Contribution à une démarche de vérification formelle d'architectures logicielles

Mohamed Graiet

Université Joseph Fourier - Grenoble 1
Université de Sfax

THÈSE

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER
Discipline : INFORMATIQUE

Titre :

Contribution à une démarche de vérification formelle d'architectures logicielles

Présentée et soutenue à Grenoble publiquement par

Mohamed GRAIET

Le 25 Octobre 2007

Jury :

Présidente du jury :	Mlle Marie-Christine Fauvet
Rapporteurs :	M. Philippe Aniorté
	M. Mourad Chabane Oussalah
Examinateurs :	M. Noureddine Belkhatir
	M. Samir Ben Ahmed
Directeurs de thèse :	M. Abdelmajid Ben Hamadou
	M. Jean-Pierre Giraudin
	M. Mohamed Tahar Bhiri

Thèse préparée dans le cadre d'une convention de co-tutelle et avec l'aide du projet franco-tunisien CMCU DEFI :

* Équipe SIGMA (**S**ystèmes d'**I**nformation - in**G**énierie et **M**odélisation **A**daptables) – LIG (**L**aboratoire d'**I**nformatique de **G**renoble)

* Laboratoire MIRACL (**M**ultimedia **I**nfo**R**mation systems and **A**dvanced **C**omputing **L**aboratory), Université de Sfax

Dédicaces

À tous ceux qui m'ont soutenu moralement pendant la rédaction de cet ouvrage. Plus particulièrement à mon père Salem, ma mère Fatama, mon oncle Abedrazek.

À mes grands parents décédés Hamadi et Aicha.

Remerciements

Mes plus sincères remerciements vont à M. Jean-Pierre Giraudin, Professeur à l'Université Pierre Mendès France pour m'avoir accueilli au sein de son équipe et avoir dirigé ce travail, ainsi que pour les nombreuses discussions et les nombreux échanges scientifiques qui ont eu lieu tout au long de cette thèse. Il a été avec son équipe un soutien permanent et efficace pendant ces années de thèse.

Mes plus sincères remerciements vont à M. Abdelmajid Ben Hamadou, Professeur à l'Université de Sfax pour m'avoir accueilli au sein de son Laboratoire et avoir dirigé ce travail. Il a montré beaucoup d'intérêt à mon travail et à la collaboration entre les deux universités. Sous sa présidence, j'ai eu l'occasion de participer à l'organisation du congrès INFORSID 2006 en Tunisie.

Je suis très reconnaissant envers M. Mohamed Tahar Bhiri, Maître assistant à l'Université de Sfax pour sa grande disponibilité, pour son soutien, pour ses conseils scientifiques et humains et ses encouragements permanents. Tu m'as enseigné et encadré depuis le Master et maintenant grâce à toi j'arrive à voir la fin de ce long parcours qui est la thèse. Je n'oublie jamais tes cours prestigieux où j'ai beaucoup appris les concepts d'une façon fine lorsque j'étais étudiant ou enseignant débutant. Je te remercie infiniment pour l'effort constant fourni durant l'encadrement de cette thèse.

Je tiens à remercier Mlle Marie-Christine Fauvet, Professeur à l'Université Joseph Fourier de Grenoble d'avoir accepté de présider le jury, ainsi que M. Philippe Aniorté, Professeur à l'Université de Pau et des Pays de l'Adour et M. Mourad Chabane Oussalah, Professeur à l'Université Nantes pour l'évaluation de ce travail. Enfin je remercie M. Noureddine Belkhatir, Professeur à l'Université Pierre Mendès France et M. Samir Ben Ahmed Professeur à l'Université de Tunis d'avoir accepté de faire partie du jury.

Je tiens à remercier les deux responsables du projet DEFI-CMCU 07G1411, M. Samir Ben Ahmed côté tunisien et Mme Jeanine Souquieres côté Français pour leur effort fourni pour mettre en place ce projet.

Je tiens à remercier l'ensemble du personnel technique et administratif du LIG (**L**aboratoire d'**I**nformatique de **G**renoble) et du Laboratoire MIRACL (**M**ultimedia **I**nfo**R**mation systems and **A**dvanced **C**omputing **L**aboratory).

Je tiens à remercier mes amis : Mourad, Walid, Kaled, Lousif, Habib, Mohamed Ali, Hichem, Abedhalek, Hafed, Fatma, Zied, Hager, Faiza et AmmEnacer pour leur soutien moral.

Je tiens à remercier M. Kais Haddar Maître assistant à l'Université de Sfax pour m'avoir accueilli au sein de son bureau pendant ces années de thèse et pour son soutien moral.

Enfin, j'associe mes remerciments à tous les membres de l'équipe SIGMA pour leur bonne humeur et l'ambiance créée ainsi que toute personne qui a contribué de près ou de loin à la réalisation de ce travail.

Table des matières

Liste des Figures

Liste des Tableaux

Introduction générale

1. Architecture logicielle

Le domaine de l'architecture logicielle est devenu un champ à part entière au niveau du génie logiciel : des workshops et des conférences spécialisés tels que EWSA (European Workshop on Software Architectures) et CAL (Conférence francophone sur les Architectures Logicielles) font maintenant le point sur ce domaine.

L'architecture logicielle fournit une description de haut niveau de la structure d'un système. Elle est définie par des composants, des connecteurs et des configurations (Garlan, 1993). La conception architecturale occupe une position clef et critique dans le processus de développement d'un système. L'architecture logicielle offre de nombreux avantages tout au long du cycle de vie du logiciel (Shaw, 1996). La présence d'une représentation de l'architecture logicielle du système étudié favorise le passage de l'étape de conception à l'étape d'implémentation. De même lors de l'étape de maintenance corrective, elle facilite la localisation des erreurs résiduelles. Egalement, elle favorise l'extensibilité du logiciel lors de l'étape de maintenance évolutive.

En tant que description de haut niveau, l'architecture logicielle permet de réduire la complexité du système étudié. Ensuite, l'adjonction des détails liés aux décisions conceptuelles et au raffinement peut être effectuée d'une façon incrémentale. Mais, ceci suppose que l'architecture logicielle possède des propriétés bien définies et prouvées formellement par des outils appropriés.

2. Description d'architectures logicielles

La communauté scientifique a développé plusieurs formalismes permettant la spécification plus ou moins précise des descriptions architecturales.

2.1. Les langages de description d'architectures (ADL : Architecture Description Language)

Un ADL se concentre plutôt sur la structure de haut niveau de l'ensemble de l'application, que sur les détails d'implémentation (Vestal, 1993). Parmi les ADL les plus connus, nous citons : Wright (Allen, 1997a), ACME (Garlan, 2000), Rapide (Luckham, 1995), Unicon (Shaw, 1995), C2 (Taylor, 1996), Darwin (Magee, 1995) et AESOP (Garlan, 1994). D'après (El Boussaidi, 2006), les propriétés principales qu'un ADL doit avoir sont divisées en trois sous-familles comme suit.

2.1.1. Les exigences minimales fondamentales

Elles couvrent la spécification des composants, des connecteurs, des configurations et éventuellement des styles architecturaux.

Un composant est une unité de calcul ou de données qui peut être atomique ou composite et qui possède une interface décrivant les points d'interaction du composant avec l'environnement. Un ADL doit permettre la spécification d'un type de composant et de son interface.

Un connecteur modélise les interactions entre les composants et aussi les règles qui régissent ces interactions. Un connecteur a les mêmes caractéristiques qu'un composant : une interface et un type qui représente une description abstraite de l'interaction qu'il

incarne. Il est important qu'un ADL permette de spécifier un connecteur comme une entité **de première classe**. Ceci permet de le réutiliser.

Une configuration décrit la structure complète d'un système sous forme d'un graphe connexe regroupant des composants et des connecteurs. Un ADL doit fournir des aptitudes pour modéliser la configuration.

Un style architectural décrit une classe générique d'architecture telles que : client-serveur, filtre-pipe et architecture par couches. Il est important qu'un ADL permette de spécifier explicitement un style architectural et un mécanisme d'instanciation adéquat afin d'engendrer des configurations conformes au style architectural.

2.1.2. Les exigences souhaitables

Ces exigences favorisent des implémentations correctes des architectures. Elles couvrent la modélisation de la sémantique, la spécification des contraintes, la composition hiérarchique et les propriétés non fonctionnelles.

La modélisation de la sémantique concerne à la fois les composants et les connecteurs. Il s'agit de décrire les comportements dynamiques des composants et des connecteurs. Par exemple, dans Wright (cf. chapitre 2 et 3), ces comportements sont décrits en CSP (Communicating Sequential Processes) de Hoare (Hoare, 1985) (Hoare, 1995) (Hoare, 2004).

La spécification des contraintes concerne les composants, les connecteurs et les configurations. Par exemple, dans Wright, les contraintes relatives aux composants et connecteurs sont spécifiées par des types d'interfaces. Tandis que celles relatives aux styles architecturaux sont exprimées par des prédicats.

La composition hiérarchique concerne les composants et les connecteurs. Un composant ou un connecteur peut être lui-même un ensemble de composants et connecteurs. Une architecture entière peut être représentée par un composant. Par exemple, dans Wright, les composants et les connecteurs peuvent être composites.

Les propriétés non fonctionnelles (sécurité, performance, etc.) couvrent des exigences qui ne sont pas inhérentes à la sémantique des concepts architecturaux (composant, connecteur, configuration). Cependant, leur description est nécessaire pour une implémentation correcte de ces concepts. Par exemple, l'ADL UniCon permet d'exprimer certaines propriétés comme la planification.

2.1.3. Les exigences désirables mais non fondamentales

Ces exigences supportent l'activité de l'architecte. Elles couvrent la réutilisation, l'évolution et une boîte à outils.

La réutilisation : il est préférable qu'un ADL offre des possibilités permettant la réutilisation des concepts architecturaux. Par exemple, dans Wright, les types des composants et des connecteurs peuvent être réutilisés. Egalement, les types peuvent être paramétrés. De même, les styles peuvent être étendus.

Evolution : Un ADL doit supporter des mécanismes permettant de faire évoluer -par le changement des propriétés- des composants, des connecteurs et des configurations. Par exemple, Darwin supporte l'instanciation dynamique des composants.

Outils de support : La disponibilité d'un outil renforce l'utilité d'un ADL. Ces outils peuvent être de divers types : analyseur de syntaxe et type, générateur de code, analyseur statique des propriétés et traducteur vers des outils de vérification formelle.

Par exemple, Wright (cf. chapitre 3) propose un outil permettant de traduire de Wright vers CSP accepté par FDR (FDR2, 2003).

2.2. La description architecturale dans UML

UML est un langage moins formel et moins rigoureux que les ADL. Cependant, plusieurs chercheurs (Garlan, 2002) (Garlan, 2001) (Medvidovic, 2002) ont étudié la possibilité de modéliser une architecture en utilisant UML1.x.

Deux approches conformes au standard UML sont proposées. La première stratégie utilise UML tel qu'il est, pour représenter les concepts architecturaux. La seconde stratégie consiste à définir des profils (cf. chapitre 6).

UML2.0 a introduit des nouvelles constructions telles que composant, interface offerte, interface requise, port, connecteur, machine de description de protocoles et structure composite (cf. chapitre 7). Ces constructions rendent UML2.0 plus adapté au développement à base de composants et à la spécification des descriptions architecturales (Ivers, 2004).

Les gains apportés par l'utilisation d'UML -surtout UML2.0- comme un langage de description d'architectures sont multiples :

- permettre à plusieurs intervenants qui ont peu de connaissance dans le domaine des spécifications formelles de comprendre et manipuler une description architecturale ;
- tirer profit des outils supportant le standard UML.

Comme inconvénient majeur, une description architecturale en UML ne peut pas être analysée d'une façon rigoureuse et formelle dans le cadre d'UML. Une ouverture d'UML sur certains ADL formels comme Wright permet de faire face à cet inconvénient.

2.3. La description architecturale dans les formalismes orientés composants

Certains formalisme orientés composants comme la méthode Symphony (cf. chapitre 9) et Ugatze (Seyler, 2004) offrent des possibilités permettant de modéliser une architecture. Mais ces formalismes comme UML ne favorisent pas une analyse rigoureuse d'une description architecturale. Des ouvertures de ces formalismes sur certains ADL formels comme Wright s'imposent.

3. Outils pour la vérification d'architectures logicielles

L'activité de vérification est l'étape qui permet de s'assurer que le système est développé correctement c'est-à-dire possède certaines propriétés. Cette activité est supportée par des notations formelles comme VDM (Bjørner, 1978), Z (Spivey, 1989), B (Abrial, 1996) et Perfect Developer (Crocker, 2003) (dans le domaine des systèmes séquentiels) et CSP (Hoare, 1985) (Hoare, 1995) (Hoare, 2004), CCS (Milner, 1980) et Réseaux de Petri (Peterson, 1981) (dans le domaine des systèmes concurrents). Pour vérifier une propriété avec ces notations formelles, il existe principalement deux pistes : la preuve et le « model checking ». La preuve consiste à démontrer mathématiquement des propriétés exprimées sous forme des théorèmes. Elle est supportée par des outils appelés prouveurs. Ces prouveurs sont plus ou moins puissants et offrent une option permettant à l'utilisateur d'intervenir au cas où le prouveur échoue. Le « model checking » est une technique de vérification automatique des systèmes modélisés à l'aide des automates à états finis. Il est supporté par des outils appelés « model checkers ». Un « model checker » implémente un

algorithme permettant d'obtenir, pour une propriété donnée, soit une preuve de sa satisfiabilité, soit un contre-exemple exposant la faute du système. Le « model checking » est applicable aussi bien sur des modèles (par exemple des spécifications CSP soumises au « model checker » FDR (FDR2, 2003)) que sur des programmes écrits dans divers langages de programmation tels que C, Ada en passant par les outils comme SPIN (Holzmann, 1991) et SMV (McMillan, 1993).

L'analyse statique par interprétation abstraite (Cousot, 2000) de programmes -écrits dans divers langages- consiste à vérifier statiquement des propriétés dynamiques des programmes. En effet, elle permet d'abstraire le programme que l'on veut analyser en s'intéressant uniquement à certains aspects qui sont précisément l'objet d'étude particulière. Plusieurs outils d'analyse statique existent tels que FLAVERS (Dwyer, 1994) (Dwyer, 1998) (Dwyer, 1999) (Cobleigh, 2002) et INCA (Corbette, 1995) (pour programmes concurrents en Ada et Java).

Une architecture logicielle peut être représentée soit par un modèle soit par un programme. Et par conséquent, les techniques de vérification formelle applicables sur les modèles et les programmes sont également applicables sur les architectures logicielles.

4. Problématique et objectifs

Cette thèse a pour objet d'aborder la problématique suivante : comment peut-on vérifier la correction d'une Architecture Logicielle (AL) décrite dans n'importe quel formalisme adéquat ? Par «formalisme adéquat», nous entendons un formalisme ayant des aptitudes pour décrire les aspects structuraux et comportementaux d'une AL. Une AL est jugée correcte si et seulement si elle possède des propriétés bien définies. Nous classons les propriétés à vérifier sur une AL en deux classes : propriétés standards et propriétés spécifiques. Les propriétés standards sont communes à n'importe quelle AL. Tandis que les propriétés spécifiques sont propres à l'AL du système étudié. Pour participer à la résolution de la problématique posée ci-dessus, nous apportons une contribution à une démarche de Vérification Formelle d'Architectures Logicielles : DVFAL. La démarche DVFAL doit répondre aux deux impératifs suivants :

- elle doit être **générique** pour permettre le traitement des AL décrites dans des formalismes assez diversifiés tels que UML, Symphony et ADL ;
- elle doit être **compatible** avec les outils de vérification formelle (model checking, preuve et analyse statique par interprétation abstraite) pour des raisons évidentes de réutilisabilité de ces outils dans le domaine des architectures logicielles.

5. Une vue d'ensemble de la démarche DVFAL

La figure 1 donne une vue d'ensemble de la démarche DVFAL.

La démarche DVFAL a retenu deux langages : l'ADL Wright et Ada. L'ADL Wright joue le rôle d'un **langage pivot** permettant de représenter une architecture logicielle décrite dans divers formalismes. Quant au langage Ada, il offre la possibilité de représenter les architectures logicielles décrites en Wright comme des programmes Ada concurrents afin de les analyser davantage avec les outils de vérification des programmes concurrents associés à Ada. En outre, cette ouverture sur le langage de programmation Ada permet l'implémentation incrémentale et correcte d'architectures logicielles en se servant progressivement des divers outils de vérification supportant Ada.

La démarche DVFAL propose trois types de traducteurs.

- **Traducteur du formalisme utilisé vers Wright**

Théoriquement, il doit y avoir autant des traducteurs que des formalismes utilisés pour la description d'architectures logicielles. Mais on peut réduire le nombre de traducteurs et la complexité de cette opération de traduction en faisant appel aux langages intermédiaires. Dans la deuxième partie de cette thèse, nous proposons un profil W-UML qui facilite la traduction d'UML2.0 vers Wright et également de Symphony vers Wright. De même l'ADL ACME (Garlan, 1997a) (Garlan, 1997b) peut être utilisé avec profit comme langage intermédiaire entre les ADL et Wright.

- **Traducteur de Wright vers CSP**

Le traducteur Wr2fdr qui accompagne Wright permet de traduire de Wright vers CSP en automatisant quelques propriétés architecturales standards définies par Wright. Mais la version actuelle Wr2fdr comporte des erreurs et elle est limitée en possibilités (cf. chapitre 2 - section 2.4.2). De plus, le CSP produit par Wr2fdr est destiné à l'outil FDR. Une action en cours est lancée pour apporter des améliorations importantes à cet outil.

- **Traducteur de Wright vers Ada**

Dans le chapitre 3 de la première partie de cette thèse, nous proposons une approche de traduction de Wright vers Ada.

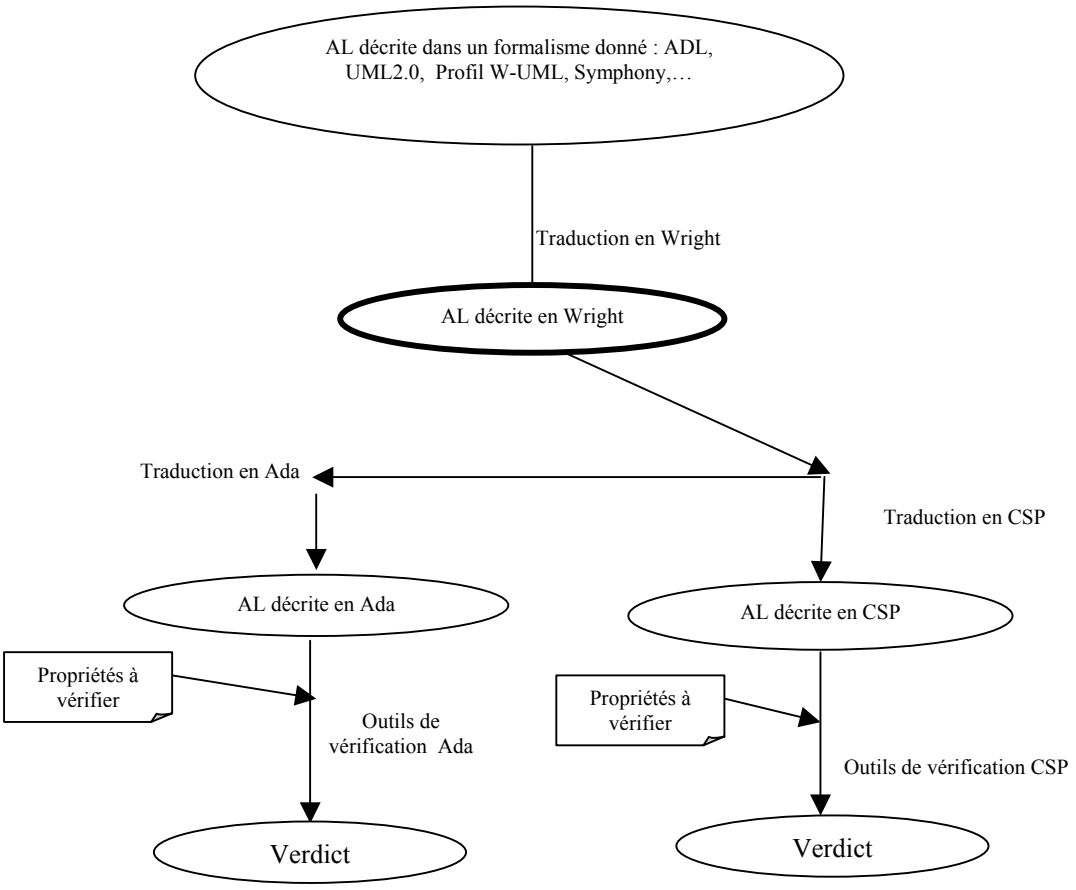


Figure 1. *Une vue d'ensemble de la démarche DVFAL*

6. Plan du mémoire

Ce mémoire de thèse est organisé en trois parties subdivisées en quatre chapitres.

La première partie est intitulée «une approche de vérification formelle d'architectures logicielles». Le premier chapitre de cette partie, présente les aspects structuraux et sémantiques de l'ADL Wright. Le deuxième chapitre évalue les possibilités de l'ADL Wright vis-à-vis de la vérification formelle des propriétés d'architectures logicielles. Le troisième chapitre préconise une Démarche de Vérification Formelle d'Architectures Logicielles. Le quatrième chapitre apporte une approche de traduction de l'ADL Wright vers Ada.

La deuxième partie intitulée, «vérification formelle d'architectures logicielles à base d'UML» est consacrée à la définition d'un langage intermédiaire -sous forme d'un profil UML2.0- entre les formalismes à base de UML et l'ADL Wright. Le cinquième chapitre présente le méta-modèle de l'ADL Wright que nous avons proposé. Le sixième chapitre présente l'extraction des fragments du méta-modèle UML2.0 nécessaires pour la définition du langage intermédiaire appelé W-UML. Le septième chapitre présente notre profil W-UML. Enfin, le huitième chapitre présente les règles de traduction depuis le profil W-UML vers Wright.

La troisième partie intitulée, «outils complémentaires et validation» commence par le chapitre 9 qui propose un outil de transformation d'expressions CSP en machines à états de description de protocoles UML supportant le profil W-UML. Le chapitre 10 apporte un outil d'automatisation de la propriété de contraintes de style de l'ADL Wright. Le chapitre 11 présente la méthode des composants métier Symphony. Enfin, le chapitre 12 propose une intégration de la méthode Symphony dans la démarche DVFAL via des connexions entre Symphony et notre profil W-UML.

Deux annexes (A et B) complètent ce document pour en illustrer l'implémentation et l'utilisation des outils développés dans le cadre de cette thèse.

Première partie : Une approche de vérification formelle d’architectures logicielles

Chapitre 1 : L'ADLWright

1.1 . Introduction

Plusieurs travaux permettant de présenter et comparer les langages de description d'architecture (ADL) existent (ACCORD, 2002) (El Boussaidi, 2006) (Oussalah, 2005). Dans ce travail, nous nous intéressons aux langages de spécification formelle d'architectures tels que Rapide (Luckham, 1995) et Wright (Allen, 1997a). Comparé à Rapide, Wright possède plusieurs points forts :

- Il propose un modèle de connecteur explicite.
- Il possède une sémantique formelle basée sur CSP de Hoare (Hoare, 1985) (Hoare, 1995) (Hoare, 2004).
- Les divers outils de vérification formelle supportant CSP peuvent être réutilisés avec profit pour l'analyse formelle d'architectures logicielles décrites en Wright.
- Des travaux liant Wright à des langages formels comme les réseaux de Petri favorisant l'accès à des outils de model checking comme SPIN et SMV existent (Jeffrey, 1999).

Dans la suite de ce travail, nous nous focalisons sur Wright. Ce chapitre comporte trois sections. La première présente les quatre concepts structuraux de Wright à savoir : composant, connecteur, configuration et style. La seconde présente les aspects syntaxiques et sémantiques de Wright. Enfin, la troisième section propose une étude de cas : fonctionnement d'une ferme (Martin, 1997b).

1.2 . Les concepts structuraux

Wright repose sur quatre concepts qui sont le composant, le connecteur, la configuration et le style.

1.2.1 . Le concept composant

Un composant en Wright est une unité abstraite localisée et indépendante. La description d'un composant contient deux parties importantes qui sont l'interface (interface) et la partie calcul (computation).

L'interface consiste en un ensemble de ports, chacun représentant une interaction avec l'extérieur à laquelle le composant peut participer. La spécification d'un port décrit deux aspects de l'interface du composant :

- Elle décrit partiellement le comportement attendu du composant : le comportement du composant est vu par la « lorgnette » de ce port particulier.
- Elle décrit ce que le composant attend du système dans lequel il va interagir.

Le calcul décrit ce que le composant fait du point de vue comportemental. Il mène à bien les interactions décrites par les ports et montre comment elles se combinent pour former un tout cohérent. C'est sur cette spécification que l'analyse des propriétés du composant va être basée (cf. chapitre 2). Un composant peut avoir plusieurs ports (et donc de multiples interfaces). Les spécifications fournissent plus que des signatures statiques de l'interface. Elles indiquent aussi des interactions dynamiques (cf. 1.3).

- **Exemple de description de composant**

L'exemple proposé concerne un système filtre de type PipeFilter permettant de lire un flot de caractères pour le transformer en flot contenant les mêmes caractères en lettres capitales. Le schéma représente ce système sous forme de diagramme :

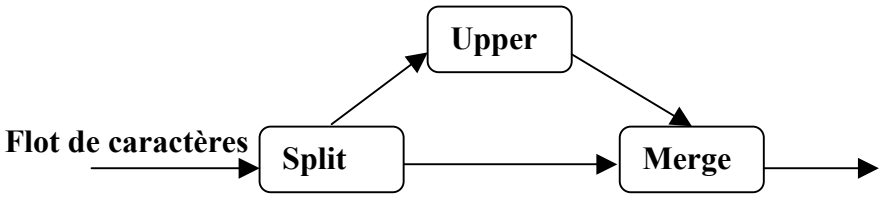


Figure 1.1. *Exemple de filtres*

Le système comprend trois composants :

- Le composant Split permettant de récupérer le flot de caractères en entrée et de créer deux flots :
  - un flot identique à celui en entrée pour le composant Upper pour la conversion en lettres capitales,
  - un flot identique à celui en entrée pour le composant Merge.
- Le composant Upper permettant de transformer un flot de caractères en flot contenant les mêmes caractères en lettres capitales,
- Le composant Merge permettant de fusionner les deux flots.

Le composant Split va être décrit sous Wright de la manière suivante (figure 1.2).

Component Split

Port Input (lire les données jusqu'à la fin des données)

Port Left (sortir les données de manière permanente)

Port Right (sortir les données de manière permanente)

Computation (lire continuellement les données à partir du port Input puis les sortir à la fois dans le port Left et le port Right)

Figure 1.2. *Exemple d'un composant*

L'interface du composant Split est formée de trois ports qui sont le port Input, Left et Right. La partie Calcul (Computation) du composant permet de mettre en rapport les trois ports. Les spécifications du comportement des ports et de la partie calcul sont écrites ici de manière informelle pour plus de compréhension. Elles sont normalement écrites avec le langage CSP de Wright (cf. 1.3).

1.2.2 . Le concept connecteur

Un connecteur représente une interaction entre une collection de composants. Il possède un type. Il spécifie le patron d'une interaction de manière explicite et abstraite. Ce patron peut être réutilisé dans différentes architectures.

Il contient deux parties importantes qui sont un ensemble de rôles et la glu :

- Chaque rôle indique comment se comporte un composant qui participe à l'interaction.
- La glu décrit comment les rôles travaillent ensemble pour créer cette interaction.

Comme pour le Calcul (au niveau du composant), la spécification de la glu (au niveau du connecteur) définit l'entière spécification du comportement. Elle coordonne le comportement des composants dans le cas où ceux-ci obéissent effectivement aux comportements indiqués par les rôles.

- **Exemple**

Si on reprend l'exemple du système de filtre de type PipeFilter, les composants filtres sont liés entre eux par des tubes. Ces tubes fonctionnent de la même façon et obéissent aux mêmes règles. On peut donc définir le connecteur suivant définissant un tube (figure 1.3) :

Connector Pipe
Role Source (délivrer les données continuellement, signaler la fin et fermer)
Role Sink (lire les données continuellement, fermer avant ou au moment de la signalisation de la fin des données)
Glue (le rôle Sink reçoit les données dans le même ordre que celui utilisé par le rôle Source pour fournir ces données).

Figure 1.3. *Exemple d'un connecteur*

1.2.3 . Le concept configuration

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composants et de connecteurs, les descriptions des liens entre les instances de composants par les connecteurs.

- **Exemple**

Si on reprend l'exemple du système de filtre de type PipeFilter, la configuration correspond à l'architecture du système Capitalize (figure 1.4).

Configuration Capitalize
Component Upper } déclaration des composants et connecteurs
Connector Pipe
Instances
Eclateur : Split
Majuscule : Upper } déclaration des instances de connecteurs et composants
Assembleur : Merge
P1, P2, P3 : Pipe
Attachments
Eclateur.Left as P1.Source
Majuscule.Input as P1.Sink
Eclateur.Right as P1.Source
Assembleur.Right as P2.Sink
Majuscule.Output as P3.Source
Assembleur.Left as P3.Sink
End Configuration

Figure 1.4. *Exemple d'une configuration*

Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants. Il en va de même pour un connecteur. Lorsqu'un composant représente un sous-ensemble de l'architecture, ce sous–ensemble est décrit sous forme de configuration dans la partie calcul du composant.

1.2.4. Le concept style

Le style d'une architecture permet de décrire un ensemble de propriétés communes à une famille de systèmes comme, par exemple, les systèmes temps réel ou les systèmes de gestion de paye. Il permet de décrire un vocabulaire commun en définissant un ensemble de types de connecteurs et de composants et un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style.

Ainsi, Wright permet de définir des types de connecteurs et composants pour une famille d'architectures à la condition que ceux-ci respectent les propriétés de cette famille. Les propriétés et les contraintes communes à une architecture peuvent être définies selon trois caractéristiques qui sont les types d'interfaces, les paramètres et les contraintes. Les types d'interfaces permettent de typer le rôle d'un connecteur ou le port d'un composant pour un système donné. Les paramètres comprennent les informations de style pour définir des composants ou des connecteurs avec des parties de leurs descriptions qui peuvent être en paramètre comme par exemple la partie calcul. Finalement, les contraintes sont des prédicats logiques de premier ordre qui doivent être satisfaits pour tous les éléments appartenant au style (figure 1.5).

- **Exemple**

```
Style PipeFilter
        Interface Type DataInput = lire des données, s'arrêter au signal end-of-data
        Interface Type Output = envoyer les données, signaler la terminaison par
close
        Connector Pipe
                Role source = DataOutput
                Role sink = DataInput
                Glue = Sink va recevoir les données dans l'ordre délivré par source
        Constraints
                ∀ c : Connectors • Type (c) = Pipe
                ∀ c : Components ; p : Port | p ∈ Ports (c) • Type (p) = DataInput ∨
                Type (p) = DataOutput
End Style
Configuration Capitalize
        Style PipeFilter
                ….
End Configuration
```

Figure 1.5. *Un style PipeFilter et une configuration Capitalize appartenant à ce style*

1.2.4.1. Les Types Composant, Connecteur et Interface

Les styles permettent d'ajouter certains aspects dans la description des types composants et connecteurs. Nous pouvons, par exemple, décrire que tous les composants du style PipeFilter doivent être des filtres qui n'utilisent en entrée et en sortie que des flots de données.

En plus de cela, les styles permettent de déclarer des types d'interfaces. Elles représentent des interfaces communes que l'architecte peut utiliser dès qu'il en a besoin. Elles peuvent être vues comme des constantes dans un langage de programmation. Ces types d'interfaces sont très pratiques et il est plus rationnel de les utiliser lorsqu'une spécification revient souvent comme c'est le cas pour les interfaces DataInput et DataOutput.

Un style a aussi l'avantage de pouvoir introduire un vocabulaire spécifique. Par exemple le style PipeFilter, le rôle qui correspond à l'entrée du pipe s'appelle communément source et le rôle qui correspond à la sortie du pipe s'appelle communément sink.

1.2.4.2. Les paramètres

Ils ont été introduits afin de fournir plus de flexibilité lors de la définition de types (au niveau des ports, des rôles, des calculs, des glus et des interfaces). Il est possible lors de la description de laisser des sortes de «trous» qui seront remplis au moment de l'instanciation pour une configuration spécifique.

Il est, par exemple, intéressant de pouvoir paramétrer le nombre de ports d'un composant. En effet, les spécifications des ports d'entrées et de sorties des filtres sont toujours les mêmes mais leur nombre peut différer. Ainsi, ces nombres pourront être déterminés au moment de l'instanciation du filtre (cf. figure 1.6).

Component SplitFilter (nout : 1 ..)
Port Input = DataInput
Port Output $_{1..\ nout}$ = DataOutput
Computation = lire des données du port Input. Envoyer ces données successivement sur les ports Output $_1$, Output $_2$,… , Output $_{nout}$

Figure 1.6. *Exemple d'un composant paramétré*

Par contre ces paramètres ne pourront pas être changés durant l'exécution ce qui reflète la nature statique de Wright. Ceci est aussi valable pour l'ensemble des composants et de leurs interactions.

1.2.4.3. Les Contraintes

C'est l'endroit où la déclaration des contraintes correspondant au style doit être définie. Par exemple pour le style PipeFilter nous indiquons que tous les connecteurs doivent être du style pipe par la spécification suivante : $\forall$ c : connectors • Type (c) = Pipe

Chaque contrainte déclarée dans un style représente un prédicat qui doit être satisfait par toutes les configurations contenant ce style. Les notations pour formuler les contraintes sont basées sur la logique du premier ordre. Nous décrivons ci-dessous la liste des ensembles et opérateurs disponibles :

- Components : l'ensemble des composants dans la configuration.
- Connectors : l'ensemble des connecteurs dans la configuration.
- Name (e) : le nom de l'élément e, où e est un composant, connecteur, port ou rôle.
- Type (e) : le type de l'élément e.
- Ports (c) : l'ensemble des ports du composant c.
- Roles (c) : l'ensemble des rôles du connecteur c.
- Glue (c) : la glu du connecteur c.

Les comportements partiels (ports, rôles) et globaux (calcul, glu) des composants et des connecteurs Wright sont décrits par des processus CSP de Hoare. Dans la suite, nous décrivons les aspects syntaxiques et sémantiques du langage formel CSP.

1.3 . CSP : Communicating Sequential Processes

1.3.1 . Les événements

Dans le modèle CSP, tout est représenté par des événements. Un événement correspond à un moment ou une action qui présente un intérêt. Pour les caractériser, nous leur choisissons un nom qui doit être unique pour pouvoir les identifier et les utiliser au moment de la spécification. Dans le cas d'un composant qui ne fait qu'envoyer des messages jusqu'à un message d'arrêt, nous avons les deux événements suivant : write et end-of-data. Comme nous le verrons plus tard, nous avons besoin de faire la différence entre un événement initialisé et un événement observé. Grâce à cette distinction, il nous est possible de mieux contrôler l'interaction en sachant quel composant a initialisé l'événement de ceux qui l'observent. Comme CSP ne fait pas cette distinction, CSP pour Wright le fait :

- Un événement qui est initialisé possède en plus une barre au dessus de lui. Par exemple, le port send utilisera l'évènement $\overline{\text{request}}$ pour indiquer que c'est lui qui l'initialise (cf. figure 1.7).
- Un événement observé sera noté comme d'habitude. Le rôle Serveur utilise l'évènement request pour indiquer qu'il observe cet évènement.
- Une dernière propriété importante des évènements est qu'il est possible de décrire des évènements qui transmettent des données et d'autres qui reçoivent des données :
    - Un évènement peut fournir une donnée. Par exemple, le port send pourra spécifier le fait qu'il veut envoyer la donnée x par l'évènement $\overline{\text{request!x}}$.
    - Un évènement peut recevoir des données. Comme par exemple pour le rôle Serveur qui veut recevoir une donnée x utilisera l'évènement request?x.

CSP définit un événement spécial √ qui indique la bonne terminaison de l'exécution.

Style ClientServeur
Component Client
Port send = $\overline{\text{request!x}}$ → resultat?y → send ∏ §
Computation = $\overline{\text{request!x}}$ → resultat?y → **Computation** ∏ §
Component Serveur
Port receive = request?x → $\overline{\text{resultat!y}}$ → receive □§
Computation = request?x → $\overline{\text{resultat!y}}$ → **Computation** □§
Connector ClientServeur
Role Client= $\overline{\text{request!x}}$ → resultat?y → Client ∏ §
Role Serveur= request?x → $\overline{\text{resultat!y}}$ → Serveur □§
Glue = Client.request?x → $\overline{\text{Serveur.request!x}}$ **→ Glue**

□ Serveur.resultat?y → $\overline{\text{Client.resultat!y}}$ → **Glue** □ §

Constraints

$\exists! s \in component, \forall c \in component : TypeServeur(s) \wedge TypeClient(c) \Rightarrow connected(c$

End Style

Figure 1.7. *Spécification formelle du style ClientServeur*

1.3.2 . Les processus

Pour définir un comportement, il faut pouvoir combiner les événements. Un processus correspond à la modélisation du comportement d'un objet par une combinaison d'événements et d'autre processus simples. L'ensemble des événements que le processus P peut exécuter est appelé son alphabet (ou interface) et est dénoté αP. Le comportement le plus simple d'un processus est de ne rien faire : un tel processus est dénoté par STOP.

Pour décrire des comportements plus élaborés, CSP offre les opérateurs donnés ci-dessous.

- **Préfixe**

Le préfixe noté → permet de définir un processus en explicitant les événements qu'il peut exécuter. Si a est un événement et P un processus, alors :

a → P est le processus qui peut exécuter a et se comporte ensuite comme le processus P. L'opérateur de préfixe est toujours utilisé sous cette forme avec un événement à la gauche de → et un processus à droite. Il est possible d'enchaîner les préfixes. Par exemple, a → b → Q correspond à l'expression de processus a → (b → Q) en appliquant l'associativité à droite. Le processus succès § est défini comme suit : √ → STOP.

- **Récursivité**

Les définitions récursives permettent de décrire en CSP des processus qui agissent sans fin. Par exemple : Alternative = On → Off → Alternative est un processus défini de manière récursive. Il exécute les événements On et Off en alternance. La récursivité croisée (ou mutuelle récursion) est également autorisée en CSP.

- **Opérateurs de choix**

Il existe en CSP trois opérateurs de choix sur les processus.

- Le plus simple, dénoté par |, agit sur les processus de la forme a → P. Le processus a → P | b → Q peut soit exécuter l'événement a et se comporter ensuite comme le processus P, soit exécuter l'événement b et se comporter ensuite comme le processus Q. les préfixes sont nécessairement distincts (a ≠ b). Il est possible d'utiliser | avec plus de deux choix : a → P | b → Q | … | Z → R.

