémantiques et enfin le mapping entre architectures.

Nous présentons dans ce qui suit les éléments de base d'une architecture SADL :

2.1 L'architecture

Le langage SADL permet la spécification de deux types d'architectures :

1 Les architectures explicites : sont des architectures dans lesquelles chaque élément architectural a un nom et un type bien déterminé.

2 Les architectures génériques : sont des architectures qui peuvent avoir des paramètres qui dénotent un type ou un élément indéfini, cela rend possible par exemple la spécification des architectures qui peuvent contenir un nombre arbitraire d'interfaces et de connexions.

D'autre part, il est important de signaler qu'une architecture explicite n'est rien qu'une instance d'une architecture générique.

2.1.1 Le composant

Un composant dans SADL peut être de différents types : une fonction, un module, un processus, ou encore une variable. Il est caractérisé aussi par une interface définie par un ensemble de ports. Chaque port identifie un point d'interaction entre le composant et son environnement. Ces ports peuvent être soit des ports d'entrées (comme par exemple le port Requête-iport du composant Serveur décrit dans la figure 2.1) soit des ports de sorties (comme le port Réponse-oport du même composant).

```
Serveur: Function
    [Requête-iport : requête –> Réponse-oport : réponse]
```

Figure 2.1 : Exemple de déclaration d'un composant dans SADL

2.1.2 Le connecteur

Un connecteur dans SADL est un élément typé (un sous type du type CONNECTOR), son rôle est de modéliser l'interaction entre les ports des composants.

Un connecteur dans SADL est caractérisé par le type d'éléments portés c'est-à-dire il ne peut recevoir d'une extrémité que des valeurs d'un certain type et de produire comme sortie des valeurs de même type.

La figure 2.2 présente un exemple de déclaration d'un connecteur Channel de type Dataflow-Channel permettant de transporter des éléments de type Data.

```
Channel :   Dataflow_Channel<Data>
```

Figure 2.2 : Exemple de déclaration d'un connecteur dans SADL

2.1.3 La configuration

La structure globale d'une architecture SADL est décrite en termes d'une configuration qui peut contenir deux types d'éléments: les connexions ("CONNECTION") permettant d'associer des connecteurs à des ports de type compatibles et les contraintes ("CONSTRAINT") qui sont utilisées pour lier des objets nommés ou pour mettre en place des restrictions sémantiques sur de telle liaison dans une architecture.

La figure suivante présente une déclaration d'une configuration qui contient une seule connexion Link, liant le port d'entrée Réponse-iport au port de sortie Réponse-oport à travers le connecteur Chanel.

```
CONFIGURATION
    Link: CONNECTION
        = Connects(Chanel, Réponse-oport, Réponse-iport)
```

Figure 2.3 : Exemple de déclaration d'une configuration dans SADL

2.2 Les styles architecturaux

La définition d'un style architectural selon Garlan [39] est la suivante :

« *Un **style architectural** permet de caractériser une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques*».

De ce fait, les styles architecturaux regroupent des éléments de conception réutilisables.

Ainsi un style est constitué d'un vocabulaire d'objets typés et des contraintes de liaison entre ces objets dans une architecture explicite, comme exemples de styles architecturaux nous pouvons citer les styles : Pipe and Filter, client/server et dataflow.

En d'autres termes, un style architectural précise les propriétés et les contraintes qui fixent les règles et les limites de construction de l'architecture. Il caractérise donc une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques. Donc l'objectif des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants et des connecteurs [09].

En plus, l'utilisation des styles architecturaux dans une description d'une architecture offre les avantages suivants [05] :

- Ø elle favorise la réutilisation dès la conception du système;
- Ø elle autorise une réutilisation significative de code;

- Ø elle favorise la normalisation des familles d'architectures, ce qui facilite la compréhension de l'organisation d'un système;
- Ø elle autorise l'utilisation d'analyses spécifiques au style concerné.

2.3 Le système de type dans SADL

Le système de type du langage SADL offre plusieurs caractéristiques utiles pour la description des architectures :

SADL est algébriquement extensible, c'est-à-dire, il est possible d'introduire de nouveaux types qui ne sont pas dérivés d'un type prédéfini, cette caractéristique est très importante en ce qui concerne la définition des styles architecturaux qui peuvent inclure de nouveaux types de composants et de connecteurs.

SADL est polymorphe, c'est-à-dire, des types peuvent être passés comme paramètres.

SADL supporte le sous typage qui permet la définition de nouveaux objets architecturaux dans une classe d'objets particulière.

Par exemple la déclaration :

Client : TYPE <= COMPONENT

Exprime que Client est un sous type du type de base COMPONENT, d'une manière similaire la déclaration :

CS_Channel : TYPE <= CONNECTOR

exprime que CS_Channel est un sous type du type de base CONNECTOR .

De plus, il est possible de créer de nouveaux types qui regroupent tous les éléments d'un type existant qui ont certaines propriétés, la déclaration :

Local_Client : TYPE = { c : Client | Local(c) }

indique que Local_Client est un sous type du type Client, et tous les éléments de ce nouveau type doivent satisfaire le prédicat Local.

SADL permet aussi d'introduire des contraintes sur les types, par exemple, une contrainte de compatibilité des types des composants et des connecteurs est déclarée de façon informelle comme suit : si un connecteur c porte une valeur depuis un port p1 d'un composant vers un port p2 d'un autre composant, alors le type de p1 doit être un sous type du type de c, qui lui a son tour doit être un sous type du type de p2.

Des contraintes additionnelles peuvent être associées aux composants et aux connecteurs, d'autre part, SADL possède un contrôleur de type qui vérifie que toutes ces contraintes sont vérifiées.

3 Un exemple d’une architecture SADL

Pour mieux comprendre les concepts de base du langage SADL, nous les présentons dans ce qui suit à travers l'exemple traditionnel de l'architecture du compilateur repris de [26], la figure 2.4 montre cette architecture.

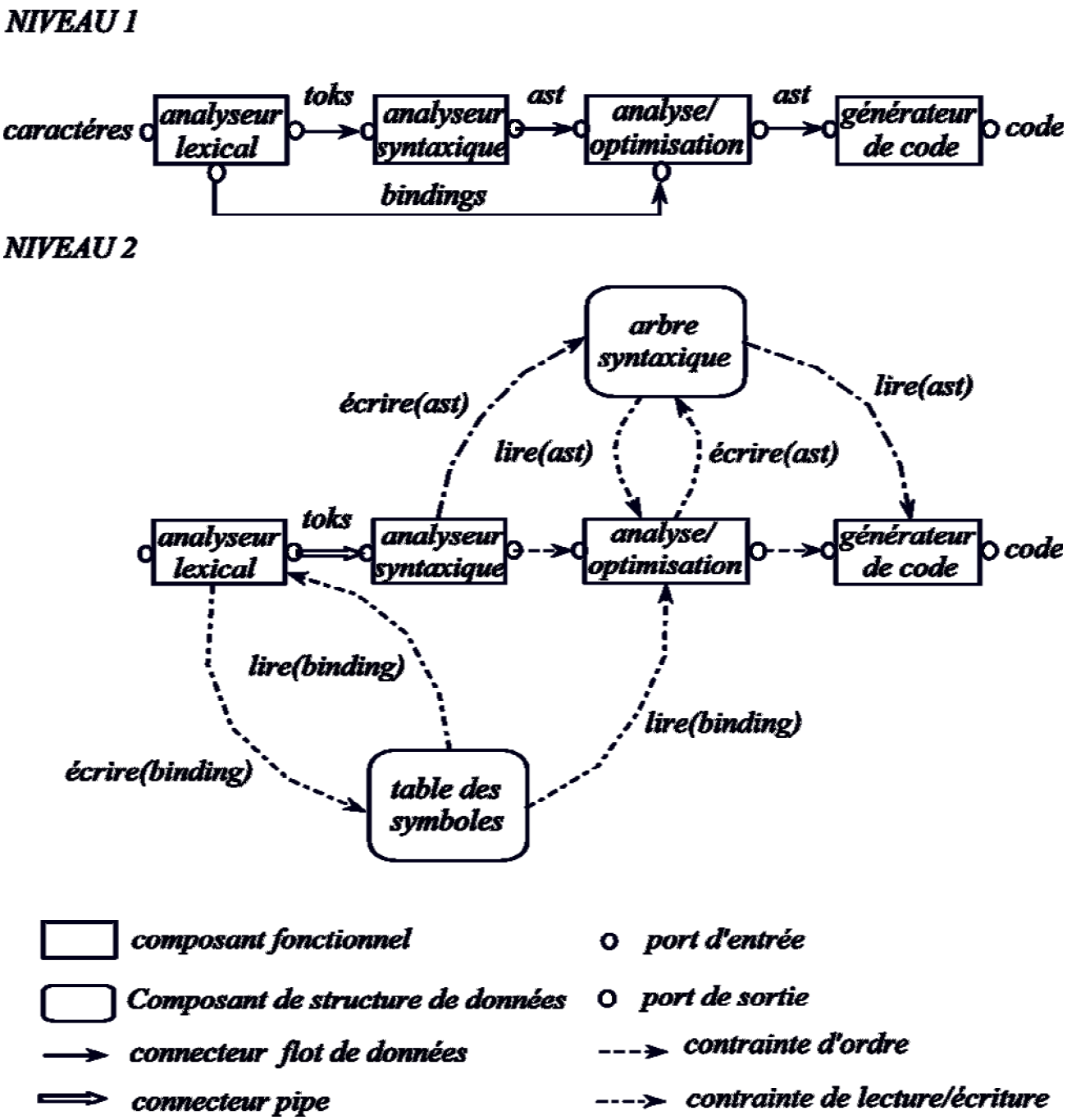


Figure 2.4 : L'architecture du compilateur

La figure ci-dessus présente l'architecture du compilateur à travers deux niveaux, le premier niveau montre les différents composants et connecteurs existants dans l'architecture, le deuxième niveau présente le raffinement du premier niveau par l'ajout de structures de données.

3.1 Le premier niveau de l'architecture du compilateur

La spécification du premier niveau de l'architecture du compilateur en utilisant la syntaxe de SADL est présentée dans la figure suivante :

```
compiler_L1:ARCHITECTURE [char_iport: SEQ(character) –> code_oport: code]
    IMPORTING character, code, token, binding, ast FROM compiler_types
    IMPORTING Function FROM Functional_Style
    IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
BEGIN
 COMPONENTS
     lexical_analyzer: Function
          [char_iport: SEQ(character)
             –> token_oport: SEQ(token) , bind_oport: SEQ(binding)]
     parser: Function
          [token_iport: SEQ(token) –> base_ast_oport: ast]
     analyzer_optimizer: Function
          [base_ast_iport: ast bind_iport: SEQ(binding)
             –> full_ast_oport: ast]
     code_generator: Function [full_ast_iport: ast –> code_oport: code]
CONNECTORS
    token_channel:     Dataflow_Channel<SEQ(token)>
    bind_channel:      Dataflow_Channel<SEQ(binding)>
    base_ast_channel:  Dataflow_Channel<ast>
    full_ast_channel:  Dataflow_Channel<ast>
CONFIGURATION
    token_flow: CONNECTION
        = Connects(token_channel, token_oport, token_iport)
    bind_flow: CONNECTION
        = Connects(bind_channel, bind_oport, bind_iport)
    base_ast_flow: CONNECTION
        = Connects(base_ast_channel, base_ast_oport, base_ast_iport)
    full_ast_flow: CONNECTION
        = Connects(full_ast_channel, full_ast_oport, full_ast_iport)
END compiler_L1
```

Figure 2.5 : Le premier niveau de l'architecture du compilateur décrit en SADL

Dans cet exemple la première ligne :

compiler_L1: ARCHITECTURE

exprime que compiler_L1 est le nom de l'architecture décrite, en plus la deuxième ligne :

[char_iport: SEQ(character) –> code_oport: code]

montre que l'architecture compiler_L1 accepte une séquence de caractères en entrée et produit du code en sortie.

Nous pouvons aussi dire à partir de cette ligne que le seul port d'entrée de l'architecture compiler_L1 est le port char_iport et le seul port de sortie est le port code_oport.

SADL permet aussi de réaliser une spécification modulaire d'une architecture cela se fait grâce à un mécanisme d'importation, dans L'exemple cité plus haut, les trois premières lignes :

```
IMPORTING character, code, token, binding, ast FROM compiler_types
IMPORTING Function FROM Functional_Style
IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
```

servent à importer les différents types utilisés dans l'architecture compiler_L1:

les types : character, code, token, binding et ast sont importés d'une autre spécification appelée compiler_types.

Le prédicat du type Function est importé du style Functional_Style,

Les prédicats : Dataflow_Channel et connects à partir du style Dataflow_Style.

Généralement une architecture peut également exporter des éléments en vue de leurs utilisations une architecture englobant.

Une autre caractéristique de l'architecture compiler_L1 est l'invisibilité de la structure interne mais si nous examinans le style Functional_Style, nous retrouvons la déclaration:

```
Function : TYPE <= COMPONENT
```

qui exprime que Function est un sous type du type prédéfini COMPONENT, de la même manière dans Dataflow_Style on trouve la déclaration :

```
Dataflow_Channel : TYPE <= CONNECTOR
```

qui énonce que Dataflow_Channel est un sous type du type CONNECTOR, et

```
Connects : PREDICATE(3)
```

qui déclare que Connects est un prédicat ternaire, Dataflow_Style contient également des contraintes de la forme :

```
connects_argtype_1 : CONSTRAINT =
(/\ x)( /\y) (/\ z)        [Connects(x, y, z) => Dataflow_Channel(x) ]
connects_argtype_2 :CONSTRAINT =
(/\ x)( /\y) (/\ z)        [Connects(x, y, z ) => Outport(y) ]
connects_argtype_3 :CONSTRAINT =
(/\ x)( /\y) (/\ z)        [Connects(x, y, z) =>  Inport(z) ]
```

Ces contraintes doivent être satisfaites par toute architecture important ce style. Ces contraintes exigent que le premier argument du prédicat Connects doit être de type Dataflow_Channel, et que le deuxième argument doit être un port de sortie et le troisième argument doit être un port d'entrée.

Le corps de compiler_L1 est constitué de trois parties séparées par les mots clés : COMPONENTS, CONNECTORS, CONFIGURATION permettant de fournir une meilleure lisibilité de cette description.

La première déclaration de composants dans la section COMPONENTS :

```
lexical_analyzer: Function
    [char_iport: (SEQ_character)
        –> token_oport: SEQ(token) ,bind_oport: SEQ(binding)]
```

introduit un composant appelé lexical_analyzer qui a un seul port d'entrée de nom char_iport acceptant une séquence de tokens, et deux ports de sortie bind_oport produisant une séquence de bindings.

Ainsi, la première déclaration dans la section CONNECTORS :

```
token_channel:   Dataflow_Channel<SEQ(token)>
```

déclare un connecteur token_channel de type Dataflow_Channel qui transporte une séquence de données de type token.

Enfin, la section CONFIGURATION définit les propriétés architecturales de connexions cela par l'introduction d'un ensemble de contraintes sur les connexions par exemple la déclaration :

```
token_flow: CONNECTION
    = Connects(token_channel, token_oport, token_iport)
```

exprime que le connecteur token_channel lie le port de sortie token_oport du composant lexical_analyzer au port de d'entrée token_iport du composant Parser .