- Le choix déterministe ou externe noté □ : nous le nommons ainsi car c'est l'environnement qui choisit le comportement du processus. Si nous avons le processus : e → P □ f → Q et que l'environnement s'engage dans l'événement f alors le processus s'engagera dans cet événement et se comportera comme le processus Q. Ce choix est typiquement utilisé entre des événements observés.
- Le choix non déterministe ou interne noté ∏. A l'inverse du choix déterministe, c'est le processus qui choisit de façon non déterministe le comportement à choisir parmi plusieurs. Le processus : e → P ∏ f → Q va choisir entre initialiser

l'événement e et continuer comme P ou initialiser l'événement f et continuer comme Q. Il décide lui-même de ce choix sans se préoccuper de l'environnement. Ce choix est typiquement utilisé entre des événements initialisés.

- **Evénements cachés**

Il est possible à partir d'un processus P de cacher certains événements de son alphabet. Si A est un ensemble d'événements et si P est un processus, alors : P\A est le processus dont les événements appartenant à A deviennent des événements internes du processus.

- **Composition parallèle**

Pour décrire plusieurs processus en concurrence, CSP introduit l'opérateur de composition parallèle entre processus.

P || Q est la composition parallèle de P et de Q. Dans ce cas, tous les événements de P et Q sont exécutés en synchronisation. Par exemple, supposons que les processus P et Q sont définis par :

P = (e → f → P) □ (g → P)

Q = e → (f→Q □ g → Q)

Dans ce cas le processus R = P || Q a le comportement suivant :

- Comme P ne peut s'engager que dans e ou g et Q que dans e, R s'engagera dans e.
- A ce moment P ne peut que continuer dans f alors que Q peut continuer dans f ou g. D'où, R s'engagera dans f.
- A ce point les deux processus ne peuvent revenir qu'à leur état initial et donc la séquence <e,f> se répétera indéfiniment.

Nous pouvons donc décrire le comportement de R ainsi : R = e → f → R.

- **Entrées/sorties**

Un événement CSP peut être représenté par un message passant par un canal. Les événements en CSP sont donc de la forme c.m, où c est le nom d'un canal et m est la valeur du message passant par le canal c.

Les valeurs des messages peuvent être exprimées en CSP en termes d'entrées ou de sorties des canaux dans les processus avec préfixe. Le processus :

c!x → P est un processus dont la valeur x est une sortie du canal c et qui se comporte ensuite comme le processus P.

De même,

c?x → P est un processus dont le paramètre x est entrée du canal c et qui se comporte ensuite comme le processus P.

- **Entrelacement**

L'opérateur d'entrelacement (|||) est similaire à l'opérateur de composition parallèle, mais sans synchronisation. On note : P ||| Q l'entrelacement des processus P et Q. Les événements de P sont alors exécutés indépendamment de l'exécution des événements de Q. Par exemple, si P et Q sont définis par :

P = a → b → P'

Q = c → Q'

Les séquences d'événements possibles pour P ||| Q sont : (a, b, c), (a, c, b) et (c, a, b). Ces séquences sont appelées les traces du processus P ||| Q.

- **Quantification**

Les opérateurs de choix interne et externe, de composition parallèle et d'entrelacement ont une forme plus générale pour considérer des combinaisons d'un nombre quelconque de processus. Si I est un ensemble fini d'indices, alors $\prod_{i \in I} P_i$ définit le choix interne entre les processus P_i , avec $i \in I$. De même, $\Box_{i \in I} P_i$ est le choix externe entre tous les processus P_i , avec $i \in I$. Si A_i est l'interface de P_i , pour chaque $i \in I$, alors $||_{i \in I}^{A_i} P_i$ est la composition parallèle de tous les P_i. L'entrelacement de plusieurs processus P_i, avec $i \in I$, est dénoté par : $|| |_{i \in I}^{P_i}$

1.3.3. Sémantique de Wright

Wright étant entièrement basé sur CSP au niveau de la spécification des interactions, sa sémantique l'est aussi. Il faut donc comprendre en premier lieu la sémantique de CSP.

1.3.3.1. Modélisation mathématique des processus CSP

Un processus CSP est un triplet (A, F, D) où A représente l'alphabet du processus, F représente ses échecs et D représente ses divergences.

- **Alphabet**

L'alphabet d'un processus représente l'ensemble des événements sur lequel le processus a une influence. De ce fait, si un événement n'est pas dans l'alphabet d'un processus, alors ce processus ne le connaît pas, ne le traite pas et donc l'ignore.

- **Echecs**

Les échecs d'un processus sont une paire de traces et de refus. Une trace est une séquence d'événements permise par le processus. L'ensemble des traces possibles d'un processus P est noté traces (P). Un refus correspond à un ensemble d'événements proposés pour lequel le processus refuse de s'engager. Cet ensemble d'événements refusés par un processus P est noté refus (P). Cette notion de refus permet une distinction formelle entre processus déterministes et non déterministes. En effet, un processus déterministe ne peut jamais refuser un événement qu'il peut entamer alors qu'un processus non déterministe le peut.

Exemple

Le processus $Q = a \rightarrow Q \prod b \rightarrow Q$ a comme échecs ($\langle\rangle$, {a} et ($\langle\rangle$, {b}) mais pas ($\langle\rangle$, {a, b}). Par contre le processus $R = a \rightarrow R \prod b \rightarrow R \prod STOP$ possède aussi l'échec ($\langle\rangle$, {a, b}).

- **Divergences**

Une divergence d'un processus est définie comme une de ses traces quelconque après laquelle il y a un comportement chaotique. Ce comportement chaotique est représenté par le processus CHAOS :

$CHAOS_A = STOP \prod (\forall x : A \bullet x \rightarrow CHAOS_A)$

Ce processus peut se comporter comme n'importe quel autre. C'est le plus non déterministe, le plus imprévisible, le plus incontrôlable de tous. La divergence est donc utilisée pour représenter des situations catastrophiques ou des programmes complètement imprédictibles (comme des boucles infinies).

1.3.3.2. Les modèles sémantiques

Les trois principaux modèles sémantiques (Roscoe, 1997) sont les traces, les échecs stables et les traces-divergences.

Le modèle des traces associe à chaque processus les séquences finies d'événements admises par ce processus. Ce modèle permet donc de représenter les comportements possibles de processus sous forme de traces. Les traces du processus P sont dénotées par traces (P).

Le modèle des échecs stables associe à chaque processus P les couples de la forme (t, E), où t est une trace finie admise par P et E est l'ensemble des événements que le processus ne peut pas exécuter après avoir exécuté les événements de t. l'ensemble de ces couples est noté failures (P). Ce modèle permet de caractériser les blocages de P. En effet, si E est égal à l'ensemble des événements exécutables par P, alors P se retrouve bloqué.

Enfin, le modèle des échecs-divergences associe à chaque processus P l'ensemble de ses échecs stables et l'ensemble de ses divergences. Un processus P n'est divergent que s'il se trouve dans un état dans lequel les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences de P noté divergences (P), est l'ensemble des traces t telles que le processus se retrouve dans un état divergent après avoir exécuté t. Si le processus est déterministe, alors divergences (P) est vide.

1.3.3.3. Le raffinement CSP

Le raffinement consiste à calculer et à comparer les modèles sémantiques de deux processus. Le raffinement dépend donc du modèle considéré. Par exemple, dans le cas du modèle des échecs-divergences, si P et Q sont deux processus, alors Q raffine P, noté

$P \sqsubseteq_{FD} Q$ si : failures (Q) $\subseteq$ failures (P) $\wedge$ divergences (Q) $\subseteq$ divergences (P)

Dans cet exemple, il n'est pas utile de comparer traces (P) et traces (Q), car par définition :

failures (Q) $\subseteq$ failures (P) $\Rightarrow$ traces (Q) $\subseteq$ traces (P)

Intuitivement, Q est égal ou meilleur que P dans le sens où il à moins de risque d'échec et divergence. Q est plus prévisible et plus contrôlable que P, car si Q peut faire quelque chose d'indésirable ou refuser quelque chose, P peut le faire aussi. Concrètement, cela signifie qu'un observateur ne peut pas distinguer si un processus a été substitué à un autre.

1.3.4. Utilisation de la sémantique de CSP dans Wright

La sémantique de Wright est entièrement basée sur CSP. En effet, chaque partie d'une AL Wright est modélisée par un processus CSP. Pour analyser l'interaction de ces processus, il faut les combiner par l'opérateur | | de CSP. Mais un problème émerge : la sémantique de CSP utilise des noms d'événements globaux. En effet, pour décrire que deux processus CSP interagissent, il suffit qu'ils partagent un nom d'événement identique. Tandis que les noms d'événements en Wright agissent comme des noms locaux propres aux composants et que l'interaction a lieu par les connecteurs.

Afin de réaliser cette correspondance d'événements locaux de Wright en événement global de CSP, Wright identifie et résout systématiquement les deux problèmes suivants :

1. Au niveau des instances : il peut exister plusieurs instances d'un même type. Ainsi des interactions indésirées peuvent avoir lieu en introduisant de multiples copies d'un même processus. Pour résoudre ce premier problème, il suffit de préfixer chaque nom d'événement par le nom de son instance. Ainsi un événement a :

- 3 niveaux : N.P.e (nom du composant, nom du port, nom de l'événement), si le Calcul (Computation) utilise un événement du port P,
- 2 niveaux : N.e (nom du composant, nom de l'événement), si le Calcul (Computation) utilise un événement interne (non associé à un port).

2. Au niveau des liens : comme les noms d'événement utilisés dans les composants et connecteurs sont locaux, pour assurer une synchronisation entre un composant et un connecteur (indiqué par un lien) il faut pouvoir changer les noms d'événements afin de respecter le fait qu'en CSP deux processus ne communiquent que s'ils partagent un même événement. Pour résoudre ce deuxième problème, il suffit de renommer des événements du connecteur par les noms des événements des composants correspondants : si nous avons un nom d'événement pour un connecteur Conn.Role.e et que le rôle de ce connecteur est lié à un port d'un composant Comp.Port, alors nous voulons que le nom de l'événement du connecteur Conn.Role.e soit renommé Comp.Port.e.

1.4 Etude de cas

1.4.1. Présentation du problème

Cette application est initialement traitée en CSP dans (Martin, 1997b). Un fermier emploie n contremaîtres. Chaque contremaître est responsable de m ouvriers. Le fermier, les contremaîtres et les ouvriers communiquent via des canaux. Un ouvrier i demande un travail à faire (une nouvelle tâche) à son contremaître j en utilisant le canal aij. Le contremaître j met au courant le fermier de la demande de l'ouvrier en utilisant le canal cj. Le fermier répond à cette demande en utilisant le canal dj. Le contremaître attribue une nouvelle tâche à l'ouvrier i en utilisant le canal bij.

Dans la suite, nous allons nous intéresser à une instance de ce problème avec n = 3 et m = 3. Le réseau d'interactions entre les différents acteurs de cette ferme est illustré par la figure 1.11.

1.4.2. Modélisation en Wright

Nous avons choisi de définir notre architecture à base de trois types de composants et de deux types de connecteurs (cf. figure 1.8).

1.4.2.1. Présentation semi-formelle des types de composants et des types de connecteurs

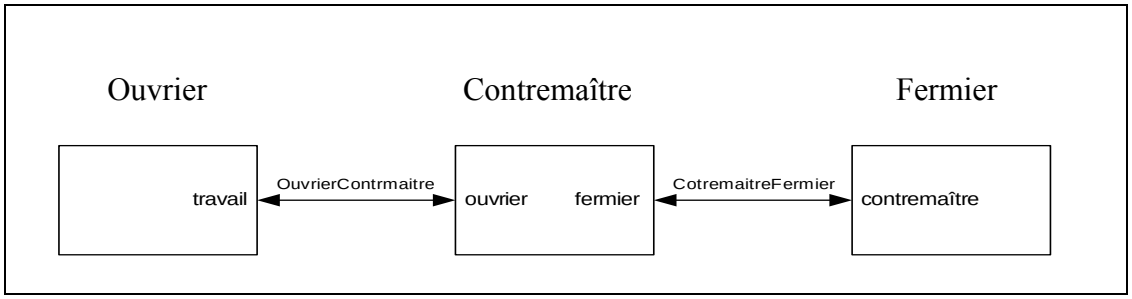


Figure 1.8. *Une architecture logicielle informelle de la ferme*

1.4.2.2. Les types de composants

Component Ouvrier
Port travail = envoyer une requête (demande tâche) et attend un résultat
Computation = envoyer la requête sur le port travail et attend un résultat

Figure 1.9. *Le composant Ouvrier*

Le composant Ouvrier possède un seul port que nous appelons travail (cf. figure 1.9).

Le composant Contremaître possède deux ports (cf. figure 1.10).

Component Contremaître
Port ouvrier = recevoir la demande de tâche et envoyer une réponse
Port fermier = envoyer la demande tâche et attendre une réponse
Computation = lire la demande du port ouvrier, l'envoyer sur le port fermier, dès qu'une réponse reçue par le port fermier alors cette réponse est envoyée sur le port ouvrier.

Figure 1.10. *Le composant Contremaître*

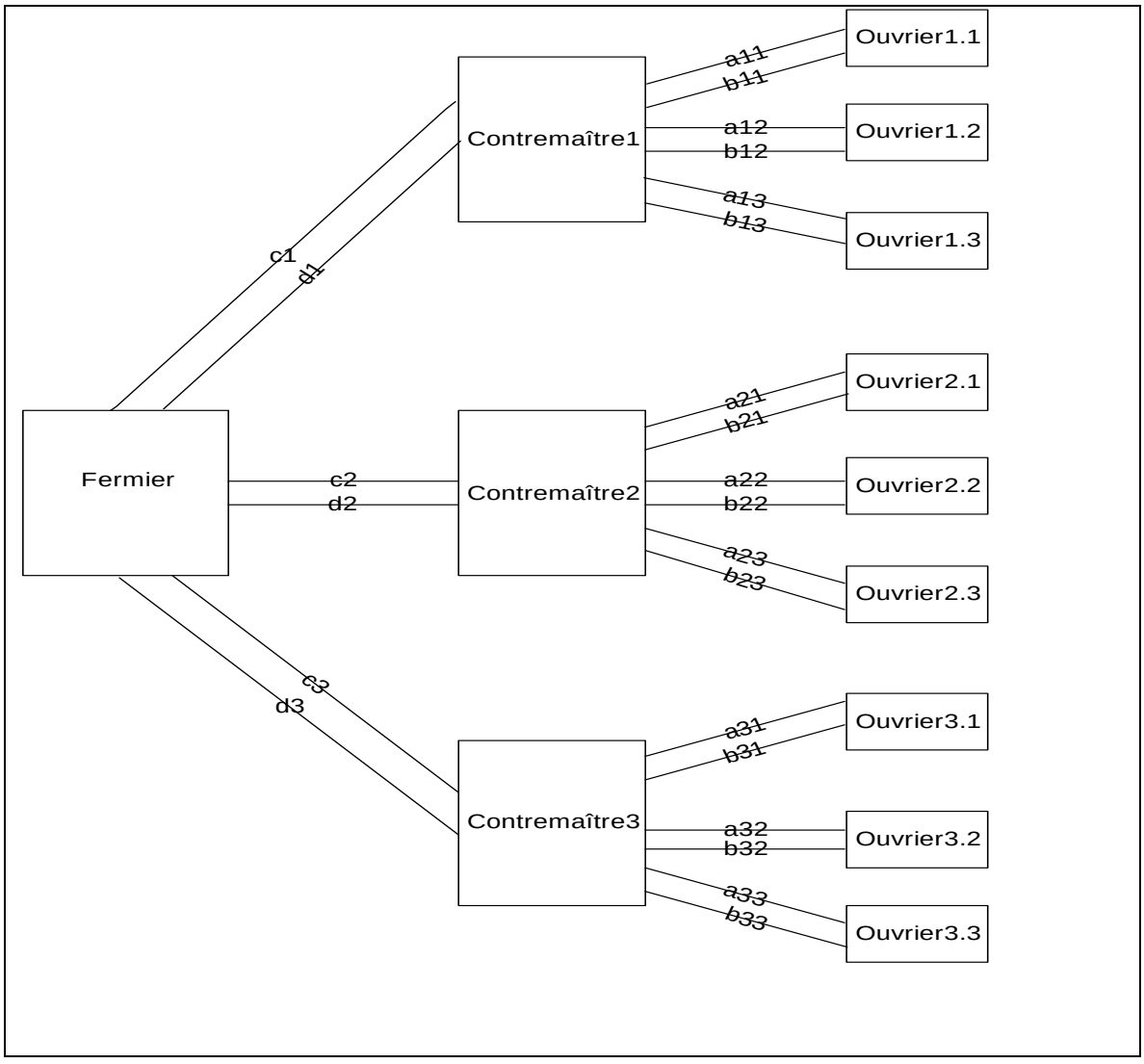


Figure 1.11. *Une architecture logicielle informelle de la ferme*

Component Fermier

Port contremaître = recevoir une requête et retourner un résultat

Computation = lire la requête du port contremaître et faire retourner sur le même port la réponse

Figure 1.12. *Le composant Fermier*

Le dernier composant est appelé Fermier. La figure 1.12 donne la présentation semi-formelle de ce composant.

1.4.2.3. Les types de connecteurs

Pour cette étude de l'architecture de la ferme, nous avons besoin de deux types de connecteurs. Le premier permet la communication entre un ouvrier et son contremaître et le second pour la communication entre un contremaître et le fermier comme le montre la figure 1.8.

Pour distinguer entre les deux types de connecteurs, nous avons utilisé des notations significatives comme OuvrierContremaître et ContremaîtreFermier.

Connector OuvrierContremaître

Role Ouvrier = envoyer une requête (demande tâche) et attendre un résultat

Role Contremaître = recevoir la demande de tâche et envoyer une réponse

Glue = le port Contremaître va recevoir la requête envoyée par le port ouvrier et de même le port ouvrier va recevoir la réponse envoyée par le port contremaître.

Figure 1.13. *Le connecteur OuvrierContremaître*

Les deux figures 1.13 et 1.14 montrent la présentation semi-formelle de ces deux types de connecteurs.

Connector ContremaîtreFermier
Role Contremaître = envoyer une requête (demande tâche) et attendre un résultat
Role fermier = recevoir la demande de tâche et envoyer réponse
Glue = le port fermier va recevoir la requête envoyée par le port Contremaître et de même le port Contremaître va recevoir la réponse envoyée par le port fermier.

Figure 1.14. *Le connecteur ContremaîtreFermier*

1.4.3. Présentation formelle pour la configuration

Dans cette partie, nous allons utiliser les processus CSP pour spécifier d'une façon formelle les comportements des ports, des rôles, de la partie calcul d'un composant et de la partie glu d'un connecteur.

La solution est donnée dans la figure 1.15.

Configuration Ferme
Component Ouvrier
Port travail = $\overline{a!x}$ -> b?x -> travail
Computation = $\overline{travail.a!x}$ -> travail.b?x -> **Computation**
Component Contremaitre
Port ouvrier1 = a?x -> $\overline{b!x}$ -> ouvrier1
Port ouvrier2 = a?x -> $\overline{b!x}$ -> ouvrier2
Port ouvrier3 = a?x -> $\overline{b!x}$ -> ouvrier3
Port fermier = $\overline{c!x}$ -> d?x -> fermier
Computation =
ouvrier1.a?x -> $\overline{fermier.c!x}$ -> fermier.d?x -> $\overline{ouvrier1.b!x}$ -> **Computation** □
Ouvrier2.a?x -> $\overline{fermier.c!x}$ -> fermier.d?x -> $\overline{ouvrier2.b!x}$ -> **Computation** □
Ouvrier3.a?x -> $\overline{fermier.c!x}$ -> fermier.d?x -> $\overline{ouvrier3.b!x}$ -> **Computation**
Component Fermier
Port contremaitre1 = c?x -> $\overline{d!x}$ -> contremaitre1
Port contremaitre2 = c?x -> $\overline{d!x}$ -> contremaitre2
Port contremaitre3 = c?x -> $\overline{d!x}$ -> contremaitre3
Computation =
contremaitre1.c?x -> $\overline{contremaitre1.d!x}$ -> **Computation** □
contremaitre2.c?x -> $\overline{contremaitre2.d!x}$ -> **Computation** □
contremaitre3.c?x -> $\overline{contremaitre3.d!x}$ -> **Computation**
Connector OuvrierContremaitre
Role ouvrier = $\overline{a!x}$ -> b?x -> ouvrier
Role contremaitre = a?x -> $\overline{b!x}$ -> contremaitre
Glue = ouvrier.a?x-> $\overline{contremaitre.a!x}$ ->contremaitre.b?x-> $\overline{ouvrier.b!x}$ -> **Glue**
Connector ContremaitreFermier
Role contremaitre = $\overline{c!x}$ -> d?x -> contremaitre
Role fermier = c?x -> $\overline{d!x}$ -> fermier
Glue = contremaitre.a?x-> $\overline{fermier.a!x}$ -> fermier.b?x -> $\overline{contremaitre.b!x}$ -> **Glue**
Instances
O11 , O12 , O13 , O21 , O22 , O23 , O31 , O32 , O33 : ouvrier

C1 , C2 , C3 : Contremaitre
F : Fermier
O11ToC1 , O12ToC1 , O13ToC1 , O21ToC2 , O22ToC2 , O23ToC2 , O31ToC3 , O32ToC3 , O33ToC3 : OuvrierContremaitre
C1ToF , C2ToF , C3ToF : ContremaitreFermier
Attachments
O11.travail as O11ToC1.ouvrier
O12.travail as O12ToC1.ouvrier
O13.travail as O13ToC1.ouvrier
O21.travail as O21ToC2.ouvrier
O22.travail as O22ToC2.ouvrier
O23.travail as O23ToC2.ouvrier
O31.travail as O31ToC3.ouvrier
O32.travail as O32ToC3.ouvrier
O33.travail as O33ToC3.ouvrier
C1.ouvrier1 as O11ToC1.contremaitre

```
C1.ouvrier2   as   O12ToC1.contremaitre
C1.ouvrier3   as   O13ToC1.contremaitre
C2.ouvrier1   as   O21ToC2.contremaitre
C2.ouvrier2   as   O22ToC2.contremaitre
C2.ouvrier3   as   O23ToC2.contremaitre
C3.ouvrier1   as   O31ToC3.contremaitre
C3.ouvrier2   as   O32ToC3.contremaitre
C3.ouvrier3   as   O33ToC3.contremaitre
C1.fermier    as   C1ToF.contremaitre
C2.fermier    as   C2ToF.contremaitre
C3.fermier    as   C3ToF.contremaitre
F.contremaitre1 as  C1ToF.fermier
F.contremaitre2 as  C2ToF.fermier
F.contremaitre3 as  C3ToF.fermier
End Configuration
```

Figure 1.15. *Spécification formelle de la ferme*

Les attachements peuvent être représentés par un arbre pour diminuer le risque d'erreur. Les noeuds ici représentent les différentes instances des ouvriers, des contremaîtres et du fermier. Par contre, les instances de connecteurs sont représentées par les arêtes. Chaque arête correspond à une instance d'un connecteur (cf. figure 1.16).

1.5 . Conclusion

Dans ce chapitre, nous avons étudié les aspects structuraux et sémantiques de Wright. Les concepts structuraux de Wright sont : composant, connecteur, configuration et style. La sémantique de Wright est entièrement basée sur CSP moyennant des correspondances entre les événements locaux de Wright et les événements globaux de CSP. En outre, nous avons proposé une modélisation progressive (passage du semi-formel vers le formel, identification des problèmes liés aux attachements entre les instances de composants et des connecteurs) d'une application en Wright. Le chapitre suivant sera consacré à l'approche de vérification d'architectures logicielles en Wright : propriétés à vérifier, outils de vérification et limites de cette approche.

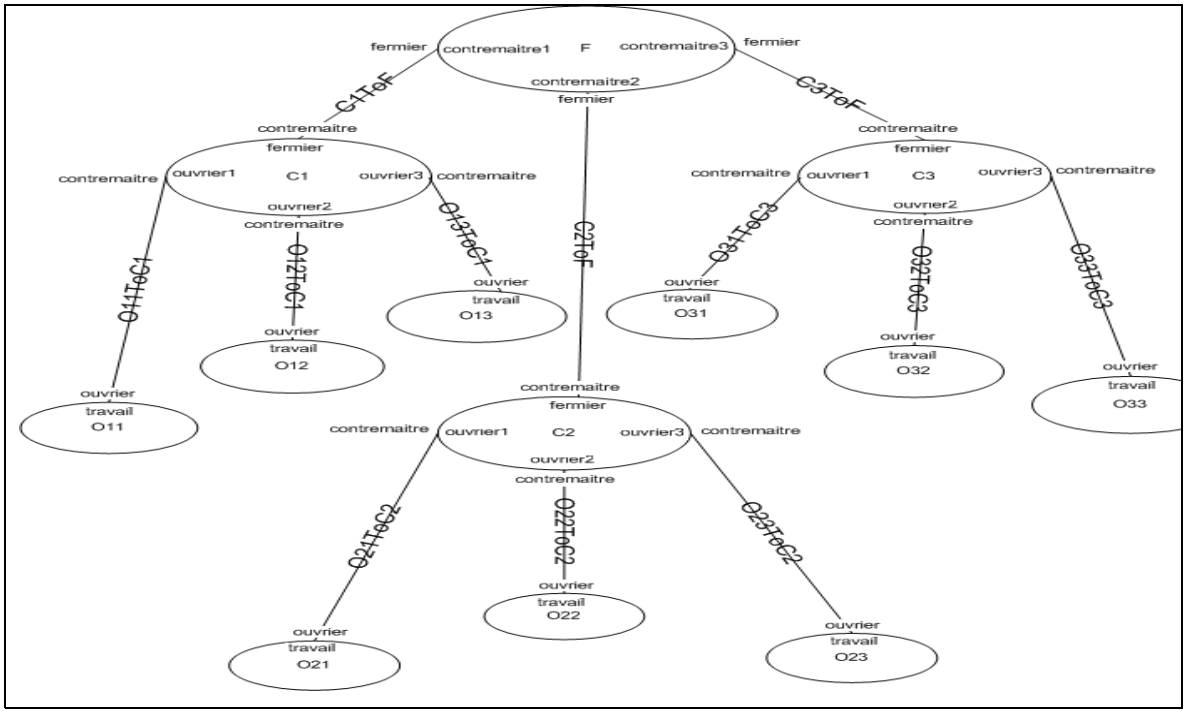


Figure 1.16. *Représentation de différentes instances de la ferme*

Chapitre 2 : Vérification d'architectures logicielles Wright

2.1 . Introduction

Dans le chapitre 1, nous avons montré comment Wright peut être utilisé pour exprimer la structure et le comportement d'une architecture logicielle ou d'un style d'architecture. Mais ces descriptions ne sont réellement intéressantes que si elles nous apportent suffisamment d'informations pour pouvoir analyser et fournir des propriétés sur l'architecture logicielle en question.

Wright permet l'étude de deux critères fondamentaux sur une architecture logicielle qui sont la **cohérence** et la **complétude**.

Informellement, la **cohérence** correspond au fait que la description a un sens, c'est-à-dire qu'aucune partie n'est contredite par une autre.

La **complétude** permet de vérifier que la description contient suffisamment d'informations pour prétendre à une analyse, c'est-à-dire que la description n'a pas omis de détails pour réaliser une analyse.

Il est important de vérifier que le système résultant de sa description est cohérent car s'il ne l'est pas, **tout raffinement ou implémentation restera incohérent**. La complétude est tout aussi importante car une analyse ne peut être basée que sur ce que nous connaissons du système. Si nous analysons la communication d'un composant, mais qu'une partie de l'interface est laissée de côté, comment est-il possible de faire une bonne analyse? Par exemple, une analyse sur une ressource partagée ne peut être faite que si tous les participants qui accèdent à cette ressource sont connus.

Le problème de la complétude est spécialement critique pour un architecte à cause de l'importance de son abstraction à ce niveau de conception. Il y a toujours une tension entre le besoin d'inclure des informations critiques nécessaires pour garantir des propriétés importantes du système et le risque de mettre du désordre dans l'architecture en mettant des contraintes et des détails qui peuvent rendre l'architecture difficile à manier et à exploiter.

Ce chapitre comporte trois sections. La première présente d'une façon informelle les propriétés architecturales standards définies par Wright. La deuxième aborde les techniques potentielles de vérification de ces propriétés. Enfin, la troisième section établit un bilan exhibant les points forts et faibles du langage Wright.

2.2 . Description informelle des propriétés Wright

Nous allons maintenant expliciter informellement l'ensemble des propriétés[1] intégrées dans Wright sur la cohérence et la complétude.

2.2.1 . Cohérence

[1] Le terme utilisé par les auteurs de Wright est test.

La cohérence consiste à vérifier que tous les éléments décrivant l'architecture logicielle (les composants, les connecteurs et la configuration) sont cohérents.

2.2.1.1. Cohérence d'un composant

Nous avons vu dans la description d'un composant qu'un composant est constitué par deux parties la partie interface décrivant les ports et la partie calcul, ainsi, nous vérifions la cohérence d'un composant en nous assurant que le calcul obéit aux règles d'interaction définies par les ports.

Le premier aspect décrit par le port correspond au comportement attendu du composant. Il faut s'assurer de la cohérence entre le comportement des ports et celui du calcul (Computation) par la notion de projection.

Un port est une projection d'un composant si ce dernier agit de la même manière que le port quand nous ignorons tous les événements n'appartenant pas à l'alphabet de ce port.

Illustrons cette notion de projection en prenant le composant suivant :

Component Double
Port Input = read?x-> Input □ close -> §
Port Output = write!x-> Output ∏ close -> §
Computation = Input.read?x -> Output. write!(2*x) -> **Computation** □ Input.close -> Output.close -> §

Si nous ignorons tous les événements qui n'appartiennent pas à l'alphabet du port Input, nous obtenons :

Computation = Input.read?x -> **Computation** □ Input.close -> §

Le port Input est bien une projection du Calcul. De la même manière, il est facile de montrer que le port Output est une projection du Calcul.

Le second aspect du port correspond à l'interaction avec l'environnement. Si la spécification des ports ne tient pas compte d'un événement, alors le composant (partie Calcul) n'a pas à s'en occuper. Mais d'un autre côté, il y a des situations dans lesquelles il peut être approprié d'avoir des spécifications d'un composant qui décrivent des comportements qui n'auront pas forcément lieu. Ceci est notamment le cas lors **de la réutilisation d'une spécification** d'un Calcul plus générale que nos besoins. Pour toutes ces raisons, la spécification des ports peut ne couvrir qu'un sous-ensemble des situations que le composant peut effectivement gérer. Illustrons nos propos par l'exemple du composant Double qui décrit un comportement d'échec indiqué par l'événement fail si nous souhaitons utiliser ce composant sans se préoccuper de cet événement, nous le spécifions comme suit :

Component Double
Port Input = read?x-> Input □ close -> §
Port Output = write!x-> Output ∏ close -> §
Computation = Input.read?x -> Output. write!(2*x) -> **Computation** □ Input.close -> Output.close -> □ Input.fail -> §

Si nous ignorons l'hypothèse du port Input qui est que l'événement fail ne va pas avoir lieu et nous projetons le port Output, nous obtenons :

Computation = $\overline{\text{Output. write!(2*x)}}$ -> **Computation** □ $\overline{\text{Output.close}}$ -> **§** □ **§**

Le port Output n'est plus une projection du Calcul. Ainsi la propriété permettant de vérifier la cohérence d'un composant Wright est formulée comme suit :

Propriété 1 : Cohérence Port / Calcul

La spécification d'un port doit être une projection du Calcul, sous l'hypothèse que l'environnement obéisse à la spécification de tous les autres ports

Intuitivement, la propriété 1 stipule que le composant ne se préoccupe pas des événements non traités par les ports (ici événement fail).

2.2.1.2. Cohérence d'un connecteur

La description du connecteur doit vérifier que la coordination des rôles par la Glu est cohérente avec le comportement attendu des composants. Prenons comme exemple le connecteur suivant :

```
Connector Bogus
Role User1 = set -> User1 ∏ get -> User1 ∏ §
Role User2 = set -> User2 ∏ get -> User2 ∏ §
Glue = User1. set-> Continue □ User2. set-> Continue □ §
where {
        Continue = User1. set-> Continue □
                   User2. set-> Continue □
                   User1. get-> Continue □
                   User2. get-> Continue □ §
        }
```

La spécification du connecteur Bogus semble raisonnable, la Glu (Glue en anglais) du connecteur exige qu'un de ses deux participants initialise l'événement set mais n'indique pas lequel, si chacun commence par set, alors que l'événement se produira d'abord et la communication peut continuer sans aucun problème, si cependant chaque participant essaye légalement d'exécuter initialement l'événement get alors le connecteur aboutira à un interblocage. Sachant qu'un processus CSP est dit en situation d'interblocage quant il peut refuser de participer à tout événement, mais n'a pas pour autant terminé correctement (en participant à l'événement §). Inversement, un processus est sans interblocage s'il ne peut jamais être en situation d'interblocage. Ainsi, Wright propose la propriété :

Propriété 2 : Connecteur sans interblocage

La glu d'un connecteur interagissant avec les rôles doit être sans interblocage.

Une autre catégorie d'incohérence est détectable comme une situation d'interblocage, lorsque la spécification d'un rôle est elle-même incohérente. Dans une spécification d'un rôle complexe, il peut y avoir des erreurs qui mènent à une situation dans laquelle aucun événement n'est possible pour ce participant, même si la Glu était prête à prendre tout événement.

Propriété 3 : Rôle sans interblocage

Chaque rôle d'un connecteur doit être sans interblocage.

Pour empêcher le conflit de contrôle, un événement ne doit être initialisé que par un unique processus, tous les autres processus ne faisant que l'observer.

Propriété 4 : Un initialiseur unique

> *Dans une spécification de connecteur, tout événement ne doit être initialisé que par un rôle ou la glu. Tous les autres processus doivent soit l'observer, soit l'oublier (grâce à leur alphabet).*

La dernière propriété pour les connecteurs vérifie que les notations pour l'initialisation et l'observation des événements sont utilisées correctement. Ceci est illustré par l'exemple du port Output dont la spécification est la suivante :

Port Output = $\overline{\text{write!(w,i)}}$ -> Output □ $\overline{\text{close}}$ -> Output

Le port Output envoie des couples (mot, numéro de ligne) jusqu'à ce qu'il n'y en ait plus, ensuite il ferme son port et s'arrête.

Cette spécification n'est pas cohérente du fait que c'est le composant qui décide s'il lui reste des couples à envoyer et non pas l'environnement.

Propriété 5 : Engagement de l'initialiseur

> *Si un processus initialise un événement alors il doit s'engager dans cet événement sans être influencé par l'environnement.*

2.2.1.3. Cohérence d'une configuration

Au niveau de la déclaration d'instances, la cohérence s'applique aux deux points suivants :

- Le nom de l'instance est-il unique?
- Des paramètres raisonnables ont-ils été donnés?

Dans l'exemple ci-dessous nous avons paramétré le nombre d'instances du port Output du composant Filtre_Texte.

Component Filtre_Texte (nout : 1 ..)
Port Input = DataInput
Port Output $_{1..\ nout}$ = DataOutput
Computation = lire des données du port Input. Envoyer ces données successivement sur les ports Output $_1$, Output $_2$,… , Output $_{nout}$

Le nombre d'instances du port Output sera déterminé au moment de l'instanciation du filtre.

Propriété 6 : Substitution des paramètres

> *Une déclaration d'instance paramétrant un type doit résulter d'une validation de ce type après avoir substitué tous les paramètres formels manquant.*

Dans le cas de paramètre numérique, il faut s'assurer que les paramètres entrent dans les bonnes données dans la description du type.

Propriété 7 : Test des valeurs sur leur intervalle donné

> *Un paramètre numérique ne doit pas être plus petit que la limite inférieure (si elle est déclarée), et pas plus grande que la limite supérieure (si elle est déclarée).*

Au niveau des liens la question suivante se pose :

Quels ports peuvent être utilisés pour ce rôle?

La vérification sur le fait que les protocoles du port et du rôle soient identiques n'est pas suffisante. En effet, nous voulons avoir la possibilité d'attacher un port qui n'a pas un protocole identique au rôle.

Considérons le rôle suivant :

Role Source = $\overline{\text{write}}$!x ->Source □ $\overline{\text{close}}$ -> §

Le rôle Source peut être attaché au port suivant :

Port Output3 = $\overline{\text{write}}$!1 -> $\overline{\text{write}}$!2 -> $\overline{\text{write}}$!3 -> $\overline{\text{close}}$ -> §

Le rôle Source et le port Output3 ne sont pas identiques. Le rôle Source qui émet des suites de x a une description plus générale que le port Output3 qui émet la suite 1 2 3.

D'autre part il faut toujours vérifier qu'il n'existe pas une incompatibilité entre le rôle et le port qui lui est attaché. Par exemple, nous ne voulons pas accepter le fait qu'un port comme BadOutput (sans l'événement close) puisse être attaché au rôle Source.

Port BadOutput = $\overline{\text{write}}$!x -> BadOutput ∏ §

Ainsi nous avons la propriété suivante :

Propriété 8 : Compatibilité port / rôle

Tout port attaché à un rôle doit toujours continuer son protocole dans une direction que le rôle peut avoir.

Les deux propriétés suivantes concernent le concept de Style d'architecture. Une configuration d'un système est cohérente avec ses styles déclarés si elle obéit à chacune de leurs contraintes.

Propriété 9 : Contraintes de Style

Les prédicats d'un style doivent être vrais pour une configuration déclarée être dans ce style.

Les contraintes d'un Style doivent être cohérentes entre elles.

- **Exemple**

  ∀ c : Components ; p : Ports (c) • Type (p) = DataOutput

  ∃ c : Components ; p : Ports (c) • Type (p) = DataOutput

Ces deux contraintes sont en contradiction, donc le Style contenant ces deux contraintes est incohérent.

Propriété 10 : Cohérence de Style

Au moins une configuration doit satisfaire les contraintes de style.

2.2.2 . Complétude

Une catégorie de complétude importante que vérifie Wright concerne la configuration :

- Au niveau des liens, si un lien est omis alors un composant va dépendre des événements qui ne vont jamais avoir lieu, ou une interaction va échouer car il manque un participant,
- D'autre part, il existe des ports de composants qui n'ont pas besoin d'être attachés et il y a des interactions qui peuvent continuer même si un participant manque.

Pour ces deux raisons, il n'est pas suffisant de contrôler que tous les ports et rôles soient bien attachés.

Propriété 11 : la complétude des liens

> *Chaque port (respectivement rôle) non attaché dans la configuration doit être compatible avec le rôle (respectivement port) §*

La figure 1.17 donne la liste complète des propriétés effectuées par Wright sur la cohérence et la complétude.