3.2 Le deuxième niveau de l'architecture du compilateur

La spécification du deuxième niveau de l'architecture du compilateur est donnée dans la figure ci-dessous.

```
compiler_L2: ARCHITECTURE [char_iport: SEQ(character) –> code_oport: code]
    IMPORTING character, code, token, binding, ast FROM compiler_types
    IMPORTING Function FROM Functional_style
    IMPORTING Pipe, Finite_Stream, Connects FROM Process_Pipeline_style
```

```
    IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
    IMPORTING Start_After_Finish_Of FROM Batch_Sequential_style
BEGIN
COMPONENTS
   lexical_analyzer_module: ARCHITECTURE
  [char_iport: SEQ(character) –> token_oport: Finite_Stream(token)]
      EXPORTING lexical_analyzer, symbol_table
      IMPORTING character, token, binding FROM compiler_types
      IMPORTING Function FROM Functional_style
      IMPORTING Finite_Stream FROM Process_Pipeline_style
      IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
   BEGIN
   COMPONENTS
      lexical_analyzer: Function
      [char_iport: SEQ(character) –> token_oport: Finite_Stream(token)]
      symbol_table: Variable (SEQ(binding)) [–>]
   CONFIGURATION
     write_bind: CONSTRAINT = Writes(lexical_analyzer, symbol_table)
     read_bind: CONSTRAINT = Reads(lexical_analyzer, symbol_table)
   END lexical_analyzer_module
   parser: Function[token_iport: Finite_Stream(token) –> ]
   analyzer_optimizer: Function[–>]
   code_generator: Function[–> code_oport:  code]
   abstract_syntax_tree: Variable(ast)[ –>]
CONNECTORS
    token_pipe: Pipe<Finite_Stream(token)>
CONFIGURATION
    token_flow: CONNECTION
       = Connects(token_pipe, token_oport, token_iport)
    read_bind: CONSTRAINT
       =Reads(analyzer_optimizer,lexical_analyzer!module_symbol_table)
    write_base_ast: CONSTRAINT
       = Writes(parser, abstract_syntax_tree)
    read_base_ast: CONSTRAINT
       = Reads(analyzer_optimizer, abstract_syntax_tree)
```

```
    write_full_ast: CONSTRAINT
       = Writes(analyzer_optimizer, abstract_syntax_tree)
    read_full_ast: CONSTRAINT
       = Reads(code_generator, abstract_syntax_tree)
    precedence_1: CONSTRAINT
       = Starts_After_Finish_Of (analyzer_optimizer, parser)
    precedence2: CONSTRAINT
       =Starts_After_Finish_Of(code_generator, analyzer_optimizer)
END compiler_L2
```

Figure 2.6 : Le deuxième niveau de l'architecture du compilateur en SADL

L'architecture de ce deuxième niveau du compilateur est hétérogène car elle importe des éléments architecturaux à partir de différents styles (Batch_Sequential_style, Shared_Memory_style, Process_Pipeline_style).

Dans ce deuxième niveau on observe que le composant lexical_analyzer du premier niveau a été remplacé par la sous architecture lexical_analyzer_module qui inclut la déclaration d'un composant lexical_analyzer et une structure de donnée appelée symbol_table. Tous deux sont exportés pour que les contraintes de la configuration globale puissent les utiliser.

En plus, la ligne :

symbol_table: Variable (SEQ(binding)) [–>]

indique que le composant symbol_table est une variable contenant une séquence de binding, et la signature [–>] indique qu'il n'a aucun port .

4 Le mécanisme de raffinement dans SADL

La description de l'architecture d'un système logiciel en une seule étape est une tâche difficile, un langage de description d'architecture doit donc fournir des mécanismes permettant aux développeurs de faciliter la spécification de logiciels complexes, parmi ces mécanismes on trouve le mécanisme de raffinement d'architectures.

Le raffinement d'une architecture logicielle permet d'obtenir à partir d'une description architecturale abstraite une description plus concrète, c'est-à-dire une architecture de plus bas niveau pouvant être proche d'une implémentation. Le raffinement d'architectures logicielles constitue une base correcte de développement des systèmes logiciels. Pour cela, le raffinement d'une architecture logicielle doit garantir que l'architecture logicielle concrète est valide au regard de l'architecture logicielle initiale. Ce raffinement conduit à définir de nouveaux composants, de nouvelles interconnections, etc. Ceci se fait au cours du processus

de développement centré architecture où sont opérées des transformations en plusieurs étapes successives [08].

Un des points fort du langage SADL est la présence du mécanisme de raffinement, en effet le langage SADL est dédié à la description structurale des hiérarchies d'architecture à différents niveaux d'abstractions grâce à un mécanisme de raffinement explicite. Ce mécanisme permet la transformation systématique d'une architecture abstraite vers une autre contenant beaucoup plus de détails, en respectant un ensemble explicite de règles de transformations.

Ainsi, nous pouvons distinguer deux types de raffinements dans SADL un raffinement vertical et un raffinement horizontal.

Le raffinement vertical permet de passer d'un niveau de description abstrait à un niveau plus concret tout en impliquant des propriétés du niveau abstrait, alors un raffinement vertical permet le développement d'une architecture concrète d'un système logiciel à partir d'une architecture abstraite, alors une hiérarchie verticale est une séquence linéaire de deux ou plusieurs architectures qui diffèrent par certains aspects, en partant d'une même architecture de départ, prenant à titre d'exemple une architecture abstraite contenant des composants fonctionnels reliés par des connections de flots de données. A un niveau concret, cette architecture peut être implémentée en termes de procédures, de connexions de contrôle et de variables partagées. Les étapes de raffinement vertical rajoutent de plus en plus de détails aux modèles abstraits jusqu'à l'obtention d'une description concrète du modèle architectural [08].

Le raffinement horizontal souvent appelé décomposition n'engendre pas de changement dans le niveau d'abstraction mais il offre la possibilité de rajouter de nouvelles parties à une description, un exemple de ce type de raffinement est l'explosion d'un composant abstrait en plusieurs sous composants qui ont le même niveau d'abstraction.

Pour permettre ces deux types de raffinement, SADL inclut un support pour des mises en correspondance (mapping) explicites entre architectures. Une mise en correspondance dans SADL est spécifiée par un ensemble de paires appelées associations de la forme :

```
arch_map : MAPPING FROM arch1 TO arch2
BEGIN
comp – –> (new_comp)
conn – –> (new_comp!subcomp)
port – –> ( )
END
```

Figure 2.7 : Forme générale d'un mapping

Plus précisément une association peut être de trois types : une association d'éléments architecturaux (composant, connecteur, port ou autres éléments), une association de contraintes ou encore une association de prédicats.

Pour les associations d'éléments architecturaux et les associations de contraintes la partie droite d'une association est constituée d'une liste (peut être vide) de noms d'éléments, ces associations expriment que la liste d'éléments de la partie droite implémente l'élément de la partie gauche, par exemple une association peut indiquer que certaine combinaison de connecteurs et de composants d'un niveau concret offre une implémentation pour un connecteur d'un niveau abstrait. Dans le cas des contraintes, la satisfaction des contraintes de la partie droite garantit la satisfaction de la contrainte de la partie gauche.

Pour les associations de prédicats, la partie droite est définie en terme d'un nouveau concept appelé prédicat virtuel, l'idée derrière l'association d'un prédicat à un prédicat virtuel est de dire que les objets désignés par le prédicat de la partie gauche sont implémentés par les objets désignés par le prédicat virtuel énoncé dans la partie droite.

Afin d'illustrer le mécanisme de raffinement d'architectures dans SADL, nous reprenons l'exemple du compilateur dont l'architecture est montrée dans la figure 2.4, afin d'obtenir le deuxième niveau de l'architecture du compilateur (figure 2.6) à partir du premier niveau (figure 2.5) nous présentons dans la figure 2.8 l'ensemble des règles de correspondances permettant de passer du premier niveau vers le deuxième niveau.

```
compiler_map: MAPPING FROM compiler_L1 TO compiler_L2
BEGIN
  lexical_analyzer  – –> (lexical_analyzer_module)
  bind_oport        – –> ( )
  base_ast_oport    – –> ( )
  base_ast_iport    – –> ( )
  bind_iport        – –> ( )
  full_ast_oport    – –> ( )
  full_ast_iport    – –> ( )
  token_channel     – –> (token_pipe)
  bind_channel      – –> (lexical_analyzer_module! symbol_table)
  base_ast_channel  – –> (abstract_syntax_tree)
  full_ast_channel  – –> (abstract_syntax_tree)
  bind_flow   – –> (lexical_analyzer_module!write_bind,read_bind)
  base_ast_flow  – –>  (write_base_ast, read_base_ast, precedence_1 precedence_2)
```

```
full_ast_flow  – –> (write_base_ast, read_base_ast, precedence_1, precedence_2)
END compiler_map
```

Figure 2.8 : Règles de mise en correspondance de l'architecture compiler_L1 vers compiler_L2

Par convention, le langage SADL prend en compte les associations implicites entre les éléments architecturaux de même nom, ainsi la première association du mapping de la figure ci-dessus :

lexical_analyzer – –> (lexical_analyzer_module)

énonce que la sous architecture lexical_analyzer_module du deuxième niveau implémente le composant lexical_analyzer du premier niveau. L'association suivante :

bind_oport – –> ( )

exprime que le port de sortie bind_oport du composant lexical_analyzer du premier niveau a été enlevé de la description compiler_L2, l'élément qui implémente le port bind_oport n'a pas été déclaré explicitement dans le deuxième niveau car le connecteur de type flots de données qui transporte l'ensemble des bindings à partir de l'analyseur lexical vers le composant responsable de l'optimisation, a été implémenté par une lecture d'une structure de donnée au niveau de l'analyseur lexical. Nous pouvons alors dire que le port bind_oport est implémenté par un emplacement mémoire dans le composant lexical_analyzer de la sous architecture lexical_analyzer_module.

5 Sémantique formelle du langage SADL

La sémantique derrière une architecture SADL est représentée par une théorie logique basée sur une logique de premier ordre étendue (appelée ω-logic) contenant des quantificateurs numériques.

Cette logique est équivalente en puissance à une faible logique de deuxième ordre, une logique ordinaire de premier ordre n'a pas une puissance suffisante pour décrire certaines propriétés des architectures génériques comme par exemple la connectivité et la finitude.

Egalement, un mapping SADL détermine une mise en correspondance entre deux ω-logic théories.

En plus, un mapping doit assurer non seulement que toutes les propriétés d'une architecture abstraite sont préservées par l'architecture concrète, mais il doit assurer aussi qu'aucune nouvelle propriété inexistante dans la description abstraite peut être inférée à partir de l'architecture concrète cela permet de prévenir l'apparition de connexions inattendues dans

l'implémentation, afin d'atteindre ce but SADL emploie un concept mathématique qui est l'interprétation des théories [36].

Dans une interprétation de théories chaque niveau de description est représenté par une théorie, par exemple si nous avons l'interprétation I et les deux théories β et β' qui correspondent respectivement à la description abstraite et à la description concrète, alors pour chaque formule F nous devons avoir :

si $F \in \beta$ alors $I(F) \in \beta'$

et

si $F \notin \beta$ alors $I(F) \notin \beta'$

6 Conclusion

Dans ce deuxième chapitre, nous avons étudié d'une manière détaillée le langage SADL. Nous avons présenté les concepts fournis par ce langage tout en se concentrant sur le mécanisme de raffinement structurel fourni par ce langage.

Le langage SADL est un des langages de description d'architecture les plus utilisés, il fournit des mécanismes permettant le raffinement d'architectures logicielles à travers différents niveaux d'abstraction, ce qui le rend bien adapté pour la spécification et le développement des systèmes logiciels complexes et de grande taille; il a été utilisé pour formaliser l'architecture de référence X/Open DTP (Distributed Transaction Processing) à travers plusieurs niveaux d'abstraction [26], le niveau le plus haut dans DTP SADL hiérarchie est décrit en utilisant un modèle de type flot de données, et le niveaux le plus bas a été spécifié en termes d'appels de procédures en langage C [26], SADL a été utilisé aussi pour assurer la consistance entre l'architecture et l'implémentation d'un système de contrôle de puissance utilisé par la compagnie Tokyo Electric Power[17] .

De point de vue syntaxique, SADL est doté d'un ensemble de concepts assez complet pour définir les structures architecturales des systèmes logiciels. Cependant la sémantique de SADL n'est pas bien définie, la logique utilisée dans ce contexte ne permet pas la définition de la dynamique et le comportement des éléments architecturaux.

En plus, malgré le besoin de la spécification des propriétés non fonctionnelles comme la qualité de service, la sécurité ou les performances d'une application, le langage SADL comme la plupart des ADLs n'offre pas de moyens ou de supports pour spécifier ces propriétés (voir tableau 1.1).

Le but de ce travail est de cadrer le langage SADL dans un formalisme mathématique approprié pour pallier ses lacunes, il s'agit de la logique de réécriture.

Chapitre III

La Logique de Réécriture et le Langage Maude

1 Introduction

La spécification formelle vise la construction de logiciels fiables par la mise en valeur d'un certain nombre de propriétés que doit respecter le produit logiciel, ainsi le développement de logiciels sera fondé sur des bases mathématiques solides. En plus, le développement basé spécification formelle permet de faciliter l'intégration des modules logiciels existants et de favoriser la réutilisation.

En effet, différents formalismes ont été introduits par plusieurs équipes de recherche dans le monde. La logique de réécriture se compte parmi les formalismes les plus utilisés. Elle a été introduite initialement par José Meseguer [31,40], comme une conséquence de son travail sur les logiques générales pour décrire les systèmes concurrents. Dans cette logique, l'aspect statique et dynamique des systèmes est représenté à l'aide des théories de réécriture. La logique de réécriture bénéficie aussi de la présence de nombreux langages et environnements opérationnels, le plus connus étant le langage Maude créé par le laboratoire SRI. C'est un langage déclaratif très expressif autour duquel se trouve un environnement composé de plusieurs outils.

Dans ce chapitre nous présentons la logique de réécriture à travers ses éléments de base ainsi que le langage Maude.

2 La logique de réécriture

La logique de réécriture est une logique de changement permettant l'expression du calcul concurrent non déterministe d'une manière très convenable. Dans cette logique l'aspect statique des systèmes est représenté par une logique sous-jacente appelée logique équationnelle d'appartenance (*MemberShip Equational Logics*). L'aspect dynamique est

représenté par des théories de réécritures décrivant les transitions possibles entre les états du système concurrent.

La logique équationnelle nous offre la possibilité de réaliser des spécifications modulaires. C'est une logique de Horn multi-sorté où les énoncés de base sont :

- soit des équations conditionnelles et/ou non conditionnelles de la forme t = t' (if c), où t, t' sont des termes algébriques de même sorte et c correspond à une expression booléenne;
- soit des assertions d'appartenance de la forme t : s où t est un terme algébrique et s correspond à une sorte.

Les théories de réécriture décrivent l'aspect dynamique des systèmes c'est-à-dire les transitions entre les états du système concurrent, cela se fait grâce à un ensemble de règles de réécritures conditionnelles et/ou non conditionnelles, de la forme : t -> t' (if c) où t et t' sont des termes algébriques de même sorte et c est une expression booléenne.

La logique de réécriture est aussi une logique réflexive, c'est-à-dire elle offre la possibilité de représenter et de simuler l'aspect métathéorie, il existe alors une théorie universelle U dans laquelle il est possible de représenter n'importe quelle théorie R (incluant U elle-même) par un terme $\overline{R}$, et tout terme t, t' dans R par des termes $\bar{t}$, $\overline{t'}$, et toute paire $\langle R,t \rangle$ par un terme $\langle \overline{R},\overline{t'} \rangle$ telle que l'équivalence suivante est vérifiée :

$$R \vdash t \to t' \Leftrightarrow U \vdash \langle \overline{R},\bar{t} \rangle \to \langle \overline{R},\overline{t'} \rangle .$$

Une autre caractéristique importante de la logique de réécriture est qu'elle représente un cadre logique et sémantique où différentes logiques et modèles de calcul bien connus peuvent être exprimés.

L'avantage essentiel de la logique de réécriture est que les spécifications réalisées peuvent être efficacement exécutées, cela offre un moyen puissant de prototypage et d'exécution. En plus, l'interopérabilité de ces logiques et modèles devient possible grâce à leurs représentations homogènes en logique de réécriture [28].

Dans ce contexte nous pouvons citer à titre d'exemple, les systèmes de transitions étiquetés, les réseaux de Petri, CCS, la machine chimique abstraite [11], et les ECATNets [37].

En ce qui concerne l'architecture logicielle Josée Meseguer et Carolyn Talcott ont jugé dans [21] que la logique de réécriture peut être utilisée aussi comme un cadre sémantique pour l'interopérabilité des différents ADLs.

Dans un autre travail [08], la logique de réécriture a été utilisée par Karim MEGZARI dans l'implémentation d'un environnement pour le raffinement des architectures logicielles. Les auteurs de [03] ont fourni au langage de description d'architecture CBABEL une sémantique basée sur la logique de réécriture.

2.1 La théorie équationnelle

2.1.1 La signature

Une signature $\sum$ de la logique équationnelle d'appartenance décrit la syntaxe de la théorie, elle est constituée de deux ensembles S et F, nous pouvons alors écrire $\sum = (S, F)$ où :

- S est un ensemble fini non vide de symboles de sortes;
- F est un ensemble non vide d'opérations (appelées aussi symboles de fonction) tel que chaque $f \in F$ possède une arité : $s_1, s_2, ..., s_n, s$.

Comme exemple de signatures nous présentons dans la figure 3.1 une signature des booléens dans laquelle on déclare la sorte bool et trois symboles de fonctions.

```
sortes: bool
operations:   true : -> bool
              False : -> bool
              Not (_) : -> bool
```

Figure 3.1 : La signature des booléens

2.1.2 $\sum$-algèbre

Une $\sum$-algèbre A d'une signature $\sum$ est donnée par :

- un ensemble A_{Si} pour toute sorte $s_i \in S$ (cet ensemble est appelé le support de s_i);
- pour chaque symbole de fonction $f \in F$ d'arité $s_1, s_2, ..., s_n, s$ une fonction

  $f^A : A_{S1} * A_{S2} * ... * A_{Sn} \rightarrow A_S$, appelée interprétation de f.

Une signature peut avoir alors plusieurs $\sum$-algèbres, dans la figure ci-dessous nous présentons deux $\sum$-algèbres pour la signature des booléens présentée dans la figure3.1.

Σ-algèbool$_{A1}$ = {0,1}	Σ-algèbool$_{A2}$ = {vrai,faux}
true$_{A1}$ =1	true$_{A2}$ = faux
false$_{A1}$ = 0	false$_{A2}$ = vrai
not$_{A1}$ (x) = 1-x	not$_{A2}$ (x) = non(x)

Figure 3.2 : Les $\sum$-algèbres

2.1.3 Les termes

Les termes sont utilisés pour designer des expressions construites à l'aide des opérations définies par une signature et enrichies par des variables.

Etant donnée une signature $\sum = (S, F)$ et une famille de variables $X = \{X_S, s \in S\}$ classés par sorte (sachant que $\sum \cap X = \{\ \}$) l'ensemble de termes de $T_\Sigma(X)$ sur X et $\sum$ est donné par :

- Ø toute variable $x \in X_S$ correspond à un terme de sorte s dans $T_\Sigma(X)$;
- Ø si f est une fonction d'arité $s_1, s_2, ..., s_n, s$ et $t_1, t_2, ..., t_n$ sont des termes de $T_\Sigma(X)$ de sorte $s_1, s_2, ..., s_n$ respectivement, alors $f(t_1, t_2, ..., t_n)$ est un terme de $T_\Sigma(X)$ de sorte s.

Un terme peut être aussi sans variables dans ce cas, il est dit clos, l'ensemble des termes clos se note $T(X)$.

A titre d'exemple la phrase not(true) correspond à un terme défini par la signature bool.

2.1.4 ∑-équation

Sachant que t et t' sont deux termes de $T_\Sigma(X)$, une $\sum$-équation est une paire de termes notée : $t = t'$.

Comme exemple de $\sum$-équation nous pouvons citer la $\sum$-équation : not(true) = false .

2.1.5 Les Règles du raisonnement équationnel

Le raisonnement dans la logique équationnelle est aussi appelé le remplacement d'égaux par égaux. Son rôle principal et de calculer la forme canonique des termes algébriques à l'aide d'un ensemble de règles dites règles de raisonnement équationnel.

Alors nous disons qu'une équations $t = t'$ est prouvable dans E et on écrit E |= $t = t'$ si et seulement $t = t'$ peut être obtenu par une application finie des règles suivantes :

Réflexivité : $t = t$.

Symétrie : $t = t' \Rightarrow t' = t$.

Transitivité : $t = t' \,\&\, t' = t'' \Rightarrow t = t''$.

Substitution : $t = t' \Rightarrow t[t''/x] = t'[t''/x]$.

Congruence : $t_1 = t_1' \,\&\, ... \,\&\, t_n = t_n' \Rightarrow f(t_1, ..., t_n) = f(t_1', ..., t_n')$.

2.2 La théorie de réécriture

Une théorie de réécriture R est un quadruplet (Σ,E,L,R), où (Σ,E) présente la signature définissant la structure des états du système, L est un ensemble d'étiquettes et R est un

ensemble de paires $L \times T_{\Sigma,E}(X)^2$ dont le premier élément est une étiquette et le deuxième correspond à une paire de classe d'équivalence de termes, et $X = \{x_1,...,x_n,...\}$ est l'ensemble des variables, les éléments de R sont appelés règles de réécriture, et ont aussi la forme $r : [t] \to [t']$.

En plus, pour indiquer que $\{x_1,...,x_n\}$ est l'ensemble des variables apparus dans t ou t' nous pouvons écrire $r : [t(x_1,...,x_n) \to t'(x_1,...,x_n)]$ ou aussi par une notation abréviée $r : [t(\bar{x}) \to t'(\bar{x})]$.

2.2.1 Les règles de déduction

Etant donnée une théorie de réécriture R nous disons qu'une règle $[t] \to [t']$ est prouvable dans R ou R implique $[t] \to [t']$ si et seulement si $[t] \to [t']$ peut être obtenue par une application finie des règles de déduction suivantes (où nous assumons que les termes sont bien formés et $t(\bar{w}/\bar{x})$ symbolise la substitution simultanée des w_i par les x_i dans t) [20] :

1 La réflexivité : pour chaque $[t] \in T_{\Sigma,E}(X)$,

$$\overline{[t] \to [t]}$$

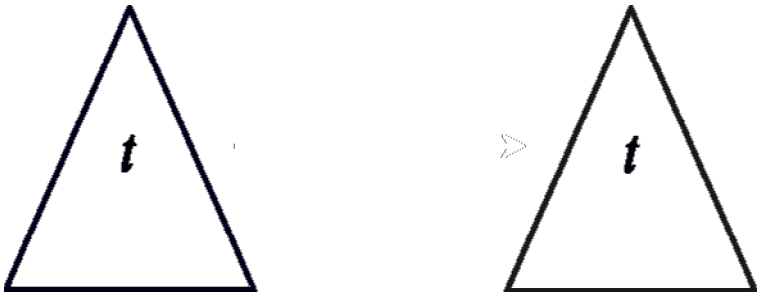


Figure 3.3 : La règle de réflexivité

2 La congruence : pour chaque $f \in \Sigma_n, n \in N$,

$$\frac{[t_1] \to [t_1'] \text{ h } [t_n] \to [t_n']}{[f(t_1, \text{h } t_n)] \to [f(t_1', \text{h }, t_n')]}$$

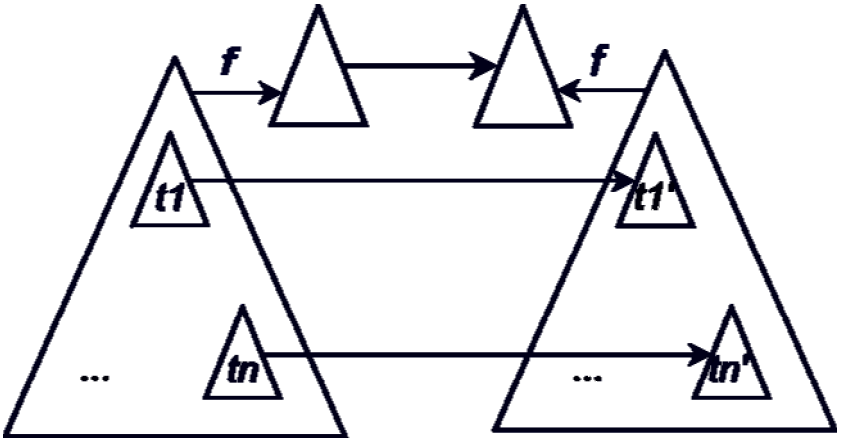


Figure 3.4 : La règle de congruence

3 Le remplacement : pour chaque règle $r : [t(x_1, \mathsf{h}, x_n)] \rightarrow [t'(x_1, \mathsf{h}, x_n)]$ dans R :

$$\frac{[w_1] \rightarrow [w_1'] \; \mathsf{h} \; [w_n] \rightarrow [w_n']}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w'}/\overline{x})]}$$

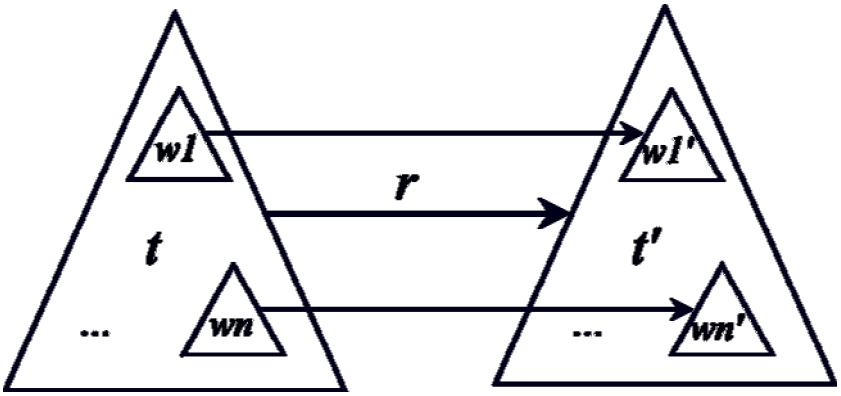


Figure 3.5 : La règle de remplacement

4 La transitivité :

$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

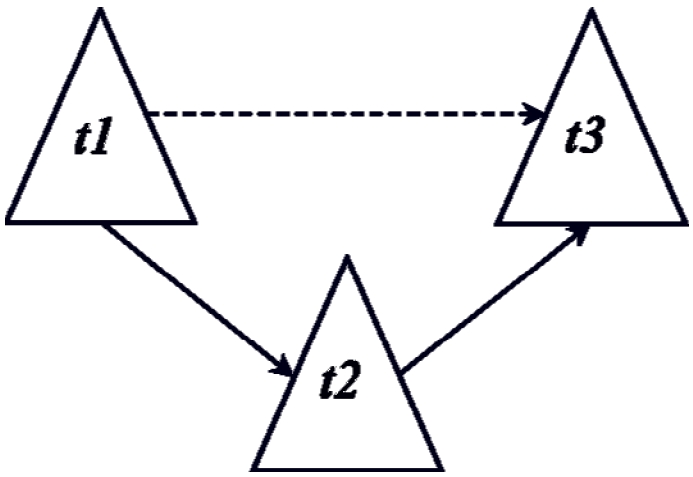


Figure 3.6 : La règle de transitivité

2.2.2 Le modèle sémantique d'une théorie de réécriture

Le modèle sémantique d'une théorie de réécriture correspond à une catégorie $T_R(X)$.

Les constituents de base des catégories sont les objets et les morphismes entre ces objets. Dans le cas des théorie de réécritures les objets correspondent aux classes d'équivalences des termes modulo l'ensemble des équations de E, et les morphismes représentent les α -preuves entre ces classes. Pour plus de détails sur l'aspect sémantique voir [20,35].

3 Le langage Maude

Le langage Maude [10] est une implémentation de haute performance de la logique de réécriture, il supporte les théories de réécriture et les théories équationnelles d'appartenance. Dans cette section nous allons présenter brièvement quelques caractéristiques de ce langage.

Maude est un langage déclaratif de haut niveau et un système de haute performance défini par J Meseguer, il constitue la seule implémentation de la logique de réécriture qui emploie systématiquement et d'une manière efficace le mécanisme de réflexivité.

Ce langage est basé sur la logique de réécriture, alors un programme écrit dans ce langage correspond à une théorie de la logique de réécriture, c'est à dire une signature et un ensemble de règles de réécriture. En plus, Maude est multi paradigme, il supporte et combine la programmation fonctionnelle, système et orienté objet et il inclus en plus un certain nombre de modules prédéfinis permettant de faciliter la tâche de spécification.

Maude est aussi utilisé pour : la spécification, le prototypage, la vérification d'un large éventail d'applications. Différents formalismes et logiques ont été décrit en Maude.

Ce langage a été grandement influencé par le langage OBJ3, plus précisément la partie équationnelle de Maude inclus OBJ3 comme sous langage.

Un autre aspect qui favorise l'utilisation du langage Maude dans la spécification des systèmes est la présence d'un ensemble d'outils basés Maude permettant de faciliter grandement la tâche de vérification, parmi ces outils nous pouvons citer :

- Le Model Checker : la dernière version du langage Maude 2.3 rend possible non seulement la vérification des propriétés temporelles linéaire des modèles de systèmes spécifiés en Maude mais en plus il est possible de vérifier des propriétés invariantes à l'aide de la commande search;
- *Prouveur de Théorème* ("Inductive Theorem Prover") (ITP) : cet outil est utilisé pour vérifier interactivement les propriétés des spécifications équationnelles;

- Ø L'analyseur de terminaison (Maude Termination Tool) (MTT) : est un outil qui vérifie la terminaison des spécification équationnelles Maude;
- Ø Le vérificateur Church-Rosser (Church-Rosser Checker) (CRC) : permet de vérifier si une spécification équationnelles à sortes ordonnées respecte ou non la propriété Church-Rosser;
- Ø Le vérificateur de cohérence Maude Coherence Checker (MCC) : pour la vérification de la cohérence des modules systèmes.

Maude est aussi un des formalismes les plus utilisés ayant les caractéristiques suivantes :

1 La simplicité : Il est simple de programmer avec Maude parce que c'est un langage déclaratif, les expressions de base du langage Maude sont très simple et facile à comprendre. Elles sont soit des équations soit des règles de réécriture, et elles ont dans les deux cas un simple sens de réécriture dans lequel une instance dans la partie gauche est remplacée par l'instance correspondante dans la partie droite.

2 L'expressivité : Le langage Maude permet de représenter sans difficulté les calculs déterministes qui amènent à un résultat final unique, et il permet en plus la représentation des calculs concurrents et non déterministes, le premier type est typiquement implémenté à l'aide des équations dans des modules fonctionnels et le deuxième avec des règles de réécritures des modules systèmes.

3 La performance : Maude est un langage compétitif en termes d'exécution avec d'autres langages de la programmation impérative, il peut exécuter des millions de réécriture par seconde, cela permet la vérification de différents chemins d'exécution possible dans une spécification.

Dans la version courante 2.3, le système Maude inclus deux composants principaux CoreMaude et FullMaude.

Le composant CoreMaude implémente un moteur de réécriture, il fournit les constructions de base du langage Maude permettant de spécifier les aspects fonctionnels et concurrents des systèmes et aussi les applications méta niveau et les modules et les théories paramétrées.

Concernant le composant FullMaude est lui-même une application méta niveau écrite en CoreMaude permettant d'étendre CoreMaude avec la notion de modules et de théories orientés objet.

3.1 CoreMaude

Le système Maude est construit autour de l'interpréteur CoreMaude implémenté en C++. Dans CoreMaude les unités de base de spécification sont appelées modules, ils ont deux types : les modules fonctionnels et les modules systèmes.

Les modules fonctionnels implémentent des théories équationnelles et les modules systèmes quant à eux implémentent des théories systèmes.

3.1.1 Concepts de base

Dans ce qui suit nous allons présenter les éléments de base communs aux modules fonctionnels et systèmes qui sont : les sortes, les relations de sous sorte, les opérations, les attributs des opérations, l'importation de modules et enfin les variables.