1. **Cohérence des ports avec le Calcul (composant)**
2. **Absence d'interblocage sur les connecteurs (connecteur)**
3. **Absence d'interblocage sur les rôles (rôle)**
4. **Initialiseur unique (connecteur)**
5. **Engagement de l'initialiseur (n'importe quel processus)**
6. **Substitution de paramètres (instance)**
7. **Bornes d'un intervalle (instance)**
8. **Compatibilité port / rôle (lien)**
9. **Contraintes pour les styles (configuration)**
10. **Cohérence de style (style)**
11. **Complétude des liens (configuration)**

Figure 1.17. *Propriétés induites par Wright*

2.3 . Techniques de vérification des propriétés Wright

Dans ce paragraphe, nous nous interrogeons sur les techniques potentielles permettant de prouver les propriétés Wright présentées précédemment.

2.3.1 . Utilisation du raffinement CSP

Le raffinement CSP (cf. 1.3.3.3) permet le développement incrémental des systèmes CSP. Un processus CSP peut être raffiné progressivement jusqu'à son implémentation (raffinement ultime). Par exemple, si une composition parallèle de deux processus P et Q raffine une spécification abstraite décrite par le processus S, alors nous écrivons :

$$S \sqsubseteq P \,||\, Q.$$

Ensuite, nous pouvons développer la spécification S en raffinant d'une façon séparée P et Q : si $P \sqsubseteq P'$ et $Q \sqsubseteq Q'$, alors la composition de P' et Q' raffine aussi S : $S \sqsubseteq P' \,||\, Q'$. Egalement le raffinement CSP peut être utilisé pour vérifier des propriétés de sûreté ou de vivacité. En effet, des propriétés Wright sont formalisées grâce au raffinement CSP. Ces propriétés sont les suivantes (Allen, 1997a) :

- Propriété 1 : Cohérence des ports avec le Calcul.
- Propriété 2 : Absence d'interblocage sur les connecteurs.
- Propriété 3 : Absence d'interblocage sur les rôles.
- Propriété 8 : Compatibilité port/rôle.

2.3.1.1. Formalisation

Afin de formaliser les propriétés Wright en utilisant le raffinement CSP, nous définissons les ensembles suivants :

- αP : l'alphabet du processus P

- $\alpha_i P$: le sous-ensemble de αP correspondant aux événements initialisés
- $\alpha_0 P$: le sous-ensemble de αP correspondant aux événements observés

Propriété 1 : Cohérence Port/Calcul

Comme nous l'avons noté, la spécification d'un port a deux aspects :

- Des exigences sur le comportement du composant (le composant accomplit le comportement décrit par le port).
- Des suppositions sur l'environnement (qu'est-ce que l'environnement, c'est-à-dire les rôles des connecteurs auxquels le composant peut être attaché, va exiger pendant l'interaction).

Ainsi, pour modéliser le processus du Calcul dans l'environnement indiqué par les ports :

1. Nous devons prendre les ports et construire un processus qui est restreint aux événements observés (ce qui extrait les suppositions de l'environnement).

Définition 1

Pour tout processus p = (A, F, D) et un ensemble d'événements E, $P \lceil E = (A \cap E, F', D')$ où $F' = \{(t', r') \mid \exists (t, r) \in F \mid t' = t \lceil E \wedge \forall r' = r \cap E\}$ et $D' = \{t' \mid \exists t \in D \mid t' = t \lceil E\}$.

La projection d'une trace ($t \lceil E$) est une trace qui contient tous les éléments de t qui sont dans E, dans le même ordre, sans tous les éléments qui ne sont pas dans E.

- **Exemple**

$< acadbcabc > \lceil \{a, b\} = < aabab >$

2. Nous devons rendre ce nouveau processus déterministe. Ainsi, nous assurons que les décisions prises dans l'interaction sont faites par le Calcul et non par les ports.

Définition 2

Pour tout processus p = (A, F, D), det(p) = (A, F', ∅) où $F' = \{(t, r) \mid t \in Traces(P) \wedge \forall e : r \bullet t \hat{} < e > \notin Traces(P)\}$.

La fonction det(P) a les mêmes traces que P, mais avec moins de refus. Ainsi, n'importe quel événement qui a lieu à tout point est entièrement contrôlable par l'environnement : det(P) est déterministe.

3. Il ne nous reste plus qu'à faire interagir ce nouveau processus déterministe (det(P)) avec celui du Calcul (C) en les mettant en parallèle : C | | det(P). Nous avons donc au moins les traces de P mais où les décisions sont prises par C.

En utilisant le raffinement, nous pouvons vérifier que le Calcul respecte bien les exigences de ports.

Propriété 1 : Cohérence Port/Calcul

Pour un composant avec un processus de Calcul C et des ports P, P_1, … P_n ; C est cohérent avec P si $P \sqsubseteq (C \mid \mid \forall i : 1 \ldots n \mid \mid det(P_i \lceil \alpha_0 P_i)) \lceil \alpha P$.

Propriété 2 et Propriété 3 : Absence d'interblocage sur les connecteurs et Absence d'interblocage sur les rôles

Ces deux propriétés reviennent à vérifier si un processus est sans interblocage. D'une façon formelle, un processus P = (A, F, D) est sans interblocage si pour toute trace t telle que (t, A) $\in$ F, last (t) =√. Mais ceci peut être exprimé par une relation de raffinement entre le processus DF_A et P $DF_A \sqsubseteq P$ avec DF_A est défini comme suit :

$DF_A = (\prod e : A \bullet e \rightarrow DF_A) \prod \S$.

Le processus DF_A permet toutes les traces possibles sur l'alphabet A mais sans jamais avoir la possibilité de refuser tous les événements : il s'agit d'un processus sans interblocage.

Propriété 8 : compatibilité port / rôle

La distinction entre un port et un rôle est que le port décrit un comportement spécifique alors qu'un rôle décrit un pattern de comportements permettant le lien de plusieurs ports.

Par contre le lien d'un port à un rôle doit toujours respecter les contraintes de spécification de ce rôle. Ainsi, le comportement d'un port attaché à un rôle est le comportement de ce processus port restreint aux traces de ce processus rôle.

Comme la restriction d'une trace est effectuée par la version déterministe d'un processus, nous testons donc le processus P | | det(R) pour exprimer cette restriction au processus rôle. Pour pouvoir utiliser le raffinement dans le test de compatibilité, il faut que les alphabets des deux processus port et rôle soient identiques. Pour cela, nous définissons comment augmenter l'alphabet d'un processus.

Définition 3

Pour tout processus P et un ensemble d'événements A, $P_{+A} = P \,||\, STOP_A$

Propriété 8 : compatibilité

Un port P est compatible avec un rôle R, noté P compat R, si

$R_{+(\alpha P - \alpha R)} \sqsubseteq P_{+(\alpha R - \alpha P)} \,||\, det(R)$

2.3.1.2. Automatisation

Les auteurs de Wright proposent un outil appelé Wr2fdr (Wr2fdr, 2005) permettant d'automatiser les quatre propriétés décrites précédemment. Pour y parvenir, l'outil Wr2fdr traduit une spécification Wright en une spécification CSP dotée des relations de raffinement à vérifier. La spécification CSP engendrée pour l'outil Wr2fdr est soumise à l'outil de Model checking FDR (Failure-Divergence Refinement) (FDR2, 2003). Dans la suite nous présentons successivement FDR, Wr2fdr et un exemple d'utilisation de ces deux outils.

- **FDR**

FDR permet de vérifier de nombreuses propriétés sur des systèmes d'états finis. FDR s'appuie sur la technique de « model checking » (Schnoebelen, 1995). Celle-ci effectue la vérification d'un modèle d'un système par rapport aux propriétés qui sont attendues sur ce modèle. Cette vérification est entièrement automatisée et consiste à explorer tous les cas possibles. Le résultat de cette analyse est soit la confirmation que chaque propriété est maintenue par le modèle, soit qu'elle ne l'est pas. Dans le dernier cas, le « model checker » (outil) renvoie un contre-exemple qui montre comment la propriété n'est pas maintenue. FDR est basé sur la théorie de CSP et précisément sur la sémantique opérationnelle de CSP (Roscoe, 1994) (Roscoe, 1997) : modèle de traces, modèle d'échecs stables et modèle d'échecs/divergences (cf. 1.3.3.2). En FDR, la méthode pour établir qu'une propriété est vérifiée revient à réaliser un raffinement entre deux processus P (processus abstrait) et Q

(processus raffiné) représentés par deux machines d'états finis. Sachant que la propriété à vérifier est traduite par la relation de raffinement entre P et Q. Le problème de FDR[2] réside dans l'exploration de tous les états à considérer pour vérifier la propriété. Très vite les limites de la machine sont atteintes. Cependant sur des espaces d'états raisonnables, FDR permet de vérifier la propriété ou, dans le cas contraire, de donner une trace correspondant à l'ensemble des transitions et des états qui conduisent au contre-exemple. Mais, dans le cas où un contre-exemple ne serait pas trouvé, il est difficile voire impossible de statuer sur la validité ou non de la propriété. Cela dépend de l'espace des états considéré et de la finitude de la machine d'états représentant le processus CSP. Ce dernier problème est non décidable. La dernière version de FDR2 améliore le passage à l'échelle de l'outil en proposant des nouvelles techniques permettant de combattre l'explosion combinatoire de la taille des machines d'états finis en utilisant des méthodes de compression (FDR2, 2003). Par exemple, la composition parallèle de deux processus P et Q ayant chacun 1000 états nécessite une machine d'états de 1000 000 d'états mais la composition compressée proposée par FDR2 (FDR2, 2003) nécessite uniquement 10000 états.

- **Wr2fdr**

Wr2fdr est un outil développé par l'université de Carnegie Mellon (Wr2fdr, 2005). Il permet de traduire une spécification Wright en une spécification CSP acceptée par l'outil FDR. Hormis les fonctionnalités lexico-syntaxiques et de génération de code CSP, l'outil Wr2fdr assure les fonctionnalités communes suivantes :

- Correspondances entre les événements locaux de Wright et les événements globaux de CSP.
- Détermination d'un processus CSP : det(P). Ceci permet de traiter l'opérateur non déterministe (∏) de CSP.
- Calcul de l'alphabet d'un processus CSP : αP, car FDR exige explicitement lors de la composition parallèle des processus (| |) leurs alphabets.
- Calcul des relations de raffinement liées aux propriétés 1, 2, 3 et 8 (cf. 2.3.1.1).

La version actuelle de l'outil Wr2fdr ne fait pas la distinction entre les événements initialisés et observés. De plus, les événements ne portent pas des informations ni d'entrée ni de sortie.

- **Exemple**

La figure 1.18 regroupe au sein de l'unité syntaxique style appelé CF le connecteur ContremaitreFermier issu de l'application traitée dans le chapitre 1 (section 1.4.3).

```
Style CF
Connector ContremaitreFermier
Role contremaitre = c -> d -> contremaitre
Role fermier = c -> d -> contremaitre
Glue = contremaitre.c -> fermier.c -> fermier.d -> contremaitre.d -> Glue
Constraints
   // no constraints
End Style
```

Figure 1.18. *Style ContremaitreFermier*

[2] Ceci est globalement vrai pour tout outil de model checking.

L'application de l'outil Wr2fdr sur ce style CF génère la spécification CSP donnée dans la figure 1.19.

```
-- FDR compression functions
transparent diamond
transparent normalise
-- Wright defined processes
channel abstractEvent
DFA = abstractEvent -> DFA |~| SKIP
quant_semi({},_) = SKIP
quant_semi(S,PARAM) = |~| i:S @ PARAM(i) ;
quant_semi(diff(S,{i}),PARAM)
power_set({}) = {{}}
power_set(S) = { union(y,{x}) | x <- S, y <-
power_set(diff(S,{x}))}
-- Style CF
-- Type declarations
-- events for abstract specification
channel d, c
-- Connector ContremaitreFermier
  -- generated definitions (to split long sets)
   ALPHA_ContremaitreFermier = {|fermier.c, fermier.d,
contremaitre.d, contremaitre.c
      |}
    Glue = ((contremaitre.c -> (fermier.c -> (fermier.d -
> (contremaitre.d
        -> Glue)))) [] SKIP)
  ALPHA_contremaitre = {c, d}
  ROLEcontremaitre = ((c -> (d -> ROLEcontremaitre)) |~|
SKIP)
  contremaitreA = ROLEcontremaitre [[ x <- abstractEvent
| x <- ALPHA_contremaitre
      ]]
  assert DFA [FD= contremaitreA

  ALPHA_fermier = {c, d}
  ROLEfermier = ((c -> (d -> ROLEfermier)) [] SKIP)
  fermierA = ROLEfermier [[ x <- abstractEvent | x <-
ALPHA_fermier ]]
  assert DFA [FD= fermierA
channel contremaitre: {c, d}
channel fermier: {c, d}
ContremaitreFermier = ( (ROLEcontremaitre[[ x <-
contremaitre.x | x <- {c
    , d } ]]
      [| diff({|contremaitre|}, {}) |]
    (ROLEfermier[[ x <- fermier.x | x <- {c, d } ]]
      [| diff({|fermier|}, {}) |]
    Glue)) )
ContremaitreFermierA = ContremaitreFermier [[ x <-
abstractEvent | x <- ALPHA_ContremaitreFermier]]
assert DFA [FD= ContremaitreFermierA
-- No constraints
-- End Style
```

Figure 1.19. *Traduction du Style ContremaitreFermier.wrt en CSP*

Les propriétés 2 et 3 relatives à la cohérence d'un connecteur (ici le connecteur ContremaitreFermier) sont traduites par les trois assertions suivantes :

(1) assert DFA [FD = contremaitreA

(2) assert DFA [FD = fermierA

(3) assert DFA [FD = ContremaitreFermierA

Les deux premières assertions (relation de raffinement DFA $\sqsubseteq_{FD}$ contremaitreA) sont liées à l'absence d'interblocage respectivement pour les deux rôles contremaitreA et fermierA. La troisième assertion est relative à l'absence d'interblocage pour le connecteur ContremaitreFermier. Ces trois assertions ont été vérifiées avec l'outil FDR (cf. figure 1.20).

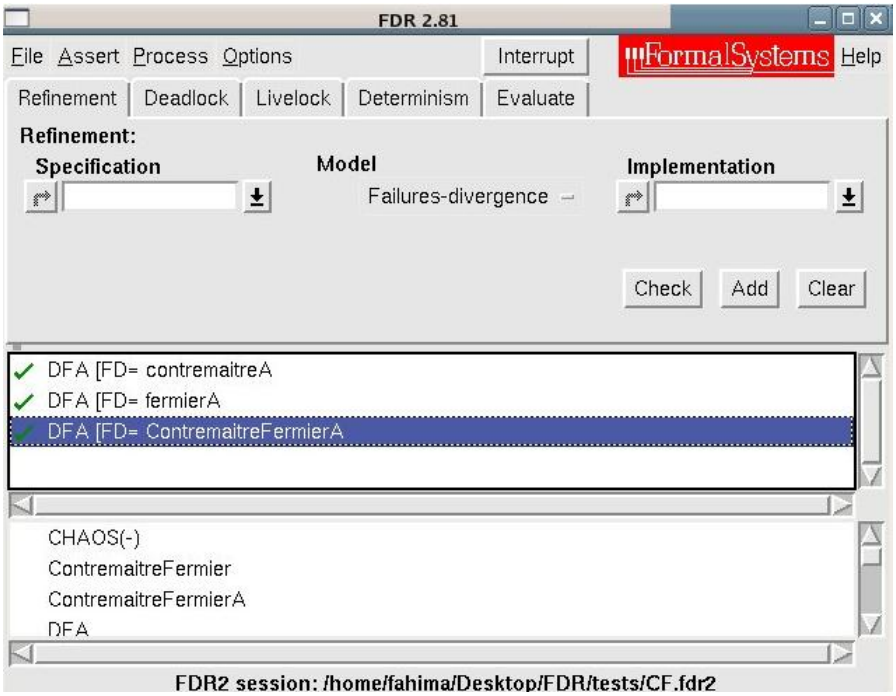


Figure 1.20. *FDR après vérification*

2.3.1.3. Autres techniques

Les auteurs de Wright ne proposent aucune automatisation des sept propriétés restantes. A notre avis la vérification automatique des propriétés 4, 6, 7 et 11 à savoir Initialiseur unique, Substitution de paramètres, Bornes d'intervalle et Complétude des liens nécessitent l'implémentation du langage Wright : analyseur lexico-syntaxique et analyseur sémantique. Ainsi la vérification des propriétés 4, 6 et 7 peut être assurée totalement par l'analyseur sémantique. Après avoir identifié les ports et les rôles non attachés au sein d'une configuration Wright, la propriété 11 peut être traitée de la même manière que la propriété 8 (compatibilité port/rôle), c'est-à-dire confiée à FDR. La vérification automatique de la propriété 9 (contraintes pour les styles) peut être obtenue par l'implantation du langage du contraintes de Wright. Dans le chapitre 10, nous proposons un prototype exploratoire permettant une implémentation orientée objet en Eiffel du langage de contraintes de Wright. La propriété 5 (engagement de l'initialiseur) ne peut pas être exprimée sous forme d'un raffinement CSP. En effet, elle exige l'inclusion d'un ensemble particulier d'échecs (t, R) et l'exclusion d'autres. Pour pouvoir vérifier la propriété 5 associée à un processus CSP P, il faut parcourir l'espace d'états du processus P et vérifier un certain pattern sur chaque état

parcouru. Ceci peut être effectué par un model checker approprié (Schnoebelen, 1995), c'est-à-dire travaillant sur des états et non pas sur des machines d'états comme FDR.

La propriété 10 (cohérence de style) peut être vérifiée par des model checkers favorisant la détection des contre-exemples tels que Nitpick (Jackson, 1996b) (Jackson., 1996a). L'outil Nitpick accepte des spécifications Z et utilise des techniques permettant d'éviter l'énumération exhaustive de l'espace d'états. Les techniques utilisées sont : énumération paresseuse (ou courte), évaluation courte, répartition de l'espace d'états en plusieurs classes d'équivalence. Chaque classe d'équivalence contient **au plus** un contre-exemple.

Nous pensons que les deux propriétés 9 et 10 peuvent être couplées et vérifiées avec des méthodes formelles avec preuve telle que B **(**Abrial, 1996**)**. En effet, les deux concepts de Wright Style et Configuration peuvent être reliés par une relation de raffinement de type B : le niveau abstrait modélise le concept Style et le niveau raffiné modélise le concept Configuration. Les contraintes associées au Style seront modélisées par des propriétés invariantes qui doivent être préservées par la Configuration. Ceci permet de surmonter les limites inhérentes à la technique de model checking : explosion combinatoire, problème de non décidabilité. Mais la preuve mathématique pose d'autres problèmes : il faut savoir répartir la preuve et avoir plus ou moins d'imagination lorsque le prouveur automatique échoue.

2.4 . Discussion

Dans ce paragraphe, nous mettons en relief les points forts et également les points faibles de l'offre Wright.

2.4.1 . Points forts

Les aspects structuraux de Wright comportent quatre concepts fondamentaux : composant, connecteur, configuration et style. Le concept connecteur joue un rôle important. En effet, il permet de définir d'une manière explicite des **patterns** d'interaction entre des composants non connus a priori. Ceci ouvre des perspectives intéressantes liées à la conception des connecteurs réutilisables inter-domaines et intra-domaines. En outre, le concept style permet de **factoriser** les caractéristiques communes (vocabulaire, propriétés invariantes, interfaces) à plusieurs configurations.

La description des comportements partiels et globaux des connecteurs et des composants est basée sur un langage élégant, formel et approprié : CSP de Hoare. Pour permettre la définition des propriétés architecturales, Wright enrichit CSP de Hoare avec deux nouveaux concepts : événement initialisé et événement observé.

Wright offre un langage de contraintes simple, basé sur la logique du premier ordre, permettant de décrire **des propriétés architecturales orientées état** telles que des propriétés invariantes attachées aux styles.

Pour permettre l'analyse rigoureuse des architectures logicielles, Wright définit **des propriétés architecturales standards** couvrant des propriétés orientées interaction et contrôle (cohérence de ports avec le Calcul, Compatibilité, …), des propriétés orientées état (contraintes pour les styles et cohérence de style) et des propriétés liées à la sémantique statique du langage Wright (substitution de paramètres, bornes d'un intervalle).

L'outil Wr2fdr de Wright permet l'automatisation de certaines propriétés (4/11) exprimées à l'aide du raffinement CSP et par conséquent vérifiées avec l'outil FDR. Ceci permet de détecter tôt des erreurs architecturales.

2.4.2 . Points faibles

La version actuelle de l'outil Wr2fdr comporte des limites : absence de distinction entre les événements initialisés et observés, événements ne portant pas d'information ni d'entrée ni de sortie, pas d'interface. De plus, en se basant sur notre expérience de cet outil, il contient des erreurs liées à la génération du code CSP et au traitement syntaxique du concept composant. Nous comptons, dans un futur proche, avec l'accord d'Allen et Garlan, participer à l'amélioration de cet outil.

Actuellement, plusieurs propriétés architecturales standards (7/11) définies par Wright ne sont pas automatisables car elles nécessitent la combinaison de techniques plus ou moins complémentaires : Nitpick, implantation du langage Wright, model checkers orientés état (section 2.3.1.3).

En Wright, la sémantique d'une configuration est définie par la composition parallèle de plusieurs processus CSP. En effet, le comportement de chaque instance (de type Composant ou Connecteur) déclarée est traduit par un processus CSP.

Configuration ABC
  Component Atype
    Port Out = ā -> Out ∏ §
    Computation = $\overline{\text{Out.a}}$ -> Computation ∏ §
  Component Btype
    Port In = c -> In □ §
    Computation = In.c -> b̄ -> Computation □ §
  Connector Ctype
    Role Origin = a -> Orgin ∏ §
    Role Target = c -> Target □ §
    Glue = Origin.a -> $\overline{\text{Target.c}}$ -> Glue□ §
  Instances
    A : Atype
    B : Btype
    C : Ctype
  Attachments
    A.Out as C.Orgin
    B.In as C.Target
End ABC

Figure 1.21. *Une Configuration Wright*

Par exemple, le processus reflétant le comportement de la configuration ABC (cf. figure 1.21) est défini :

A = $\overline{\text{A.Out.a}}$ -> A ∏ §

|| C = A.Out.a -> $\overline{\text{B.In.C}}$-> C □ §

|| B = B.In.c -> $\overline{\text{B.b}}$ -> B □ §

Les expressions des processus CSP associés à chaque instance sont obtenues selon des règles formelles (Allen, 1997a). Mais Wright ne définit pas de propriétés standards liées aux configurations telle que l'absence d'interblocage. Les raisons évoquées par les auteurs de Wright touchent à l'efficacité de la vérification de l'absence de l'interblocage pour une

configuration de taille (en nombre d'instances) importante. En effet, FDR est inefficace (complexité exponentielle) pour la vérification de l'absence d'interblocage des processus CSP utilisant intensivement la composition parallèle (| |). L'apparition des outils dédiés à la vérification de l'absence d'interblocage des processus CSP élaborés (cf. chapitre 3) -grâce à la composition parallèle- ouvre des perspectives intéressantes pour l'analyse des configurations Wright.

Wright en tant que langage «généraliste» ne peut pas proposer des propriétés architecturales spécifiques c'est-à-dire liées à un domaine d'application ou à des applications particulières. L'architecte qui désire vérifier des propriétés architecturales spécifiques doit travailler sur des spécifications CSP produites par l'outil Wr2fdr. De plus, il faut que les propriétés à vérifier soient naturellement exprimables sous forme de raffinement CSP. Mais les propriétés spécifiques potentiellement vérifiables dans le cadre de CSP ne couvrent pas toutes les classes de propriétés. En effet, CSP ne peut pas traiter naturellement les propriétés d'équité[3], les propriétés orientées état (invariant du système) et les propriétés qui incluent des événements et excluent d'autres. La technique de model checking appliquée sur des programmes -notamment sur des programmes concurrents- ouvre des perspectives intéressantes pour vérifier des propriétés spécifiques plus ou moins diversifiées. Mais ceci suppose le passage d'une représentation architecturale sous forme d'un **modèle** (AL en Wright) vers une représentation architecturale sous forme **d'un programme concurrent**. Pour faciliter un tel passage, il faut que l'écart sémantique entre Wright et le langage d'écriture du programme concurrent soit réduit. Un bon candidat est Ada (Burns, 1998) (Booch, 1991).

2.5 . Conclusion

Dans ce chapitre, nous avons étudié d'une façon approfondie les propriétés architecturales standards définies par Wright. Après avoir présenté informellement ces propriétés, nous nous sommes penchés sur les techniques de vérification des propriétés Wright : model checking basé sur le raffinement (FDR), model checking orienté état, implantation du langage Wright et outils de vérification des propriétés orientées état (Nitpick). Enfin, nous avons identifié les points forts et également les points faibles de Wright en indiquant des pistes permettant de faire face à ces points faibles. Dans le chapitre suivant, nous proposons une Démarche de Vérification Formelle d'Architectures Logicielles (DVFAL) basée en partie sur les pistes exposées dans le paragraphe 2.4.

[3] Une propriété d'équité (fairness en anglais) énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois. On parle de vivacité répétée (Schnoebelen, 1995).

Chapitre 3 : Démarche de Vérification Formelle d'Architectures Logicielles (DVFAL)

3.1 . Introduction

Dans ce chapitre, nous proposons une Démarche de Vérification Formelle d'Architectures Logicielles : DVFAL. Dans la première section, nous établissons les exigences de la démarche DFVAL : pluralité des formalismes de description des AL, pluralité des classes de propriétés, propriétés standards/propriétés spécifiques et pluralité des techniques de vérification. Dans la seconde section nous présentons les constituants de la démarche DVFAL : les langages retenus, les traducteurs, les outils de vérification et les différentes représentations des AL.

3.2 . Exigences

3.2.1 . Pluralité des formalismes de description des AL

La communauté scientifique a développé plusieurs formalismes (ou notations) permettant de décrire des AL. Ces formalismes peuvent être classés en quatre catégories :

- **Langages de description d'architecture (ADL)**

Un ADL est un langage qui modélise l'architecture du système sous forme graphique ou textuel. Il existe un nombre important voire considérable d'ADL tels que : C2, Acme, Unicon, Wright, Rapide, ArchJava, Fractal, Darwin. Plusieurs études comparatives des ADL existent (Medvidovic et, 2000), (ACCORD, 2002), (El Boussaidi, 2006) sur les ADL. Les ADL partagent au moins trois concepts fondamentaux : composant, connecteur et configuration. Ces ADL sont plus ou moins outillés et diffusés aussi bien dans le monde académique qu'industriel. Rares sont les ADL qui permettent de raisonner formellement sur les architectures logicielles tels que Wright (cf. chapitre 1 et chapitre 2) et Rapide (Luckham, 1995). Mais contrairement à Wright, Rapide ne supporte pas de modèle de connecteur c'est-à-dire la notion de connecteur n'est pas explicite. Nous citons également le modèle COSA (Alti, 2006) (Component-Object based Software Architecture) qui est une architecture logicielle à base d'objets et de composants. Sa stratégie est de s'intéresser, dans le cadre du développement de nouvelles architectures logicielles ouvertes, adaptables et évolutives, au problème de conception et de développement d'un composant en empruntant et en étendant les formalismes de description des ADL et en les projetant sur des modèles objets. Cela permet de développer des systèmes à base de composants de meilleure qualité, offrant un meilleur potentiel de réutilisation et limitant la distance sémantique entre conception et implémentation.

- **UML2.0 et ses extensions**

UML2.0 (OMG, 2005) propose un modèle de composants englobant des concepts tels que : composant, port, structure composite, connecteur, interface offerte, interface requise et protocol state machine (cf. chapitre 6). Ainsi, UML2.0 permet de modéliser des architectures logicielles. Le concept «structure composite» d'UML2.0 est similaire au concept «configuration» offert par les ADL. De plus, les aspects comportementaux d'une

AL peuvent être décrits par des machines à états de description de protocoles (protocol state machine). Des profils UML2.0 peuvent être utilisés avec profit pour décrire des AL et permettent d'adapter d'avantage UML2.0 au domaine des architectures (cf. chapitre 7).

- **Modèles de composant académiques**

Plusieurs modèles de composants sont issus du monde académique tel que Kmelia (André, 2006) et Ugatze (Seyler, 2004). Par exemple Ugatze comporte trois concepts fondamentaux permettant de décrire des AL : composant, interaction et graphe d'interactions.

- **Méthodes orientées composant**

Le paradigme composant a donné naissance à des méthodes orientées composant (Oussalah, 2005). Par exemple, dans la méthode Symphony (cf. chapitre 11), un composant métier comporte trois parties :

- une partie contrat avec l'exécution (ce que je sais faire),
- une partie structurelle (ce que je suis),
- une partie collaboratrice (ce que j'utilise).

Ces modèles et méthodes basés composant sont tout à fait compatibles avec des implantations selon de type EJB, CORBA ou autres.

3.2.2 . Pluralité des classes de propriétés

L'appréciation d'une architecture logicielle passe inévitablement par l'étude des propriétés que cette architecture devrait satisfaire. Dans le domaine de la vérification de logiciels (Schnoebelen, 1995), les propriétés à vérifier sont regroupées en plusieurs classes.

- **Les propriétés d'atteignabilité**

Une propriété d'atteignabilité énonce qu'une certaine situation peut être atteinte. Par exemple, «on peut entrer en section critique sans passer par n=0». L'atteignabilité peut être simple, conditionnelle ou aussi s'appliquer à tout état atteignable.

- **Les propriétés de sûreté**

Une propriété de sûreté énonce que, sous certaines conditions, quelque close ne se produit jamais. Par exemple : «les deux processus ne seront jamais en section critique». En général, il s'agit d'énoncer qu'une chose indésirable ne se produira pas, d'où la terminologie de «propriété de sûreté» (en anglais : safety property).

- **Les propriétés de vivacité**

Une propriété de vivacité énonce que, sous certaines conditions, quelque chose finira par avoir lieu. Par exemple : «toute requête finira par être satisfaite». En anglais, vivacité se dit liveness. Il s'agit en général d'énoncer qu'une chose souhaitée finira par avoir lieu.

- **L'absence de blocage**

L'absence de blocage est une propriété particulière, énonçant que le système ne se trouve jamais dans une situation où il lui est impossible de progresser. C'est une propriété de correction pour des systèmes supposés ne jamais terminer. L'absence de blocage (de deadlock en anglais) est souvent cataloguée comme une propriété de sûreté. En effet, elle respecte «que quelque chose de non souhaité ne se produira jamais».

- **Les propriétés d'équité**

Une propriété d'étiqueté énonce que, sous certaines conditions, quelque chose aura (ou n'aura pas lieu) un nombre infini de fois. Par exemple : «si l'on demande l'accès en section

critique en nombre infini de fois, alors il sera accordé un nombre infini de fois». En anglais, équité se dit fairness. On parle aussi de vivacité répétée.

Bien entendu, les classes de propriétés citées ci-dessus sont applicables sur les architectures logicielles. En effet, l'architecture du système étudié peut avoir des propriétés invariantes exprimées comme des propriétés de sûreté De même, les architectures logicielles de type client-serveur possèdent des propriétés de vivacité de forme : «tôt ou tard un client doit être servi». En outre, ces architectures logicielles de type client-serveur doivent souvent satisfaire des propriétés d'équité ayant la forme générale suivante : «un client doit être servi autant de fois qu'il le désire». Enfin, les architectures logicielles comportent des entités qui évoluent en parallèle et doivent respecter des protocoles de synchronisation afin d'éviter des situations d'interblocage.

3.2.3. Propriétés standards/Propriétés spécifiques

Les propriétés architecturales standards sont communes à toutes les architectures logicielles. Elles peuvent être attachées aux trois principaux concepts architecturaux : composant, connecteur et configuration. Ces propriétés sont liées à la **cohérence** de chaque concept architectural : pas de contradiction entre ses différentes parties. La description et la vérification des propriétés architecturales standards dépendent du formalisme utilisé (ADL, UML2.0 et ses extensions,…). Par exemple, l'ADL Wright (cf. chapitre 1 et 2) propose cinq propriétés permettant de vérifier la cohérence des composants (propriété 1), des connecteurs (propriétés 2 et 3) et des configurations (propriétés 8 et 11).

Les propriétés architecturales spécifiques dépendent du système étudié. Elle sont plus ou moins nombreuses et conditionnent tout rafinement ultérieur de l'architecture du système étudié. Une hiérarchisation des propriétés architecturales spécifiques est possible en tenant compte des critères tels que : domaine (factorisation des propriétés communes) et type d'architecture (Shaw, 1996) tels que client-serveur, pipe-filtre.

L'automatisation des propriétés architecturales standards est souvent possible. Tandis que la définition des propriétés architecturales spécifiques est à la charge de l'architecte.

3.2.4. Pluralité des techniques de vérification

Plusieurs techniques peuvent être envisagées pour vérifier que **le système** (modèle ou programme) est conforme à la spécification. Dans la suite, nous allons passer en revue les techniques de vérifications associées aussi bien aux modèles qu'aux programmes.

3.2.4.1. Techniques de vérification de modèles

En vérification de modèles, les techniques sont usuellement classées en trois catégories : simulation, preuve de théorèmes, model checking.

- **La simulation**

La simulation soumet le modèle à un ensemble de scénarios qui visent à mettre en échec le modèle pour les propriétés souhaitées. Par exemple, il existe des outils permettant d'animer (ou de simuler) des modèles UML/OCL (Ziemann, 2003), (Richters, 2000). Mais cette technique ne permet pas de prouver l'exactitude du modèle car les scénarios testés ne sont généralement pas exhaustifs c'est-à-dire ne couvrent pas la totalité des comportements possibles.

- **La preuve de théorèmes**

La preuve de théorèmes permet d'apporter la preuve mathématique de correction du modèle. Le théorème représente une propriété générale. Une fois ce théorème démontré, il vaut pour la totalité du modèle. Par exemple, les méthodes formelles basées sur la théorie des ensembles et la logique du premier ordre comme VDM, (Bjørner, 1978), Z (Spivey, 1989) et B (Abrial, 1996) proposent des prouveurs automatiques permettant de prouver une bonne partie des lemmes et des propriétés attachées aux modèles. Ainsi, pour prouver la cohérence d'un modèle abstrait -appelé machine abstraite en B-, le langage B (Abrial, 1996), (Abrial, 2003) génère les deux obligations de preuve suivantes :

(op1) La clause d'initialisation établit l'invariant de la machine abstraite.

(op2) Chaque opération offerte par la machine abstraite doit préserver l'invariant sous réserve que sa précondition, soit satisfaite.

Ensuite, ces obligations de preuves sont traitées par le prouveur automatique B. Cependant, les preuves « intéressantes » nécessitent généralement une intervention humaine plus ou moins importante : preuve semi-automatique (Cansell, 2003).

- **Le model checking**

Le model checking (Merz, 2006), (Schnoebelen, 1995) est une technique automatique basée sur l'exploration de tous les comportements du modèle : vérification exhaustive. Les données à fournir à un model checker sont la description du modèle à analyser et la propriété à vérifier. L'outil (model checker) confirme que la propriété est vraie ou bien informe l'utilisation qu'elle n'est pas vérifiée. Dans le cas négatif, le model checker exhibe une exécution du modèle qui montre que la propriété est fausse. Ceci simplifie l'analyse de l'échec. En général, un model checker supporte deux formalisme plus ou moins différents : l'un pour décrire le modèle à analyser et l'autre pour décrire la propriété à vérifier. Par exemple, le model checker SPIN (Holzmann, 2003) supporte le langage PROMELA pour décrire les modèles et la logique temporelle pour décrire les propriétés. Tandis que, le model checker FDR (FDR2, 2003) accepte des modèles écrits en CSP dotés des propriétés à vérifier exprimées par des relations de raffinement CSP. Cependant, la technique de model checking souffre de deux limites importantes. La plupart des model checkers supposent que l'espace d'état du système étudié est **fini**. De plus, même si l'espace d'état est fini, il peut être **si grand** qu'il ne peut pas être exploré par des algorithmes implantés par les model checker faute de temps et de mémoire. Ce problème est connu sous le nom **d'explosion combinatoire** du nombre d'états. En effet, soit un système S composé de p processus indépendants, si chacun de ces processus peut être dans k états différents, alors la taille de l'espace des états de S sera de k^p. Une telle taille croit exponentiellement par rapport au nombre de processus. Ceci rend le problème du model checking très difficile même pour des petites valeurs de k et p. Cependant, il existe trois approches globales permettant de combattre l'explosion combinatoire (Merz, 2006) : techniques de compression, optimisations basées sur les réductions et techniques d'abstraction.

3.2.4.2. Techniques de vérification des programmes

Il existe plusieurs techniques permettant de vérifier l'exactitude du programme (code ou implémentation). Parmi les techniques, nous citons :

- **Le test**

L'activité de test (Xanthakis, 2000) permet de vérifier si l'implémentation est conforme à la spécification. On distingue le test fonctionnel et structurel. Le test fonctionnel considère l'implémentation comme boîte noire et génère les cas de test à partir de la spécification. Tandis que le test structurel engendre les cas de test à partir de l'implémentation considérée comme boîte blanche. Les principaux aspects tels que génération automatique des cas de

test (Agedis, 2003) (Jéron, 2002), automatisation de l'oracle et apports de la conception par contrat (précondition, postcondition et invariants) à l'activité de test (Meyer, 1992) font l'objet d'efforts constants par la communauté de test. Cependant, comme la simulation, le test ne permet pas d'affirmer avec certitude que le programme est correct mais de trouver des erreurs (ou bogues). De plus, les programmes concurrents comportant plusieurs processus qui interagissent (compétition et coopération) sont extrêmement difficiles à tester. Les erreurs de concurrence comme l'interblocage, famine et exclusion mutuelle peuvent être provoquées par des entrelacements des processus très subtils et difficiles à reproduire.

- **L'analyse statique par interprétation abstraite**

L'interprétation abstraite est une théorie de l'approximation discrète de sémantiques de systèmes informatiques principalement utilisée pour l'analyse et la vérification statique de logiciels (Cousot, 2000). L'idée de base de l'analyse statique des programmes et d'utiliser l'ordinateur pour trouver des erreurs de programmation. Mais la sémantique concrète du programme qui formalise l'ensemble des exécutions possibles de ce programme dans tous ses environnements d'exécution possible est non calculable. En effet, le calcul de la sémantique concrète des programmes est non décidable (Blanchet, 2006). La sémantique abstraite consiste à ne considérer qu'une approximation des comportements possibles du programme à analyser. Il s'agit de construire un sur-ensemble de la sémantique concrète du même programme. Ceci offre deux avantages majeurs :

- La sémantique abstraite couvre tous les chemins d'exécution possibles relatifs à la sémantique concrète. Toute propriété satisfaite sur la sémantique abstraite d'un programme P est forcément (implication) satisfaite sur la sémantique concrète du même programme ;
- La représentation informatique de la sémantique abstraite d'un programme P est souvent compacte.