1 Sorte et sous sorte : La première chose à déclarer dans une spécification est l'ensemble de nouveaux types de données utilisés, dans la communauté de spécification algébrique ces types sont appelés sorte.

Dans Maude une sorte est déclarée de la manière suivante :

sort < Sort > .

où < Sort > est l'identificateur de la sorte, de multiple sortes peuvent être déclarées en suivant la forme :

sorts < Sort-1> … <Sort-K> .

Les sortes dans Maude sont partiellement ordonnée via une relation de sous sorte, cette relation est équivalente à la relation de sous ensemble dans le type de donnée ensemble, alors nous pouvons déclarer une relation de sous sorte en employant le mot clé subsort ou subsorts comme suit :

subsort <Sort-1> < <Sort-2> .

subsorts <Sort-1> … <Sort-j> < … < <Sort-k> … <Sort-l> .

Dans la figure ci-dessous nous présentons un exemple de déclaration de trois sortes : Zero pour représenter la constante zero, Nat pour représenter l'ensemble des nombres naturels et enfin NzNat pour représenter les nombres naturels non nuls et la relation de sous sorte existe ente elles.

```
sort Zero .
sort Nat .
sort NzNat .
subsort Zero < Nat .
subsort NzNat < Nat .
```

Figure 3.7 : Déclaration de sorte et de sous sorte

2 La déclaration des opérations : Décrire une algèbre consiste principalement à déclarer les sortes mais aussi les opérations qui agissent sur ces sortes.

Dans le langage Maude une opération est déclarée à l'aide du mot clé op suivi par le nom de l'opération, suivi par deux points, suivi par la liste des sortes arguments de l'opération (appelé le domaine), suivi par le symbole à suivi par la sorte du résultat de l'opération (appelée le co-arité ou le rang de l'opération) et optionnellement suivi par la déclaration des attributs de l'opération suivi par un espace et un point, donc le schéma général est de la forme :

op <OpName> : <Sort-1> … <Sort-k> à > <Sort> [<attributs de l'opération>] .

Dans la figure suivante nous présentons un exemple de déclaration de quelques opérations en Maude.

```
op zero : -> Zero .
op s_ : Nat -> NzNat .
op sd : Nat Nat -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

Figure 3.8 : Déclaration des opérations

3 Attributs des opérations : Le langage Maude permet la déclaration d'un ensemble d'attributs pour les opérations, ces attributs sont introduits entre crochet après la sorte résultat de l'opération et avant le point final.

Le rôle essentiel de ces attributs est d'offrir des informations : syntaxiques, sémantiques, pragmatiques additionnelles sur les opérations.

Maude support plusieurs types d'attributs nous citons à titre d'exemple :

- Ø ctor : pour exprimer que l'opération est constructeur de la sorte résultat.
- Ø assoc : pour exprimer l'associativité d'une opération.
- Ø comm : pour représenter la commutativité
- Ø idem : pour exprimer l'idempotence.
- Ø id:<term> : pour énoncer que le terme <term> est l'élément neutre de l'opération.

Dans la portion du code Maude suivante nous présentons un exemple d'utilisation de quelques attributs.

```
sorts List Elt .
subsort Elt < List .
op nil : -> List [ctor] .
op __ : List List -> List [ctor assoc comm id :nil] .
```

Figure 3.9 : Attributs des opérations

Dans l'exemple de la figure 3.9 nous avons introduit deux sortes : la sorte List pour représenter les listes et la sorte Elt pour représenter les éléments de ces listes.

Nous avons en plus déclaré l'opération nil avec l'attribut ctor cela signifie que nil est une opération constructeur du type algébrique List, en plus nous avons introduit l'opération " __ " qui permet de réaliser la concaténation des éléments de listes avec les attributs : ctor assoc comm id :nil pour exprimer que l'opération est constructeur, elle est aussi associative et commutative est que le terme nil est l'élément neutre de la concaténation.

4 Variables : Dans Maude une variable représente une valeur indéfinie d'une sorte, les variables sont déclarées avec une syntaxe constituée du mot clé var suivi par l'identificateur de la variable suivi par deux points, suivi par le nom de la sorte et se termine par un point.

À titre d'exemple une variable N de sorte Nat est déclarée comme suit :

```
var N : Nat .
```

Une déclaration de plusieurs variables de même sorte peut se faire aussi en une seule ligne comme suit :

```
vars N M : Nat .
```

Une variable peut être déclarée aussi à la volée, ceci par l'ajout de deux points superposés et la sorte de la variable au nom de la variable comme suit N:Nat.

5 Importation de modules : La plupart des langages de programmation offrent au développeur la possibilité d'importer d'autres modules, cette façon de spécification permet d'inclure toutes les déclarations et définitions des modules importés, par conséquent cela permet de minimiser la redondance dans la spécification mais aussi permet d'offrir une meilleure modularité.

Le langage Maude permet l'inclusion de modules par l'usage de trois clauses : including, extending et protecting.

La syntaxe d'utilisation de ces trois clauses est comme suit :

```
including NomModule ou
extending NomModule ou encore
protecting NomModule
```

Le mode protecting signifie que les éléments déclarés dans le module importés n'accepte pas la modification c'est-à-dire toutes les opérations et toutes les sortes sont utilisées strictement comme elles sont définies initialement.

Utiliser including veut dire qu'on peut changer le sens des éléments déclarés dans le module importé, par exemple si nous voulons redéfinir l'opération d'addition du module INT (un module prédéfini pour les entiers) à l'aide d'une nouvelle équation nous devrons utiliser le mode including, concernant l'inclusion en mode extending elle se situe entre les deux modes précédents, lorsque on utilise le mode extending l'ajout de nouveaux termes (constructeurs et constants) à une sorte est autorisé mais la modification de la signification des opérations à l'aide des équations n'est pas autorisé.

3.1.2 Les modules fonctionnels

Un module fonctionnel définit les types de données et les opérations sur ces types grâce à un ensemble d'équations conditionnelles et/ou non conditionnelles.

Les équations non conditionnelles sont déclarées en utilisant le mot clé eq, suivi par un terme, suivi par le signe d'égalité =, suivi par un autre terme, et optionnellement suivi par une liste d'attributs équationnels, et terminé par un espace et un point, alors le schéma général d'une équation non conditionnelles et comme suit :

eq <Term-1 > = <Term-2 > [<attributs de l'équation>] .

Concernant les équations conditionnelles leur forme est la suivante :

ceq <Term-1 > = <Term-2 > if <EqCondition-1 > /\ ... /\ <EqCondition-k > [<attributs de l'équation >] .

Où EqCondition-i représente la condition d'exécution de l'équation et elle peut avoir trois variantes :

- Ø Des équations ordinaires t = t';
- Ø Des équations d'affectation t:=t';
- Ø Des équations booléennes abrégées de la forme t, où t est un terme de sorte Bool.

Plus précisément un module fonctionnel représente une théorie équationnelle dont ses équations sont utilisées de gauche à droite comme des règles de simplifications pour atteindre la forme canonique.

Un module fonctionnel a donc la forme : fmod (Σ, E) endfm avec (Σ, E) correspond à une théorie équationnelle d'appartenance.

Dans la figure 3.10 nous présentons un exemple d'un module fonctionnel Maude appelé SET introduisant deux sortes : Set pour représenter le type de donnée ensemble, et la sorte Elt pour représenter les éléments des ensembles.

La constructeur empty permet d'obtenir l'ensemble vide et l'opération lenght servira donc à donner la taille d'un ensemble, dans ce module se trouve aussi la déclaration de deux variables E et S et une équation non conditionnelle et autre conditionnelle dont leur rôle est de calculer la cardinalité d'un ensemble.

```
fmod SET is
protecting NAT .
sorts Elt Set .
subsort Elt < Set .
op empty : -> Set [ctor] .
op __ : Set Set -> Set [ctor assoc comm id: empty] .
op lenght : Set -> Nat .
var E : Elt .
vars S : Set .
eq lenght(E S) = lenght(S) + 1 .
ceq lenght(S) = 0 if(S == empty) .
endfm
```

Figure 3.10 : le module fonctionnel SET

3.1.3 Les modules systèmes

Les modules systèmes quant à eux spécifient des théories de réécriture et ils ont donc la forme mod $(\sum, E, R)$ endm où $(\sum, E, R)$ correspond à une théorie de réécriture.

Ce type de modules étend les modules fonctionnels par l'introduction des règles de réécriture conditionnelles et non conditionnelles.

Une règle de réécriture non conditionnelle dans Maude a la syntaxe suivante :

rl [<Label >] : <Term-1 > => <Term-2 > [< attributs de la règle >] .

Et les règles de réécritures conditionnelles sont déclarées comme suit:

crl [<Label >] : <Term-1 > => <Term-2 > if <Condition-1 > /\ ... /\ <Condition-k >
[< attributs de la règle >] .

Où les conditions d'application des règles peuvent contenir en plus des équations et des assertions d'appartenance des expressions de réécriture de la forme t à t'.

Ce type de règle permet de décrire les transitions locales concurrentes des systèmes.

Dans la figure ci-dessous nous présentons un module système appelé VENDING-MACHINE permettant de spécifier le comportement concurrent d'une appareil de vente automatique, la règle buy-c permettant de simuler le vente d'un gâteau qui coûte un dollar, concernant la règle buy-a elle permet de décrire le vente d'une pomme qui coûte un quart de dollar.

```
mod VENDING-MACHINE is
sort Place Marking .
subsort Place < Marking .
op null : -> Marking .
ops $,q,a,c : -> Place .
op _ _ : Marking Marking -> Marking [assoc comm id: null] .
rl[buy-c]: $ => c .
rl[buy-a]: $ => a q .
rl[change]: q q q q => $ .
endm
```

Figure 3.11 : Le module système VENDING-MACHINE

3.1.4 La programmation paramétrée

La version actuelle du langage CoreMaude permet la définition des spécifications structurées. En plus de l'usage des mots clé protecting, including et extending et les opérations de renomage et d'unions de modules, CoreMaude offre aussi une forme de programmation paramétrée très riche.

Dans ce contexte la matière de base de la programmation paramétrée est constituée des théories, des modules paramétrés et des vues.

Comme dans le langage OBJ, une théorie dans CoreMaude définit l'interface d'un module paramétré c'est-à-dire l'ensemble des propriétés requises par le paramètre formel actuel.

Concernant l'instanciation des paramètres formelles d'un module paramétré par un module ou une théorie elle se fait grâce à une vue, alors les vues offrent une interprétation des paramètres formels.

1 Les théories : Les théories sont utilisées pour déclarer l'interface d'un module paramétré, quant aux types de modules, CoreMaude supporte deux différents types de théories : les théories fonctionnelles et les théories systèmes avec la même structure du module correspondant.

Les théories fonctionnelles sont déclarées en utilisant les mots clés fth … endfth, et les théories systèmes sont déclarés avec les mots clés th … endth. Ces deux types de théories

peuvent avoir : des sortes, des relations de sous sorte, des opérations, des variables, des assertions d'appartenance, et des équations et peuvent aussi importer d'autres modules ou théories. Les théories systèmes peuvent avoir des règles de réécriture en plus.

Comme exemple de théorie, nous présentons une théorie fonctionnelle prédéfinie dans CoreMaude appelée TRIV contenant que la déclaration d'une seule sorte Elt :

```
fth TRIV is
sort Elt .
endfth
```

Figure 3.12 : La théorie TRIV

La théorie TRIV est utilisée fréquemment surtout dans la spécification des types de données.

2 Les modules paramétrés : Les modules systèmes et fonctionnels peuvent être paramétrées.

Un module système paramétré par exemple a la syntaxe :

mod M{X1 :: T1 , . . . , Xn :: Tn} is ... endm avec $n \geq 1$

Un module fonctionnel paramétré a une syntaxe complètement analogue.

La patrie {X1 :: T1 , . . . , Xn :: Tn} est appelée l'interface où chaque paire Xi :: Ti correspond à un paramètre, chaque Xi correspond à un identificateur et chaque Ti correspond à une théorie.

Chaque identificateur dans une interface doit être unique, bien que il n y a aucune restriction sur l'unicité d'une théorie dans une interface, nous pouvons avoir par exemple deux théories TRIV déclarées dans la même interface mais avec des identificateurs différents.

Dans CoreMaude les théories paramètres d'un module fonctionnel ne peuvent être que des théories fonctionnelles.

Dans un module paramétré les sortes des théories paramètres doivent être qualifier par le nom de ces dernières, par exemple si on a le paramètre Xi :: Ti chaque sorte S dans Ti doit être écrit comme suit Xi$S, et chaque nouvelle sorte introduite dans le module paramétré doit contenir tous les identificateurs de toutes les théories paramètres, si par exemple l'interface du module paramétré est comme suit : {X1 :: T1 , . . . , Xn :: Tn} alors chaque nouvelle sorte S' introduite dans ce module doit avoir la syntaxe : sort S'{X1, …, Xn} .

Dans la figure 3.14 nous présentons la version paramétrée du module fonctionnel présenté dans la figure 3.10, maintenant la sorte Elt correspond aux éléments des ensembles, et elle est représentée par la sorte Elt de la théorie TRIV, ceci offre une spécification plus structurée et extensible. Nous pouvons alors dès maintenant spécifier à l'aide de simples vues des ensembles de différents types de données.

```
fmod SETP {X :: TRIV} is
protecting NAT .
sorts Set{X} .
subsort X$Elt <  Set{X} .
op empty : -> Set{X} [ctor] .
op __ : Set{X} Set{X} -> Set{X} [ctor assoc comm id: empty] .
op lenght : Set{X} -> Nat .
var E : X$Elt .
vars S : Set{X} .
eq lenght(E S) = lenght(S) + 1 .
ceq lenght(S) = 0 if(S == empty) .
endfm
```

Figure 3.13 : Le module paramétré SETP

3 Les vues : Une vue établit un ensemble de correspondances entre une théorie et un module, une vue dans CoreMaude est déclarée en suivant la syntaxe :

view <nom de la vue > from <théorie> to < théorie ou module >
<Mappings>
endv

où Mappings correspond à l 'ensemble des correspondances.

L'instanciation du module paramétré SETP pour spécifier le type algébrique ensemble de chaîne de caractères peut se faire l'aide de la vue String présenté dans la figure suivante.

```
view String from TRIV to STRING is
sort Elt to String .
endv
```

Figure 3.14: La vue String

3.2 Full Maude

FullMaude est une extension du CoreMaude implémentée en exploitant l'aspect métalangage inhérent au langage CoreMaude, alors tous les concepts présents dans CoreMaude sont aussi présents dans FullMaude mais avec quelques restrictions syntaxiques à respecter, à titre d'exemple toutes les commandes et modules du langage CoreMaude doivent être écrits entre parenthèse pour être utilisés dans FullMaude.

En plus des modules fonctionnels et systèmes FullMaude offre au développeur un troisième type de modules qui sont les modules orientés objet, ce type de module permet de supporter la programmation orienté objet par l'usage d'une notation convenable.

3.2.1 Les modules orientés objet

Les modules orientés objet offrent une syntaxe plus appropriée pour décrire les concepts de base du paradigme objet, comme entre autre l'objet, le message, et la configuration.

Dans un système orienté objet concurrent l'état concurrent usuellement appelé configuration a typiquement la structure d'un multi-ensemble constitué d'objets et de messages qui se développent par des réécritures modulo l'associativité, la commutativité et l'identité, en utilisant des règles qui décrivent l'effet des événements de communication entre les objets et les messages.

Un objet dans une configuration donnée est représenté par un terme :

< O : C | a1 :v1,…, an :vn >

Où O est le nom ou l'identificateur de l'objet, C est l'identificateur de la classe, les ai représentent les identificateurs des attributs de l'objet et les vi représentent leurs valeurs.

Concernant les messages ils n'ont pas une forme syntaxique préfinie, leur syntaxe est définie par l'utilisateur pour chaque type d'application, la seule contrainte est que le premier argument du message doit être l'identificateur de l'objet destination.

Une autre caractéristique importante du langage FullMaude et qu'il permet la paramétrisation des modules orienté objet.

Dans la figure 3.15 nous présentons un exemple d'un module orienté objet, dans ce module nous introduisons une classe appelée Account qui représente un compte bancaire qui a l'attribut bal représentant le solde, nous déclarons aussi dans ce module trois types de messages permettant d'ajouter de retirer et de transférer de l'argent et les différents règles de réécriture permettant de simuler le comportement de ce système.

```
(omod ACCOUNT is
protecting QID .
protecting INT .
subsort Qid < Oid .
class Account | bal : Int .
msgs credit debit : Oid Int -> Msg .
msg from_to_transfer_ : Oid Oid Int -> Msg .
vars A B : Oid .
```

```
var M : Nat .
vars N N' : Int .
rl [credit] : credit(A, M) < A : Account | bal : N > => < A : Account | bal : N + M > .
crl [debit] : debit(A, M) < A : Account | bal : N > => < A : Account | bal : N - M >
 if N >= M .
crl [transfer] : (from A to B transfer M) < A : Account | bal : N > < B : Account | bal : N'
>
=> < A : Account | bal : N - M > < B : Account | bal : N' + M > if N >= M .
endom)
```

Figure 3.15 : Le module orienté objet ACCOUNT

4 Conclusion

Dans ce troisième chapitre nous avons présenté premièrement la logique de réécriture en définissant les éléments de base de ces deux constituants à savoir les théories équationnelles et les théories systèmes, nous avons aussi présenté les règles de raisonnement pour ces deux constituants ainsi que le modèle sémantique associé aux théories de réécriture.

Dans une deuxième partie nous avons donné les différents concepts de base du langage Maude, nous avons cité en détail des notions importantes concernant la syntaxe de déclaration des variables, des opérations, des équations, des règles de réécriture.

Nous avons en plus présenté brièvement l'ensemble d'outils construit autour de ce langage favorisant son utilisation dans la spécification et la vérification des systèmes.

Dans le chapitre suivant nous allons procéder à l'utilisation de la logique de réécriture via son langage Maude pour fournir au langage SADL un cadre sémantique formel.

Chapitre IV

Un Cadre Sémantique Formel pour le Langage SADL

1 Introduction

De nos jours les modèles formels proposés pour les systèmes informatiques sont trop complexes pour assurer un suivi, une compréhension, et une analyse correcte. De plus, les parties de tels systèmes peuvent être réutilisées au sein d'un autre système similaire, ou lors du remplacement d'un sous-système par un autre. Les langages de description d'architectures [17] (Architectural Description Languages, ADL) ont ouvert la voie à nombre d'applications concernant le développement et le maintien des systèmes logiciels complexes.

Ces langages servent de support à l'architecte logiciel en répondant à un certain nombre de fonctionnalités, celles que nous retenons dans le contexte de ce chapitre sont:

- Ø Fournir une sémantique bien définie au plan architectural et ne pas se limiter à une description à base «Box-and-Line»;
- Ø Permettre l'analyse de l'architecture, soit par l'utilisation de critères liés aux respects de contraintes de style, soit par l'utilisation de techniques formelles (exemple : vérification de l'absence d'inter blocages dans une architecture);
- Ø Aider à l'implantation du système, par exemple en permettant la construction automatique du code des interactions entre composants du système.

Certains ADL formels existants tels que Wright [29], Rapide [30] et Darwin [34], permettent la spécification formelle et la vérification des descriptions architecturales. En se basant

principalement sur des algèbres de processus comme le π-calcul, CSP ou FSP, ces ADL supportent l'expression et l'analyse du comportement des composants, des connecteurs et des topologies (configurations) d'architectures. En contrepartie ces ADL sont plus complexes à utiliser, ils font recours à plusieurs formalismes, et sont parfois jugés insuffisants pour décrire formellement le bon fonctionnement d'architectures de composants.

L'expérience a montré que les formalismes sémantiques utilisés actuellement présentent des limites concernant la formulation de certains concepts inhérents aux ADLs tels que la synchronisation et la connexion dynamique entre les composants architecturaux.

L'objectif du présent chapitre est de proposer un seul formalisme sémantique, *la logique de réécriture*, pour décrire formellement les éléments architecturaux d'un logiciel: composant, connecteur et configuration, et analyser ensuite leurs comportements selon des propriétés fonctionnelles (par exemple, leur non blocage)

Comme nous avons vu dans le deuxième chapitre le langage SADL est un langage de description d'architectures dédié à la description structurale des hiérarchies d'architectures à différents niveaux d'abstractions grâce à un mécanisme de raffinement explicite, cependant SADL n'est accompagné que d'un outil de vérification syntaxique [23], ce qui rend impossible la vérification des propriétés architecturales de logiciels que ce soit fonctionnelles ou non fonctionnelles.

Notre objectif dans ce chapitre est de définir un cadre sémantique formel à base de la logique de réécriture permettant de décrire tous les concepts clés d'une architecture SADL (composant, connecteur, configuration, raffinement, etc.) à travers lequel nous pouvons analyser une telle architecture.

Le présent chapitre est organisé comme suit : La section 2 est dédiée, tout d'abord à la présentation de l'approche générique de transformation d'une architecture SADL vers une théorie de réécriture équationnelle, l'extension de cette approche pour décrire le mécanisme de raffinement des architectures SADL fera l'objet de la section 4. Dans la section 5, nous montrons comment analyser formellement les propriétés fonctionnelles des architectures SADL en utilisant le model-checker de l'environnement Maude. Enfin, une conclusion termine ce chapitre.

2 Modèle sémantique d'une architecture SADL

Notre but dans ce chapitre consiste à définir un cadre sémantique formel, basé logique de réécriture, pour décrire tous les concepts de base du langage SADL (composant, connecteur, configuration, raffinement, etc.) permettant l'analyse des architectures SADL à un niveau

purement statique. La logique de réécriture à travers son langage Maude offre des mécanismes sémantiques adéquats pour vérifier le respect des contraintes de liaison entre les ports et les rôles d'une architecture (ou des styles architecturaux). En outre, le mécanisme de raffinement du langage SADL peut être facilement intégré dans le formalisme considéré, vue que le comportement des architectures est absent dans SADL, la logique de réécriture permet aussi d'encourager l'extension formelle de ce langage avec ce concept pour permettre une vérification rigoureuse des propriétés comportementales des architectures logicielles.

2.1 La formalisation des éléments architecturaux

Le modèle théorique que nous associons à une architecture SADL représente une théorie équationnelle de la logique équationnelle d'appartenance " membership equational logic " qui est une sous logique de la logique de réécriture.

Ce modèle est alors noté : $(\sum, E \cup A)$, où $\sum$ est la signature de notre modèle, c'est-à-dire la spécification de l'ensemble de sortes, de sous sortes et de l'ensemble des opérations utiles pour décrire statiquement une architecture, E représente l'ensemble des équations de notre modèle, et enfin A représente l'ensemble des attributs équationnels des opérations [01].

En effet, nous avons adopté une approche générique qui consiste à spécifier séparément chaque élément architectural SADL dans un module fonctionnel Maude, par conséquent nous avons cinq modules Maude génériques présentés dans la figure 4.1.

```
fmod Port is
sort DataType .
sorts IPortName OPortName .
sorts IPort OPort .
sorts SetIPort SetOPort .
subsort IPort < SetIPort .
subsort OPort < SetOPort .
op none : -> IPort [ctor] .
op none : -> OPort [ctor] .
op _:_ : IPortName DataType -> IPort [ctor prec 21] .
op _:_ : OPortName DataType -> OPort [ctor prec 21] .
op _;_ : SetIPort SetIPort -> SetIPort [ctor assoc id: none comm prec 22] .
op _;_ : SetOPort SetOPort -> SetOPort [ctor assoc id: none comm prec 22] .
endfm
```

***

```
fmod Component is
extending Port .
sort ComponentName .
sort ComponentType .
sort Component .
sort SetComponent .
subsort Component < SetComponent .
op none : -> Component [ctor] .
op _:_`[_->_`]  : ComponentName ComponentType SetIPort SetOPort -> Component [ctor prec 23] .
op __ : SetComponent SetComponent -> SetComponent [ctor assoc id: none comm prec 24] .
endfm
```

***

```
fmod Connector is
extending Port .
sort ConnectorName .
sort ConnectorType .
sort Connector .
sort SetConnector .
subsort Connector < SetConnector .
op _:_<_> : ConnectorName ConnectorType DataType -> Connector [ctor prec 23] .
op none : -> Connector [ctor] .
op __ : SetConnector SetConnector -> SetConnector [ctor assoc id: none comm prec 24] .
endfm
```

***

```
fmod Configuration is
extending Component .
extending Connector .
sorts ConnectionName ConstraintName .
sorts Connection Constraint .
sorts ConstraintRelation ConnectionRelation .
sort SetCon .
subsorts Connection Constraint < SetCon .
```

```
op _:CONNECTION_`(_`,_`,_`) : ConnectionName ConnectionRelation ConnectorName
OPortName IPortName -> Connection  [ctor prec 23] .
op _:CONSTRAINT_`(_`,_`) : ConstraintName ConstraintRelation ComponentName
ComponentName -> Constraint [ctor prec 23] .
op none : -> SetCon [ctor] .
op __ : SetCon SetCon -> SetCon [assoc id: none  comm prec 24] .
endfm
                                        ***
fmod Architecture is
extending Configuration .
sort Head .
sort Architecture .
subsort Architecture < Component .
op `[_->_`] : SetIPort SetOPort -> Head [ctor] .
op ARCHITECTURE_COMPONENTS_CONNECTORS_CONFIGURATION_ : Head
SetComponent SetConnector SetCon -> Architecture [ctor prec 25] .
endfm
```

Figure 4.1 : Les modules Maude formalisant les éléments architecturaux SADL

Dans la figure ci-dessus :

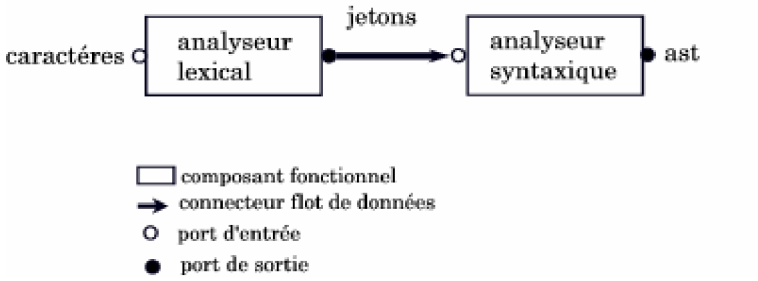
- Ø Le module fonctionnel Maude appelé Port permet la spécification de la notion de port d'une architecture SADL en spécifiant les ports d'entrée et de sortie;
- Ø Le module Maude Component permet de spécifier la structure d'un composant SADL;
- Ø Le module Connector permettant de modéliser l'interaction entre les composants;
- Ø Le module Maude Configuration importe les deux modules Component et Connector en mode " Extending " pour décrire la relation de deux ports de deux composants par un connecteur de type compatible;
- Ø Le module appelé Architecture permet de représenter une architecture SADL grâce à l'opération : " ARCHITECTURE_COMPONENTS_CONNECTORS_CONFIGURATION_ " permettant de déclarer une telle architecture.

La figure 4.3 présente la transformation de l'architecture SADL présentée dans la figure 4.2a qui est une partie de celle décrite dans la figure 3.5 vers un module fonctionnel Maude Parser. En effet, cette transformation se fait par la déclaration d'un module qui importe le

module générique Architecture au moyen de la clause "extending", ainsi que la spécification des opérations constructeurs pour identifier dans ce cas, les noms des ports (char-iport, token-iport, token-oport, base-ast-oport), les noms des composants (lexical-analyzer, parser), le nom du connecteur (token-channel), le nom de la connexion (token-flow), et enfin le nom de notre architecture (parser-L1). En effet, cette approche de transcription d'une architecture SADL en Maude est assez générale, les contraintes de style concernant cet exemple sont données sous forme d'opérations et une seule équation Maude.

```
parser_L1: ARCHITECTURE
                    [char_iport: SEQ(character) –> base_ast_oport: ast]
   IMPORTING character, token, ast FROM compiler_types
   IMPORTING Function FROM Functional_Style
   IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
BEGIN
COMPONENTS
   lexical_analyzer: Function
       [char_iport: SEQ(character) –> token_oport: SEQ(token)]
   parser: Function
       [token_iport: SEQ(token) –> base_ast_oport: ast]
CONNECTORS
   token_channel:    Dataflow_Channel<SEQ(token)>
CONFIGURATION
   token_flow: CONNECTION
     = Connects(token_channel, token_oport, token_iport)
END compiler_L1
```

a : L'architecture SADL



b: le diagramme box_line

Figure 4.2 : L'architecture SADL parser-L1

```
fmod Parser is
extending Architecture .
ops SEQ-character SEQ-token ast : -> DataType [ctor] .
ops char-iport  token-iport  : -> IPortName [ctor] .
ops token-oport base-ast-oport : -> OPortName [ctor] .
ops lexical-analyzer parser : -> ComponentName [ctor] .
op Function : -> ComponentType [ctor] .
op token-channel : -> ConnectorName [ctor] .
op Dataflow-Channel : -> ConnectorType [ctor] .
op token-flow : -> ConnectionName [ctor] .
op Connects : -> ConnectionRelation [ctor] .
op parser-L1 : -> Architecture .
eq parser-L1 = ARCHITECTURE
          [ char-iport : SEQ-character -> base-ast-oport : ast ]
COMPONENTS
     lexical-analyzer : Function
        [ char-iport : SEQ-character -> token-oport : SEQ-token ]
    parser : Function
        [ token-iport : SEQ-token -> base-ast-oport : ast ]
CONNECTORS
     token-channel : Dataflow-Channel < SEQ-token >
CONFIGURATION
    token-flow :CONNECTION Connects ( token-channel , token-oport , token-iport ) .
endfm
```

Figure 4.3 : L'architecture compiler-L1 en Maude

À travers les différents modules présentés dans cette section, nous constatons que nous avons réalisé une spécification modulaire et lisible d'une architecture SADL cela est due à la flexibilité de la logique de réécriture, de même cette spécification peut être enrichie facilement, nous pouvons en particulier rajouter d'autres éléments pour spécifier par exemple le comportement des composants architecturaux du langage SADL.

2.2 La formalisation de la relation du raffinement

SADL fournit un support pour des mises en correspondance (mapping) explicites entre architectures, une mise en correspondance dans SADL est spécifiée par un ensemble de paires appelées associations de la forme :

élément-architectural1 --> élément-architectural2

Dans cette partie nous présentons une approche pour la formalisation de la relation de raffinement du langage SADL. La figure 4.4 présente un module système appelé REFINER permettant de réaliser cette relation. L'opération la plus importante de ce module est Refine, qui a comme paramètres un terme de sorte Architecture et un autre terme de sorte Mapping et elle a comme but la génération à partir d'une architecture abstraite l'architecture raffinée grâce aux associations données. Cela se fait grâce à un mécanisme de propagation de modifications à l'intérieur des éléments architecturaux en utilisant trois règles de réécriture conditionnelles : refine-component, refine-connector et refine-connexion[01].

Plus précisément le processus de raffinement se déroule comme suit :

Etant donnée une architecture Arch et un mapping Map, les associations de Map seront analysées une par une et suivant leurs types on applique soit la règle de réécriture refine-component si l'association est de sorte MapComp ou la règle de réécriture refine-connector si l'association est de sorte MapConnec, ou encore la règle de réécriture refine-connexion si l'association est de sorte MapConx.

La démarche de raffinement adoptée par les trois règles de réécriture refine-component, refine-connector et refine-connexion est la même, elle consiste à tester si l'élément à raffiné est égale à la partie gauche de l'association en question, si oui alors il serait remplacé par la partie droite de l'association.

Concernant la règle de réécriture non conditionnelle no-refinement elle permet de donner une représentation lisible d'une architecture raffinée.