L'interprétation abstraite a été très récemment industrialisée. Des sociétés comme «AbsInt Angewandte Informatik GmbH» (Allemagne) et «Polyspace Technologies» (France) développent des analyseurs statiques de logiciels qui sont disponibles commercialement. De même des analyseurs statiques de programmes issus du milieu académique existent. Par exemple, l'analyseur ASTREE (Cousot, 2007) permet de prouver l'absence des erreurs d'exécution des programmes écrits en C telles que division par zéro, débordement, accès en dehors des bornes d'un tableau. De même, l'analyseur ESC/Java (Cok, 2004) permet de prouver l'absence des exceptions d'exécution dans des programmes écrits en Java.

La technique d'interprétation abstraite souffre d'un problème connu sous le nom de «fausses alarmes» (en anglais false alarms). En effet, cette technique peut inventer des erreurs inexistantes (vis-à-vis de la sémantique concrète) à cause du compromis difficile entre précision et efficacité : le sur-ensemble des chemins d'exécution possibles provoque une certaine imprécision. Mais ce problème de «fausses alarmes» peut être résolu moyennant un réglage manuel des paramètres des analyseurs statiques conçus selon l'approche d'interprétation statique comme ASTRÉE (Cousot, 2005).

- **La preuve**

Elle permet de démontrer que le programme vérifie bien certaines propriétés exprimées sous formes de théorèmes. Cette preuve peut être effectuée manuellement ou assistée par ordinateur (prouveurs automatiques, semi-automatiques intégrées dans des méthodes formelles comme B). Les assertions externes (précondition, postcondition et invariant) et les assertions internes (invariant et variant de boucle et contrôle à un certain point d'exécution du programme) jouent des rôles importants pour la construction et la preuve des algorithmes (Guyomard, 1999) (Meyer, 2003).

- **Le model checking**

Il peut aussi être appliqué au programme. La propriété à vérifier est testée de façon exhaustive sur l'ensemble des exécutions possibles du programme. En cas, d'erreur, une exécution incorrecte est montrée par le model checker.

3.3 . La démarche DVFAL

En tenant compte de la pluralité des formalismes de description d'architectures logicielles, des classes des propriétés, des techniques de vérification applicables aussi bien sur les modèles que les programmes et de la distinction entre propriétés standards et spécifiques, nous proposons dans ce paragraphe une Démarche de Vérification Formelle d'Architectures Logicielles : DVFAL.

3.3.1 . Objectifs généraux et vue d'ensemble

La démarche DVFAL doit :

- Traiter d'architectures logicielles décrites dans divers formalismes : ADL (Wright, Rapide, C2, Olan, Darwin), UML2.0, des profils UML2.0, Symphony, Ugatze, etc.
- Permettre la vérification formelle des propriétés architecturales standards et spécifiques.
- Faciliter l'implémentation incrémentale d'architectures logicielles.

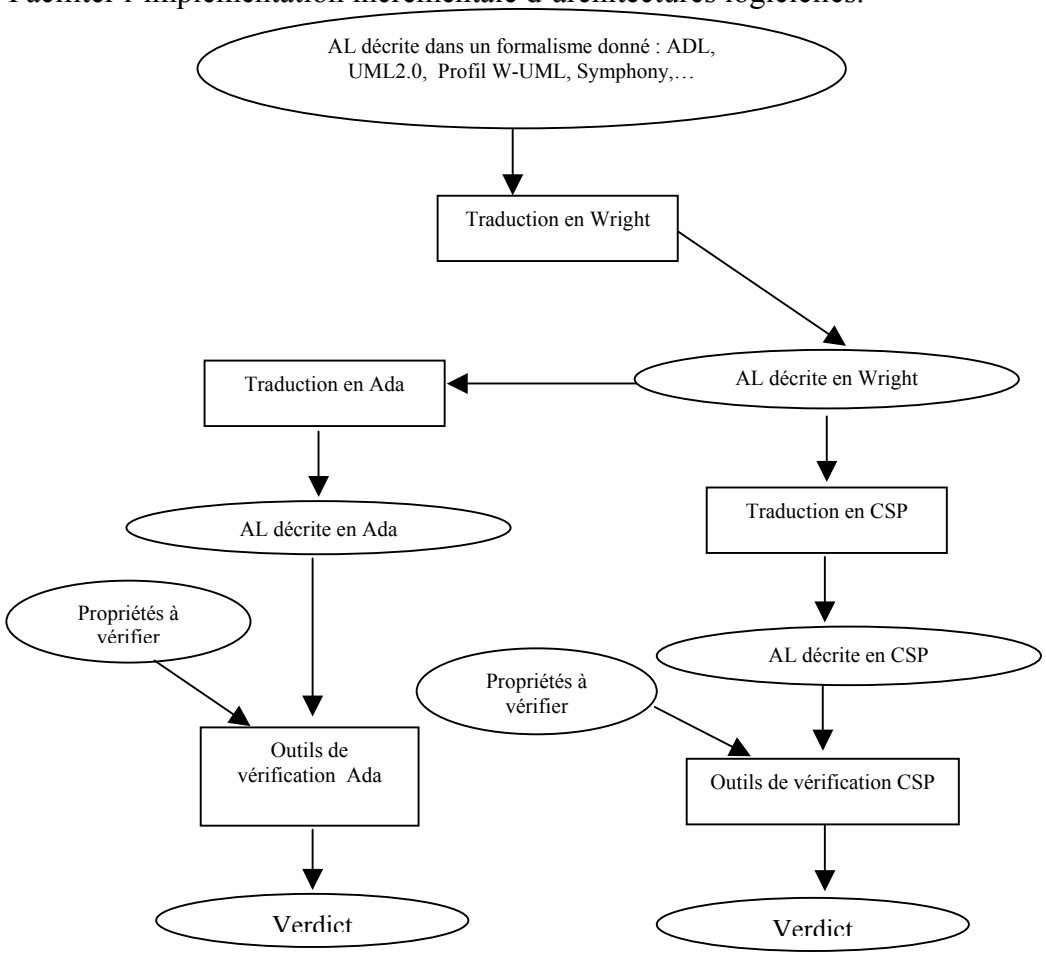


Figure 1.22. *La structure macroscopique de la démarche DVFAL*

La figure 1.22 donne la structure macroscopique de la démarche DFVAL. Celle-ci comporte deux types d'outils logiciels : les traducteurs et les outils de vérification formelle.

Les traducteurs assurent le passage d'un formalisme à un autre. Et par conséquent, ils permettent la manipulation de plusieurs formalismes de description d'architectures logicielles et donnent accès à divers outils de vérification formelle.

La démarche DFVAL a retenu[4] deux langages : l'ADL Wright et le langage Ada. L'ADL Wright joue le rôle d'un langage pivot permettant de représenter une architecture logicielle décrite dans divers formalismes de modélisation. Quant au langage Ada, il offre la possibilité de représenter les architectures logicielles décrites en Wright comme des programmes Ada concurrents afin de les analyser davantage avec les outils de vérification des programmes concurrents associés à Ada. En outre, cette ouverture sur le langage de programmation Ada permet l'implémentation incrémentale et correcte d'architectures logicielles en se servant progressivement des divers outils de vérification supportant Ada.

L'enchaînement des étapes de la démarche DVFAL est :

1. Traduction du formalisme utilisé vers Wright
2. Traduction de Wright vers CSP
3. Vérification des propriétés architecturales standards définies par Wright et de propriétés spécifiques en utilisant les outils CSP.
4. Traduction de Wright vers Ada
5. Vérification des propriétés spécifiques en utilisant les outils Ada

La démarche DVFAL ne traite pas d'une façon explicite les retours en arrière liés à la correction des erreurs montrées par les outils de vérification formelle utilisés. Ainsi, les correspondances entre Wright-formalisme utilisé, CSP-Wright et Ada-Wright sont totalement manuelles c'est-à-dire à la charge de l'architecte en fonction du système étudié.

3.3.2. Architecture Logicielle comme modèle Wright

La démarche DVFAL propose l'ADL Wright (cf. chapitre 2 et chapitre 3) comme langage pivot permettant de représenter d'une façon uniforme des architectures logicielles décrites dans divers formalismes y compris en Wright. Les principales raisons qui militent en faveur de l'ADL Wright comme langage pivot sont les suivantes.

- **Concepts architecturaux**

Wright supporte des concepts architecturaux bien définis : composant, connecteur, configuration et style. Les connecteurs de Wright jouent un rôle important : ils sont explicitement présents dans les descriptions architecturales et ils peuvent encapsuler des sémantiques arbitraires d'échange potentiellement réutilisables. En outre, les styles de Wright permettent de factoriser des caractéristiques communes (types de composants, types de connecteurs, des propriétés sémantiques) à un ensemble des configurations.

- **Langage de spécification formelle**

Wright est un langage de spécification formelle d'architectures. Il est basé sur le calcul formel CSP et définit des propriétés architecturales standards (cf. chapitre 2).

- **Ouverture sur CSP**

La sémantique de Wright s'appuie sur la sémantique formelle de CSP. En outre, l'outil Wr2fdr (cf. chapitre 2) supportant Wright permet de convertir une description architecturale

[4] Nous reviendrons ultérieurement sur les raisons qui militent en faveur de ces deux choix.

décrite en Wright vers une description décrite en CSP. Ceci permet d'utiliser avec profit les divers outils de vérification formelle supportant CSP (cf. 3.3.4).

3.3.3 . Architecture Logicielle comme programme Ada concurrent

La démarche DVFAL offre la possibilité de représenter des architectures logicielles décrites en Wright par des programmes Ada[5] concurrents. Ceci permet :

- D'utiliser les divers outils de vérification supportant Ada (cf. 3.3.5) pour prouver des propriétés spécifiques qui ne peuvent pas être naturellement traitées dans le cadre de CSP telles que les propriétés d'équité ou les propriétés orientées état ;
- D'aller vers des implantations concrètes et incrémentales d'architectures logicielles décrites en Wright en utilisant progressivement les outils de vérification formelle associés à Ada. Ainsi, le champ d'utilisation de l'ADL Wright est élargi : il couvre à la fois la modélisation et l'implémentation.

3.3.4 . Outils de vérification CSP

Le langage CSP de Hoare est une notation permettant la description des patterns de communication en utilisant une algèbre de processus. Ce langage est largement utilisé pour la conception des systèmes parallèles et distribués, et pour la preuve formelle des propriétés vitales -notamment de sûreté et de vivacité- de tels systèmes. Cependant, sans l'assistance des outils de vérification, il n'est pas pratique de vérifier de telles propriétés sur des systèmes complexes. Les outils de vérification formelle supportant CSP peuvent être classés comme en quatre catégories :

- **Les outils basés sur le raffinement**

La propriété à vérifier est exprimée par une relation de raffinement entre deux processus P et Q ; P $\sqsubseteq$ Q. Elle admet deux niveaux de description : niveau abstrait traduit par P et niveau moins abstrait (ou raffiné) traduit par Q. L'outil FDR décrit dans 2.3.1.2 incarne cette approche de vérification. CSP et FDR sont notamment utilisés pour le développement incrémental avec preuve des systèmes concurrents et distribués (Lawrence, 2005).

Mais la vérification des propriétés en utilisant le raffinement CSP souffre des problèmes suivants :

- La description abstraite de la propriété à vérifier (Processus P) n'est pas facile à établir. Une connaissance approfondie des concepts abstraits de CSP notamment liés au non déterminisme est largement exigée.
- La description de la propriété à vérifier sous forme d'une relation de raffinement liant deux processus est peu naturelle.
- Certaines propriétés ne peuvent pas être décrites en utilisant le raffinement CSP : par exemple, la propriété 5 « engagement de l'initialiseur » définie par Wright (cf. 2.3.1.3).

- **Les outils basés sur la satisfaction des prédicats**

La propriété à vérifier est exprimée par un prédicat attaché à un processus CSP à analyser. Un tel prédicat est une expression logique utilisant des ensembles (traces, échecs et

[5] La concurrence en Ada sera traitée dans le chapitre 4.

divergences, cf. 1.3.3.2), des opérateurs applicables sur ces ensembles assimilés à des séquences et des opérateurs logiques et relationnels. La notion P **sat** f(tr, ref) spécifie que tous les échecs (tr, Ref) du processus P doivent satisfaire le prédicat f, formellement nous avons l'équivalence suivante :

P **sat** f(tr, ref) <==> ∀ (tr,ref) : failures(P).f(tr,ref).

L'outil satchecker incarne cette approche de vérification (Martin, 2000). La spécification des propriétés à vérifier sous forme des prédicats possède plusieurs points forts. Elle exige moins de connaissances approfondies de CSP que l'approche basée sur le raffinement. En outre, elle est plus naturelle. Enfin, son champ d'utilisation est plus élargi : elle couvre plus de propriétés que l'approche basée sur le raffinement.

- **Les outils dédiés pour la détection d'interblocage**

Une propriétés vitale à vérifier lors de la conception des systèmes parallèles et distribués est l'absence d'interblocage. Mais la vérification de cette propriété sur des systèmes contenant plusieurs entités (processus) évoluant en parallèle se heurte au problème d'explosion combinatoire liée à la technique de model checking (cf. 3.2.4.1). Des outils spécialisés pour la détection d'interblocage et de divergence pour des systèmes massivement parallèles existent (Martin, 1997a). Ces outils proposent des solutions permettant de surmonter les limites inhérentes à l'approche exhaustive : un seul système d'états-transitions pour tous les processus (approche utilisée par FDR). Par exemple, l'outil Deadlock Checker (Martin, 1997a) utilise un système d'états-transitions pour chaque processus et opère progressivement pour détecter l'interblocage : vérifier son absence sur chaque processus pris individuellement et par couples de processus. Ainsi, l'approche exhaustive exige $10^{\sim 50}$ états pour un réseau de 100 philosophies et fourchettes pour le dîner des philosophes. Par contre, l'outil Deadlock Checker nécessite uniquement un graphe bi-parti de 800 états pour détecter s'il contient ou non un circuit.

- **Les outils d'animation**

L'outil ProBE (ProBE, 2003) est un animateur des spécifications CSP. Il permet d'explorer le comportement d'un processus CSP en provoquant les événements qui entraînent le passage d'un état à un autre. ProBE offre une structure hiérarchique permettant de visualiser les actions et les états possibles du processus CSP animé. L'outil ProBE est facile à utiliser. De plus il fournit des fonctionnalités importantes permettant d'analyser le comportement d'un processus telles que : la recherche des états stables, des événements exprimés par des expressions régulières et la sauvegarde de la trace du processus.

3.3.5. Outils de vérification Ada

Le langage Ada offre des constructions intéressantes permettant d'écrire des programmes concurrents telles que : tâche (task), choix non déterminisme (select) et des primitives de synchronisation/communication (rendez-vous et types protégés). Une présentation plus détaillée des possibilités d'Ada vis-à-vis de la concurrence fera l'objet du chapitre 4.

L'analyse statique est une technique qui permet d'analyser un programme sans toutefois l'exécuter. Plusieurs techniques d'analyse statique des programmes concurrents ont été proposées (Chamillard, 1996). Ces techniques englobent : analyse d'accessibilité, model checking symbolique, analyse de flot d'équations et analyse de flot de données. L'analyse d'accessibilité (ou analyse exhaustive ou model-checking) permet de générer un espace d'états représentant la sémantique du programme concurrent et vérifie ensuite la propriété sur cet espace d'états. Le model checking symbolique vérifie la propriété sur une représentation symbolique (implicite grâce aux OBDD (Bryant, 1986)) compacte de

l'espace d'états. L'analyse de flot d'équations représente le programme et la propriété sous forme d'un système d'inégalités entières à résoudre. Enfin, l'analyse de flot de données représente le programme sous forme d'un graphe orienté et vérifie la propriété décrite en utilisant les expressions régulières sur cette représentation graphique.

Ces quatre techniques ont été implémentées dans divers outils. Parmi ces outils, nous présentons : SPIN, SMV, INCA et FLAVERS.

- **SPIN**

Le programme concurrent à analyser est décrit en utilisant le langage PROMELA (Holzmann, 1991). Celui-ci est un langage impératif comportant quelques primitives de communication. PROMELA permet ainsi de décrire le comportement de chacun des processus d'un système, et les interactions entre ces systèmes. Pour communiquer, les processus peuvent utiliser des canaux de communication fifo, prendre rendez-vous ou utiliser des variables communes.

En partant d'une description PROMELA du programme et de la propriété à vérifier, SPIN (Chamillard, 1996) génère un programme écrit en C. L'exécution de ce programme permet d'effectuer l'analyse demandée. SPIN permet de vérifier automatiquement l'absence d'interblocage. Les autres propriétés à vérifier doivent être spécifiées en utilisant les **assertions** ou **les automates d'états finis**. Les assertions sont insérées dans le programme PROMELA à des endroits choisis par l'utilisateur.

- **SMV**

SMV (McMillan, 1993) permet le model-checking symbolique (à base de BDD). Le programme concurrent à analyser est soumis à SMV sous forme d'un réseau d'automates. La propriété à vérifier est spécifiée en utilisant la logique temporelle CTL (Computation Tree Logic). Dans les cas où une propriété demandée n'est pas satisfaite SMV donne un exemple d'exécution violant la propriété.

- **INCA**

L'outil INCA implémente la technique flot d'équations (Corbette, 1995). Le programme à analyser est spécifié en Ada-like ou en SEDL (S-Expression Design langage). La propriété à vérifier est spécifiée sous forme d'une séquence d'événements. INCA génère un ensemble des conditions nécessaires pour l'existence d'une exécution violant la propriété. Ces conditions nécessaires sont formulées comme un système d'inégalités résolu en utilisant la technique de programmation entière linéaire.

- **FLAVERS**

L'outil FLAVERS (Dwyer, 1994) (Dwyer, 1998) (Dwyer, 1999) (Cobleigh, 2002) effectue une analyse de flot de données pour vérifier des propriétés sur des programmes concurrents. FLAVERS accepte un ensemble de graphes de flots de contrôle (Control Flow Graphs : CFG) annotés avec des événements pertinents (les événements qui nous intéressent) comme spécification du programme concurrent à analyser. La propriété à vérifier est spécifiée en utilisant les expressions régulières quantifiées (QRE : Quantified Regular Expression). La vérification de l'absence d'interblocage n'est pas couramment supportée par FLAVERS. Plusieurs autres propriétés notamment de vivacité et d'équité peuvent être exprimées en utilisant QRE. Dans les cas où une propriété demandée n'est pas satisfaite, FLAVERS fournit un exemple d'exécution violant la propriété.

Il existe une chaîne d'outils permettant aux programmes écrits en Ada d'être analysés par SPIN, SMV, INCA et FLAVERS (Dwyer, 1998). La figure 1.23 illustre cette chaîne d'outils.

Ces quatre outils SPIN, SMV, INCA et FLAVERS sont des représentants des quatre approches permettant l'analyse statique des programmes concurrents Ada ; ils sont complémentaires. En effet, vis-à-vis de la spécification des propriétés à vérifier, les outils SPIN et SMV favorisent les propriétés orientées **état**. Tandis que INCA et FLAVERS favorisent les propriétés orientées **chemin**.

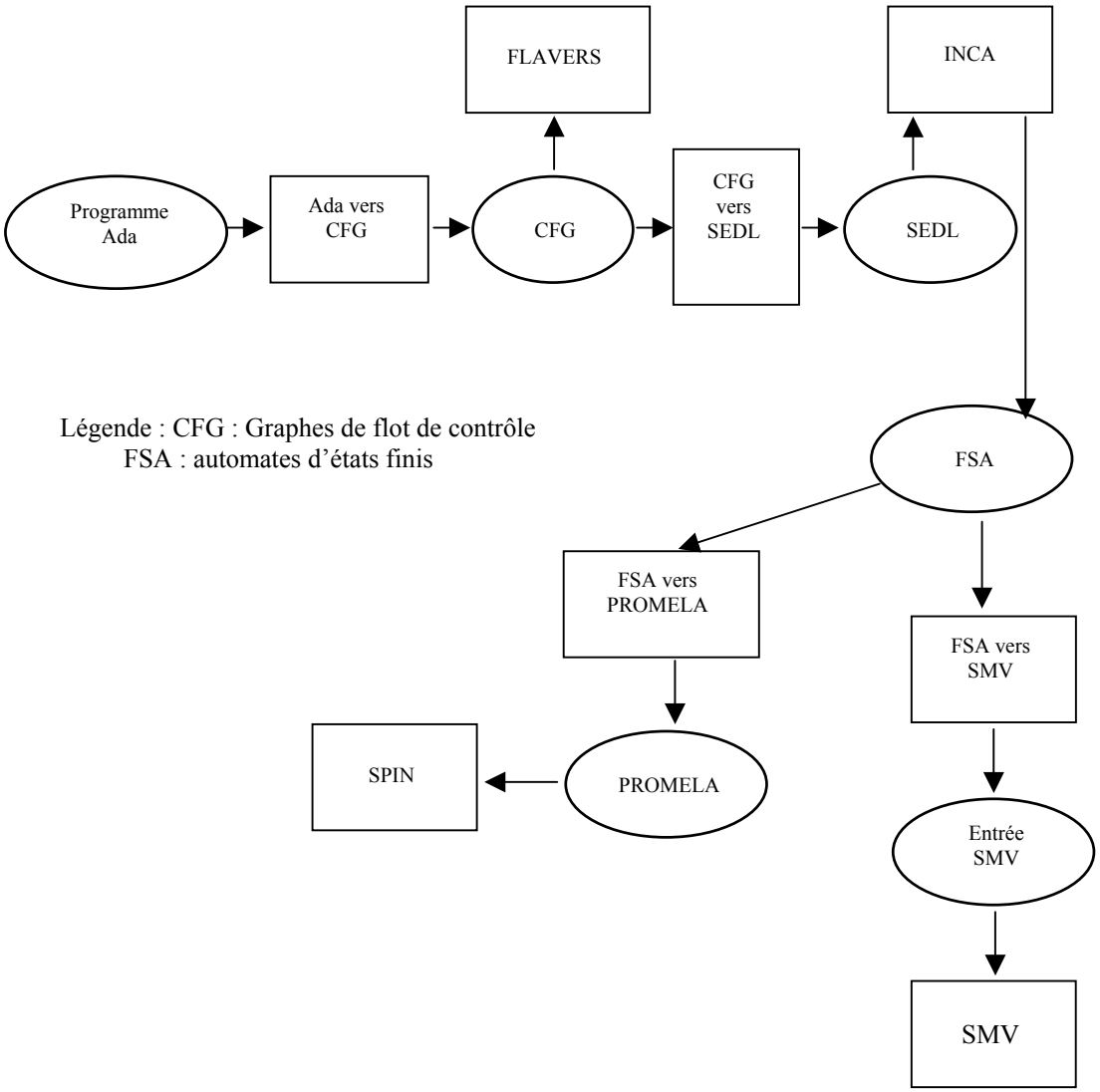


Figure 1.23. *Processus de traduction*

Les propriétés orientées état peuvent être vérifiées en considérant chaque état du système d'une façon isolée. Les propriétés orientées chemin exigent la considération d'un chemin d'exécution du programme souvent en termes d'événements survenus tout au long de ce chemin. Les propriétés orientées état couvrent plutôt des propriétés de sûreté et d'absence d'interblocage. Et les propriétés orientées chemin couvrent plutôt des propriétés de vivacité et d'équité. Toutes ces classes de propriétés sont nécessaires pour une analyse approfondie d'architectures logicielles.

3.3.6. Les traducteurs de la démarche DFVAL

La démarche DFVAL propose trois types de traducteurs.

- **Traducteurs de formalismes utilisés vers Wright**

Théoriquement, il doit y avoir autant des traducteurs que de formalismes utilisés pour la description d'architectures logicielles. Mais on peut réduire le nombre de traducteurs et la

complexité de cette opération de traduction en faisant appel aux langages intermédiaires. Dans la deuxième partie de cette thèse, nous proposons un profil W-UML qui facilite la traduction d'UML2.0 vers Wright et également de Symphony vers Wright. De même l'ADL Acme (Garlan, 1997a) (Garlan, 1997b) peut être utilisé avec profit comme langage intermédiaire entre les ADL et Wright.

- **Traducteur de Wright vers CSP**

Le traducteur Wr2fdr qui accompagne Wright permet de traduire de Wright vers CSP en automatisant quelques propriétés architecturales standards définies par Wright. Mais la version actuelle Wr2fdr comporte des erreurs et elle est limitée en possibilités (cf. chapitre 2, section 2.4.2). De plus, le CSP produit par Wr2fdr est destiné à l'outil FDR. Une action en cours est lancée pour apporter des améliorations importantes à cet outil.

- **Traducteur de Wright vers Ada**

Dans le chapitre suivant, nous proposons une approche de traduction de Wright vers Ada.

3.4 . Conclusion

Dans ce chapitre, nous avons établi une Démarche de Vérification Formelle d'Architectures Logicielles : démarche DVFAL. Celle-ci permet de traiter d'architectures logicielles décrites dans divers formalismes : ADL, UML2.0, des profils W-UML, Symphony, etc. La démarche DVFAL a retenu deux principaux langages : l'ADL Wright et Ada. L'ADL Wright joue le rôle d'un langage pivot permettant de représenter une architecture logicielle décrite dans divers formalismes. Quant au langage Ada, il offre la possibilité de représenter les architectures logicielles décrites en Wright comme des programmes Ada concurrents afin de les analyser davantage. En outre la démarche DVFAL propose divers outils de vérification supportant aussi bien CSP de Hoare qu'Ada. Enfin, elle réutilise et propose de nouveaux traducteurs. Le chapitre suivant préconise une approche permettant de traduire une architecture logicielle Wright en un programme Ada.

Chapitre 4 : Traduction d'une architecture logicielle Wright vers Ada

4.1 . Introduction

Les travaux permettant d'établir des connexions automatiques entre Wright et les outils de vérification ne sont pas nombreux. Les auteurs de l'article (Jeffrey, 1999) proposent une approche permettant de traduire les constructions Wright en réseaux de Petri. Ensuite, les réseaux de Petri sont traduits dans les langages d'entrée des outils comme SPIN et SMV. D'autres travaux (Naumovich, 1997) proposent une traduction manuelle de Wright vers Ada afin de comparer les outils de vérification FLAVERS et INCA. Dans ce chapitre, nous apportons une contribution permettant de traduire d'une façon systématique une architecture logicielle formalisée en Wright vers Ada. Notre contribution comporte un ensemble des règles permettant de traduire les constructions de Wright (configuration, composant, connecteur et processus CSP) en Ada. Ce chapitre comporte quatre sections. La première présente les concepts Ada susceptibles d'être utilisés pour traduire les concepts Wright. La deuxième est consacrée au concept rendez-vous de CSP afin de le situer par rapport au concept rendez-vous d'Ada. La troisième propose une approche de traduction de Wright vers Ada. Enfin, la dernière section donne un exemple de traduction d'une configuration Wright en programme concurrent Ada.

4.2 . La concurrence dans Ada

4.2.1 . Généralités sur Ada

Le langage Ada (Booch, 1991) (Le Verrand, 1982) a été conçu par l'équipe dirigée par Jean Ichbiah pour répondre aux besoins du Département de la Défense américain (DoD). Une révision d'Ada a conduit à la création d'un nouveau langage, appelé Ada 95 (De Bondeli, 1998) (Rousseau, 1994). Tout au long de ce chapitre, le nom Ada sans qualification supplémentaire fait référence à la version précédente, Ada 83, qui est la plus couramment utilisée de nos jours.

Le langage Ada offre des possibilités importantes vis-à-vis :

- De la programmation structurée : structuration de données (types prédéfinis et constructeurs de types simples et structurés) et structuration de traitements (structures de contrôle et les sous-programmes avec une distinction nette entre procédure et fonction). De plus, Ada est un langage fortement typé : il est plus sévère que Pascal!
- De la programmation modulaire en offrant un concept puissant, appelé package, doté de deux parties : interface (package) et implémentation (package body).
- De la programmation générique en offrant la possibilité de concevoir et de réaliser des unités génériques : sous-programmes et paquetages. Ces unités peuvent être paramétrées sur de types, variables et sous-programmes.
- De la gestion des exceptions. Ceci permet d'écrire des logiciels robustes dans divers domaines critiques : temps réel, systèmes embarqués.

4.2.2 . Les tâches en Ada

Un programme concurrent en Ada comporte plusieurs entités indépendantes qui coopèrent pour la résolution d'un problème concurrent (ou non séquentiel). Ces entités sont appelées tâches (**task**). En Ada, une tâche est une unité de programmation comportant deux parties : interface et implémentation. L'interface d'une tâche peut exporter des services appelés des entrées (**entry**). Ces entrées peuvent avoir des paramètres formels : in, out et in out. La partie implémentation d'une tâche Ada réalise ses fonctionnalités. Les services offerts par une tâche peuvent être utilisés par d'autres tâches moyennant le respect des règles de visibilité du langage Ada.

```
task serveur is                  task body serveur is          task t1
        entry prendre ;            begin                       task body t1 is
        entry liberer ;              loop                      begin
end serveur                              accept prendre ;            …
                                         accept liberer ;          serveur.prendre ;
                                       end loop                      …
                                     end serveur                   serveur.liberer ;
                                                                     …
                                                                 end t1
        (a)                              (b)                           (c)
```

Figure 1.24. *(a) Interface serveur ; (b) Implémentation serveur ; (c) Utilisation*

Les entrées exportées par une tâche Ada indiquent des possibilités des rendez-vous (cf. 4.2.3). Par exemple la tâche *serveur* propose deux types de rendez-vous : *prendre* et *liberer* (cf. figure 1.24 partie (a)). Ada fournit deux instructions permettant l'acceptation d'un rendez-vous (*accept* cf. figure 1.24 partie (b)) et la demande d'un rendez-vous (cf. figure 1.24 partie (c)). L'instruction *accept* a la forme générale suivante :

accept nom_entree (paramètres formels) ;

La demande d'un rendez-vous a la forme générale suivante :

tache_appelee.nom_entree (paramètres effectifs) ;

En Ada, une tâche accepte des rendez-vous sur ces entrées venant de n'importe quelle tâche : la tâche appelante est anonyme. Mais pour pouvoir demander un rendez-vous, la tâche appelante doit nommer **explicitement** la tâche appelée. Un tel principe régissant l'acception et la demande d'un rendez-vous est connu sous le nom de **désignation asymétrique**. Une tâche Ada ne peut traiter à la fois qu'un seul rendez-vous. On dit qu'il y a exclusion mutuelle sur les rendez-vous. Chaque entrée (ou type de rendez-vous) proposée par une tâche est dotée d'une file d'attente FIFO (Structure de Données First In First Out) gérée par l'exécutif d'Ada. Cette file FIFO permet de stoker les demandes sur cette entrée dans l'ordre d'arrivée.

Sur le plan conceptuel, les tâches peuvent être classées en trois catégories :

- Les tâches serveuses qui proposent des rendez-vous et n'en demandent pas : par exemple la tâche *serveur* (cf. figure 1.24).
- Les tâches actrices qui ne proposent pas des rendez-vous. Mais elles en demandent : par exemple la tâche t1 (cf. figure 1.24).
- Les tâches actrices/serveuses qui proposent et demandent des rendez-vous.

4.2.3 . Mécanisme de rendez-vous simple

Le mécanisme de rendez-vous simple est décrit ci-dessous (Booch, 1991).

Il y a rendez-vous entre une tâche APPELANTE et une tâche APPELEE, lorsque la tâche APPELANTE appelle une ENTREE de la tâche APPELEE et que cette dernière accepte cet appel. Le rendez-vous est terminé lorsque les actions associées à une instruction spécifique *accept* de la tâche APPELEE ont été exécutées. Les deux tâches reprennent alors leur exécution respective, indépendamment l'une de l'autre.

Le mécanisme de rendez-vous Ada permet à la fois la synchronisation et la communication, nécessaires à la solution des problèmes de compétition et de coopération entre tâches. La communication entre tâches est réalisée par la transmission de paramètres typés dans les deux sens (in, out et in out), lors de l'occurrence d'un rendez-vous. La figure 1.25 illustre l'utilisation exclusive (variable entière encapsulée par la tâche serveuse variable_protegee) d'une ressource critique par deux tâches actrices t1 et t2.

```
task variable_protegee is
        entry lire (v : out integer);
        entry ecrire (v : in integer);
end variable_protegee
task body variable_protegee is
   x : = integer : = 0;
  begin
   loop
      select
        accept lire (v : out integer) do
                v :=x ;
         end lire ;
       or
        accept ecrire (v : in integer) do
                x :=v ;
        end ecrire ;
        or
           terminate ;
      end select ;
     end loop ;
end variable_protegee ;
```

```
task t1
task t2
 task body t1 is
            a : integer ;
   begin
    …
    variable_protegee.ecrire(a);
     …
 end t1
 task body t2 is
  b: integer;
    begin
      …
     variable_protegee.lire(b);
       …
 end t2;
```

Figure 1.25. *Utilisation d'une variable protégée*

La figure 1.26 issue de (Zaffalon, 1999) illustre deux cas. Dans le cas a) la tâche APPELEE est prête après que la tâche APPELANTE ait effectué une demande de rendez-vous sur une entrée. Alors que dans le cas b) c'est la tâche APPELEE qui est prête avant la demande de rendez-vous. Lors d'un appel, si la tâche APPELEE attend sur l'instruction *accept*, le rendez-vous a lieu immédiatement, sinon, la demande de la tâche APPELANTE est mémorisée dans une file FIFO et la tâche est mise en attente passive de l'acceptation du rendez-vous. Ainsi, la forme de communication entre les tâches Ada est donc **bloquante**. La durée du rendez-vous est la durée nécessaire à l'exécution des instructions attachées à l'entrée :

accept nom_entree do

<Instructions exécutées>

end nom_entree ;

Le langage Ada autorise l'acceptation des rendez-vous imbriqués : *accept* englobant et *accept* englobé.

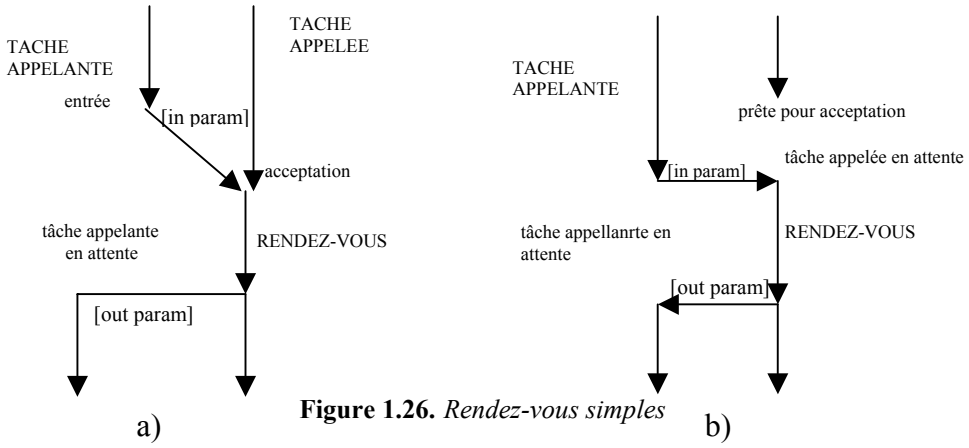


Figure 1.26. *Rendez-vous simples*

4.2.4. Le non déterminisme

Pour gérer le non déterminisme, le langage Ada dispose d'une instruction adéquate, l'instruction d'attente sélective **select**. Cette instruction permet à la tâche appelée de sélectionner une demande de rendez-vous parmi plusieurs branches possibles :

```
select
        accept ENTREE_A;
        or
        accept ENTREE_B;
        or
        accept ENTREE_C;
        -- etc.
end select;
```

Le choix du rendez-vous est **arbitraire**. Tout algorithme qui suppose un choix particulier est erroné. Par exemple, la tâche variable_protegee (cf. figure 1.25) accepte les rendez-vous selon l'ordre {lire | ecrire}*. Le dernier appel peut être soit *lire* soit *ecrire*.

Les différentes branches formant l'instruction select peuvent être gardées. Une garde est une expression de type booléen et par défaut cette garde est vide (true).

- **Exemple d'instruction select** (Le Verrand, 1982)

```
select
        when not occupe ⇒           -- garde
                accept prendre do   -- branche de rendez-vous
                                occupe := true;
                end prendre;        -- fin du rendez-vous
                x := x+ 1;          -- fin de la branche de rendez-vous
                or
                accept prendre do   -- branche de rendez-vous
                        occupe := false; --dépourvue de garde
                end prendre;
end select ;
```

L'instruction select peut comporter une branche de terminaison réduite au mot réservé **terminate**. La tâche qui exécute l'instruction terminate passe dans un état terminé : elle

disparaît. Cette instruction est très utile pour la terminaison implicite d'une tâche serveuse : lorsque son environnement (formé des autres tâches) n'est pas susceptible de la solliciter, la tâche serveuse n'est plus utile et par conséquent elle doit disparaître.

4.2.5. Les types tâches

A l'instar des types de données usuels ou des types abstraits, il est possible de déclarer des types de tâches (task type). Les types tâches en Ada sont considérés comme **limited private** : seule la transmission est permise. Les affectations et les comparaisons des tâches ne sont point permises. Les types tâches permettent de factoriser plusieurs tâches ayant le même comportement. De plus, ils ouvrent la perspective de créer dynamiquement des tâches.

- **Exemple de type tâche**

```
task type allocateur is                -- type de tâche et non une tâche
        entry prendre;
        entry liberer;
end allocateur;
task body allocateur is
begin
        loop
                select
                        accept prendre;
                        accept liberer;
                        or
                        terminate;
                end select;
        end loop;
end allocateur;
-- définition statique des tâches
ecran : allocateur;
imprimante : allocateur;
```

Les deux tâches serveuses *ecran* et *imprimante* permettent l'accès exclusif respectivement aux deux ressources critiques écran et imprimante. Il est aussi possible de créer dynamiquement des instances de types allocateur :

```
-- type pointeur
type ptr_allocateur is access allocateur;
-- variable de type pointeur
mutex : ptr_allocateur;
-- création dynamique d'une tâche
mutex := new ptr_allocateur;
```

4.2.6. Le langage Ada 95

Le langage Ada 95 (De Bondeli, 1998) (Rousseau, 1994) offre des nouvelles fonctionnalités liées : à la programmation par objets, à l'organisation des bibliothèques des packages, au temps réel et à la concurrence. En ce qui concerne la concurrence Ada 95 fournit une nouvelle primitive de synchronisation/communication appelée types protégés (protected type) qui sont des moniteurs (Zaffalon, 1999) à l'Ada. Les types protégés permettent notamment l'implémentation efficace des tâches serveuses comme des éléments passifs : ils n'entrent pas en compétition avec les autres tâches pour obtenir le processeur.