```
mod Refiner is
protecting Compiler .
sorts MapComp MapConnec MapConx Mapping .
subsorts MapComp MapConnec MapConx < Mapping .
op nil : -> Mapping [ctor] .
op _-->_ : Component Component -> MapComp [ prec 26] .
op _-->_ : Connector Connector -> MapConnec [ prec 26] .
op _-->_ : Connection Connection -> MapConx [ prec 26] .
op __ : Mapping Mapping -> Mapping [assoc prec 27 id: nil] .
op Refine : Architecture Mapping -> Architecture .
var map : Mapping .
var arch : Architecture .
vars head : Head .
```

```
vars setcom : SetComponent .
vars setcon : SetConnector .
vars setcx : SetCon .
vars comp1 comp2 comp3: Component .
vars conct1 conct2 conct3 : Connector .
vars cx1 cx2 cx3 : Connection .
crl [refine-component] : Refine(ARCHITECTURE head COMPONENTS comp1 setcom CONNECTORS setcon CONFIGURATION setcx, comp2 --> comp3 map) => Refine(ARCHITECTURE head COMPONENTS comp3 setcom CONNECTORS setcon CONFIGURATION setcx, map) if comp1 == comp2 .
crl [refine-connector] : Refine(ARCHITECTURE head COMPONENTS setcom CONNECTORS conct1 setcon CONFIGURATION setcx, conct2 --> conct3 map) => Refine(ARCHITECTURE head COMPONENTS setcom CONNECTORS conct3 setcon CONFIGURATION setcx, map) if conct1 == conct2 .
crl [refine-connexion] : Refine(ARCHITECTURE head COMPONENTS setcom CONNECTORS setcon CONFIGURATION cx1 setcx, cx2 --> cx3 map) => Refine(ARCHITECTURE head COMPONENTS setcom CONNECTORS setcon CONFIGURATION cx3 setcx, map) if cx1 == cx2 .
rl [no-refinement] : Refine(arch, nil) => arch .
endm
```

Figure 4.4 : Le module système Refiner

La figure 4.5 montre un exemple d'utilisation de l'opération Refine pour raffiner le connecteur token-channel et la connexion token-flow de l'architecture parser-L1 décrite dans la figure 4.2 et le résultat obtenu après exécution.

```
                     \|||||||||||||||||||/
                   --- Welcome to Maude ---
                     /|||||||||||||||||||\
            Maude 2.3 built: Feb 21 2007 14:55:47
          Copyright 1997-2007 SRI International
                 Fri May 25 21:10:21 2007
Maude> rew Refine ( parser-L1 ,
token-channel : Dataflow-Channel < SEQ-token > -->
pipe-channel : Pipe < Finite-Stream-token >
token-flow :CONNECTION Connects ( token-channel , token-oport , token-iport ) -->
token-pipe :CONNECTION Connects ( pipe-channel , token-oport , token-iport ) ) .
```

```
rewrite in Refiner : Refine(compiler-L1, token-channel : Dataflow-Channel < SEQ-token
> --> pipe-channel : Pipe < Finite-Stream-token > token-flow :CONNECTION
Connects(token-channel,token-oport,token-iport) --> token-pipe :CONNECTION
Connects(pipe-channel,token-oport,token-iport)) .
rewrites: 6 in 1625750371000ms cpu (0ms real) (0 rewrites/second)
result Architecture:
ARCHITECTURE  [char-iport : SEQ-character -> base-ast-oport : ast]
COMPONENTS
    lexical-analyzer : Function
        [char-iport : SEQ-character -> token-oport : SEQ token]
    parser : Function
        [token-iport : SEQ-token -> base-ast-oport : ast]
 CONNECTORS
    pipe-channel : Pipe < Finite-Stream-token >
 CONFIGURATION
   token-pipe :CONNECTION Connects(pipe-channel,token-oport,token-iport)
Maude>
```

Figure 4.5 : Exemple d'utilisation de l'opération Refine sous Maude

Dans cette portion du code Maude nous avons montré comment profiter du mécanisme de réécriture inhérent aux modules système Maude pour réaliser le raffinement d'architectures, plus précisément nous avons montré comment utiliser l'opération Refine présentée précédemment pour raffiner le connecteur token-channel et la connexion token-flow de l'architecture parser-L1.

2.3 Analyse d'une architecture SADL

Pour montrer l'expressivité de la logique de réécriture, nous avons jusqu'ici considéré une partie de l'exemple connu de l'architecture statique d'un compilateur. Tous les concepts syntaxiques du langage ont été naturellement intégrés dans le formalisme proposé. Les vérifications syntaxiques nécessitant des outils spécifiques, peuvent facilement être déduites dans le cadre de cette formalisation. En effet, on peut facilement exprimer et vérifier que les arguments de la clause Connects ont le même type de données transportées.

De plus, dans la formalisation du processus raffinement, on peut détecter, dans le prototypage de la spécification sous Maude, que les remplacements des objets architecturaux exprimés

dans les règles de correspondance sont valides ou non. Nous comptons dans un futur proche exploiter théoriquement et au mieux le raffinement d'architecture SADL.

L'extension de SADL à l'expression du comportement en fait un langage d'analyse. Pour l'heure, aucune suggestion complète et efficace orientée dans ce sens n'a été développée pour être intégrée. Le but de cette section est de proposer une extension du modèle sémantique associé au langage SADL pour supporter l'expression et l'analyse du comportement de composants architecturaux. Ceci permettra d'engendrer un processus de développement graduel et validé des architectures logicielles, en plus l'utilisation de la logique de réécriture via son langage Maude, offrira une spécification exécutable et analysable en profitant des outils construits au tour de cet environnement comme par exemple l'utilisation du model-checker pour la vérification des propriétés temporelles linéaires.

Nous explicitons notre démarche à travers un exemple générique très simple qui spécifie deux processus (procA et procB) en exclusion mutuelle, la variable partagée turn qui a initialement la valeur 0 permet de garder la trace du processus qui va entrer dans la section critique et examine ou met à jour la valeur de cette variable.

L'architecture SADL du système est présentée dans la figure 4.6.

```
Mutex: ARCHITECTURE [ –> ]
IMPORTING nat FROM compiler_types
IMPORTING Variable Read\Write FROM Shared_Memory_style
IMPORTING Process FROM Process_style
BEGIN
COMPONENTS
   procA: Process [ –> ]
   procB: Process [ –> ]
   turn: Variable(nat)[ -> ]
CONFIGURATION
   rwA-var: CONSTRAINT Read\Write (procA, turn)
   rwB-var: CONSTRAINT Read\Write (procB, turn)
END Mutex
```

Figure 4.6 : Le système architectural Mutex

Les modules Maude décrivant l'aspect structure et comportement du système Mutex sont donnés au niveau de la figure 4.7. L'aspect structure du système Mutex est défini telle que l'on a détaillée dans la section précédente (voir figure 4.7a). Nous remarquons l'utilisation d'une version enrichie du module Component avec la sorte ComponentState qui permet de

définir l'état d'un composant. L'opération qui génère un composant SADL se présente maintenant ainsi:

```
op _:_`[_->_`]`(_`) : ComponentName ComponentType SetIPort SetOPort
ComponentState -> Component [ctor prec 23] .
```

Par conséquent, nous avons déclaré dans le module Mutex l'états que peut prendre un processus (wait ou critical) ainsi que l'états ou la valeur du variable partagée turn (0 ou 1) à l'aide des opérations constructeurs.

Concernant la lecture ou la mise à jour de la variable partagée turn par les deux processus procA et procB elle sera définie à l'aide des deux contraintes rwA-var et rwB-var.

Le comportement de l'architecture SADL Mutex sera défini principalement par les quatre règles de réécritures définies dans le module système Behav-Mutex en Maude (voir figure 4.7b).

Le comportement de l'architecture Mutex peut être alors décrit comme suit :

- Ø Au départ le processus procA examine la valeur de turn et si la trouve égale à 0 il entre à sa section critique (la règle de réécriture A-enter);
- Ø Lorsque le processus procA décide de quitter sa section critique il modifie la valeur de turn à 1 (la règle de réécriture A-exit);
- Ø Lorsque le processus procB examine la valeur de turn et si la trouve égale à 1 il entre à sa section critique (la règle B-enter);
- Ø Et enfin, Lorsque le processus procB désire quitter sa section critique il met la valeur de turn à 0 (la règle de réécriture B-exit).

```
fmod Mutex is
extending Architecture .
ops wait critical 0 1 : -> ComponentState [ctor] .
ops procA procB turn : -> ComponentName [ctor] .
ops Process Variable : -> ComponentType [ctor] .
op Read\write : -> ConstraintRelation [ctor] .
ops rwA-var rwB-var : -> ConstraintName [ctor] .
op Mutex : -> Architecture .
eq Mutex = ARCHITECTURE [ none -> none ]
  COMPONENTS
        procA : Process [ none -> none ](wait)
        procB : Process [ none -> none ](wait)
```

```
        turn : Variable [ none -> none ](0)
  CONNECTORS
        none
  CONFIGURATION
        rwA-var :CONSTRAINT Read\write ( procA , turn )
        rwB-var :CONSTRAINT Read\write ( procB , turn ) .
endfm
```

a: Le module Maude Mutex

```
mod Behav-Mutex is
protecting Mutex .
var setc : SetCon .
var st : ComponentState .
rl [A-enter] : ARCHITECTURE [ none -> none ] COMPONENTS
  procA : Process [ none -> none ](wait)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc
  =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](critical)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc .
rl [A-exit] : ARCHITECTURE [ none -> none ] COMPONENTS
  procA : Process [ none -> none ](critical)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc
 =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](wait)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](1)
  CONNECTORS none CONFIGURATION setc .
rl [B-enter] : ARCHITECTURE [ none -> none ] COMPONENTS
```

```
  procA : Process [ none -> none ](st)
  procB : Process [ none -> none ](wait)
  turn : Variable [ none -> none ](1)
  CONNECTORS none CONFIGURATION setc
 =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](st)
  procB : Process [ none -> none ](critical)
  turn : Variable [ none -> none ](1)
  CONNECTORS none CONFIGURATION setc .
rl [B-exit] : ARCHITECTURE [ none -> none ] COMPONENTS
  procA : Process [ none -> none ](st)
  procB : Process [ none -> none ](critical)
  turn : Variable [ none -> none ](1)
  CONNECTORS none CONFIGURATION setc
 =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](st)
  procB : Process [ none -> none ](wait)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc .
Endm
```

b: Le module Maude spécifiant le comportement de **Mutex**

Figure 4.7 : Expression du comportement dans un composant architectural

En plus, ces modules ont été testés syntaxiquement et analysé formellement avec le mdel-checker du système Maude, particulièrement, nous avons concentré sur la vérification de la propriété d'exclusion mutuelle des deux processus procA et procB.