- **Exemple de type protégé**

La tâche variable_protegee (cf. figure 1.25) peut être définie en Ada 95 par :

```
-- interface
protected type variable_protegee (valeur_initiale : integer := 0) is
        function lecture return integer;
        procedure modifier (x :  in integer);
private
        v : integer := valeur_initiale;
end variable_protegee;
-- implémentation
protected body variable_protegee is
        function lecture return integer is
        begin
                        return v;
        end lecture;
        procedure modifier (x :  in integer) is
        begin
                        v := x;
        end modifier;
end variable_protegee;
-- utilisation
a : variable_protegee; -- valeur initiale par défaut
b : variable_protegee (10); -- valeur initiale 10
```

L'exécutif d'Ada 95 permet de doter chaque objet protégé (ici a et b) de deux verrous. Sachant qu'un verrou peut être soit ouvert, soit fermé. Un verrou assure l'accès exclusif des opérations de modification (procedure et entry) et des opérations de consultation (function). Et l'autre assure l'accès simultané des opérations de consultation et l'exclusion des opérations de modification.

4.3 . Les rendez-vous CSP

Le modèle CSP précise que la communication entre processus se fait **uniquement** via des échanges de messages. Ces échanges ne peuvent être réalisés qu'à l'aide de deux événements spéciaux appelés encore commandes : émission et réception. Une émission ou sortie est décrite par :

```
p! constr (expr)
```

et une réception ou entrée :

```
q?var
```

où, expr est une expression quelconque dont la valeur est le message transmis, constr est une étiquette ou identité de constructeur de type et var est la variable cible à qui est assignée la valeur du message reçu. L'échange entre deux processus CSP exige les conditions suivantes (Zaffalon, 1999) :

- une commande d'entrée dans un processus spécifie comme source le nom de l'autre processus ;
- une commande de sortie dans l'autre processus spécifie comme destination le premier processus ;

et

- le type de la variable cible, dans la commande d'entrée, correspond (compatible) à la valeur dénotée par l'expression de la commande de sortie.

Si ces trois conditions sont réalisées, les deux commandes sont exécutées en **même temps** et la valeur de l'expression émise et affectée à la variable de réception. Dans le cas où un des deux processus n'est pas encore prêt à exécuter sa commande d'entrée, respectivement de sortie, l'autre processus doit attendre jusqu'à ce que cela soit le cas : c'est le principe du **rendez-vous** strict. Ainsi, les primitives d'entrée-sortie en CSP sont **bloquantes**. Ce qui signifie que le mécanisme est de type **synchrone** (cf. figure 1.27).

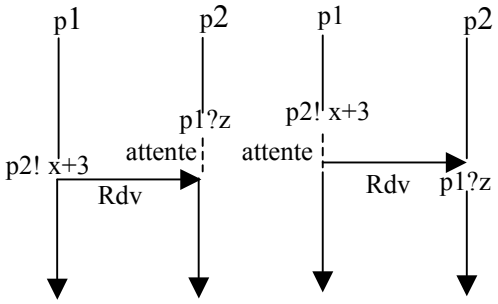


Figure 1.27. *Principe du rendez-vous CSP*

Lors de l'échange, les deux processus concernés se désignent mutuellement. C'est le principe **de désignation symétrique**.

Le langage CSP a fortement influencé la conception des langages de programmation tels que Occam et Ada. Le langage Occam (Occam, 2000) est une implémentation directe de CSP. Quant à Ada, il en reprend le concept de rendez-vous, pour la mise en œuvre de la communication et la synchronisation entre tâches. Ce rapprochement sémantique entre CSP et Ada sera exploité pour faciliter la traduction entre Wright dont la sémantique est basée sur CSP et Ada.

4.4 . De Wright à Ada

Dans cette section, nous proposons une approche de traduction de Wright vers Ada. Nous suivons une démarche descendante pour présenter le processus de traduction de Wright vers Ada.

4.4.1 . Traduction des configurations

- **Traduction 4.1 (Configuration)**

Wright

```
Configuration ClientServeur
        Component Client
                …
        Component Serveur
                …
        Connector CS
                …
        Instances
                c : Client
                s : Serveur
                cls : CS
        Attachements
                …
end Configuration
```

Ada

```
procedure ClientServeur is
  task c is
  …
  end c;
  task s is
  …
  end s;
  task cls is
  …
  end cls;

  task body c is
  …
  end c;
  task body s is
  …
  end s;
  task body cls is
  …
  end cls;
  begin
  null;
end ClientServeur
```

Figure 1.28. *Traduction d'une configuration Wright*

Une configuration Wright est traduite en Ada par un programme concurrent dans lequel :

- Chaque instance de type Composant est traduite par une tâche Ada.
- Chaque instance de type Connecteur est traduite également par une tâche Ada.
- Les tâches de même type ne communiquent pas entre elles.

La figure 1.28 illustre le principe de la traduction d'une configuration Wright en Ada. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification Wright. La traduction proposée possède un avantage majeur. Elle permet de conserver la sémantique d'une configuration Wright. En effet, celle-ci est définie formellement en CSP (cf. chapitre 1 section 1.3) comme la composition parallèle des processus modélisant les composants et les connecteurs formant cette configuration. Et un programme concurrent en Ada peut être modélisé en CSP comme la composition parallèle des tâches formant ce programme.

- **Remarque**

La configuration ClientServeur (cf. figure1.28) comporte 3 instances (deux composants et un connecteur). Le programme concurrent Ada équivalant comporte 4 tâches. Les trois tâches c, s et cls traduisent leurs équivalents en Wright et la tâche ClientServeur correspond au programme principal d'Ada (procedure). Mais cette tâche en plus est considérée comme une structure d'accueil et elle ne fait rien (null).

4.4.2 . Traduction des événements

Nous distinguons les deux cas suivants :

- un événement observé de la forme e?x ou e
- une émission ou encore événement initialisé de la forme $\overline{e!x}$ ou $\overline{e}$

- **Traduction 4.2 (Evénement observé)**

Un événement observé de la forme e?x est traduit par une entrée (**entry**) et par une acceptation de rendez-vous (instruction **accept**).

La figure 1.29 illustre le principe de la traduction d'une réception CSP en Ada.

```
Component Client
Port appelant = request!x -> result?y -> appelant |~|§
        ….
Instances
c : Client
                a) Spécification Wright

task c is
        entry result(y: in message);
end c;
task body c is
begin
        …
        accept result (y: in message) do
…
        end result ;
        …
end c ;
                b) Code Ada
```

Figure 1.29. *Traduction d'une réception*

- **Remarques**

- Les données portées par les événements en CSP de Wright ne sont pas typées explicitement. En Ada, les données sont typées explicitement. Nous supposons l'existence d'un type Message défini comme suit : type Message is…
- Les échanges de données entre deux processus CSP sont unidirectionnels. Donc lors d'une réception, le paramètre x doit être de nature **in**.

- **Traduction 4.3 (Evénement initialisé)**

Un événement initialisé de la forme $\overline{e!x}$ est traduit par une demande de rendez-vous sur l'entrée e exportée par une tâche de type différent (seules les tâches de types différents communiquent) à identifier. Pour y parvenir, il faut analyser la partie Attachments de la configuration.

La figure 1.30 illustre le principe de la traduction d'une émission.

```
Component Client
        Port appelant = request!x -> result?y -> appelant |~|§
                ….
        Connector cs
                Role client = request!x -> result?y-> client |~|§
                Role serveur = request?x  -> result!y -> serveur □ §
        Instances
                c : Client
                cls: cs
        Attachements
                Client. appelant as cls.client
                                ....
```

a) Spécification Wright

```
        task c is
                entry result(y: in Message);
        end c;
        task  cls is
                entry result(y: in Message);
                entry request(y: in Message);
        end cls;
        task  body c is
                 x : Message;
        begin
                cls.request(x);
        end c;
```

b) Code en Ada

Figure 1.30. *Traduction d'une émission*

4.4.3 . Traduction des interfaces des composants

L'interface d'un composant Wright comporte un ensemble de ports. Chaque port contient deux ensembles d'événements observés et initialisés.

- **Traduction 4.4 (Interface d'un composant)**

L'interface d'un composant Wright est traduite par une interface d'une tâche Ada. Cette interface est obtenue de la manière suivante :

pour chaque port ∈ au composant Wright
faire
pour chaque événement ∈ au port
faire
si événement est un événement observé de la forme e[?x]

alors
créer une entrée ayant le nom suivant : port_e
finsi
finfaire
finfaire

La figure 1.31 illustre le principe de la traduction de l'interface d'un composant Wright.

Component Client

Port appelant = $\overline{\text{request!x}}$-> result?y-> appelant |~|§
….
Instances
c : Client

a) Spécification Wright

task c **is**
entry appelant_result(y: in Message);
end c;

b) Code en Ada

Figure 1.31. *Traduction d'une interface composant*

4.4.4 . Traduction des interfaces des connecteurs

L'interface d'un connecteur Wright comporte un ensemble de rôles. Chaque rôle contient deux ensembles d'événements observés et initialisés.

- **Traduction 4.5 (Interface d'un connecteur)**

L'interface d'un connecteur Wright est traduite par une interface d'une tâche Ada. Cette interface est obtenue de la manière suivante :

pour chaque rôle ∈ au connecteur Wright
faire
pour chaque événement ∈ au rôle
faire
si événement est un événement initialisé de la forme $\overline{e[!x]}$
alors
créer une entrée ayant le nom suivant : rôle _e
finsi
finfaire
finfaire

Connecteur CS
Role client = $\overline{\text{request!x}}$-> result?y-> client |~|§
Role serveur = request?x -> $\overline{\text{result!y}}$->serveur □ §
…
Instances
cls: CS

a) Spécification Wright

task cls **is**
entry client_request(x: in Message);
entry serveur_result(y: in Message);
end cls;

b) Code en Ada

Figure 1.32. *Traduction d'une interface connecteur*

La figure 1.32 illustre le principe de la traduction de l'interface d'un connecteur Wright.

4.4.5. De CSP Wright à Ada

Dans ce paragraphe, nous proposons des règles permettant de traduire en Ada les opérateurs CSP couramment utilisés en Wright.

- **Traduction 4.6 (Opérateur de préfixage ->)**

On distingue deux cas :

Cas 1 : a[?x] -> P

La traduction en Ada est :

```
accept a(…) do
null ;
end a ;
traiter P
```

Cas 2 : $\overline{a[!x]}$ -> P

La traduction en Ada est :

```
nom_ tache.a(…);
traiter P
```

Dans les deux cas l'action *traiter P* sera effectuée (ou exécutée) après avoir exécuté l'événement a. Ceci correspond à la sémantique de l'opérateur a -> P qui commence par réaliser l'événement x puis se comporte exactement comme P.

- **Traduction 4.7 (Opérateur de récursion)**

La récursion en CSP permet la description des entités qui continueront d'agir et d'interagir avec leur environnement aussi longtemps qu'il le faudra. On distingue les trois cas suivants :

Cas 1 : P= a[?x] -> Q -> P

La traduction en Ada est basée sur une boucle infinie :

```
loop
    accept a(…) do
        null ;
    end a;
    traiter Q
end loop;
```

Cas 2 : P= $\overline{a[!x]}$ -> Q -> P

La traduction en Ada est également basée sur une boucle infinie :

```
loop
    nom_ tache.a(…);
    traiter Q
end loop;
```

Cas 3 : P= a -> Q->P ∏ §

La traduction en Ada est :

```
loop
        exit when condition_interne ;
        accept a(…) do null; end a; ou nom_tache a(…);
        traiter Q
end loop;
```

- **Traduction 4.8 (Opérateur choix non déterministe)**

La notation P ∏ Q dénote un processus qui se comporte soit comme P soit comme Q, la sélection étant réalisée de façon arbitraire, hors du contrôle ou de la connaissance de l'environnement extérieur. En CSP, on a l'égalité suivante : a -> (P∏ Q) = (a-> p) ∏(a-> Q). On distingue les cas suivant :

Cas 1 : (a[?x] -> P) ∏ (a[?x] -> Q)

La traduction en Ada est basée sur la construction non déterministe select :

```
select
        accept a(…) do
                null ;
        end a;
        traiter P;
or
        accept a(…) do
                null ;
        end a;
        traiter Q;
end select;
```

Cas 2 : (a[?x] -> P) ∏ (b[?x] -> Q) avec a ≠ b

La traduction est:

```
select
        accept a(…) do
                null ;
        end a;
        traiter P;
 or
accept b(…) do
                null ;
        end b;
        traiter Q;
end select;
```

Cas 3 : $\overline{a[!x]}$ -> P ∏ §

Toute traduction qui pourrait exécuter ou non $\overline{a[!x]}$ -> P est acceptable. Par exemple :

```
if condition_interne then
        nom_tache.a(…);
        traiter P;
 else
        stop
end if;
```

Stop est une procédure permettant de mettre fin à l'exécution du processus courant.

Cas 4 : $\overline{a[!x]}$ -> P ∏ $\overline{b[!x]}$ -> Q

La traduction est basée sur un schéma conditionnel :

```
if condition_interne then
        nom_tache.a(…);
        traiter P;
 else
        nom_tache.b(…);
        traiter Q;
end if;
```

Cas 5 : $\overline{a[!x]}$ -> P ∏ b[?x] -> Q

La traduction est basée sur un schéma conditionnel :

```
if condition_interne then
        nom_tache.a(…);
        traiter P;
else
        accept b(…) do
        null;
        end b;
        traiter Q;
end if;
```

Cas 6 : $\overline{a[!x]}$ -> P ∏ $\overline{a[!x]}$ -> Q

La traduction est basée sur un shéma conditionnel :

```
        nom_tache.a(…);
if condition_interne then
        traiter P;
else
        traiter Q;
end if;
```

- **Traduction 4.9 (choix déterministe □)**

Le processus P □ Q introduit une opération par laquelle l'environnement peut contrôler celui de P ou de Q qui sera sélectionné, étant entendu que ce contrôle s'exerce sur la toute première action ou événement. On distingue les cas suivants :

Cas 1 : (a-> P □ a -> Q)

Un tel cas est équivalent à (a-> P ∏ a -> Q) et par conséquent on peut utiliser les traductions fournies en 4.8 : cas 1 et cas 6.

Cas 2 : a[?x] -> P □ b[?x] -> Q avec a ≠ b

Un tel cas est traduit à base de la construction non déterministe select :

```
select
        accept a(…) do
                null ;
        end a;
        traiter P;
 or
```

```
        accept b(…) do
                        null ;
                end b;
                traiter Q;
        end select;
```

4.4.6 . Traduction de la partie calcul des composants

- **Traduction 4.10 (Computation)**

La partie Computation d'un composant Wright est traduite en un corps (partie implémentation) d'une tâche Ada conformément aux règles de traduction des opérateurs CSP établis dans 4.4.5.

4.4.7 . Traduction de la partie Glu des connecteurs

- **Traduction 4.11 (Glu)**

La partie « Glue » d'un connecteur Wright est traduite en un corps (partie implémentation) d'une tâche Ada conformément aux règles de traduction des opérateurs CSP établis dans 4.4.5.

4.5 . Exemple

Ici, nous donnons une spécification Wright d'une application de type Client/Serveur et sa traduction en Ada.

4.5.1 . Spécification Wright

```
Configuration client_serveur
        Component client
                port appelant = request!x -> result?y -> appelant |~|§
                computation = appelant.request!x -> appelant.result?y -> computation
                |~|§
        Component serveur
                port  appele = request?x -> result!y -> appele □ §
                computation = appele.request?x -> appele.result!y -> Computation□ §
         Connector cs
                role client = request!x->result?y -> client |~| §
                role serveur = request?x -> result!y -> serveur □  §
                glue = client.request?x -> seveur.request !x -> serveur.result?y ->
                client.result!y -> glue □  §
        Instances
                c : client
                s : serveur
                cls: cs
        Attachments
                c.appelant as cls.client
                s.appele as cls.serveur
end Configuration
```

4.5.2 . Traduction en Ada

Voici le programme concurrent Ada correspondant :

```
Procedure client_serveur is
        --type des messages échangés
                type message is … --à compléter
        --les interfaces des tâches
        task c is
                entry appelant_result(y: in message);
        end c;

        task s is
                entry appele_request(x: in message);
        end s;
        task  cls is
                entry client_request(x: in message);
                entry serveur_result(y: in message);
        end cls;
        --les corps des tâches
        task body c is
                x : message ;
              do
                loop
                        exit when condition_interne;
                        cls.client_request(x);
                        accept appelant_result(y :in message) do
                                null ;
                        end appelant_result ;
                end loop ;
        end c;
        task body s is
                y: message;
        begin
                loop
                  select
                        accept appele_request(x : in message) do
                                null ;
                        end appele_request ;
                        cls .serveur_result(y) ;
                     or
                        terminate ;
                  end select ;
                end loop ;
         task body cls is
                y: message;
         begin
                loop
                  select
                        accept client_request(x : in message) do
                        null ;
                        end client_request ;
                                s.appele_request(y) ;
        accept serveur_result(y : in message) do
                                null ;
                        end serveur_result ;
```

```
                        c.appelant_result(y);
                    or

                        terminate;
                end select;
        end loop;
        end cls;
begin
                null;
end client_serveur;
```

- **Explications**

1) L'exécution d'un événement initialisé est traduite par une demande de rendez-vous. L'identité de la tâche appelée est déduite à partir de la partie «attachments» de la configuration Wright. De même le nom de l'entrée à appeler (entry) est déduit en partie de la même section. Par exemple, l'événement initialisé appelant.request!x, figurant dans la partie Computation du composant c de type client dont le port appelant est attaché au rôle client du connecteur cls de type cs (c.appelant as cls.client), est traduit en Ada de la manière suivante :

- appelant.request!x est renommé client.request!x
- client.request!x est renommé client_request!x pour respecter la définition d'un identificateur Ada
- client_request!x est traduit comme une demande de rendez-vous : cls.client_request(x) ;

2) Le programme concurrent Ada correspondant à la spécification Wright peut être compilé moyennant des modifications mineures : compléter la définition de type message et l'implémentation du prédicat condition_interne.

Par exemple :

type message is new integer ; --les messages à échanger sont de type entier

exit when false ; --la tâche c ne s'arrête jamais

Dans une implémentation future de notre traducteur de Wright vers Ada, nous pourrions retenir ces deux choix comme des choix par défaut.

3) Le programme concurrent Ada obtenu peut être soumis aux divers outils de vérification des programmes concurrents an Ada (cf. chapitre 3 section 3.3.5). Ensuite, après vérification des propriétés spécifiques, ce programme va constituer le point de départ d'un processus incrémental avec preuve d'implémentation de l'architecture logicielle.

4.6 . Conclusion

Dans ce chapitre, nous avons proposé un ensemble de règles permettant de traduire des constructions Wright vers Ada. Sur le plan macroscopique, une configuration Wright est considérée comme un programme concurrent en Ada. Ceci autorise l'utilisation de nombreux et divers outils de vérification des programmes concurrents supportant Ada tels

que : SPIN, SMV, FLAVERS et INCA (cf. chapitre 3 section 3.3.5). En outre, le programme concurrent issu de la configuration Wright peut être amélioré d'une façon incrémentale et sûre -grâce aux outils de vérification- jusqu'à l'implémentation correcte de l'architecture logicielle.

Deuxième partie : Vérification formelle d’architectures logicielles à base d’UML

Chapitre 5 : Un Métamodèle pour l'ADL Wright

5.1 . Introduction

Le document de base de Wright (Allen, 1996) (Allen, 1997b) comporte une description syntaxique en BNF. Celle-ci est peu exploitable dans une approche orientée **modèle**. Nous avons donc décidé d'élaborer un métamodèle Wright (syntaxe abstraite) représentant la plupart des concepts issus de ce langage. Ce métamodèle permet, dans notre contexte, de cerner d'une façon semi-formelle les concepts Wright à modéliser en UML2.0.

Pour élaborer le métamodèle Wright souhaité, nous avons suivi une démarche comportant les étapes suivantes :

- La première étape a pour objectif de traiter les aspects structuraux de Wright : component, connector et configuration.
- La deuxième étape a pour objectif de traiter les aspects comportementaux de Wright : le langage CSP pour Wright.
- Enfin, lors de la troisième étape, nous relions les deux aspects structuraux et comportementaux de Wright pour obtenir le métamodèle que nous utilisons par la suite.

5.2 . Aspects structuraux de Wright

Wright repose sur les abstractions architecturales de base qui sont les **composants**, les **connecteurs** et la **configuration**.

5.2.1 . Le concept Composant

Un composant Wright est une unité abstraite localisée et indépendante. La description d'un composant contient deux parties importantes, l'**interface** et la partie **calcul** (computation).

- L'interface consiste en un ensemble de ports, chacun représente une interaction avec l'extérieur à laquelle le composant peut participer. Chaque port décrit deux aspects : le comportement partiel attendu du composant (services offerts par le composant via ce port) et ce que le composant attend du système (services requis).
- La partie calcul, quant à elle, consiste à décrire ce que le composant fait du point de vue comportemental, en indiquant comment celui-ci utilise les ports. Ainsi, les ports qui sont décrits indépendamment dans l'interface, sont utilisés pour décrire le comportement du composant dans le calcul.

La figure 2.1 illustre le métamodèle du concept composant de Wright selon une vue structurelle.

Ce métamodèle, représenté par un diagramme de classes, est composé de trois métaclasses : **Component**, **Port** et **Computation**.

Component et **Port** contiennent chacune un méta-attribut nommé name, dont le type est une chaîne de caractères.

Il existe deux méta-associations composition, qui ont pour source la métaclasse Component et pour cible les deux métaclasses : **Port**, avec multiplicité un ou plusieurs, et **Computation**, avec multiplicité un. Ceci explique qu'un composant peut avoir un ou plusieurs ports et une et une seule partie calcul ou computation.

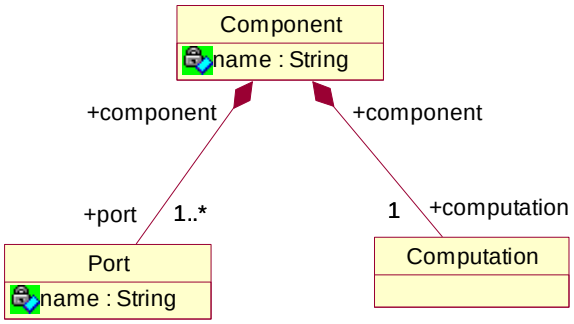


Figure 2.1. *Métamodèle Component de l'ADL Wright*

La figure 2.2 décrit un composant appelé Titulaire selon la syntaxe concrète de Wright.

Component Titulaire

Port consultation = …

Computation = …

Figure 2.2. *Composant « Titulaire » selon la syntaxe concrète de Wright*

La figure 2.3 donne, sous forme d'un diagramme d'objets, le composant "Titulaire" considéré comme un modèle conforme au métamodèle Component.

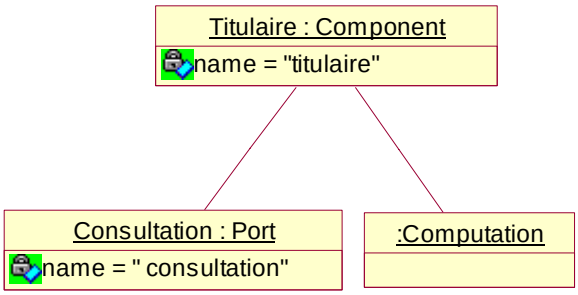


Figure 2.3. *Modèle objet du composant "Titulaire"*

5.2.2 . Le concept Connecteur

Un connecteur représente une interaction entre une collection de composants. Il possède un type et contient deux parties qui sont un ensemble de rôles et la glu :

- Chaque rôle décrit les attentes du connecteur dans ses interactions avec les composants.
- La glu décrit comment les rôles travaillent ensemble pour créer cette interaction.

La figure 2.4 illustre le métamodèle du concept connector de l'ADL Wright selon la définition citée précédemment. Ce métamodèle est composé de trois métaclasses. Les métaclasses **Connector** et **Role** contiennent un méta-attribut appelé name de type chaîne de caractères et la métaclasse **Glue**.

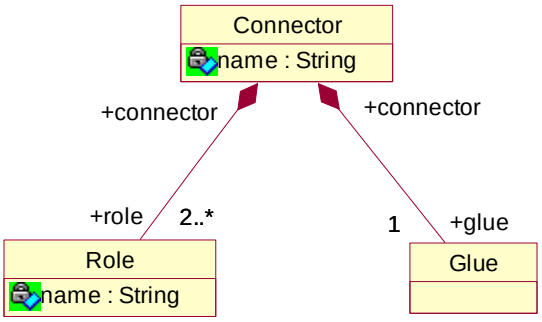


Figure 2.4. *Métamodèle Connector de l'ADL Wright*

Il existe deux méta-associations composition, qui ont pour source la métaclasse Connector et pour cible les métaclasses Role et Glue.

La figure 2.5 décrit un connecteur appelé CS (Client-Serveur) selon la syntaxe concrète de Wright.

Connector cs

Role client= …

Role serveur = ….

Glue = …

Figure 2.5. *Connecteur « CS » selon la syntaxe concrète de Wright*

La figure 2.6 donne, sous forme d'un diagramme d'objets, le connecteur "cs" considéré comme un modèle conforme au métamodèle Connector.

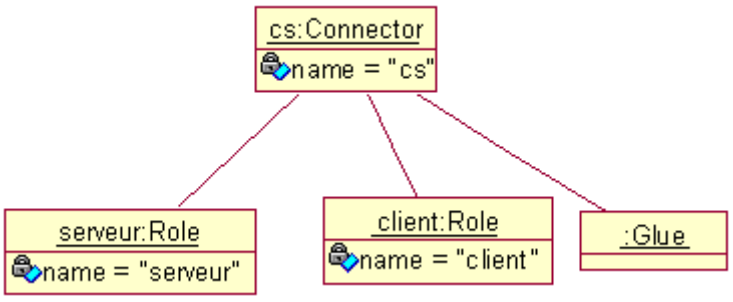


Figure 2.6. *Modèle objet du connecteur "CS"*

5.2.3 . Le concept Configuration

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des composants et des connecteurs utilisés

dans l'architecture, la déclaration des instances de composants et de connecteurs, les descriptions des liens (ou attachements) entre les instances de composants et les instances de connecteurs (Sanlaville, 1997).

Dans la suite, nous décrivons les différentes sections formant une configuration Wright.

5.2.3.1. Les types de composants et connecteurs

Une configuration doit en premier lieu déclarer les **types** de **composants** et **connecteurs** qu'elle va utiliser. Pour cela, elle utilise les conventions mentionnées auparavant.

5.2.3.2. Les instances

Pour pouvoir utiliser des types de composants et de connecteurs dans la création d'un système spécifique, l'architecte doit d'abord définir et nommer un ensemble **d'instances** de ces types.

5.2.3.3. Les liens

Une fois que ces instances de composants et de connecteurs ont été déclarées, une configuration est créée en décrivant un ensemble de **liens** entre les différents **ports** et **rôles** de ces **instances**. Ces liens décrivent la topologie du système en indiquant quel composant participe à quelle interaction.

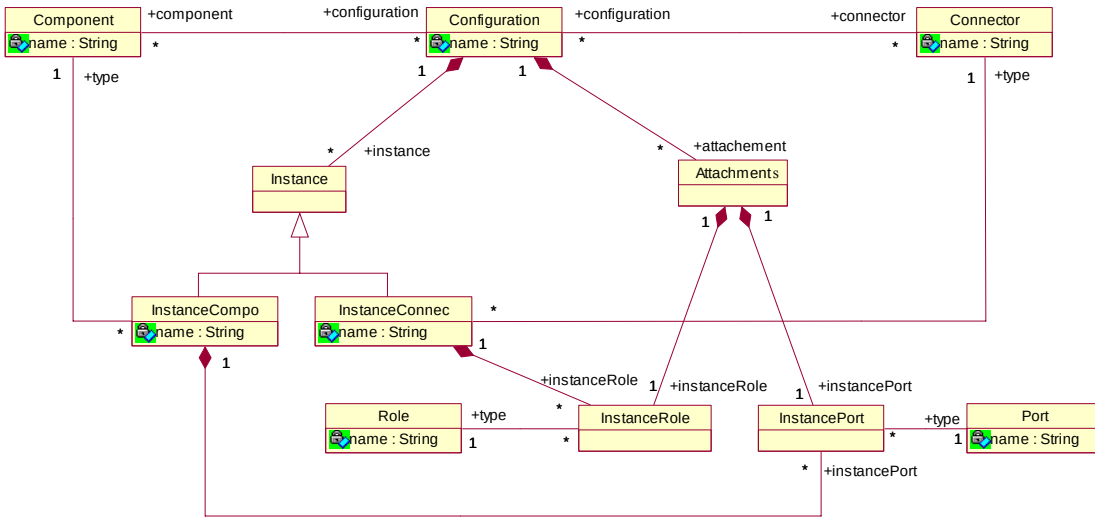


Figure 2.7. *Métamodèle Configuration de l'ADL Wright*

La figure 2.7 illustre le métamodèle du concept configuration de l'ADL Wright. Le diagramme de classes de ce métamodèle est composé de onze métaclasses.

D'après la définition semi-formelle du concept configuration, il existe deux méta-associations navigables, qui ont pour source la métaclasse **Configuration** et pour cible les deux métaclasses Component et Connector. Ceci traduit le fait qu'une configuration Wright peut utiliser plusieurs types de composants et de connecteurs.

Et puisqu'on peut avoir plusieurs instances de composants et ou de connecteurs, et plusieurs attachements ou liens, il existe dans notre diagramme de classes deux méta-associations composition, qui ont pour source la métaclasse Configuration et pour cible les deux métaclasses **Instance** et **Attachements**. Ceci traduit le fait qu'en Wright, les instances des composants et des connecteurs sont **propres** à une configuration donnée et également pour les attachements.

Il existe deux métaclasses qui héritent de la métaclasse **Instance** : **InstanceComp** et **InstanceConnec**. **InstanceComp** est reliée par une méta-association navigable avec la métaclasse Component, et par une méta-association composition avec la métaclasse **InstancePort**. **InstanceConnec** est reliée par une méta-association navigable avec la métaclasse Connector, et par une méta-association composition avec la métaclasse **InstanceRole**.

La figure 2.8 décrit une configuration appelée AppBancaire selon la syntaxe concrète de Wright.

```
Configuration   AppBancaire
 Component  Titulaire
      Port  Consultation = …
      Computation = …
 Component Compte
      Port  Solde = …
      Computation = …
 Connector CS
      Role client =  …
      Role serveur = …
      Glue = …
 Instances
       Titu   :   Titulaire
      Comp :   Compte
      c1     :   CS
 Attachements
       Titu.Consultation    as     c1.client
       Comp.Solde           as     c1.serveur
End Configuration
```

Figure 2.8. *Expression Wright de la configuration "AppBancaire"*

La figure 2.9 donne, sous forme d'un diagramme d'objets, la configuration "AppBancaire" considérée comme un modèle conforme au métamodèle configuration.

5.3 . Aspects Comportementaux de Wright

La formulation du comportement des composants et des connecteurs de façon informelle ne permet pas de prouver des propriétés non triviales sur l'architecture d'un système. Ainsi pour spécifier le comportement et la coordination des composants, Wright utilise une notation formelle basée sur CSP (Hoare, 1985) (Hoare, 1995) (Hoare, 2004).

CSP est un modèle mathématique qui a pour but de formaliser la conception et le comportement de systèmes qui interagissent avec leur environnement de manière permanente. Il est basé sur de solides fondements mathématiques qui permettent une analyse rigoureuse.

Les notions essentielles de CSP utilisées dans Wright sont décrites dans le chapitre 1.

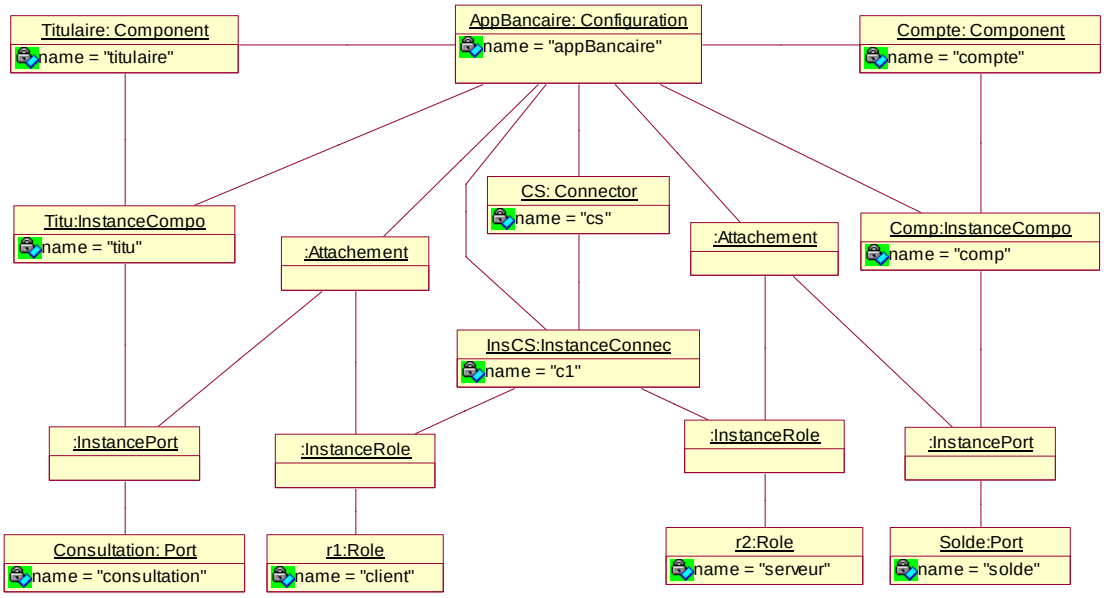


Figure 2.9. *Modèle objet de la configuration "AppBancaire"*

5.3.1. Un métamodèle CSP pour Wright

La figure 2.10 représente le métamodèle Processus CSP de l'ADL Wright, dont la métaclasse principale est **Process**, qui désigne une expression CSP.

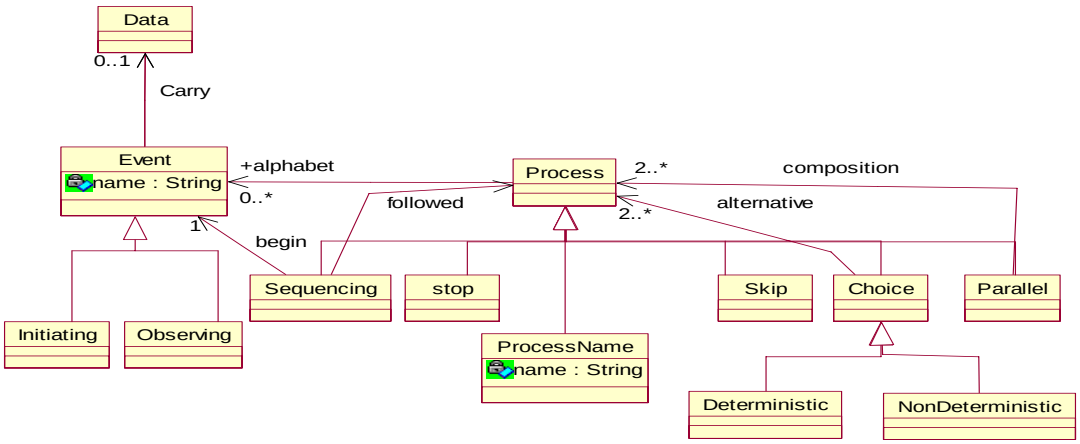


Figure 2.10. *Métamodèle Processus CSP de l'ADL Wright*

Une expression CSP comporte l'expression de terminaison avec succès, l'interblocage, le préfixage, le nommage des processus, le parallélisme et le choix déterministe et non déterministe. Ceci est modélisé par les métaclasses **Skip**, **Stop**, **Sequencing**, **ProcessName**, **Parallel** et **Choice** (avec deux classes descendantes **Deterministic** et **NonDeterministic**) héritant de la métaclasse **Process**.

L'ensemble des événements sur lequel le processus a une influence est modélisé par une méta-association entre les deux métaclasses **Process** et **Event**.

Un événement CSP peut porter une donnée. Ceci est traduit par la méta-association **carry** entre les deux métaclasses **Event** et **Data**.

Les deux métaclasses **Initiating** et **Observing** héritant de la métaclasse **Event** modélisent respectivement les événements initialisés et observés.

Les deux opérateurs de processus □ (choix déterministe) et ∏ (choix non déterministe) sont modélisés respectivement par les deux métaclasses **Deterministic** et **NonDeterministic**. La méta-association **alternative** entre **choice** et **process** traduit le fait que ces opérateurs (□ et ∏) exigent au moins deux processus.

La composition parallèle (| |) de CSP est modélisée par la métaclasse **Parallel**.

La méta-association **composition** traduit le fait que l'opérateur de composition parallèle exige au moins deux processus.

Les deux méta-associations **begin** (entre **sequencing** et **Event**) et **followed** (entre **sequencing** et **process**) traduisent la structure syntaxique de l'opérateur de préfixage (x -> P : x puis P).

5.4 . Aspects structuraux et comportementaux de Wright

Dans ce paragraphe, nous allons relier les aspects structuraux et comportementaux de Wright.

5.4.1 . Le concept Component

Comme étant signalé dans la section 5.2.1, un composant est formé par un ensemble de ports et d'une partie calcul ou (*computation*). Ces deux parties sont décrites en CSP.

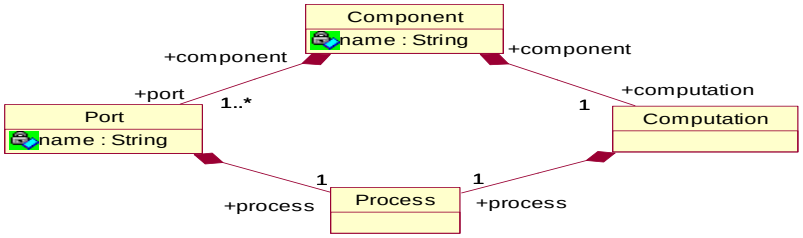


Figure 2.11. *Les deux aspects du métamodèle Component*

On peut alors enrichir le métamodèle Component de la figure 2.1, en lui ajoutant la métaclasse **Process** qui modélise une expression CSP. Cette dernière est la classe cible de deux méta-associations composition de multiplicité un, qui ont pour source les classes **Port** et **Computation**. Ainsi, la figure 2.11 illustre le métamodèle résultant.

5.4.2 . Le concept Connector

La structure d'un connecteur est similaire à celle d'un composant. Elle consiste en un ensemble de rôles et une glu. Les rôles et la glu sont également décrits en CSP. De même, on peut enrichir le métamodèle Connector de la figure 2.2, en lui ajoutant la métaclasse **Process**. Cette dernière est la classe cible de deux méta-associations composition de multiplicité un, qui ont pour source les métaclasses **Role** et **Glue**. Ainsi, la figure 2.12 donne le métamodèle du Connector.