```
Maude> red modelCheck(init, [] ~(crit(procA) /\ crit(procB))) .
reduce in Behaviour-CHECK :
modelCheck(init, []~ (crit(procA) /\ crit(procB)))   .
rewrites: 23 in 7714480714ms cpu (3ms real) (0 rewrites/second)
result Bool: true
```

Figure 4.8 : la vérification de la propriétés d'exclusion mutuelle

3 Intérêt du modèle proposé

A travers de ce chapitre nous avons présenté un cadre sémantique exécutable basé sur la logique de réécriture pour le langage de description d'architecture SADL. Bien que ce cadre est défini initialement pour le langage SADL, il peut être aussi adapté pour supporter d'autres ADLs.

Le but d'un tel formalisme est d'offrir au langage SADL une sémantique bien définie permettant d'engendrer un processus de développement validé des architectures logicielles. En plus, l'utilisation de la logique de réécriture via son langage Maude nous offre une spécification exécutable et nous permet aussi de profiter de l'ensemble des outils construits au tour de cet environnement comme par exemple l'utilisation du model-checker pour la vérification des propriétés temporelles linéaires.

Ainsi, à travers les différents modules présentés précédemment nous constatons que nous avons réalisé une spécification modulaire et lisible d'une architecture SADL cela est due à la flexibilité de la logique de réécriture, de même cette spécification peut être enrichie facilement et nous pouvons par exemple rajouter d'autres éléments pour spécifier de nouveaux concepts architecturaux.

Dans une autre partie de ce chapitre nous avons étendu notre cadre par un module système Maude permettant de réaliser le processus raffinement des architectures.

Enfin, il est important de noter que le cadre réalisé constitue aussi un outil pour la vérification syntaxique des architectures SADL, concernant la vérification sémantique elle peut être implémentée en utilisant par exemple la programmation méta-niveau du langage Maude.

4 Conclusion

Depuis le début des années 90, la communauté de l'ingénierie de logiciels a développé plusieurs langages de description d'architectures, le but principal de ces langages est de faciliter la tâche de conception de logiciels. Dans ce contexte la présence d'un cadre formel pour un ADL permet d'offrir un développement validé d'une architecture logicielle.

Dans une première partie de ce chapitre nous avons présenté un cadre sémantique formel basé logique de réécriture approprié pour le langage de description d'architecture SADL, en fait nous avons présenté d'une manière détaillée comment chaque élément d'une architecture SADL sera transformé vers un terme de la logique de réécriture tout en préservant sa syntaxe initiale, et cela grâce à l'expressivité de la logique de réécriture.

Puis nous avons présenté une démarche de formalisation du processus de raffinement offert par le langage SADL, plus précisément nous avons indiqué comment la puissance de la logique de réécriture pourra être utilisée pour la propagation du raffinement, en plus nous avons enrichi le modèle proposé pour permettre la modélisation du comportement des composants d'une architecture SADL.

En effet l'intérêt d'une telle formalisation est d'offrir la possibilité de vérification des propriétés architecturales.

Dans le chapitre suivant nous présentons l'extension de ce formalisme pour permettre la spécification et la vérification des contraintes non fonctionnelle.

Chapitre V

Intégration des Propriétés non Fonctionnelles dans le Langage SADL

1 Introduction

Face à la croissance de la taille et de la complexité des systèmes logiciels, il est devenu de plus en plus nécessaire l'adoption de la description architecturale dans leur processus de développement. La description architecturale a joué toujours un rôle important dans le développement de logiciels complexes et maintenables à moindre coût et avec respect des délais. Les ADLs, ou encore langages de description d'architectures sont des notations qui servent à définir et à exprimer la structure haut niveau de l'architecture du logiciel en termes d'une collection de composants et de connecteurs et de leur structure d'interconnexion globale [17]**.** Ils fournissent un modèle abstrait indépendant des détails d'implémentation permettant au développeur de raisonner correctement sur les propriétés du logiciel.

Les travaux menés jusqu'à présent sur le développement centré architecture de logiciels portent essentiellement sur la spécification formelle des architectures. La vérification et l'analyse des architectures logicielles restent un point ouvert de recherche. En effet, il est indispensable d'associer à une architecture logicielle des spécifications précises tant fonctionnelles que non fonctionnelles. Ces spécifications décrivent des services offerts par les différents composants de cette architecture, mais aussi ceux dont ils ont besoin pour fonctionner. L'environnement possède des propriétés qui ont une influence sur le fonctionnement du logiciel. Il peut s'agir de propriétés temporelles, des propriétés sur le mode

de panne ou sur les attaques possibles en termes de sécurité, etc. Ces spécifications sont rarement complètes et formelles.

En général, le terme contrat est de plus en plus utilisé pour décrire les propriétés d'un composant architectural. Les propriétés non fonctionnelles, devant être exprimées à part, permettent de caractériser le degré de satisfaction d'un logiciel, c'est-à-dire comment ses fonctionnalités sont réalisées. Elles sont associées à différents concepts comme par exemple la performance, la qualité de service, la sécurité, la robustesse, la portabilité, etc.

De par leur complexité et leur haut niveau d'abstraction, elles sont rarement prises en compte dans le processus de développement d'un logiciel, particulièrement sa conception architecturale. En effet, le concepteur doit pouvoir considérer ce type de propriétés dès le début du processus, et les estimer avant la phase d'implémentation du système.

En fait, les ADLs qui sont censés décrire de façon formelle l'architecture d'un système à un niveau d'abstraction élevé peuvent être dotés de mécanismes pour l'expression et la vérification de ce type de propriétés. Dans ce contexte, nous suggérons de fournir un support formel permettant l'expression et la vérification de ces propriétés au niveau des architectures logicielles décrites dans le langage SADL, tout en exploitant l'expressivité et la puissance de la logique de réécriture via son langage Maude.

Le langage SADL est un des ADLs utilisés dans la modélisation des architectures logicielles, il offre une notation textuelle précise permettant de décrire les architectures logicielles en faisant une séparation claire entre les différents éléments architecturaux : composants, connecteurs, et configuration. Cependant, comme la plupart des ADLs, ce langage souffre de limites concernant la prise en compte de l'aspect non fonctionnel.

L'objectif de ce chapitre et d'élargir le modèle formel, basé logique de réécriture, du langage SADL présenté dans le chapitre précédent pour permettre la spécification des propriétés non fonctionnelles des composants et des connecteurs de SADL ainsi que l'analyse d'une application architecturale garantissant ces propriétés.

Dans la suite de ce chapitre, la section 2 présente les travaux antérieurs relatifs à notre problématique. Dans la section 3, nous détaillons notre approche d'intégration des propriétés non fonctionnelles dans le langage SADL. Nous montrons comment étendre le modèle des architectures SADL présenté dans le chapitre précédent pour permettre d'une part, la spécification des propriétés non fonctionnelles des composants et des connecteurs du langage SADL, et d'autre part, l'analyse d'une application architecturale garantissant ces propriétés. Finalement, la conclusion propose une synthèse du travail réalisé.

2 Travaux antérieurs

La définition d'une architecture logicielle est une étape importante dans la conception d'un logiciel. Elle permet d'avoir un niveau d'abstraction élevé des applications conçues, en se détachant des détails techniques propres à l'environnement et respectant les contraintes des futurs utilisateurs. Pour répondre à ce besoin de description d'architectures, différents langages de description d'architecture ont été développés, chacun offre des capacités complémentaires pour le développement et l'analyse architecturale [27], à titre d'exemple Aesop [32] supporte l'utilisation des styles architecturaux, METAH [24] offre la possibilité de conception des systèmes de contrôle avionique temps réel, C2 [33] supporte la conception des interfaces graphiques, Wright [29] supporte la spécification et l'analyse comportementale des éléments architecturaux, Rapide [30] permet la vérification et la validation des architectures logicielles par simulation, SADL [26] offre un mécanisme explicite de raffinement d'architectures.

Cependant, dans ces ADLs, la prise en compte des spécifications non fonctionnelles est incomplète ou mal traitée. Il existe un manque notable de supports et de techniques permettant l'expression, l'évaluation, et la prédiction de ce type de propriétés dans les ADLs existants.

Les ADLs ACME, Aesop, et Weaves permettent la spécification des propriétés et/ou des annotations arbitraires sur les éléments architecturaux, mais malheureusement ils n'offrent aucun outil permettant l'interprétation de ces propriétés [17].

D'un autre côté, UniCon et Rapide offrent un support partiel de modélisation des propriétés non fonctionnelles. UniCon permet la spécification des propriétés liées à certains aspects comme le ″scheduling″ de l'application [13], tandis que Rapide offre la possibilité de la modélisation des informations temporelles dans son langage de contraintes [17].

D'autre part, les auteurs de [22] ont présenté une approche qui porte sur l'intégration de l'information non fonctionnelle dans l'architecture logicielle des systèmes, leur objectif est de mesurer et vérifier les propriétés non fonctionnelles du produit logiciel final. L'approche adoptée utilise une notation ad hoc, indépendante de tout ADL, pour décrire les composants et les connecteurs, chacun avec une partie spécification et une partie implémentation. La modélisation de l'aspect non fonctionnel est réalisée via un autre langage appelé *NoFun*, basé sur les trois concepts clés suivants: *Non-functional attribute (NF-attribute)* représentant n'importe quel attribut du logiciel permettant de décrire et/ou évaluer ce dernier. Les attributs les plus connus dans ce cas, sont l'efficacité en temps d'exécution et espace mémoire, la

réutilisation, la fiabilité; *Non-functional behavior (NF-behaviour)* correspond à toute valeur assignée à un attribut non fonctionnel lié à un élément architectural particulier; *Non-functional requirements (NF-requirement)* permettant de spécifier les valeurs permises des attributs non fonctionnels.

Par ailleurs, un langage appelé $Process^{NFL}$ dédié à l'expression des besoins non fonctionnels de logiciels a été proposé dans [14]. Ce langage porte plutôt sur la modélisation des aspects de conflit et de corrélation entre les propriétés non fonctionnelles. Un exemple de deux propriétés non fonctionnelles qui peuvent être en conflit sont la sécurité et la performance, un haut niveau de sécurité peut conduire à une perte de performance et vice versa. Dans ce langage trois abstractions sont aussi utilisées pour modéliser une propriété non fonctionnelle: *NF-Attribute, NF-Property* et *NF-Action*. *NF-Attribute* modélise les propriétés non fonctionnelles qui peuvent être simples ou dérivées à partir d'autres *NF-Attribute*. *NF-Action* modélise les aspects logiciels (algorithmes, structures de données) ou les mécanismes matériels (les ressources disponibles) qui ont un effet sur les *NF-Attributes*. L'abstraction *NF-Property* permet d'exprimer des contraintes sur les *NF-Attribute*.

Dans sa thèse Aagedal [15] a défini aussi un langage de modélisation de la qualité de service appelé (CQML), ce langage permet la spécification des offres et des exigences en terme de qualité de service pour les systèmes à base de composants. Une caractéristique importante de ce langage est ses notations sont purement syntaxiques.

Une autre approche plus prometteuse visant l'intégration d'une combinaison de notations CSP, de phrases structurées et de notations WRIGHT pour spécifier des propriétés non fonctionnelles, dans l'ADL WRIGHT a été proposé dans [04]. L'idée de base derrière cette approche est que la satisfaction des besoins non fonctionnels d'une unité architecturale dépend de la satisfaction d'un ensemble de besoins fonctionnels liés à d'autres unités.

L'approche que nous proposons s'inspire de cette dernière. Elle fournit un cadre formel unique pour la spécification des propriétés non fonctionnelles d'une application architecturale en SADL ainsi que leur vérification en simulant le comportement ce cette architecture. L'apport principal de cette contribution se situe au niveau de la satisfaction d'une propriété non fonctionnelle qui nécessite au préalable la satisfaction d'un ensemble de propriétés non fonctionnelle ou/et besoins fonctionnels de la part d'autres unités architecturales.

3 Extension du modèle sémantique avec l'aspect non fonctionnel

Les systèmes informatiques sont essentiellement construits pour satisfaire certains besoins fonctionnels. Pourtant, ces systèmes ont aussi des besoins additionnels qui caractérisent leur aptitude d'exister et de s'adapter aux variations de l'environnement dans lequel ils sont situés. Ces besoins sont dans la plupart du temps qualifiés d'être non fonctionnels.

Deux types d'approches sont utilisés pour traiter l'aspect non fonctionnel dans les systèmes informatiques, les approches orientées produit (″product-oriented″) et les approches orientées processus (″process-oriented″) [41]. Le premier type (orientées produit) portent sur l'évaluation des propriétés non fonctionnelles dans le produit logiciel final, c'est-à-dire déterminer s'il satisfait ou non certains besoins non fonctionnels par la définition généralement d'un ensemble d'attributs quantitatifs. Les approches orientées processus, prennent en considération les propriétés non fonctionnelles en même temps que les propriétés fonctionnelles durant le processus du développement de logiciel. Notre contribution se situe dans ce deuxième type d'approches et son objectif est de fournir au concepteur de l'application des notations formelles permettant de caractériser l'aspect non fonctionnel de son logiciel cela au cours de la phase de conception architecturale.

En fait, les ADLs qui sont censés décrire de façon formelle l'architecture d'un système à un niveau d'abstraction élevé peuvent être dotés de mécanismes pour l'expression et la vérification de ce type de propriétés. Actuellement, les recherches sur les ADLs, issus de milieux académiques, changent d'orientation et visent l'extension de ces langages avec des notations syntaxiques (généralement connues sous le nom de contrats) décrivant les services offerts par un élément architectural (composant, connecteur ou configuration) et aussi ceux dont il a besoin pour fonctionner. Dans ce contexte, nous suggérons un support mathématique formel permettant l'expression et la vérification des propriétés non fonctionnelles au niveau des architectures logicielles décrites en SADL, tout en exploitant l'expressivité et la puissance de la logique de réécriture via son langage Maude. La présentation notre approche se fera en deux étapes pour une meilleure compréhension. Nous présentons tout d'abord l'extension faite sur le modèle générique associé aux architectures SADL présenté dans le chapitre précédant pour prendre en compte l'aspect non fonctionnel dans une architecture SADL. Nous procédons ensuite à présenter le module système Maude permettant de valider cet aspect.

Notre approche est présentée dans la section suivante, à travers l'exemple simple et assez générique considéré jusqu'ici : architecture SADL du compilateur (figure 4.3).
Notons que le formalisme logique de réécriture via son langage Maude offre des spécifications exécutables et prêtes à l'analyse en tirant profit des outils de cet environnement.

3.1 La spécification

Actuellement, aucune suggestion efficace et complète n'a été faite pour considérer l'aspect non fonctionnel dans une architecture logicielle. Pour pouvoir décrire cet aspect dans les architectures SADL, nous proposons d'intégrer une nouvelle partie appelée configuration non fonctionnelle à la structure globale d'une architecture SADL. Ceci est réalisable en modifiant l'opération principale qui génère une architecture SADL comme suit:

```
op ARCHITECTURE_COMPONENTS_CONNECTORS_CONFIGURATION_
NFCONFIGUATION_ : Head SetComponent SetConnector SetCon NfConf ->
Architecture [ctor prec 25] .
```

où NFCONFIGURATION est l'entête indiquant le début de la spécification non fonctionnelle et NfConf dénote la sorte du terme algébrique spécifiant la structure statique de l'information non fonctionnelle dans l'architecture logicielle.
En fait, notre approche de modélisation des propriétés non fonctionnelles repose sur un principe très général qui déclare que les besoins de satisfaction d'une propriété non fonctionnelle peuvent être fonctionnels ou non fonctionnels. Pour cela nous introduisons une autre construction aussi importante que la précédente:

```
NfReq REQUIRE { SetReq }
```

Où NfReq représente une propriété non fonctionnelle liée à un port de sortie (point d'accès au service) ou à un connecteur (support du service) et SetReq représente une liste des besoins fonctionnels et/ou non fonctionnels permettant de garantir cette propriété.
La spécification des offres en termes de services fonctionnels est définie par la construction :

```
element1 PROVIDE{ element2 . FReq }
```

Pour exprimer que l'élément architectural "element1" fournit le service fonctionnel FReq à l'élément architectural element2.
Ces constructions supplémentaires utiles pour décrire l'information non fonctionnelle dans une architecture SADL sont implémentées dans un module fonctionnel Maude présenté dans la figure 5.1 :

```
fmod NfConfiguration is
extending Component .
```

```
extending Connector .
sorts NfProp FProp NfReq FReq SetReq NfStatement NfConf .
subsort NfReq FReq < SetReq .
subsort NfStatement < NfConf .
op _._ : ConnectorName NfProp -> NfReq [ctor prec 21 ] .
op _._ : OPortName NfProp -> NfReq [ctor prec 21] .
op _._ : ConnectorName FProp -> FReq [ctor prec 21] .
op _._ : OPortName FProp -> FReq [ctor prec 21] .
op _._ : IPortName FProp -> FReq [ctor prec 21] .
op none : -> SetReq [ctor] .
op none : -> NfConf [ctor] .
op __ : SetReq SetReq -> SetReq [ctor assoc id: none comm prec 22] .
op _REQUIRE{_} : NfReq SetReq -> NfStatement [ctor prec 23 ] .
op _PROVIDE{_} : IPortName FReq -> NfStatement [ctor prec 23 ] .
op _PROVIDE{_} : ConnectorName FReq -> NfStatement [ctor prec 23] .
op __ : NfConf NfConf -> NfConf [ctor assoc id: none comm prec 24 ] .
endfm
```

Figure 5.1 : Le module Maude pour gérer l'information non fonctionnelle

Pour expliciter notre approche de conception du module NfConfiguration, nous l'illustrons à travers l'exemple du compilateur (figure 4.3). Une configuration non fonctionnelle possible peut être décrite au niveau de la figure 5.2.

```
fmod CompilerNfConf is
including Compiler NfConfiguration .
ops NoTokenLost NoAstLost : -> NfProp [ctor] .
ops ReadSpeed BufferSize : -> FProp [ctor] .
op nfconf : -> NfConf [ctor] .
eq nfconf =
   token-oport . NoTokenLost REQUIRE{ token-channel . BufferSize }
   token-iport PROVIDE{ token-channel . ReadSpeed }
   base-ast-oport . NoAstLost REQUIRE{ token-channel . NoTokenLost }
   token-channel PROVIDE{ token-oport . BufferSize }
   token-channel . NoTokenLost REQUIRE{ token-oport . NoTokenLost
      token-iport . ReadSpeed } .
endfm
```

Figure 5.2 : Configuration non fonctionnelle pour l'architecture parser-L1

Dans cette configuration, l'information non fonctionnelle est définie par trois propriétés (non fonctionnelles) ainsi que des besoins fonctionnels et/ou non fonctionnels garantissant ces propriétés.

Ainsi, dans ce module (figure 5.2), l'opération nfconf décrit, à l'aide d'une équation, deux propriétés non fonctionnelles NoTokenLost et NoAstLost et deux services fonctionnel BufferSize et ReadSpeed. La propriété non fonctionnelle NoTokenLost (première ligne de l'équation) est attachée au port de sortie token-oport du composant lexical-analyzer, la satisfaction de cette propriété nécessite un service fonctionnel BufferSize de la part du connecteur token-channel. La deuxième ligne de l'équation spécifie un service fonctionnel ReadSpeed offert par le port d'entrée token-iport au connecteur token-channel.

Afin d'illustrer mieux l'approche de spécification nous présentons dans la figure ci-dessous un schéma explicatif contenant l'architecture parser-L1 annotée avec l'information non fonctionnelle.

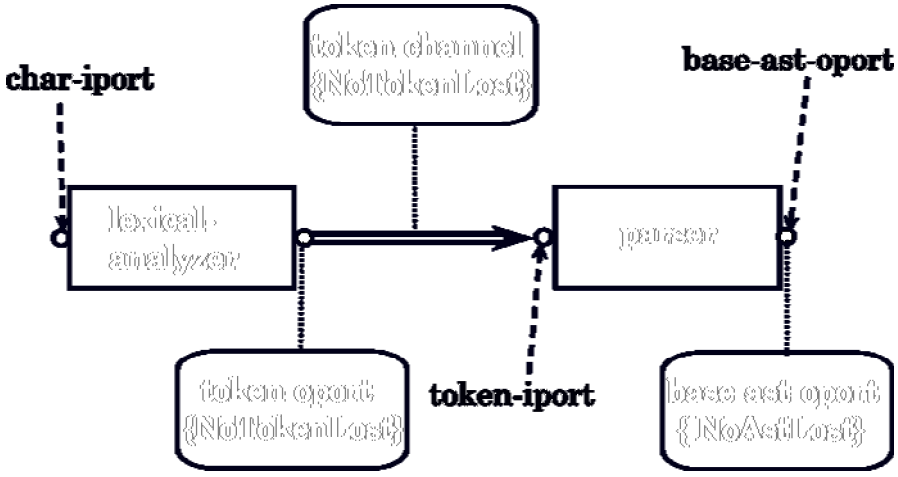


Figure 5.3 : L'architecture parser-L1 annotée avec l'information non fonctionnelle

3.2 La vérification

L'extension d'une architecture SADL par l'expression de l'information non fonctionnelle facilite aisément l'analyse des propriétés fonctionnelles ou non fonctionnelles qui y lui sont reliées. Dans notre cas le processus de vérification consiste à tester l'inclusion de l'ensemble des besoins requis par une propriété non fonctionnelle dans l'ensemble des besoins contenus dans l'information non fonctionnelle. La mise en oeuvre de ce processus peut être réalisée par un module système Maude appelée validate donné au niveau de la figure 5.4, contenant l'ensemble des règles de réécritures formalisant ce processus.