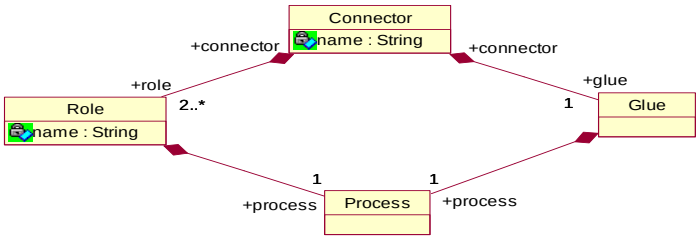


Figure 2.12. *Les deux aspects du métamodèle Connector*

5.5 . Conclusion

En suivant une démarche progressive, nous avons établi un métamodèle permettant de décrire d'une façon explicite aussi bien les aspects structuraux que comportementaux de Wright. Les fragments du métamodèle de Wright proposés ont été testés sur des modèles écrits en Wright.

Dans le chapitre suivant, nous allons identifier et justifier les fragments du métamodèle UML2.0 utilisés comme cibles pour l'adaptation d'UML2.0 à l'ADL Wright.

Chapitre 6 : Le Métamodèle UML2.0

6.1 . Introduction

UML (Unified Modeling Language) est un langage de modélisation graphique, semi-formel et largement répandu dans le monde industriel. UML est adopté comme un standard industriel de fait.

UML1.x est un langage de modélisation orienté objet (OMG, 2003a). Il propose neuf diagrammes différents, aussi appelés vues. UML1.x est basé sur un métamodèle unique.

UML2.0 (OMG, 2005) propose un modèle de composants permettant de définir les spécifications des composants, ainsi que l'architecture des systèmes à développer. UML2.0 décrit treize types de diagrammes officiels. Ces derniers sont classifiés comme l'indique la figure 2.13. UML2.0 représente le changement le plus important qu'ait jamais connu UML, les modifications les plus profondes concernant le métamodèle.

Ce chapitre comporte trois volets. Le premier volet est consacré à l'étude du modèle des composants préconisé par UML2.0. Quant au deuxième volet, il présente l'organisation générale du métamodèle UML2.0 et le dernier présente les fragments concernés pour adapter UML2.0 à l'ADL Wright.

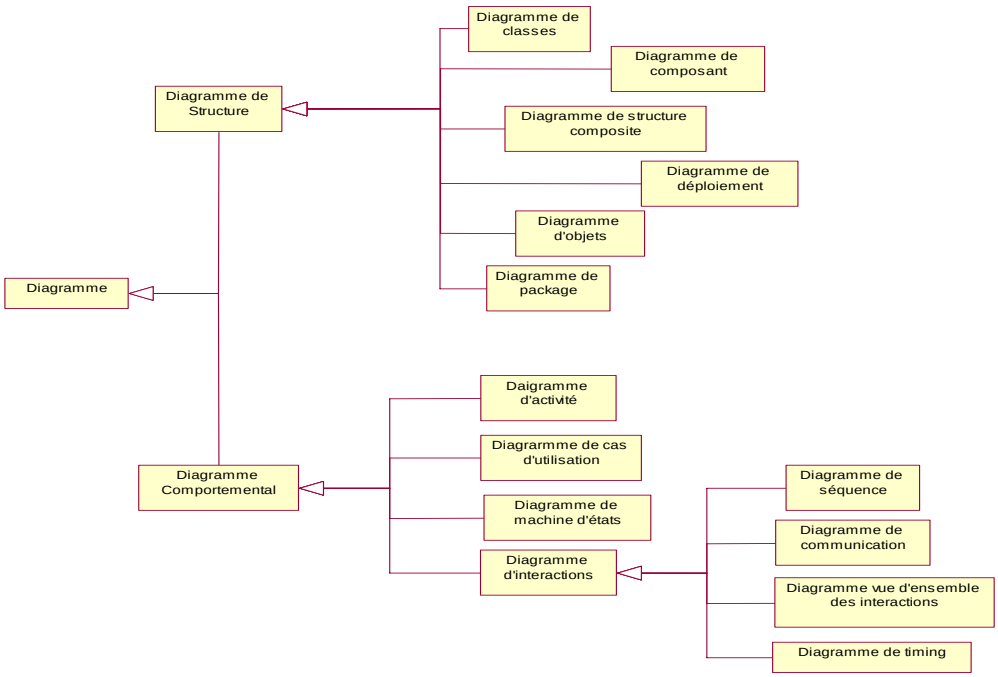


Figure 2.13. *Classification des diagrammes UML2.0*

6.2 . Modèle de composants UML2.0

6.2.1 . Composant

Un **composant** UML2.0 représente une partie modulaire d'un système qui encapsule son contenu et qui est remplaçable au sein de son environnement.

Un composant définit un comportement en terme d'interfaces fournies ou requises. Ces interfaces contiennent un ensemble d'opérations, d'attributs et de contraintes OCL2.0 (langage formel d'expression de contraintes pour UML).

Le comportement d'une interface est décrit par une machine à états de description de protocoles (protocol state machine). Un composant UML2.0 est voué à être déployé un certain nombre de fois, dans un environnement non connu a priori lors de la conception.

La figure 2.14 montre trois façons pour représenter un composant doté d'une interface offerte et d'une interface requise.

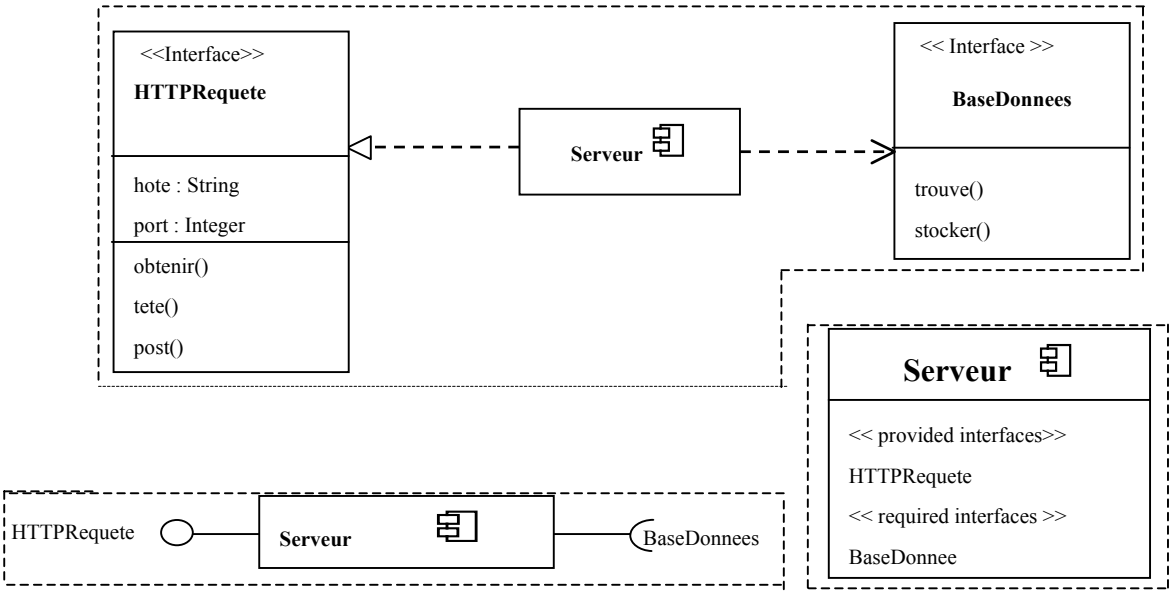


Figure 2.14. *Multiples représentations d'un composant Serveur*

6.2.2 . Port

Un composant UML2.0 interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Un **Port** est une caractéristique structurelle d'un classificateur encapsulé qui spécifie un point d'interaction entre le classificateur et son environnement ou entre un classificateur et ses parties internes appelées **parts**. Seules les classes et les composants peuvent contenir des ports.

Des interfaces requises ou offertes peuvent être associées aux ports afin de spécifier les opérations respectivement attendues de l'environnement ou offertes par le classificateur.

Le comportement d'un port est issu de la composition des comportements de ces interfaces. Un tel comportement est décrit à l'aide d'une machine à états de description de protocoles. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports. Les ports installés sur des composants ou classes peuvent être fournis ou requis.

La figure 2.15 montre un composant appelé Serveur doté d'un ou plusieurs ports (multiplicité 1..*). Une instance de type Serveur peut avoir un ou plusieurs ports p.

Figure 2.15. *Port sans interfaces*

La figure 2.16 montre le même composant Serveur dont le port p est doté d'une interface offerte *requete* et d'une interface requise *reponse*.

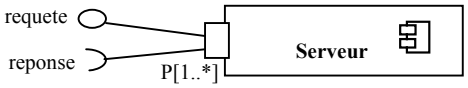


Figure 2.16. *Port avec interfaces*

6.2.3 . Structure composite

Il existe deux types de modélisation de composants dans UML2.0 : le composant atomique (ou basique) et le composant composite (structure composite). La première catégorie définit le composant comme un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme un ensemble cohérent des parties appelées **parts**. Chaque partie représente une instance d'un autre composant.

La figure 2.17 montre la vue externe dite encore boîte noire d'un composant FusionEtTri permettant de fusionner et trier deux flots d'entrée. Le composant FusionEtTri offre deux ports *entree1* et *entree2* et exige un port *sortie*.

Figure 2.17. *Vue externe d'un composant FusionEtTri.*

La figure 2.18 montre la vue interne dite encore boîte blanche d'un composant FusionEtTri. Ce dernier est composé de deux sous-composants : Fusion et Tri. La direction des connecteurs de délégation (cf. section 2.4) indique que les deux ports *entree1* et *entree2* sont typés avec des interfaces offertes et le port *sortie* est typé avec des interfaces requises. Les parties Fusion et Tri sont connectées par un connecteur d'assemblage non nommé.

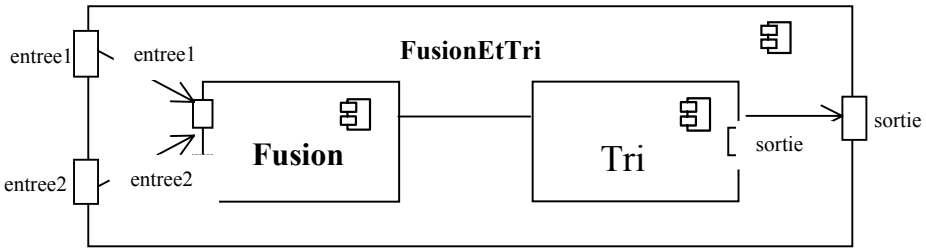


Figure 2.18. *Vue interne d'un composant FusionEtTri*

6.2.4 . Connecteur

La connexion entre les ports requis et les ports fournis se fait au moyen de connecteurs. Deux types de connecteurs existent : le connecteur de délégation et le connecteur d'assemblage.

Le connecteur de délégation est un connecteur qui relie le contrat externe d'un composant (spécifié par ses ports) à la réalisation de ce comportement par les parties internes du composant. Il permet de lier un port du composant composite vers un port d'un composant situé à l'intérieur du composant composite : relier par exemple un port requis à un autre port requis. Un connecteur de délégation doit uniquement être défini entre les interfaces utilisées ou des ports de même type, c'est-à-dire entre deux ports ou interfaces fournies par le composant ou entre deux ports ou interfaces requis par le composant.

Un connecteur d'assemblage est un connecteur entre deux composants qui définit qu'un composant fournit le service qu'un autre composant requiert. Un connecteur d'assemblage doit uniquement être défini à partir d'une interface requise ou d'un port vers une interface fournie ou un port.

Figure 2.19. *Un connecteur d'assemblage entre deux ports*

La figure 2.19 montre un connecteur d'assemblage entre deux ports installés sur deux composants.

La figure 2.20 montre un connecteur d'assemblage entre deux interfaces : l'une est offerte par le composant C1 et l'autre est requise par le composant C2.

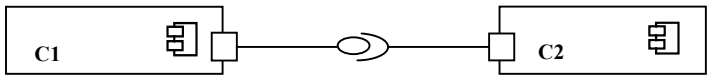


Figure 2.20. *Un connecteur d'assemblage entre deux interfaces*

La figure 2.21 montre un connecteur de délégation entre un port externe et un port interne.

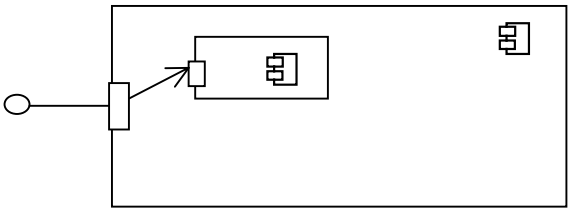


Figure 2.21. *Un connecteur de délégation*

6.3 . Organisation générale du Métamodèle UML2.0

UML2.0 est composé de deux standards, UML2.0 Superstructure, qui définit la vision utilisateur et UML2.0 Infrastructure, qui spécifie l'architecture de métamodélisation d'UML ainsi que son alignement avec MOF (Meta-Object Facility) (OMG,200). Dans la suite de ce paragraphe, nous nous intéressons à UML2.0 Superstructure notée tout simplement UML2.0 (Blanc, 2005).

Tout d'abord, nous présentons le canevas de description des parties du métamodèle UML2.0. Ensuite, nous décrivons l'architecture générale du métamodèle UML2.0.

6.3.1 . Canevas de description

Chaque élément du langage UML2.0 (classe, association, attribut, opération, composant,…) est représenté dans le métamodèle UML2.0 par la métaclasse correspondante (Class, Association, Attribute, Operation, Component, …). Les relations entre éléments sont représentées par des méta-associations. Par exemple, une classe UML2.0 peut comporter plusieurs attributs. Un tel fait est représenté par une méta-association (précisément une métacomposition) entre les deux métaclasses Attribute et Class dans le métamodèle UML2.0.

Une description complète du métamodèle UML2.0 peut être trouvée dans (OMG, 2005). Chaque partie du métamodèle est décrite selon le canevas suivant comportant trois sections :

- **Syntaxe abstraite** (Abstract Syntax) : contient plusieurs diagrammes de classes décrivant des éléments du langage UML2.0, suivis par des description textuelles (en anglais) des métaclasses, de ses méta-attributs, des méta-associations, etc.
- **Règles de bonne utilisation** (Well-Formedness Rules : WFR) : la notation graphique d'UML (y compris d'UML2.0) n'est pas assez précise pour définir le métamodèle. Ainsi, il est nécessaire de compléter ces diagrammes (syntaxe abstraite) par une liste de règles permettant d'imposer une bonne utilisation du modèle. Ces règles sont décrites d'une façon informelle (en anglais) et/ou d'une façon formelle en OCL (Object Constraint Language).
- **Des détails sémantiques** (Detailed Semantics) : les diagrammes et les règles donnent une description abstraite du métamodèle. Cette section comporte certaines explications textuelles (en anglais) ayant pour objectif de fournir une meilleure compréhension de la signification (ou de la sémantique) de certains éléments du langage, de leurs relations avec d'autres éléments et de l'utilisation "juste" de ces éléments pour la construction de modèles.

Si on prend à tire d'illustration l'exemple connecteur (Connector), La figure 2.22 donne une partie de la syntaxe abstraite, sous forme d'un diagramme de classes, du concept connecteur d'UML2.0.

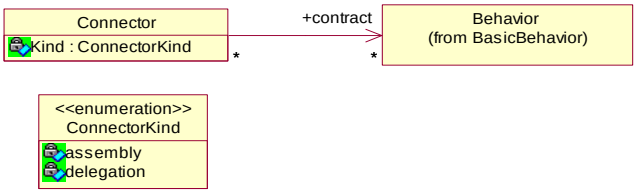


Figure 2.22. *Les métaclasses définissant en partie le concept connecteur*

La métaclasse Behavior modélise un comportement au sens général. En UML, il existe trois sortes de comportements, les machines à états, les activités et les interactions.

Parmi les règles de bonne utilisation (WFR ou encore contraintes) associées à la construction connecteur, nous citons :

- Un connecteur de délégation doit être défini entre des interfaces ou ports de même genre, c'est-à-dire entre deux ports ou interfaces requis ou entre deux ports ou interfaces fournis ;
- Un connecteur d'assemblage doit être défini entre une interface ou un port **requis** et une interface ou un port **fourni**.

6.3.2 . Architecture générale du Métamodèle UML2.0

Le métamodèle UML2.0 est extrêmement complexe, avec une cinquantaine packages et une centaine de métaclasses.

Au plus haut niveau, le métamodèle UML2.0 est découpé en trois parties :

- La partie **Structure** définit les concepts statiques (classe, attribut, opération, instance, composant, package) nécessaires aux diagrammes statiques, tels que les diagrammes de classes, de composants et de déploiement ;
- La partie **Behavior** définit les concepts dynamiques (interaction, action, état, etc.) nécessaires aux diagrammes dynamiques, tels que les diagrammes d'activités, d'états et de séquences ;

- La partie **Supplement** définit les concepts additionnels, tels que types primitifs et les templates. Cette partie définit aussi le concept de profil.

UML2.0 introduit une nouvelle relation entre packages : la relation *merge* (Blanc, 2005). Cette relation est fortement utilisée par UML2.0. Elle nécessite un package source et un package cible. Le package cible de la relation est appelé package mergé et le package source package mergeant.

La sémantique de la relation **merge** est la suivante :

- Si une classe est définie dans le package mergé, il faut définir une classe qui lui corresponde (même nom, mêmes attributs, etc.) dans le package mergeant, à moins qu'il n'existe déjà une classe correspondante, ayant même nom, mêmes attributs, etc. ;
- Si un package est défini dans le package mergé, il faut définir un package qui lui corresponde (même nom) dans le package mergeant, à moins qu'il n'y ait déjà un package correspondant (même nom). Après cela, une relation *merge* doit être établie entre les packages correspondants.

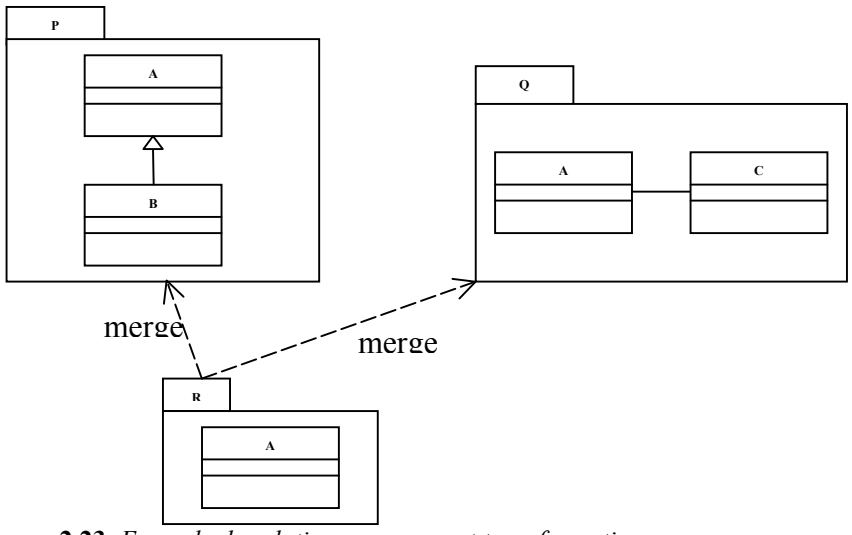


Figure 2.23. *Exemple de relation merge avant transformation*

La relation *merge* est très utilisée pour la réutilisation de packages. Elle peut être vue comme une sorte de calque entre le package mergé et le package mergeant. La figure 2.23 illustre un exemple de relation *merge*. La figure 2.24 donne le contenu du package R après transformation.

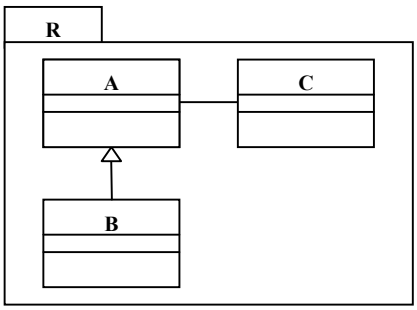


Figure 2.24. *Exemple de relation merge après transformation*

La figure 2.25 donne les différents packages reliés par la relation *merge* formant la partie **Structure** du métamodèle UML2.0.

L'organisation multipackages et la relation *merge* rendent la lecture du métamodèle UML2.0 **difficile**.

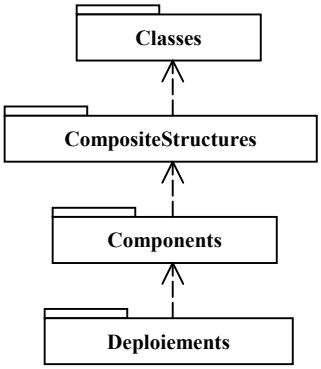


Figure 2.25. *Packages de la partie Structure*

6.4 . Fragments d'UML2.0 pertinents

Rappelons que l'objectif recherché de cette partie de la thèse est d'établir un profil UML2.0 pour l'ADL Wright. Tout d'abord, nous allons présenter la partie du métamodèle UML2.0 qui contient les métaclasses relatives aux profils. Ensuite, nous allons identifier les métaclasses cibles du métamodèle UML2.0 pour stéréotyper les concepts de Wright.

6.4.1 . Profils UML2.0

UML2.0 facilite la création de nouveaux profils en simplifiant la définition des concepts de profils et de stéréotypes dans le métamodèle.

La figure 2.26 illustre la partie du métamodèle UML2.0 qui contient les métaclasses relatives aux profils.

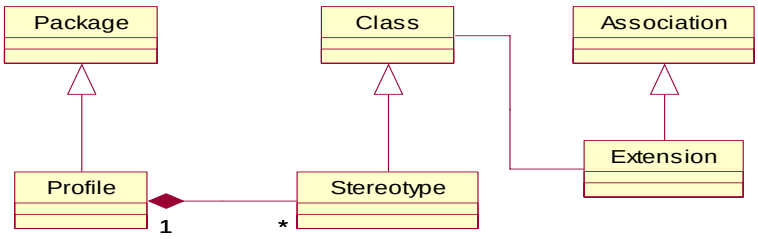


Figure 2.26. *Les profils dans UML2.0*

Dans ce métamodèle, les définitions de nouveaux profils sont considérées comme des packages. Ceci explique la relation d'héritage entre les deux métaclasses Profile et Package. La métaclasse Stereotype hérite de la métaclasse Class. Ceci a comme conséquence que les définitions de stéréotypes sont considérées comme étant des classes UML2.0.

Les classes étendues par des stéréotypes introduisent de nouvelles métaclasses dans le métamodèle. Ceci explique le lien -via la métaclasse Extension- entre les deux métaclasses Stereotype et Class. Sachant que la métaclasse Class représente aussi bien le concept de classe que celui de métaclasse.

6.4.2 . Concepts structuraux de Wright et UML2.0

Dans cette section, nous nous interrogeons sur les concepts les plus proches d'UML2.0 permettant d'être utilisés comme cibles pour stéréotyper les deux concepts structuraux de Wright : composant et connecteur.

6.4.2.1. Concepts structuraux de Wright

Les deux concepts structuraux de Wright composant et connecteur sont considérés comme **des types**. De plus, ils sont traités comme des entités occupant le **même niveau** (entité de première classe). Enfin, la vision externe de ces deux concepts est basée respectivement sur un ensemble de **ports** et un ensemble de **rôles**.

Les caractéristiques relatives aux concepts composant et connecteur de Wright sont similaires aux concepts composant et classe d'UML2.0. En effet, en UML2.0, composant et classe sont considérés comme des **classificateurs** (classifier en anglais). Sachant qu'un classificateur permet de classifier un ensemble d'objets. En outre, en UML2.0, composant et classe sont considérés comme des classificateurs encapsulés (Encapsulated classifier en anglais) sur lesquels on peut installer des ports au sens d'UML2.0.

Bien que les deux concepts composant et classe d'UML2.0 possèdent le même pouvoir expressif, ils sont utilisés comme base pour stéréotyper respectivement les deux concepts composant et connecteur de Wright (cf. figure 2.27 et 2.28). Ceci permet de distinguer les deux constructions de Wright du point de vue utilisation. En effet, composant et classe d'UML2.0 possèdent deux icônes différentes.

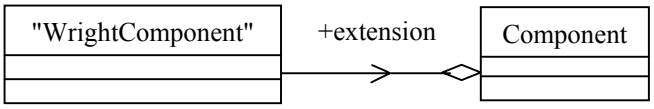


Figure 2.27. *Métaclasse cible du composant Wright*

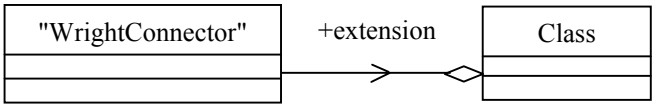


Figure 2.28. *Métaclasse cible du connector Wright*

Dans la suite nous allons identifier les parties du métamodèle UML2.0 relatives aux concepts classe et composant.

6.4.2.2. Classe et Composant comme classificateurs

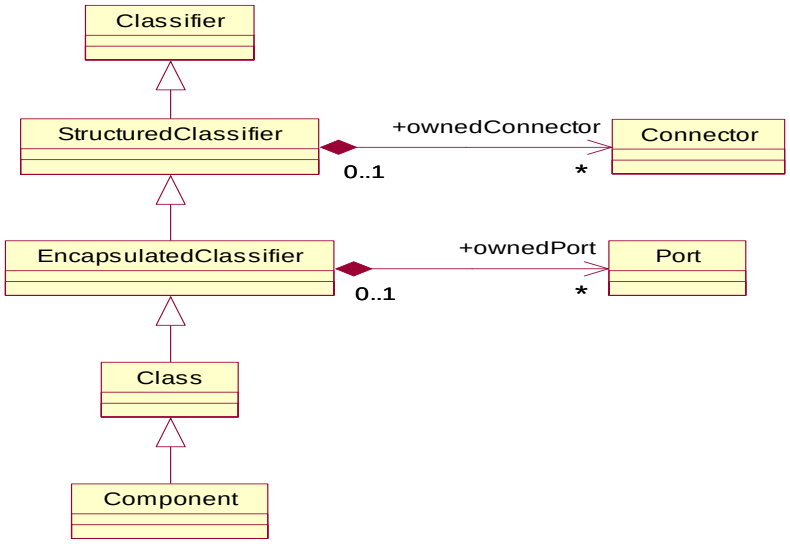


Figure 2.29. *Classe et composant comme classificateurs*

La figure 2.29 illustre comment les deux concepts classe et composant sont considérés comme des classificateurs. En effet, les deux métaclasses Class et Component héritent

d'une façon indirecte de la métaclasse Classifier. Celle-ci est une métaclasse abstraite qui représente des classificateurs tels que Class, Component, Interface, DataType, UseCase, etc. De plus, les deux métaclasses Class et Component dérivent indirectement de la métaclasse StructuredClassifier. Ceci précise que Class et Component peuvent contenir des connecteurs. Enfin, les deux métaclasses Class et Component héritent de la métaclasse EncapsulatedClassifier. Ceci précise que Class et Component peuvent contenir des ports.

Dans la suite, on va détailler respectivement les concepts connecteur, port, classe et composant.

6.4.2.1.1. La métaclasse Connector dans le métamodèle UML2.0

Dans ce métamodèle (cf. figure 2.30), la métaclasse Connector hérite de la métaclasse Feature. Cela signifie qu'un connecteur est une caractéristique (feature) qui peut être attachée à des instances appartenant à des classificateurs. La métaclasse Connector est reliée à la métaclasse ConnectorEnd par une méta-association qui précise qu'un connecteur doit contenir au moins deux instances de types ConnectorEnd.

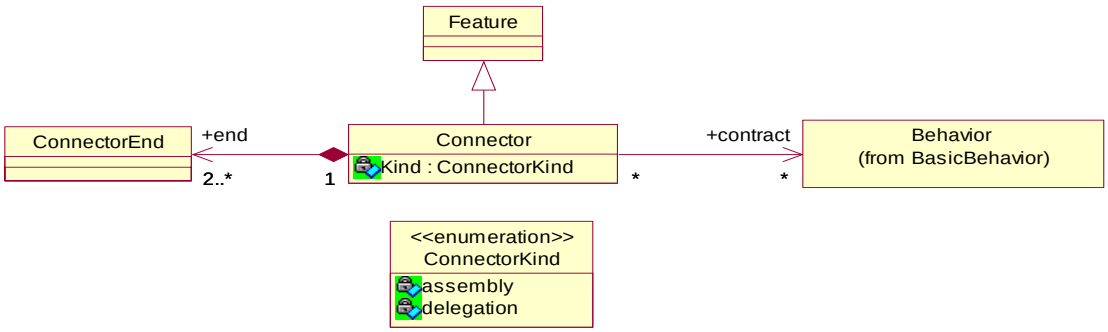


Figure 2.30. *La métaclasse Connector dans le métamodèle UML2.0*

Dans ce métamodèle, la métaclasse Connector hérite de la métaclasse Feature. Cela signifie qu'un connecteur est une caractéristique (feature) qui peut être attachée à des instances appartenant à des classificateurs. La métaclasse Connector est reliée à la métaclasse ConnectorEnd par une méta-association qui précise qu'un connecteur doit contenir au moins deux instances de types ConnectorEnd.

Les autres aspects liés au concept connecteur d'UML2.0 ont été expliqués précédemment (cf. figure 2.29).

6.4.2.1.2. La métaclasse Port dans le métamodèle UML2.0

La figure 2.31 illustre la partie du métamodèle UML2.0 qui définit la métaclasse Port.

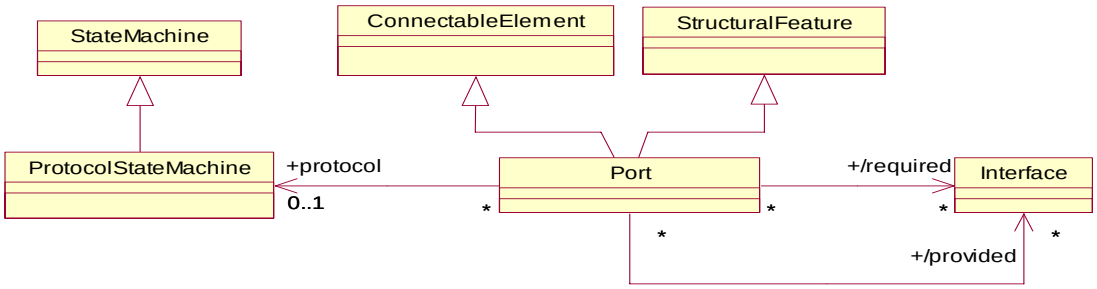


Figure 2.31. *La métaclasse Port dans le métamodèle UML2.0*

La métaclasse Port héritant à la fois de la métaclasse StructuralFeature et de la métaclasse ConnectableElement. Un port est considéré comme une caractéristique (feature)

structurelle. D'autre part, un port peut être connecté par des connecteurs. De plus, la métaclasse Port est reliée à la métaclasse Interface par deux méta-associations, qui précisent qu'un port peut identifier des interfaces requises ou offertes. Enfin, la métaclasse Port est reliée à la métaclasse ProtocolStateMachine par une méta-association, qui précise que le comportement d'un port peut être décrit par une machine à états de description de protocoles (Protocol State Machine).

6.4.2.1.3. La métaclasse Class dans le métamodèle UML2.0

La figure 2.32 illustre la partie du métamodèle UML2.0 qui définit la métaclasse Class.

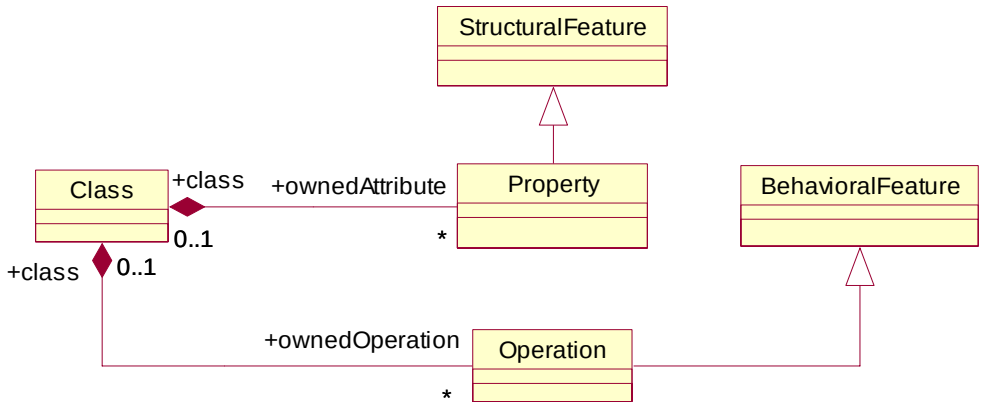


Figure 2.32. *La métaclasse Class dans le métamodèle UML2.0*

La métaclasse Class possède des caractéristiques structurelles et comportementales. Ceci est traduit par deux méta-associations entre la métaclasse Class et respectivement la métaclasse Property et la métaclasse Operation.

6.4.2.1.4. La métaclasse Component dans le métamodèle UML2.0

La figure 2.33 illustre la partie du métamodèle UML2.0 qui définit la métaclasse Component.

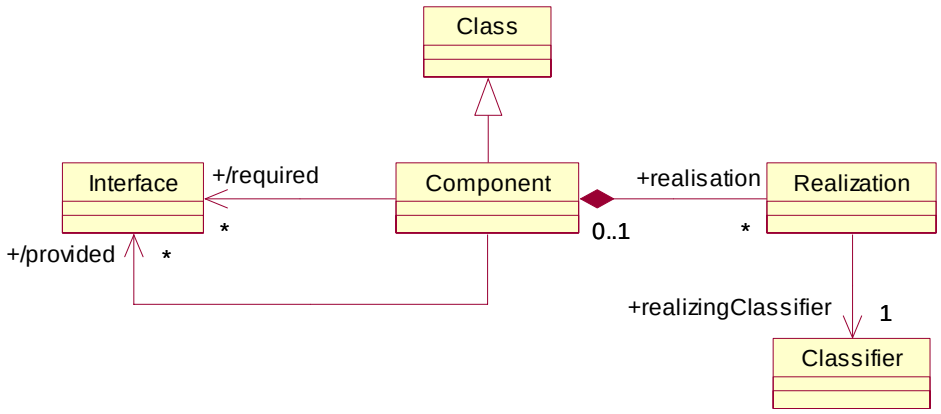


Figure 2.33. *La métaclasse Component dans le métamodèle UML2.0*

La métaclasse Component est reliée à la métaclasse Interface par deux méta-associations. Ainsi, un composant a des interfaces offertes et/ou requises.

La métaclasse Component est reliée à la métaclasse Realization par une méta-association qui précise qu'un composant peut être réalisé par une ou plusieurs instances de type Realization.

La métaclasse Realization est reliée à la métaclasse Classifier par une méta-association qui précise qu'une Realization est décrite par un classifier.

6.4.2.3. Les concepts comportementaux de Wright

Les comportements partiels associés aux ports d'un composant Wright sont exprimés à l'aide des processus CSP (chapitre 1). De même, les comportements partiels associés aux rôles d'un connecteur Wright sont exprimés en utilisant des processus CSP. Egalement, les comportements globaux aussi bien des composants que des connecteurs Wright sont exprimés en CSP.

Les résultats de plusieurs travaux permettent de relier dans les deux sens CSP aux machines à états d'UML (Engels, 2001) (Engels, 2002) (Küster, 2004).

Les événements CSP peuvent être modélisés à l'aide des machines à états d'UML2.0. Ceci est illustré par la figure 2.34. Sachant qu'en UML2.0, chaque transition porte une étiquette qui comprend trois parties : signature-déclencheur [garde]/activité.

Toutes ces parties sont optionnelles. Signature-déclencheur est généralement un unique événement qui déclenche potentiellement un changement d'état. La forme complète de signature-déclencheur peut comprendre plusieurs événements et plusieurs paramètres.

Figure 2.34. (a) *Un événement CSP avec une donnée d'entrée* **(b)** *Un événement CSP avec une donnée de sortie*

a- Un événement CSP avec une donnée d'entrée noté e?x est modélisé en UML comme une machine à états comportant deux états reliés par une transition dotée uniquement d'une signature-déclencheur.
b- Un événement CSP avec une donnée de sortie notée e!x est modélisé en UML comme une machine à états comportant deux états reliés par une transition dotée uniquement d'une activité (ou action) : e(x). Sachant que ε dénote un événement nul en UML

La garde, si elle est présente, est une condition booléenne qui doit être vraie pour que la transition soit empruntée. L'activité est un comportement quelconque qui est exécuté pendant la transition.

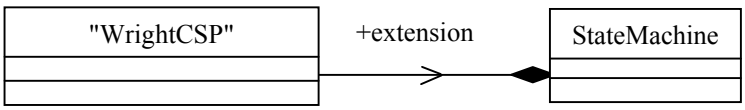
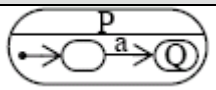
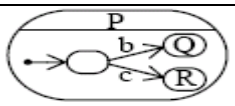
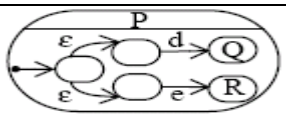
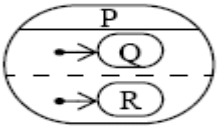
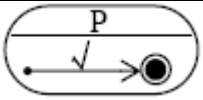


Figure 2.35. *Métaclasse cible des expressions CSP de Wright*

Tableau 2.1. ***Correspondances entre constructions CSP et machines à états d'UML***

Concept CSP	Notation CSP	Machine à états UML
Préfixage	P = a → Q	
Choix déterministe	P = b → Q □ c → R	
Choix non déterministe	P = d → Q ⊓ e → R	
Parallélisme	P = Q || R	
Evénement Succès	P = √	

Les auteurs de (Medvidovic, 2002) proposent des schémas généraux permettant d'établir des correspondances entre les constructions CSP et les machines à états d'UML (cf. tableau 2.1).

Le concept machine à états d'UML2.0 (State Machine) est utilisé comme base pour stéréotyper les aspects comportementaux de Wright ou plus précisément des expressions CSP (cf. figure 2.35).

La figure 2.37 illustre la partie du métamodèle UML2.0 qui définit la métaclasse StateMachine. Dans ce métamodèle, la métaclasse StateMachine est reliée à la métaclasse Region par une méta-association. Cela signifie qu'une machine à états comporte une ou plusieurs régions. La métaclasse Region représente une partie orthogonale dont l'exécution est autonome. Elle contient des états et des transitions. Ceci explique respectivement les deux méta-associations entre Region et Vertex et entre Region et Transition.

Chaque transition a un état source et un état destination. Ceci explique les deux méta-associations entre Transition et Vertex. Les trois caractéristiques (signature-déclencheur, garde et activité) attachées à une transition sont précisées par les trois méta-associations entre Transition et respectivement Trigger, Constraint et Activity.

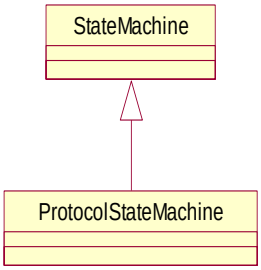


Figure 2.36. *La Métaclasse ProtocolStateMachine*

UML2.0 introduit une nouvelle machine à états connue sous le nom de machine à états de description de protocoles. Celle-ci est représentée par la métaclasse ProtocolStateMachine héritant de la métaclasse StateMachine (cf. figure 2.36).

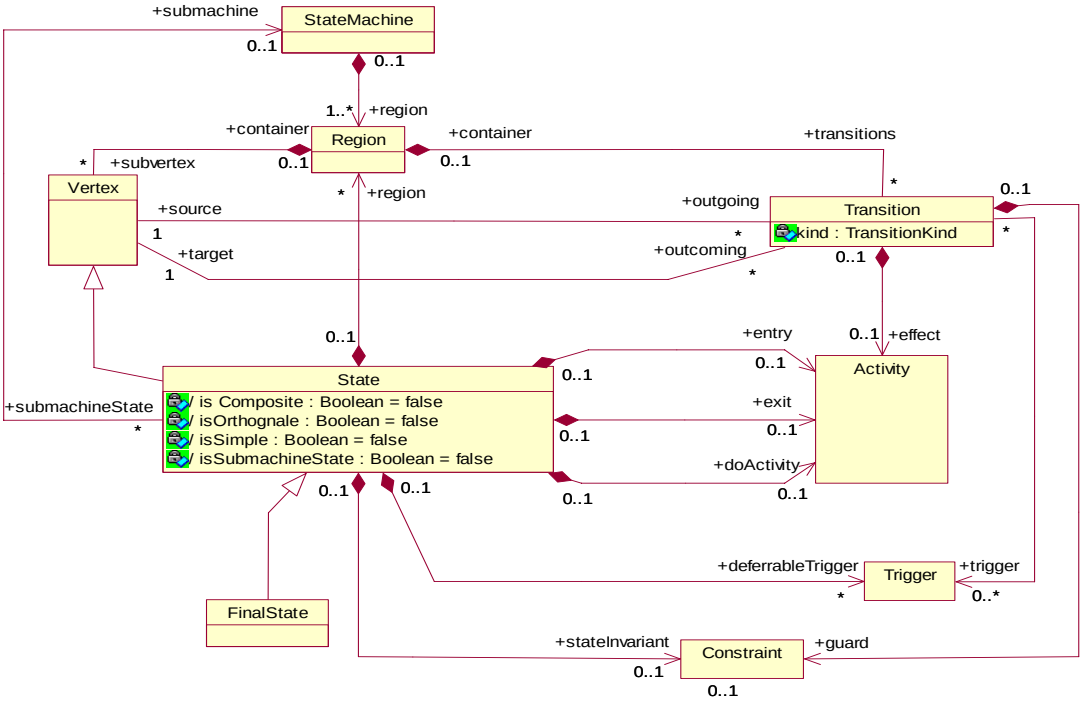


Figure 2.37. *La Métaclasse StateMachine dans le métamodèle UML2.0*

UML2.0 introduit une nouvelle transition connue sous le nom de transition de protocole. Celle-ci est une spécialisation des transitions présentes dans les machines à états. Une transition de protocole appartient toujours à une machine à états de description de protocoles. La figure 2.38 illustre la partie du métamodèle UML2.0 qui définit la métaclasse ProtocolTransition.

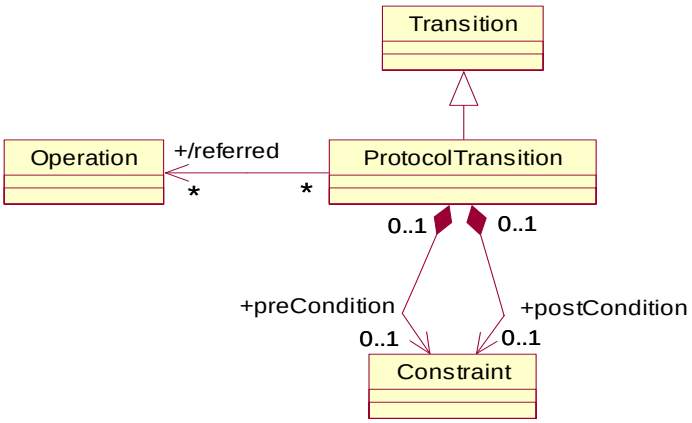


Figure 2.38. *La Métaclasse ProtocolTransitition dans le métamodèle UML2.0*

Dans ce métamodèle, des préconditions et postconditions peuvent être associées à des transitions de protocole. Ceci justifie les deux méta-associations entre ProtocolTransition et Constraint. De même, des opérations peuvent être associées aux transitions de protocole. Ceci est traduit par la méta-association entre ProtocolTransition et Operation. Enfin, une transition de protocole n'a jamais d'action associée.

En UML2.0, les machines à états peuvent être utilisées pour spécifier le comportement de plusieurs éléments des modèles décrits en UML2.0 tels que des instances d'une classe UML2.0. Tandis que les machines à états de description de protocoles peuvent être utilisées avec profit pour exprimer des protocoles liés à des scénarios d'utilisation des services offerts par des interfaces ou des ports (cf. figure 2.39).

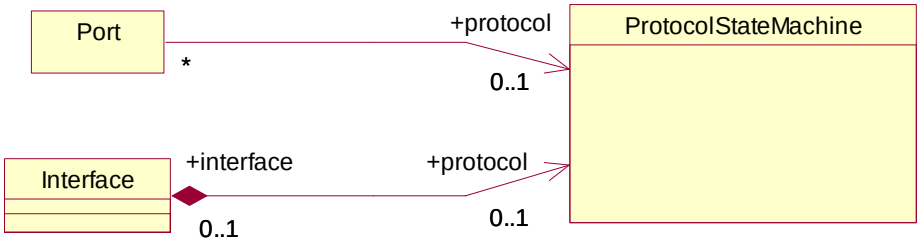


Figure 2.39. *Machine à états de description de protocoles associée à des interfaces ou à des ports*

6.5 . Conclusion

Dans ce chapitre nous avons étudié l'approche par composant préconisée par UML2.0, l'organisation générale du métamodèle UML2.0 et nous avons identifié et justifié les fragments du métamodèle UML2.0 nécessaires pour adapter UML2.0 à l'ADL Wright.

Dans le chapitre suivant, nous allons proposer un profil UML2.0 pour l'ADL Wright en se basant sur le métamodèle Wright établi dans le chapitre 5 et les fragments du métamodèle UML2.0 identifiés dans ce chapitre.

Chapitre 7 : Un profil UML pour l'ADL Wright : W-UML

7.1 . Introduction

Dans ce chapitre nous définissons un langage intermédiaire sous forme d'un profil UML afin de faciliter la correspondance entre des modélisations d'architectures de systèmes en UML et l'ADL Wright.

La définition d'un profil exige deux préalables : la description du domaine couvert par le profil et l'identification du sous-ensemble UML2.0 concerné.

Le domaine couvert par notre profil, à savoir l'ADL Wright, est décrit, sous forme d'un métamodèle (cf. chapitre 5). Des concepts avancés supportés par Wright comme composant composite (ou non atomique), connecteur composite, style et reconfiguration dynamique ne sont pas traités dans ce travail.

Le sous-ensemble d'UML2.0 concerné par notre profil a été identifié conformément aux choix arrêtés dans le chapitre 6.

Ce chapitre comporte quatre sections. La première présente et discute les différentes stratégies potentielles permettant de modéliser en UML des architectures logicielles. La deuxième présente les aspects fondamentaux d'OCL nécessaires à la compréhension du profil proposé. La troisième introduit les stéréotypes, tagged values et les contraintes OCL formant notre profil UML2.0 pour l'ADL Wright : W-UML. Enfin, la quatrième section présente des exemples d'utilisation de notre profil.

7.2 . Architectures logicielles Modélisées en UML

L'architecture de métamodélisation en quatre niveaux appliquée au langage UML induit trois stratégies potentielles pour modéliser des architectures logicielles en UML (Medvidovic, 2002). Ces stratégies sont :

- l'utilisation UML2.0 en tant que tel ;
- l'utilisation des mécanismes d'extensibilité d'UML2.0 pour contraindre le métamodèle UML afin de l'adapter aux concepts architecturaux ;
- l'augmentation du métamodèle UML afin de supporter directement les concepts architecturaux.

Chaque stratégie a certains avantages et inconvénients. Dans la suite, nous allons présenter et évaluer ces trois stratégies en s'inspirant des travaux effectués par (Medvidovic, 2002), (Zarras, 2001), (Roh, 2004), (Goulão, 2003), (Kandé, 2000) et (Ivers, 2004).

7.2.1 . Utilisation directe d'UML

Cette stratégie consiste à utiliser les constructions offertes par UML pour représenter les concepts architecturaux venant des ADL tels que composant, connecteur, rôle, port et configuration.

Garlan et al. (Garlan, 2002) (Garlan, 2001) proposent et évaluent quatre approches permettant de représenter les principaux concepts architecturaux en UML1.x. Ces

approches sont centrées autour de la représentation **des types** de composant et **des instances** de composant en UML1.x. Ainsi, ces auteurs proposent les options suivantes :

- classes et objets : les types de composants sont représentés par des classes UML1.x et les instances de composants par des objets ;
- classes et classes : les types de composants et les instances de composants sont représentés par des classes UML1.x ;
- composants UML1.x : les types de composants sont représentés par des composants UML1.x et les instances de composants par des instances de composants UML1.x;
- subsystems : les types de composants sont représentés par des subsystems UML et les instances de composants par des instances de subsystems.

En fonction de l'option, les autres concepts architecturaux comme port, connecteur, rôle et configuration peuvent être représentés par d'autres constructions UML adéquates à l'option choisie. Par exemple, le concept port dans l'option 1 peut être représenté par plusieurs choix comme interface, attribut, classe contenue (contained class).

L'utilisation d'UML en tant que tel est également employée pour représenter des ADL particuliers. Par exemple, dans (Medvidovic, 2000), UML1.x est utilisé pour modéliser l'ADL C2. Dans (Ivers, 2004), UML2.0 est utilisé pour modéliser des concepts architecturaux issus de l'ADL ACME tels que component, port, connector, role et attachement. Les auteurs de cette recherche utilisent avec profit les nouveaux concepts proposés par UML2.0 tels que component, port et assembly connector.

L'avantage majeur de l'utilisation d'UML en tant que tel pour la modélisation des architectures logicielles décrites par des ADL est la compréhension de cette modélisation par n'importe quel utilisateur UML. De plus, une telle modélisation peut être manipulée par des ateliers UML supportant les autres étapes du développement : conception et implémentation. Quant à l'inconvénient inhérent à cette stratégie, il concerne l'incapacité d'UML en tant que tel -notamment UML1.x- à traduire d'une façon explicite les concepts architecturaux.

7.2.2 . Utilisation des mécanismes d'extensibilité

UML est un langage de modélisation "généraliste" pouvant être adapté à chaque domaine grâce aux mécanismes d'extensibilité offerts par ce langage tels que stéréotypes, tagged values (valeurs marquées) et contraintes (partie II, chapitre 6, cf. section 4.1). Les extensions UML ciblant un domaine particulier forment des profils UML. Les mécanismes d'extensibilité offerts par UML permettent d'étendre UML sans toucher au métamodèle UML.

Plusieurs travaux permettent d'adapter aussi bien UML1.x qu'UML2.0 aux architectures logicielles. Ces travaux peuvent être classés en deux catégories : des travaux visant des ADL particuliers et d'autres ciblant des ADL génériques c'est-à-dire comportant des concepts architecturaux communs à tous les ADL.

Les auteurs de (Medvidovic, 2000) proposent trois profils UML1.x -précisément UML1.3- respectivement pour les ADL C2, Wright et Rapide. Nous nous sommes inspirés de cette recherche intéressante pour l'élaboration de notre profil UML2.0 pour l'ADL Wright (cf. 7.4).

Les auteurs de (Goulão, 2003) établissent un profil UML2.0 pour l'ADL ACME (Garlan, 2000). Celui-ci est considéré comme un ADL pivot permettant l'interopérabilité entre les différents ADL. Ainsi, il regroupe des concepts communs et minimaux à tous les ADL. Les

auteurs de (Roh, 2004) signalent quelques faiblesses de ce travail liées notamment à la représentation proposée du connecteur (au sens d'ADL) en UML2.0 et proposent un ADL générique sous forme d'un profil UML2.0. Dans ce travail, on note notamment l'utilisation des collaborations UML2.0 pour représenter des connecteurs ADL. De plus, un autre aspect intéressant se dégage : type et instance de connecteurs sont modélisés par deux stéréotypes. En effet, le type de connecteurs est défini comme stéréotype à base de métaclasse Collaboration d'UML2.0. Et l'instance de connecteurs est définie comme stéréotype à base de métaclasse Connector d'UML2.0. Mais ce travail ne traite pas les aspects comportementaux des ADL.

L'utilisation des profils pour adapter UML au domaine des architectures logicielles a comme avantage important **d'expliciter** la représentation des concepts architecturaux sans toucher au métamodèle UML. De plus, ces profils peuvent être manipulés par des ateliers UML supportant le concept profil. Actuellement, ces ateliers sont de plus en plus nombreux tels que Softeam UML Modeler (http://www.objecteering.com) et IBM Rational Software Modeler (http://www.rational.com). La contrepartie vis-à-vis des utilisateurs est de connaître notamment les contraintes -décrites souvent en OCL- associées aux stéréotypes formant ces profils.

7.2.3 . Augmentation du métamodèle UML

Cette stratégie consiste à ajouter de nouveaux éléments de modélisation (de nouvelles constructions syntaxiques) en modifiant directement le métamodèle UML. Ceci donne un nouveau langage de modélisation supportant d'une façon native les concepts architecturaux.

Les auteurs de (Enrique, 2003) augmentent le métamodèle UML afin de supporter directement les concepts issus de l'ADL C3.

L'augmentation du métamodèle UML afin de couvrir des besoins architecturaux se heurte aux deux inconvénients majeurs suivants :

- la nouvelle version d'UML devient de plus en plus complexe. Ceci a un impact sur la facilité d'utilisation de ce langage.
- la nouvelle version d'UML perd son caractère standard et par conséquent elle devient incompatible avec les ateliers UML existants.

7.3 . OCL

Dans cette section, nous présentons les aspects fondamentaux d'OCL nécessaires à la compréhension du profil UML2.0 pour l'ADL Wright proposé dans la section 4.

Depuis sa version 1.3, UML intègre un langage de spécification de contraintes OCL (Object Constraint Language) (Warmer, 2003). Ce langage permet l'écriture d'expressions booléennes sans effet de bord qui restreignent le domaine de valeurs d'un modèle UML. Ces expressions peuvent être attachées à n'importe quel élément UML. De telles expressions OCL peuvent aussi s'appliquer au niveau du métamodèle (OMG, 2005) permettant de décrire les règles de bonne utilisation (Well Formedness Rules : WFR) (cf. chapitre 6). De plus, les expressions OCL sont utilisées pour définir des stéréotypes (cf. section 4). En outre les expressions OCL favorisent la conception par contrat (Design by contract) préconisée par l'auteur d'Eiffel (Meyer, 1992) (Meyer, 2000).

La nouvelle spécification d'OCL notée OCL2.0 (OMG, 2003b) fait partie intégrante d'UML2.0.

Le langage OCL2.0 -noté dans ce mémoire OCL- ajoute aux anciens types de contraintes (inv, pre, post) des nouveaux types de contraintes (init, def, body). Contrairement aux anciens types de contraintes, les nouveaux sont considérés comme des expressions non forcément booléennes.

Les contraintes d'utilisation qui accompagnent les stéréotypes formant notre profil W-UML (cf. section 4.) sont toutes des contraintes booléennes de type **inv** (invariant). Sachant qu'une contrainte de type inv, attachée à une métaclasse, doit être observée pour toutes les instances de cette métaclasse.

7.3.1. Éléments du langage OCL

7.3.1.1. Contrainte

Les contraintes OCL sont utilisées en UML pour spécifier des restrictions sur les éléments des différents modèles. Elles peuvent être attachées à tout type d'élément du modèle et doivent être respectées dans toute modification ou raffinement d'un diagramme.

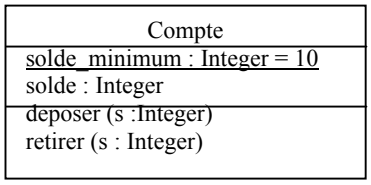
7.3.1.2. Contexte

Toute contrainte OCL est liée à un contexte spécifique : l'élément auquel la contrainte est attachée qu'on peut spécifier explicitement à l'intérieur d'une expression à l'aide du mot clé : **« self»**.

- Si le contexte est un type alors self se rapporte à un objet de ce type.
- Si le contexte est une opération alors self désigne le type qui possède cette opération.

  - **Exemple**

Soit la classe compte **:**



Des contraintes attachées à cette classe peuvent être :

context Compte

inv solde_minimum : self.solde >= Compte.solde_minimum

context Compte :: deposer (s : Integer)

pre : s > 0

post : self.solde = self.solde @pre + s

L'opérateur @pre dénote l'état précédent c'est-à-dire avant l'exécution de l'opération.

7.3.2. Types de base

Dans OCL, plusieurs types de base sont prédéfinis et disponibles. Ces types prédéfinis sont indépendants de tout modèle objet et font partie de la définition d'OCL. Ces types prédéfinis sont : Integer, Real, String et Boolean.

7.3.2.1. Types du Modèle UML

Chaque expression OCL est écrite dans le contexte d'un modèle UML ainsi que dans plusieurs classificateurs (types/classes,...) avec leurs caractéristiques et associations, et leurs généralisations. Tous les classificateurs du modèle UML sont des types dans les expressions OCL qui sont attachées au modèle.

7.3.2.2. Types énumérés

Les énumérations sont des types de données dans UML et ont un nom. Dans OCL, on peut faire référence à la valeur d'une énumération.

Exemple: **enum {value1, value2, value3}**

Pour éviter les conflits de nom, on utilise dièse «#» : **#value1**

7.3.3. Les collections

Le langage OCL permet de manipuler des collections d'objets afin de naviguer dans un diagramme de classes en tenant compte de l'arité des associations, des compositions et des agrégations. En OCL, les collection sont modélisées comme suit :

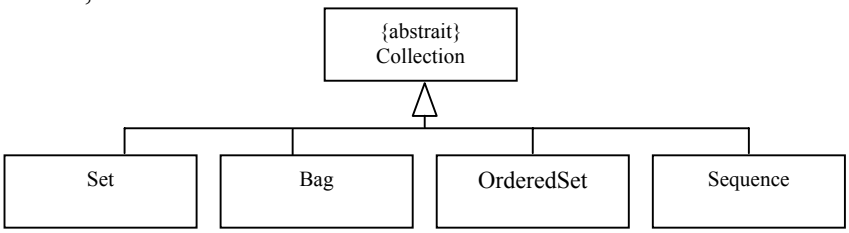


Figure 2.40. *Les collections en OCL*

Le langage OCL offre des opérations communes à tout type de collection et également des opérations spécifiques liées à chaque type de collection. En outre, OCL fournit des opérations de conversion entre les différents types de collection.

Les opérations applicables sur toutes les collections en produisant un type de base (entier ou booléen) sont :

- Collection -> size() : Integer -- nombre d'éléments dans la collection.
- Collection -> isEmpty() : Boolean -- vrai si la collection est vide.
- Collection -> notEmpty() : Boolean -- vrai si la collection a au moins un élément.
- Collection -> includesAll (collection2) -- voir si collection2 est incluse ou non dans Collection
- Collection -> exists (expr) -- quantificateur existentiel (expr est une expr. booléenne)
- Collection -> forAll (elem:T | bool−expr) : Boolean -- quantificateur universel

L'opération *forAll* s'évalue à vrai si l'expression du corps est évaluée à vrai pour chaque élément dans la collection de la source, autrement le résultat est faux.

Les opérations sur une collection produisant une autre collection sont :

- Collection -> select(expr) -- éléments de la collection qui satisfont expr (expression booléenne)
- Collection -> reject(expr) -- éléments de la collection qui ne satisfont pas expr

- Collection -> collect(expr) -- collection obtenue en évaluant expr pour chacun des éléments de la collection

7.3.4 . Propriétés prédéfinies

Il existe plusieurs propriétés qui s'appliquent à tous les objets. Nous citons :

- p.oclIsKindOf(t : OclType) : Boolean

  -- retourne vrai si le type de « p » est « t » ou s'avère être un sous-type de « t ».

- p.oclIsTypeOf(t : OclType) : Boolean

  -- retourne vrai si le type de « p » est « t ».

- p.oclAsType(t : OclType) : oclType

  -- similaire à la conversion de type en Java, « p » devient de même type que « t ».

7.4 . Définition technique du Profil

Cette partie est consacrée à la définition technique du profil W-UML. Un tel profil comporte un ensemble de stéréotypes appliqués sur des métaclasses UML2.0 et définis par un ensemble de contraintes OCL (Graiet, 2006a) (Graiet, 2006b). Les métaclasses cibles des stéréotypes proposés sont identifiées et justifiées dans le chapitre précédent.

Dans la suite, nous allons établir des stéréotypes permettant de modéliser respectivement des concepts comportementaux et structuraux de Wright tels que : expressions CSP des interfaces des composants et des connecteurs, des ports, des rôles, des composants, des connecteurs et des attachements. Nous avons accordé un soin particulier à l'élaboration des contraintes formelles en OCL associées aux stéréotypes. Ceci permet de mieux cerner le cadre d'utilisation de ces stéréotypes.

7.4.1 . Expressions CSP de Wright

Une expression CSP de Wright est décrite par une machine à états de description de protocoles (Protocol State Machine) UML2.0 stéréotypée par «WrightProtocolStateMachine». Mais la définition de ce stéréotype nécessite l'introduction d'autres stéréotypes tels que «WrightOperation», «WrightProtocolTransition», «WrightRegion» et «WrightVertex».

7.4.1.1. Stéréotype « WrightOperation »

Un événement CSP peut transmettre ou recevoir des données notées respectivement e!x et e?x. Nous avons décidé de modéliser un événement CSP de Wright par une opération UML (cf. figure 2.41) stéréotypée par «WrightOperation» avec les contraintes OCL suivantes :

- Une WrightOperation contient un seul paramètre formel.

  self.formalParameter -> size() = 1

- Le paramètre formel associé à une WrightOperation peut être soit in, soit out.

self.formalParameter -> forAll (p:Parameter | p.direction = #in or p.direction = #out)

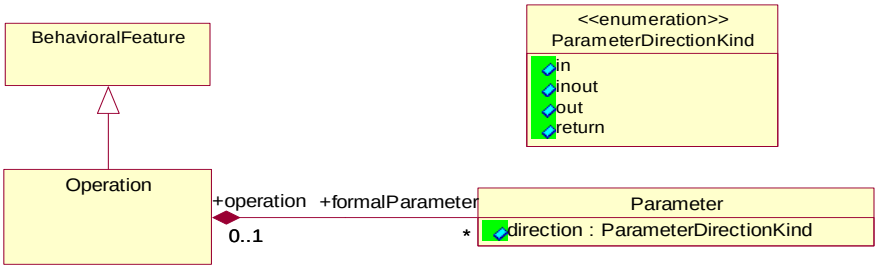


Figure 2.41. *La métaclasse Operation dans le métamodèle UML2.0*

7.4.1.2. Stéréotype « WrightProtocolTransition »

La combinaison des événements CSP est décrite par une transition de protocole UML2.0 (cf. figure 2.42) stéréotypée par «WrightProtocolTransition» avec les contraintes OCL suivantes :

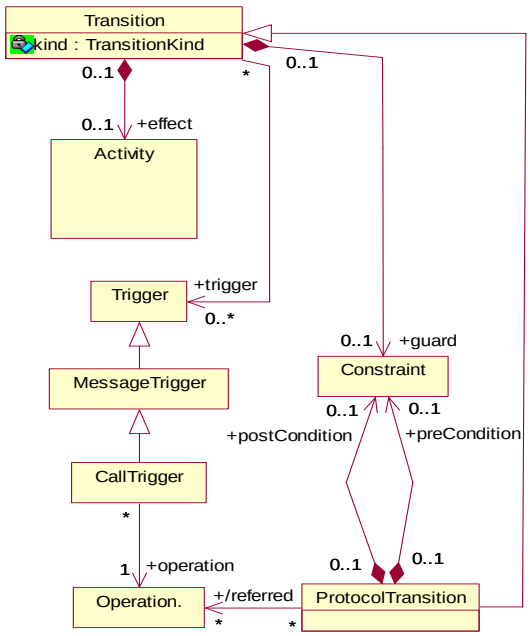


Figure 2.42. *La métaclasse ProtocolTransition dans le métamodèle UML2.0*

– Une WrightProtocolTransition comporte uniquement un événement qui consiste à un appel CallTrigger. De plus, elle ne comporte ni précondition ni postcondition.

self.trigger -> size() = 1 implies (self.trigger.oclIsKindOf(CallTrigger) and self.precondition -> isEmpty() and self. postcondition -> isEmpty())

– Une WrightProtocolTransition ne comporte que des opérations de type WrightOperation.

self.referred -> forAll(O | O.stereotype = WrightOperation)

7.4.1.3. Stéréotype « WrightVertex »

L'enchaînement des événements CSP entraîne des états intermédiaires. Un état CSP est décrit par un sommet UML2.0 (cf. figure 2.43) stéréotypée par «WrightVertex» avec les contraintes OCL suivantes :

– Toutes les transitions sortantes de WrightVertex doivent être WrightProtocolTransition.

self.outgoing -> forAll (t | t.oclAsType(ProtocolTransition).stereotype = WrightProtocolTransition)

– Toutes les transitions entrantes de WrightVertex doivent être WrightProtocolTransition.

self.incoming -> forAll (t | t.oclAsType(ProtocolTransition).stereotype = WrightProtocolTransition)

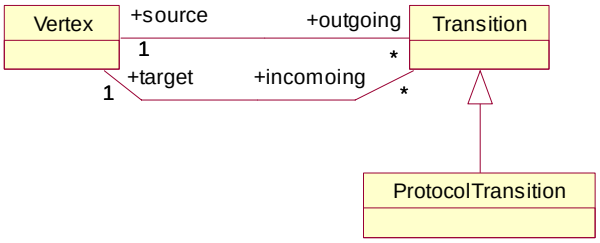


Figure 2.43. *La métaclasse Vertex dans le métamodèle UML2.0*

7.4.1.4. Stéréotype « WrightRegion »

Le stéréotype « WrightRegion » appliqué à la métaclasse Region (cf. figure 2.44) est défini par les contraintes OCL suivantes :

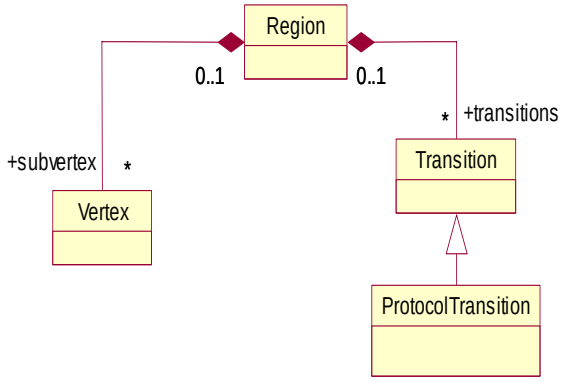


Figure 2.44. *La métaclasse Region dans le métamodèle UML2.0*

– Tous les sommets appartenant à WrightRegion doivent être WrightVertex

self.subvertex -> forAll(s | s.stereotype = WrightVertex)

– Toutes les transitions appartenant à WrightRegion doivent être WrightProtocolTransition

self.transitions -> forAll(t | t.oclAsType(ProtocolTransition).stereotype = WrightProtocolTransition)

7.4.1.5. Stéréotype « WrightProtocolStateMachine »

Le stéréotype « WrightProtocolStateMachine » appliqué à la métaclasse StateMachine (cf. figure 2.45) est défini par les contraintes OCL suivantes :

– Toutes les régions appartenant à WrightProtocolStateMachine doivent être WrightRegion.

self.oclAsType(ProtocolStateMachine).region -> forAll(r | r.stereotype = WrightRegion)

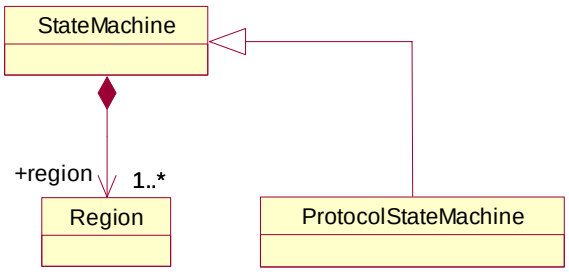


Figure 2.45. *La métaclasse StateMachine dans le métamodèle UML2.0*

7.4.2 . Interfaces des composants et des connecteurs de Wright

Chaque interface Wright (un port d'un composant ou un rôle d'un connecteur) possède une ou plusieurs opérations. En Wright, ces opérations sont modélisées implicitement, comme partie intégrante du protocole CSP associé au port ou au rôle. Nous avons choisi de modéliser explicitement ces opérations en UML2.0.

Les protocoles CSP associés au port ou au rôle sont modélisés comme <<WrightProtocolStateMachine>>.

Une interface Wright est décrite par une interface UML2.0 (cf. figure 2.46) stéréotypée par <<WrightInterface>> avec les contraintes OCL suivantes :

- Toutes les opérations associées à <<WrightInterface>> sont des opérations sans paramètre.

  self.ownedOperation -> forAll (o | o.formalParameter -> isEmpty())

- Aucun attribut n'est associé à une <<WrightInterface>>.

  self.ownedAttribute -> isEmpty()

- Exactement un et seul <<WrightProtocolStateMachine>> est associé à chaque <<WrightInterface>>.

  self.protocol -> size() = 1 and self.protocol -> forAll (psm | psm.stereotype = WrightProtocolStateMachine)

- Toutes les opérations utilisées dans le <<WrightProtocolStateMachine>> associé à <<WrightInterface>> figurent dans l'ensemble des opérations associées à la même <<WrightInterface>>.

  self.protocol.region -> forAll (r | r.transitions -> forAll (t | t.oclAsType(ProtocolTransition).referred -> forAll (o | self.ownedOperation -> exists (o1| o1=o))))

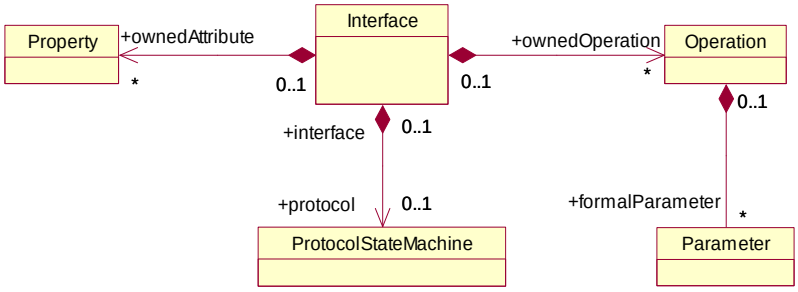


Figure 2.46. *Interface dans le métamodèle UML2.0*

7.4.3. Ports et rôles de Wright

Un port Wright est décrit par un port UML2.0 stéréotypé par <<WrightPort>>. De même un rôle Wright est décrit par un port UML2.0 (cf. figure 2.47) stéréotypé par <<WrightPort>>. Les contraintes OCL suivantes sont définies :

- Les <<WrightPorts>> sont typés comme des ports ou des rôles.

  WrightPortType : enum {port, role}

- Toutes les interfaces requises associées au <<WrightPort>> sont des WrightInterface.

  self.required -> forAll (i | i.stereotype = WrightInterface)

- Toutes les interfaces offertes associées au <<WrightPort>> sont des <<WrightInterface>>.

  self.provided -> forAll (i | i.stereotype = WrightInterface)

- <<WrightPort>> à une seule interface offerte et une seule interface requise.

  self.provided -> size() = 1 and self.required -> size() = 1

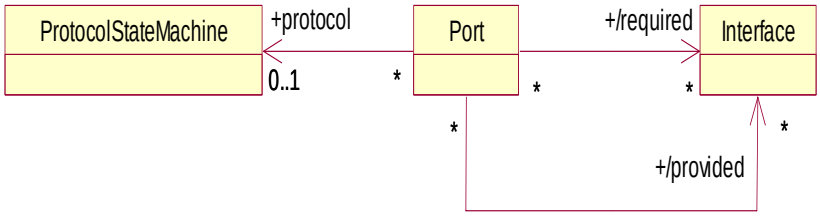


Figure 2.47. *La métaclasse Port dans le métamodèle UML2.0*

- Un seul <<WrightProtocolStateMachine>> est associé au <<WrightPort>>

  self.protocol -> size() >= 1 and self.protocol -> forAll (psm | psm.stereotype = WrightProtocolStateMachine)

- <<WrightProtocolStateMachine>> est associé au <<WrightPort>>, c'est la composition des <<WrightInterfaces>> attachées à ce <<WrightPort>>. Cette contrainte ne peut pas être exprimée en OCL à cause du caractère déclaratif de ce langage.

7.4.4. Composants de Wright

Un composant Wright est décrit par un composant UML2.0 (cf. figure 2.48) stéréotypé par <<WrightComponent>>.

Le stéréotype <<WrightComponent>> est défini par les contraintes OCL suivantes :

- Aucune interface offerte ou requise n'est associée à <<WrightComponent>>.

  self.provided -> isEmpty() and self.required -> isEmpty()

- Tous les ports associés à <<WrightComponent>> sont des <<WrightPorts>> et doivent être de type port.

  self.ownedPort -> forAll(p | p.stereotype = WrightPort and p.WrightPortType = #port)

- Un <<WrightComponent>> ne possède aucune réalisation

self.realisation -> isEmpty()

– Un <<WrightProtocolStateMachine>> est associé au <<WrightComponent>> traduisant le protocole CSP de calcul attaché à un composant Wright.

self.stateMachine -> size() = 1 and self.stateMachine.oclAsType (ProtocolStateMachine).stereotype = WrightProtocolStateMachine

– Le <<WrightProtocolStateMachine>> associé au <<WrightComponent>> est une composition des <<WrightProtocolStateMachine>> attachés aux ports de ce <<WrightComponent>>. Cette contrainte ne peut pas être exprimée en OCL à cause du caractère déclaratif de ce langage.

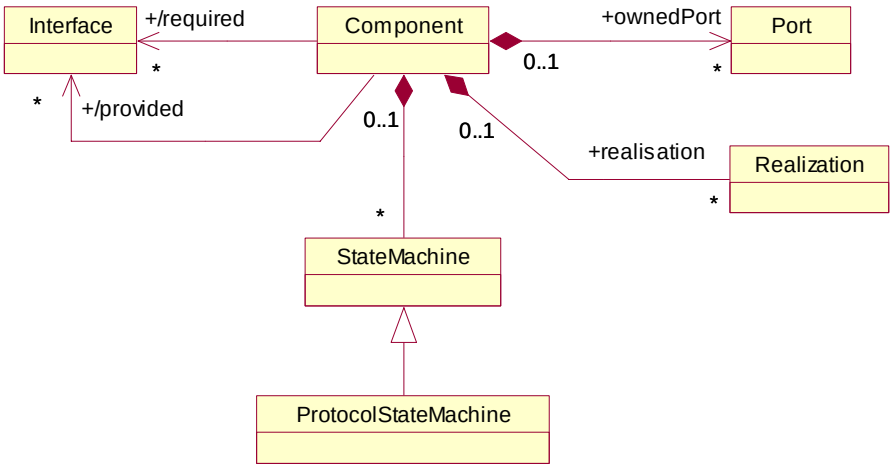


Figure 2.48. *La métaclasse Component dans le métamodèle UML2.0*

7.4.5. Connecteurs de Wright

Un connecteur Wright est décrit par une classe UML2.0 (cf. figure 2.49) stéréotypée par <<WrightConnector>>.

Le stéréotype <<WrightConnector>> est défini par les contraintes OCL suivantes :

– Un <<WrightConnector>> ne comporte ni attributs ni opérations.

self.ownedAttribute -> isEmpty() and self.ownedOperation -> isEmpty()

– Aucune interface offerte ou requise n'est associée à <<WrightConnector>>.

self.provided -> isEmpty() and self.required -> isEmpty()

– Tous les ports associés à <<WrightConnector>> sont des WrightPort et doivent être de type rôle.

self.ownedPort -> forAll (p | p.stereotype = WrightPort and p.WrightPortType = #role)

– Un <<WrightProtocolStateMachine>> est associé au <<WrightConnector>> traduisant le protocole CSP de glue attaché à un connecteur Wright

self.stateMachine -> size() = 1 and self.stateMachine.oclAsType (ProtocolStateMachine).stereotype = WrightProtocolStateMachine

– Le <<WrightProtocolStateMachine>> associé au <<WrightConnector>> est une composition des <<WrightProtocolStateMachine>> attachés aux rôles de ce

<<WrightConnector>>. Cette contrainte ne peut pas être exprimée en OCL à cause du caractère déclaratif de ce langage.

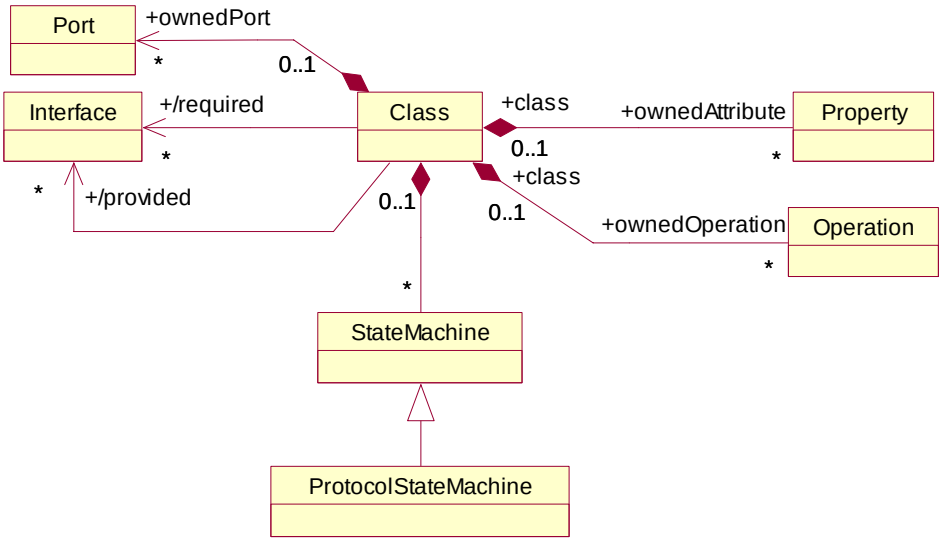


Figure 2.49. *La métaclasse Class dans le métamodèle UML2.0*

7.4.6. Attachements de Wright

Un attachement Wright est décrit par un connecteur UML2.0 (cf. figure 2.50) stéréotypé par <<WrightAttachement>>.