```
mod validate is
including CompilerNfConf .
```

```
op check : NfConf NfReq -> Bool .
op _IN_ :  NfStatement NfConf -> Bool .
var oprt  : OPortName .
var iprt : IPortName .
var fp : FProp .
var concn : ConnectorName .
var nfp : NfProp .
var setrq : SetReq .
var conf : NfConf .
var nfs : NfStatement .
vars nfr1 nfr2 : NfReq .
eq nfs IN nfs conf = true .
eq nfs IN conf = false [owise] .
crl [1] : check(oprt . nfp REQUIRE{ concn . fp setrq }  conf , oprt . nfp) => check(oprt . nfp
REQUIRE{ setrq } conf , oprt . nfp) if concn PROVIDE{ oprt . fp } IN conf .
crl [2] : check(concn . nfp REQUIRE{ iprt . fp setrq } conf , concn . nfp) => check(concn .
nfp REQUIRE{ setrq } conf , concn . nfp) if iprt PROVIDE{ concn . fp } IN conf  .
crl [3] : check(nfr1 REQUIRE{ nfr2 setrq } conf, nfr1) => check(nfr1 REQUIRE{ setrq }
conf, nfr1) if check(conf , nfr2) => true .
crl [4] : check(oprt . nfp REQUIRE{ concn . fp setrq }  conf , oprt . nfp) => false if not(
concn PROVIDE{ oprt . fp } IN conf ) .
crl [5] : check(concn . nfp REQUIRE{ iprt . fp setrq } conf , concn . nfp) => false if not( iprt
PROVIDE{ concn . fp } IN conf ) .
crl [6] : check(nfr1 REQUIRE{ nfr2 setrq } conf, nfr1) => false if check(conf , nfr2) =>
false .
rl [7] : check( nfr1 REQUIRE{ none } conf, nfr1) => true .
endm
```

Figure 5.4 : Le module validate

L'opération principale dans ce module est appelée check**,** elle est utilisée pour vérifier si une propriété non fonctionnelle est garantie ou non par rapport à une information non fonctionnelle décrite. Alors le processus de vérification consiste à éliminer chaque besoin requis si il est garanti, plus précisément on élimine les besoins fonctionnels s'ils sont offerts par autres éléments (règles 1 et 2) et les besoins non fonctionnels si ils sont valides par rapport aux éléments qui lui sont associés (règle 3), concernant la dernière règle de réécriture son rôle est de confirmer la validité d'une propriétés non fonctionnelle.

Un exemple d'utilisation de ce module est donné dans la figure 5.5 pour garantir la propriété non fonctionnelle token-oport. NoTokenLost déclarée dans l'information non fonctionnelle associée à la configuration de l'architecture Compiler. Notons que le modèle théorique étendu est assez générique, il peut être appliqué à n'importe quelle architecture SADL et quelque soit la propriété non fonctionnelle considérée. Nous projetons dans un travail futur, d'appliquer cette modélisation à un exemple significatif de grandeur naturelle.

```
Maude> rew check(nfconf , token-oport . NoTokenLost ) .
rewrite in validate : check(nfconf, token-oport . NoTokenLost) .
rewrites: 4 in 1625750370999ms cpu (0ms real) (0 rewrites/second)
result Bool: true
```

Figure 5.5 : Un exemple d'utilisation de l'opération check

4 Conclusion

Dans ce dernier chapitre nous avons présenté une approche permettant l'intégration de l'information non fonctionnelle dans l'architecture logicielle. Tout d'abord, nous avons enrichi le modèle sémantique des architectures SADL par des constructions permettant de qualifier l'aspect non fonctionnel des architectures SADL, plus précisément nous nous sommes concentré sur la spécification des besoins de satisfactions des propriétés non fonctionnelles.

Dans une autre partie nous avons présenté une formalisation de l'approche de vérification des propriétés non fonctionnelles.

Conclusion Générale

Face aux limites rencontrées dans l'approche de conception par objet, une nouvelle méthode de conception des applications logicielles est apparue, elle correspond à la définition de l'architecture des systèmes logicielles en ce basant sur un ensemble d'unités réutilisables. Dans ce contexte différents langages de définition d'architectures ont été proposés, chacun entre d'eux propose des constructions pour améliorer la description et l'analyse des architectures logicielles. Les travaux menés jusqu'à présent sur le développement centré architecture de logiciels portent essentiellement sur la spécification formelle des architectures. La vérification et l'analyse des architectures logicielles restent un point ouvert de recherche. En effet, il est indispensable d'associer à une architecture logicielle des spécifications précises tant fonctionnelles que non fonctionnelles. Ces spécifications décrivent des services offerts par les différents composants de cette architecture, mais aussi ceux dont ils ont besoin pour fonctionner. L'environnement possède des propriétés qui ont une influence sur le fonctionnement du logiciel. Il peut s'agir de propriétés temporelles, des propriétés sur le mode de panne ou sur les attaques possibles en termes de sécurité, etc. Ces spécifications sont rarement complètes et formelles.

En général, le terme contrat est de plus en plus utilisé pour décrire les propriétés d'un composant architectural. Les propriétés non fonctionnelles, devant être exprimées à part, permettent de caractériser le degré de satisfaction d'un logiciel, c'est-à-dire comment ses fonctionnalités sont réalisées. Elles sont associées à différents concepts comme par exemple la performance, la qualité de service, la sécurité, la robustesse, la portabilité, etc.

De par leur complexité et leur haut niveau d'abstraction, elles sont rarement prises en compte dans le processus de développement d'un logiciel, particulièrement sa conception architecturale. En effet, le concepteur doit pouvoir considérer ce type de propriétés dès le début du processus, et les estimer avant la phase d'implémentation du système.

En fait, les ADLs qui sont censés décrire de façon formelle l'architecture d'un système à un niveau d'abstraction élevé peuvent être dotés de mécanismes pour l'expression et la vérification de ce type de propriétés.

L'objectif principal de ce travail est de fournir au langage SADL les moyens nécessaires pour permettre une spécification explicite et une vérification rigoureuse des propriétés non fonctionnelles.

Pour cela nous avons tout d'abord défini un modèle sémantique extensible basé logique de réécriture permettant décrire tous les concepts de base du langage SADL (composant, connecteur, configuration, raffinement, etc.). Ce modèle s'est avère un cadre théorique approprié pour décrire d'une part, les propriétés non fonctionnelles des éléments architecturaux et d'autre part, l'assemblage de ces éléments pour garantir les propriétés non fonctionnelles spécifiées. En plus, la réalisation de cette spécification en Maude offre une spécification exécutable, lisible et extensible, du à la flexibilité et la puissance de ce langage.

Ainsi, nous avons étudié une première solution au problème de la prise en compte des propriétés non fonctionnelles à un niveau haut et abstrait dans le processus de conception des systèmes logiciels. En effet, notre approche adoptée est basée sur la description des besoins de satisfaction des propriétés non fonctionnelles, qui peuvent être fonctionnels ou non fonctionnels. Pour cela, nous avons suggéré l'introduction au modèle formel du langage SADL de nouvelles constructions permettant de décrire ces besoins.

Nos futurs travaux dans ce cadre, porteront sur l'utilisation de ces résultats pour élargir l'éventail des exemples à des applications significatives de la réalité (systèmes embarqués, transactionnels, etc.). Nous intéresserons en particulier aux architectures logicielles reconfigurables.

Les spécifications Maude écrites sont assez génériques et peuvent être considérées pour n'importe quel langage de description d'architecture logicielle.

Nous envisageons également comme travaux futurs la formalisation de la relation de raffinement d'architectures en utilisant les fonctions méta niveaux de la logique de réécriture.

Bibliographie

[01] Faiza Belala, Fateh Latreche, Malika Benammar, *A Formal Semantic Framework for SADL Language*, To appear in Proc. of ACIT'07, Lattakia Syria, 2007.

[02] Anne-Marie Déplanche, Sébastien Faucou, *Les Langages de Description d'Architecture pour le Temps Réel*, Institut de recherche en communications et cybernétique de Nantes, École d'été Temps Réel 2005 (ETR'05), 2005.

[03] Alexandre Rademaker, Christiano Braga, Alexandre Sztajnberg, *A Rewriting Semantics for a Software Architecture Description Language*, Electronic Notes in Theoretical Computer Science, Vol 130, pp 345–377, 2005.

[04] Christopher Van Eenoo, Osama Hylooz, Khaled M. Khan, *Addressing Non-Functional Properties in Software Architecture using ADL*, Proc. 6th Australasian Workshop on Software and Systems Architectures (AWSA 2005), Brisbane, 2005.

[05] Jérôme Revillard, *Approche centrée architecture pour la conception logicielle des instruments intelligents***,** Thèse de Doctorat, Université de Savoie, décembre 2005.

[06] Christophe Mareschal, Onera-Supaéro, *Adaptation d'un Langage de Description d'Architecture à l'Expression du Comportement, Applications*, Journée FAC, Toulouse, janvier 2004.

[07] Humberto Cervantes, *Vers un Modèle a Composants Orienté Services pour Supporter la Disponibilité Dynamique*, Thèse de Doctorat, Université joseph fourier - grenoble 1, Mars 2004.

[08] Karim Megzari, *Refiner : Environnement Logiciel pour le Raffinement d'Architectures Logicielles Fondé sur une Logique de Réécriture*, Thèse de Doctorat, Université de Savoie, Annecy, décembre 2004.

[09] Lionel Blanc, *Modélisation des Processus « Métier » mis en œuvre dans une Approche EAI en vue de leur Pilotage « Le Pilotage des Applications Iintégrées»*, Thèse de Doctorat, Université de Savoie, décembre 2004.

[10] Manuel Clavel, Franciso Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer, and José Quesada, *Maude: Specification and Programming in Rewriting Logic*, Theoretical Computer Science, vol 285, pp 187–243, 2002.

[11] José Meseguer, *Rewriting Logic Revisited*, Slides of tutorial presented at WRLA 2002, Pisa, Italy, September 2002.

[12] Michel Riveill, Aline Senart, *Coopération dans les Systèmes à Objets*, volume 8 de RSTI - L'Objet, chapitre Aspects dynamiques des langages de description d'architecture logicielle, Hermes, 2002.

[13] Projet ACCORD: Etat de l'Art sur Les Langages de Description d'Architecture (ADL), 2002.
http:// www.infres.enst.fr.projet/accord/.

[14] Nelson S. Rosa, Paulo R. F. Cunha, George R. R. Justo, *Process^{NFL}: A Language for Describing Non-functional Properties*, Proc. of the 35th Annual Hawaii International Conference (HICSS), pp 3676–3685, 2002.

[15] Jan Øyvind Aagedal, *Quality of Service Support in Development of Distributed Systems*. Thesis doctor Scientiarum, Department of Informatics, University of Oslo, 2001.

[16] Ariel D. Fuxman, *A Survey of Architecture Description Languages*. In Collection of Reports from CSC2108 Automatic Verification, University of Toronto, 2000.

[17] Nenad Medvidovic, Richard M. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Vol 26, no1, pp70–93, Janvier 2000.

[18] Benjamin Blanc, *Modélisation et Spécification d'Architectures Logicielles*, Rapport de Stage, laboratoire LSV de l'école normale supérieure de cachan, 1999.

[19] Vorgelegt Von Vean-Guy Schneider, *Components, scripts, and glue: a conceptual framework for software composition*, Ph.D. thesis, University of Berne, Switzerland, 1999.

[20] José Meseguer, *Research Directions in Rewriting Logic*, NATO Advanced Study Institute Series F, Vol 165, pp 345–398, Springer Verlag, 1999.

[21] José Meseguer, Carolyn Talcott, *Formal Foundations for Compositional Software Architectures*, Position Paper, in proc. of OMG-DARPA-MCC Workshop on Compositional Software Architecture, 1998.

[22] Xavier Franch, Pere Botella, *Putting Non-functional Requirements into Software Architecture*, Proc. of the 9th International Workshop on Software Specification, and Design, pp 60–67, 1998.

[23] Milica Barjaktarovic, *The State-of-the-Art in Formal Methods*, Rapport Technique, Wilkes University and WetStone Technologies Inc, Janvier 1998.

[24] Steve Vestal, *MetaH Programmer's Manual, Version 1.27*, Rapport Technique, Honeywell Technology Center, 1998.

[25] Luc Bellisard, *Construction et configuration d'applications réparties*, Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1997.

[26] Mark Moriconi et R. A. Riemenschneider, *Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*, Rapport Technique SRI-CSL-97-01, Computer Science Laboratory, SRI International, mars 1997.

[27] David Garlan, Robert Monroe, David Wile, *ACME: An Architecture Description Language*, Proc. of CASCON'97, November, 1997.

[28] José Meseguer, *Rewriting logic as a Semantic Framework for concurrency*, A Progress Report, SRI international, 1996.

[29] Allen Robert, Garlan David, *The Wright Architectural Specification Language*, Rapport technique MU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pillsburgh,PA, September 1996.

[30] David C. Luckham, *Rapide: A Language And Toolset For Simulation of Distributed Systems by Partial Orderings of Events*, Proc. of the DIMACS Workshop on Partial order methods in verification, august 1996.

[31] Narciso Marti-Oliet, José Meseguer, *Rewriting Logic as a Logical and Semantic Framework*, Electronic Notes in Theoretical Computer Science, Vol 4, no1, pp1-36, 1996.

[32] David Garlan, *An Introduction to the Aesop System*, Carnegie Mellon University, the ABLE Project, juillet 1995.
http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesopoverview.ps

[33] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robins, *A Component and Message-Based Architectural Style for GUI Software*, Proc. of the Sevenbteenth International Conference on Software Engineering (ICSE17), pp 295–304, Seattle WA, April 1995.

[34] Magee J., Dualy N., Eisonbach S. and Kramer J., *Specifying Distributed Software Architectures*, In Proc. of the Fifth Symposium on the Foundations of Software Engineering (FSE4), vol 989,pp 137–153,1995.

[35] Narciso Marti-Oliet, José Meseguer, *From Abstract Data types to Logical Framework*, Lecture Notes in Computer Science, Springer-Verlag, Vol 906, pp 48–80, 1995.

[36] Mark Moriconi, X. Qian, *Correctness and Composition of Software Architectures*, proc. of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering, Computer Science Laboratory, SRI International, December 1994.

[37] Mohamed Bettaz, Mourad Maouche. *How to Specify Non Determinism and True Concurrency with Algebraic Term Nets*, Lecture Notes in Computer Science, Springer-Verlag, vol 655, pp 164-180, 1993.

[38] Narciso Marti-Oliet, José Meseguer, *Rewriting Logic as a Logical and Semantic Framework*, Technical Report SRI-CSL-93-05, Menlo Park, CA 94025, and Center for the study of language and Information Stanford University, Stanford, CA 94305, 1993.

[39] Gregory Abowd, Robert Allen, David Garlan., *Using Style to Give Meaning to Software Architecture,* In Proc. of SIGSOFT 93: Foundations of Software Engineering, Software Engineering Notes 118(3), pp 9–20, ACM Press, 1993.

[40] José Meseguer, *Conditional Rewriting Logic as a unified Model of Concurrency,* Theoretical Computer Science 96, pp 73–55, 1992.

[41] John Mylopoulos, Lawrence Chung and Brian Nixon, *Representing and Using Non-Functional Requirements: A Process-Oriented Approach*, IEEE Transactions on Software Engineering, 18(6), pp. 483–497, 1992.