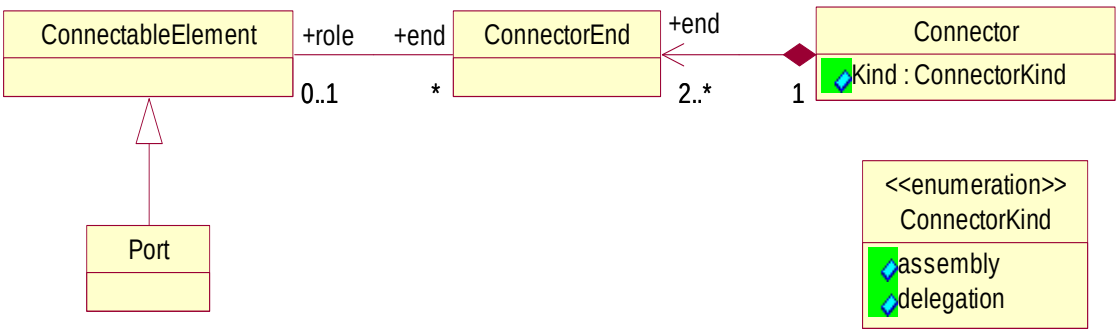


Figure 2.50. *La métaclasse Connector dans le métamodèle UML2.0*

Le stéréotype <<WrightAttachement>> est défini par les contraintes OCL suivantes :

- Un attachement dans Wright permet de relier deux éléments.

  self.end -> size () = 2

- Un <<WrightAttachement>> est un connecteur d'assemblage.

  self.Kind = #assembly

- Les éléments connectables de <<WrightAttachement>> sont tous des <<WrightPorts>>

  self.end -> forAll (ce | ce.role-> forAll (elt | elt.oclIsKindOf (Port) implies elt.stereotype = WrightPort))

- Un <<WrightAttachement>> permet de relier un <<WrightPort>> de type port et un WrightPort de type rôle.

self.end -> (exists (cp | cp.role.oclAsType (Port).stereotype = WrightPort and cp.role.oclAsType (Port).WrightPortType = #port) and (exists (cr | cr.role.oclAsType (Port).stereotype = WrightPort and cr.role.oclAsType (Port).WrightPortType = #role)

7.5 . Exemples d'utilisation

Dans cette section, nous illustrons les concepts introduits par notre profil W-UML à l'aide de deux exemples consacrés respectivement aux concepts structuraux et comportementaux introduits par notre profil.

7.5.1 . Génération d'un index

7.5.1.1. Présentation

L'application permettant de générer un index prend en entrée une suite de caractères (texte) et renvoie un ensemble de couples (mot, numéro de ligne). Ces couples doivent être ordonnés par un tri alphabétique au niveau des mots.

Une architecture logicielle en pipeline pour la génération d'un index est donnée dans la figure 2.51 extraite de (Sanlaville, 1997). On distingue trois composants de type filtre : Filtre_Texte, Filtre_Tri et Filtre_Fusion et deux connecteurs de type pipe : Pipe1 et Pipe2.

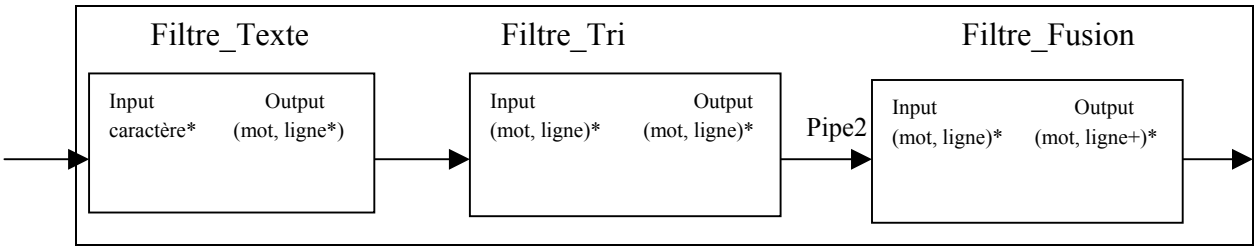


Figure 2.51. *Une architecture logicielle d'un index*

Le composant Filtre_Texte prend en entrée une suite de caractères et produit un ensemble de couples (mot, numéro de ligne). Ces couples sont envoyés au connecteur Pipe1 par son port de sortie un à un.

Le composant Filtre_Tri doit lire tous les couples (mot, numéro de ligne) que lui envoie le composant Filtre_Texte par l'intermédiaire du connecteur Pipe1. Ensuite, il trie tous ces couples par ordre alphabétique pour les mots et par ordre croissant pour les numéros de ligne lorsque les mots sont identiques. Enfin, il envoie ces couples triés au connecteur Pipe2 par son port de sortie un à un.

Le composant Filtre_Fusion reçoit les couples (mot, numéro de ligne) triés en entrée par l'intermédiaire du connecteur Pipe2. Dès qu'il récupère un couple :

- il l'oublie si ce couple est identique au précédent. Ceci évite les doublons.
- il fusionne son numéro de ligne avec celui du couple précédent si les mots de deux couples sont identiques.
- il envoie le couple (mot, numéro(s) de ligne) du couple précédent sur son port de sortie et stocke ce nouveau couple si les mots des deux couples sont différents.

7.5.1.2. Spécification en Wright

Les aspects structuraux de l'application index sont regroupés dans la figure 2.52.

```
Configuration Index
  Component  Filtre_texte
     Port Input =
    Port Output =
    Computation =
 Component  Filtre_Tri
    Port Input =
    Port Output =
    Computation =
 Component  Filtre_Fusion
    Port  Input =
    Port Output =
    Computation =
 Connector   Pipe
    Role Source =
    Role Sink    =
    Glue =
 Instances
    Texte :   Filtre_texte
     Tri    :   Filtre_tri
     Fusion :   Filtre_fusion
     P1, P2 :   Pipe
 Attachments
   Texte.Output      as     P1.Source
    Tri.Input          as     P1.Sink
   Tri.Output        as     P2.Source
    Fusion.Input     as     P2.Sink
End Configuration
```

Figure 2.52. *Application index en Wright*

7.5.1.3. Utilisation du profil W-UML

La figure 2.53 décrit les différents types de composants utilisés dans l'application index.

La figure 2.54 décrit le connecteur utilisé dans l'application index.

La figure 2.55 décrit la configuration de l'application index.

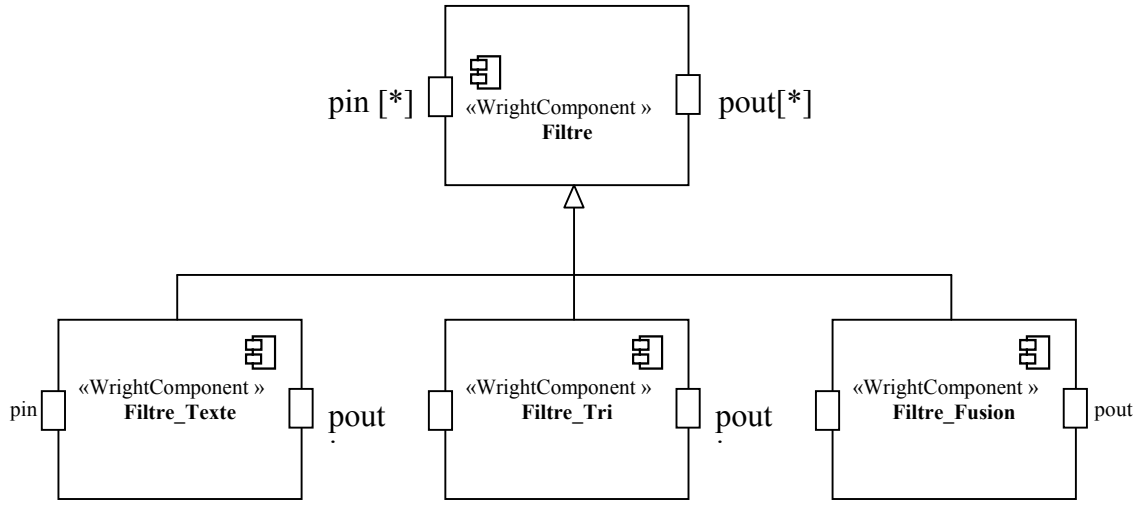


Figure 2.53. *Types de composants*

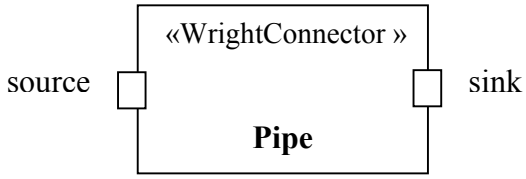


Figure 2.54. *Le connecteur Pipe*

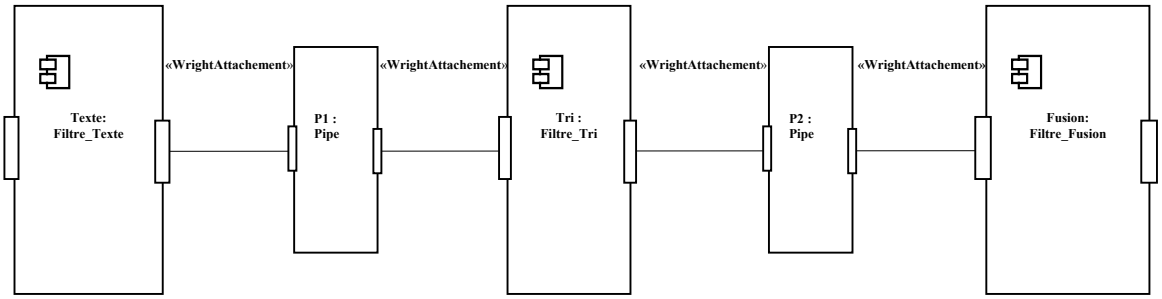


Figure 2.55. *Configuration de l'application index*

7.5.2. Application bancaire

Dans cette section, nous allons décrire en Wright puis en W-UML un exemple simple d'une application bancaire. Celle-ci est décrite d'une façon informelle par la figure 2.56.

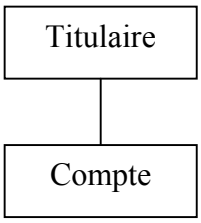


Figure 2.56. *Architecture informelle d'une application bancaire simple*

7.5.2.1. Spécification des composants en Wright

Les deux figures 2.57 et 2.58 donnent respectivement en Wright la spécification des composants Titulaire et Compte.

Component Titulaire
Port Consultation = $\overline{requete!u} \rightarrow reponse\ ?r \rightarrow$ Consultation $\sqcap$ §
Computation = calcul_interne $\rightarrow$ Consultation.*requete ! u*
$\rightarrow$ Consultation.*reponse?r* $\rightarrow$ **Computation** $\sqcap$ §

Figure 2.57. *Le composant Titulaire*

Le Titulaire fait une demande ($\overline{\text{Consultation.requete !u}}$) et attend en réponse (Consultation.reponse?r) sur son port Consultation puis éventuellement termine avec succès (§).

Component Compte
Port Solde = *requete ? u* → $\overline{\textit{reponse ! r}}$ → Solde □ §
Computation = Solde.*requete ? u* → calcule_interne
→ Solde.*requete ! u* → **Computation** □ §

Figure 2.58. *Le composant Compte*

L'usage de choix non déterministe (Π) dans la spécification indique que c'est le Titulaire qui décide s'il fait une demande ou s'arrête. Par contre, l'usage de choix déterministe (□) dans le composant Compte indique que ce dernier ne connaît pas a priori le nombre de requêtes provenant du Titulaire. C'est pour cela que l'environnement agit sur le comportement du processus.

La partie Computation du composant Titulaire ressemble beaucoup à la modélisation du port consultation étant donné qu'il est le seul port du composant. Ceci est également vrai pour le composant Compte doté d'un seul port solde.

7.5.2.2. Spécification du connecteur CS en Wright

La figure 2.59 présente un connecteur qui servira à relier les deux composants Titulaire et Compte de l'application bancaire décrite précédemment.

Connector CS
Role client = $\overline{\textit{requete ! u}}$ → *reponse ?r* → client ⊓ §
Role serveur = *requete ? u* → $\overline{\textit{reponse ! r}}$ → serveur □ §
Glue = (client.*requete ? u* → $\overline{\textit{serveur.requete!u}}$
→*serveur.reponse !r* → $\overline{\textit{client. reponse ! r}}$ → **Glue**) □ §

Figure 2.59. *Le Connecteur CS*

Le rôle client décrit le comportement d'un composant qui va utiliser ce connecteur. Le composant connecté via le rôle client sera un processus qui peut respectivement envoyer une requête et recevoir le résultat ou terminer.

De façon similaire, le rôle serveur définit un processus qui, soit accepte répétitivement une requête, retournant par la suite une réponse, soit peut terminer avec succès. La glu décrit comment les activités des rôles client et serveur sont coordonnées et présente leur séquencement.

7.5.2.3. Architecture logicielle

Configuration AppBancaire
Component Titulaire
Port Consultation = $\overline{\textit{requete!u}}$ → *reponse ?r* → Consultation ⊓ §
Computation = calcul_interne → $\overline{\text{Consultation.}\textit{requete ! u}}$
→ Consultation.*reponse?r* → **Computation** ⊓ §
Component Compte
Port Solde = *requete ? u* → $\overline{\textit{reponse ! r}}$ → Consultation □ §
Computation = Solde.*requete ? u* → calcul_interne
→ Solde.requete*!u* → **Computation** □ §

Connector CS
Role client = *requete ! u* → *reponse ?r* → client ⊓ §
Role serveur = *requete ? u* → *reponse ! r* → serveur □ §
Glue = (client.*requete ? u* → *serveur.requete!u*
→*serveur.reponse !r* →*client. reponse ! r* → **Glue**) □ §
Instances
Titu : Titulaire
Comp : Compte
c1 : CS
Attachments
Titu.Consultation **as** c1.client
Comp.Solde **as** c1.serveur
End Configuration

Figure 2.60. *Spécification formelle de la configuration AppBancaire*

La figure 2.60 présente une configuration de l'application bancaire contenant trois instances respectivement de type Titulaire, Compte et CS.

L'instance c1 de type CS permet de relier les deux autres instances titu et comp.

7.5.2.4. Utilisation du profil W-UML

Dans cette application bancaire nous nous limiterons au composant Titulaire . Un tel composant comporte un port appelé Consultation défini comme par :

Port Consultation = *requete!u* → *reponse ?r* → Consultation ⊓ §

La figure 2.61 utilise le profil W-UML pour modéliser l'expression CSP associée au port Consultation

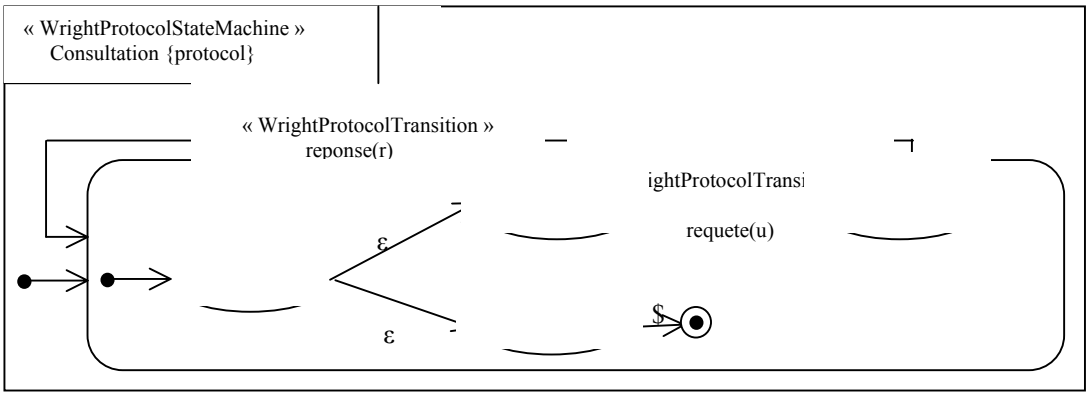


Figure 2.61. *Machine à états de description de protocoles associée au port Consultation*

7.6 . Conclusion

Dans le cadre de la démarche DFVAL (cf. chapitre 3), nous pouvons utiliser le profil W-UML comme langage intermédiaire permettant de faciliter la traduction entre les formalismes à base d'UML2.0 et Wright. Une telle utilisation est illustrée par la figure 2.62. En effet, dans des travaux antérieurs (Graiet, 2005a) (Graiet, 2005b), nous avons souligné les difficultés de traduire directement des modèles UML2.0 en Wright : identification des connecteurs et traitement des aspects comportementaux.

D'autres utilisations du profil W-UML sont également possibles :

- La récupération (ou réutilisation) des architectures logicielles décrites en Wright venant du monde académique ;
- La conception et la réalisation des systèmes logiciels ayant des architectures logicielles explicites et documentées ;
- La transformation de modèles selon l'approche MDA (Bézivin, 2002a) (Bézivin, 2002b) (Blanc, 2005) (OMG, 2003c). Par exemple la transformation d'un PIM (Platform Independent Model) décrit dans ce profil vers un autre PIM ou un PSM (Platform Specific Model) décrit en UML2.0 ou à l'aide d'autres profils.

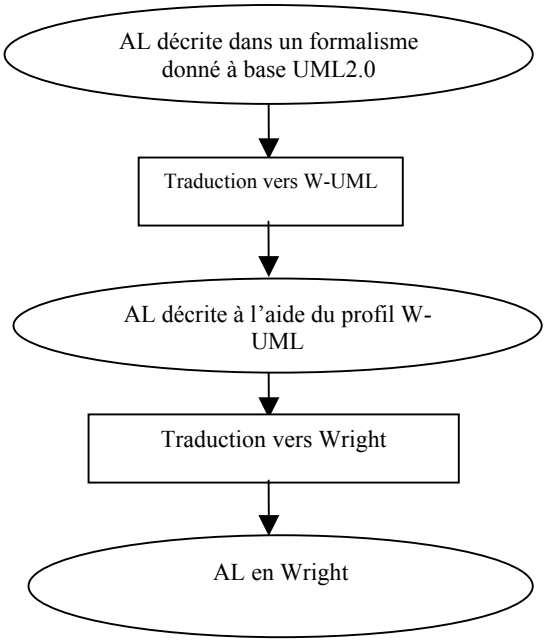


Figure 2.62. *Le profil W-UML comme langage intermédiaire dans la démarche DFVAL*

Chapitre 8 : Traduction W-UML vers Wright

8.1 . Introduction

L'utilisation du profil W-UML au sein de la démarche DFVAL exige la traduction des architectures logicielles décrites à l'aide de W-UML vers des architectures logicielles décrites en Wright. Une telle traduction est simple car l'écart sémantique entre les deux formalismes est minime. Dans ce chapitre, nous proposons des règles simples de traduction W-UML vers Wright. Ces règles sont illustrées par des exemples.

8.2 . Traduction de <<WrightComponent>>

Les constituants de <<WrightComponent>> sont :

- un ou plusieurs ports de type <<WrightPort>> ;
- une ou plusieurs machines à états de description de protocoles de type <<WrightProtocolStateMachine>> : autant de machines que de ports ;
- une machine à états de description de protocoles de type <<WrightProtocolStateMachine>>.

- **Traduction 8.1 (<<WrightComponent>>)**

Un <<WrightComponent>> du profil W-UML est traduit comme un type de composant (component) Wright selon les règles suivantes :

i) Tout <<WrightPort>> est traduit par un port Wright ayant le même nom.
ii) Tout <<WrightProtocolStateMachine>> est traduit par une expression CSP associée au port correspondant.
iii) Le <<WrightProtocolStateMachine>> associé au <<WrightComponent>> est traduit par une expression CSP associée à la partie Computation du composant Wright.

- **Exemple**

La figure 2.63 illustre le principe de la traduction d'un <<WrightComponent>> Titulaire de l'application bancaire (cf. chapitre 7) vers Wright. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification W-UML.

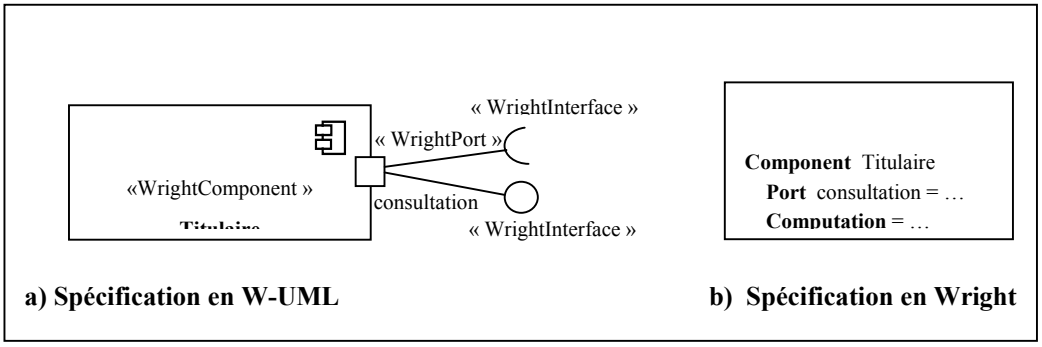


Figure 2.63. *Traduction du <<WrightComponent>>*

8.3 . Traduction de <<WrightConnector>>

Les éléments formant <<WrightConnector>> sont :

- deux ou plusieurs rôles de type <<WrightPort>> ;
- deux ou plusieurs machines à états de description de protocoles de type <<WrightProtocolStateMachine>> : autant de machines que de rôles ;
- une machine à états de description de protocoles de type <<WrightProtocolStateMachine>>.

- **Traduction 8.2 (<<WrightConnector>>)**

Un <<WrightConnector>> du profil W-UML est traduit comme un type de connecteur (connector) Wright selon les règles suivantes :

i) Tout <<WrightPort>> est traduit par un rôle Wright ayant le même nom.
ii) Tout <<WrightProtocolStateMachine>> est traduit par une expression CSP associée au rôle correspondant.
iii) Le <<WrightProtocolStateMachine>> associé au <<WrightConnector>> est traduit par une expression CSP associée à la partie Glue du connecteur Wright.

- **Exemple**

La figure 2.64 illustre le principe de la traduction d'un <<WrightConnector>> CS de l'application bancaire vers Wright. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification W-UML.

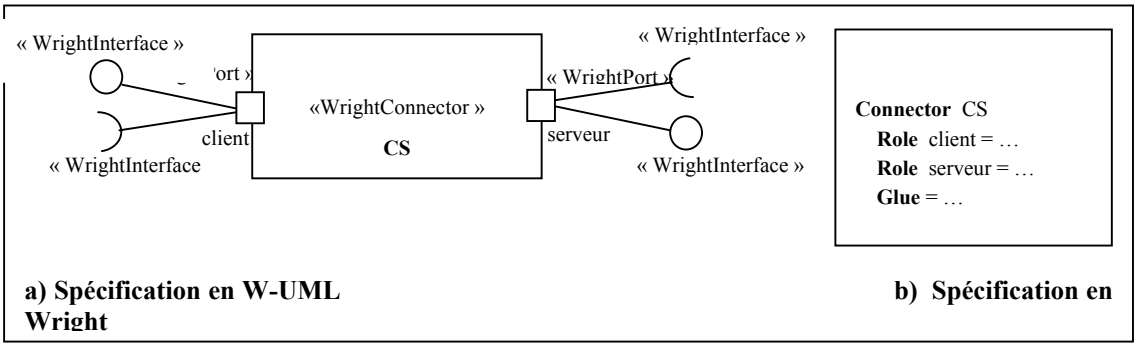


Figure 2.64. *Traduction d'un <<WrightConnector>> en Wright*

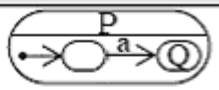
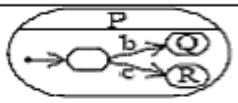
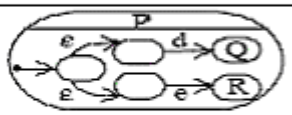
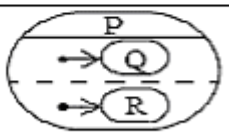
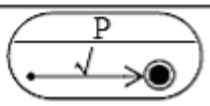
8.4 . Traduction de <<WrightProtocolStateMachine>>

- **Traduction 8.3 (<<WrightProtocolStateMachine>>)**

Un <<WrightProtocolStateMachine>> est traduit par une expression CSP conformément aux correspondances suivantes :

i) un <<WrightOperation>> e ayant un paramètre formel x de type in est traduit comme événement observé : e?x
ii) un <<WrightOperation>> e ayant un paramètre formel x de type out est traduit comme événement initialisé : $\overline{e!x}$
iii) Les autres correspondances entre les machines à états et les notations CSP sont regroupées dans le Tableau 2.2.

Tableau 2.2. ***Correspondances entre constructions machines à états d'UML et CSP***

Machine à états UML	Notation CSP
	P = a → Q
	P = b → Q □ c → R
	P = d → Q ⊓ e → R
	P = Q ‖ R
	P = √

- **Exemple**

La figure 2.65 illustre le principe de la traduction d'un << ProtocolStateMachine>> associé au port Consultation du Composant Titulaire de l'application bancaire en CSP de Wright. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification W-UML.

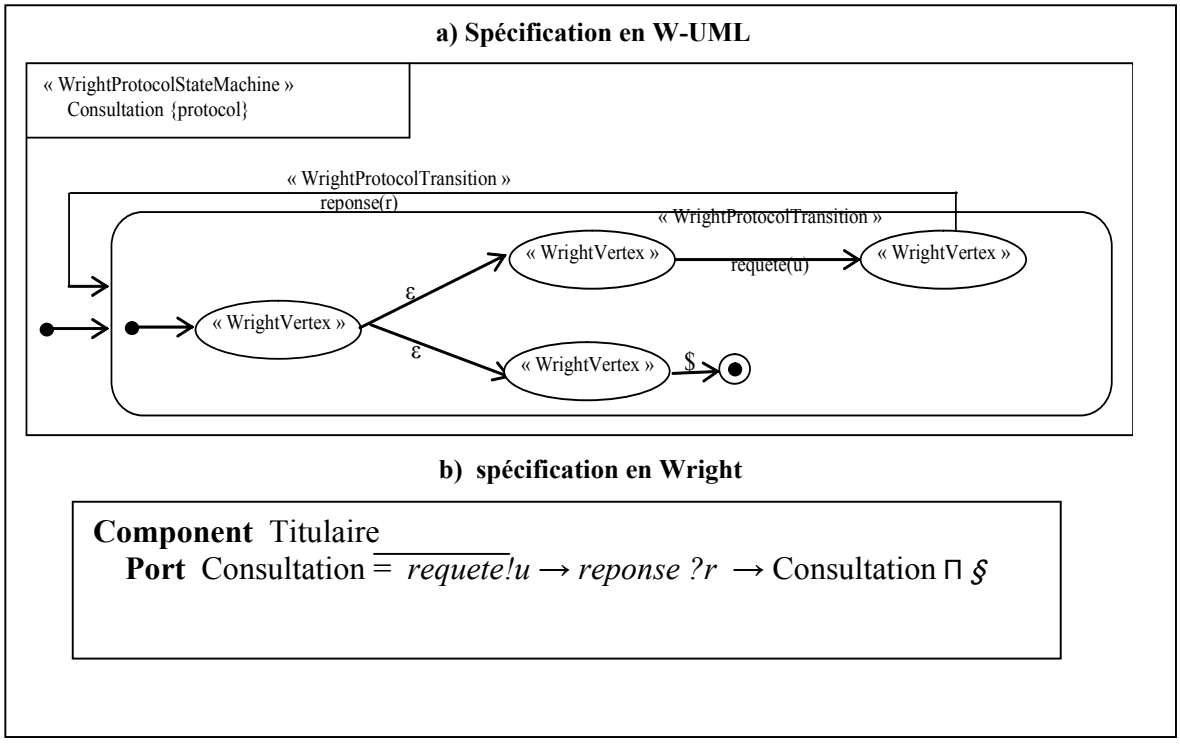


Figure 2.65. *Traduction d'un <<WrightProtocolStateMachine>> en CSP Wright*

8.5 . Traduction d'un diagramme d'instances W-UML

Un diagramme d'instances W-UML comporte :

- des composants de divers types à base de <<WrightComponent>> ;
- des connecteurs de divers types à base de <<WrightConnector>> ;
- des attachements liant des ports appartenant à des composants et des rôles appartenant à des connecteurs. Ces attachements sont à base de <<WrightAttachement>>.

- **Traduction 8.4 (diagramme d'instances W-UML)**

Un diagramme d'instances W-UML est traduit par une configuration Wright conformément aux correspondances suivantes :

- Toute instance appartenant au diagramme d'instance W-UML est traduite comme instance Wright.
- Tout <<WrightAttachement>> est traduit comme un attachement Wright.

- **Exemple**

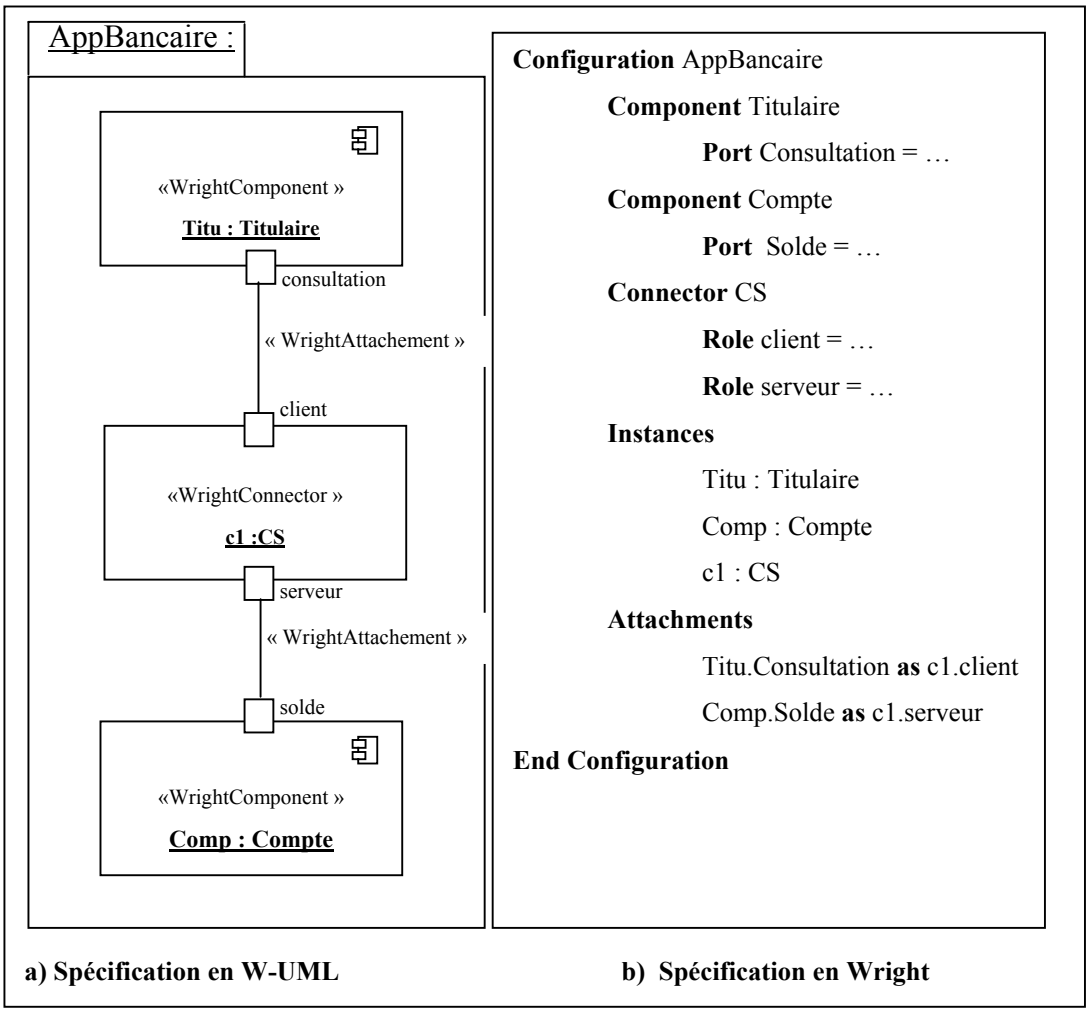


Figure 2.66. *Traduction d'un diagramme d'instances W-UML en Wright*

La figure 2.66 illustre le principe de la traduction d'un diagramme d'instances W-UML de l'application bancaire vers Wright. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification W-UML.

8.6 . Conclusion

Nous avons proposé des règles simples de traduction du profil W-UML vers Wright. Ces règles concernent la traduction en Wright des concepts<<WrightComponent>>, <<WrightConnector>>, <<WrightPort>>, <<WrightOperationt>>, <<WrightProtocolStateMachine>>> et <<WrightAttachement>> du profil W-UML. Des exemples illustratifs montrant l'application des règles proposées ont été fournis dans ce chapitre.

Troisième partie : Outils complémentaires et Validation

Chapitre 9 : Vers un outil de transformation d'expressions CSP en machines à états de description de protocoles UML

9.1 . Introduction

Nous avons outillé notre profil W-UML avec un transformateur d'expressions CSP en machines à états de description de protocoles stéréotypées par WrightProtocolStateMachine. Les fonctionnalités souhaitées de ce transformateur sont présentées dans la section suivante. Ensuite, nous proposons un analyseur lexico-syntaxique des expressions CSP écrit en Eiffel en se servant des deux bibliothèques LEX et PARSE (Meyer, 1994). Nous aurions souhaité valider notre profil W-UML directement sur des ateliers UML2.0 avec support OCL2.0, mais ces ateliers sont peu ouverts.

9.2 . Transformation des expressions CSP en machines à états de description de protocoles UML2.0

La vérification des architectures logicielles venant de l'ADL Wright à partir de spécifications UML exige la traduction des expressions CSP -décrivant les aspects comportementaux d'une architecture Wright- en machine à états de description de protocoles UML2.0. Une automatisation de cette activité de traduction entraîne les points positifs suivants :

- Les erreurs potentielles induites par une traduction manuelle peuvent être écartées.
- Les utilisateurs UML souhaitant récupérer des architectures décrites en Wright peuvent ne pas connaître CSP.

La structure macroscopique d'un outil de transformation des expressions CSP en machines à états de description de protocoles est donnée par la figure 3.1.

Le module "Analyse lexico-syntaxique" a pour objectif de vérifier si l'expression CSP soumise est bien formée ou non. Un tel module est basé sur les règles lexicales et syntaxiques du langage CSP utilisé par Wright.

Le module "Génération d'une machine à états de description de protocoles UML2.0" accepte en entrée une expression CSP jugée correcte et produit une machine à états de description de protocoles UML2.0 équivalente. Un tel module doit proposer une structure de données adéquate pour la mémorisation des machines à états d'UML.

Le module "Visualisation" permet de générer une représentation externe d'une machine à états de description de protocoles UML2.0 en se servant de la représentation interne proposée par le module précédent.

9.3 . Un analyseur lexico-syntaxique des expressions CSP

Nous avons développé un analyseur lexico-syntaxique des expressions CSP en réutilisant les deux bibliothèques LEX et PARSE fournies par l'environnement Eiffel. Dans un premier temps, nous allons présenter les aspects généraux de LEX et PARSE. Dans un deuxième temps, nous allons proposer une grammaire des expressions CSP adaptée aux exigences de la bibliothèque d'analyse syntaxique PARSE. Enfin, nous allons présenter les principales classes formant notre analyseur lexico-syntaxique des expressions CSP.

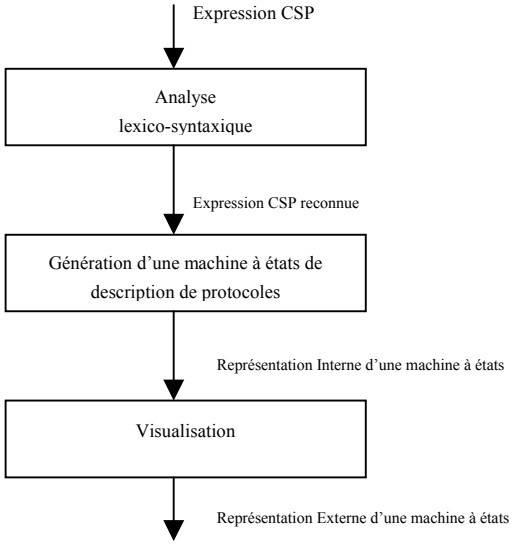


Figure 3.1. *Structure macroscopique d'un outil de transformation des expressions CSP*

9.4 . Bibliothèque LEX

La bibliothèque LEX est un ensemble de classes permettant la construction et l'utilisation d'analyseurs lexicaux de divers langages. Ces classes peuvent ainsi définir des grammaires lexicales pour beaucoup d'applications différentes, et produire des analyseurs lexicaux pour ces grammaires.

De point de vue utilisateur, les principales classes sont TOKEN, LEXICAL, METALEX et SCANNING (Meyer, 1994).

La classe TOKEN modélise une unité lexicale. Elle regroupe des caractéristiques relatives à une unité lexicale telles que : type (identificateur, mot réservé, constante,…), lexème et coordonnées dans le texte d'entrée.

La classe LEXICAL modélise un analyseur lexical relatif à une grammaire lexicale donnée c'est-à-dire à un ensemble d'unités lexicales.

La classe METALEX offre des possibilités permettant de construire des analyseurs lexicaux appartenant à la classe LEXICAL. En particulier, elle fournit des services permettant de lire la grammaire lexicale stockée dans un fichier et de construire l'analyseur lexical correspondant.

La classe SCANNING dérive de la classe METALEX. Elle apporte des possibilités liées à l'application d'un analyseur lexical au texte d'entrée.

La bibliothèque LEX propose un mécanisme basé sur les expressions régulières permettant de décrire d'une façon formelle les unités lexicales. De plus, LEX offre plusieurs expressions régulières prédéfinies décrivant des unités lexicales courantes telles que des constantes naturelles signées ou non signées et des constantes réelles.

9.4.1. Bibliothèque PARSE

La bibliothèque PARSE offre trois sortes de productions appelées agrégat, choix et liste permettant la description des constructions non terminales formant la grammaire syntaxique. Les sections suivantes décrivent les trois sortes de productions et les principales classes de la bibliothèque PARSE.

9.4.1.1. Agrégat

La production de type agrégat définit une construction dont les spécimens sont constitués d'un nombre fixé de composants. Par exemple, une expression CSP mise entre parenthèses est définie avec la production de type agrégat suivante :

Expression_parenthesee $\triangleq$[6]"(" Exp_csp")"

9.4.1.2. Choix

Une production de type choix définit une construction à l'aide d'un ensemble d'alternatives. Par exemple, dans CSP utilisé par Wright, un événement peut être un événement initialisé ou observé. Ceci est défini avec la production de type choix suivante :

Evenement $\triangleq$Ev_observe | Ev_initialise

Un spécimen d'Evenement est un spécimen de l'une des constructions données dans la partie droite.

9.4.1.3. Liste

Une construction peut avoir des spécimens faits d'une série de zéro, un ou plusieurs spécimens d'une autre construction déterminée. Ces spécimens sont séparés par un séparateur. Par exemple, un choix déterministe en CSP est défini par la production de type liste suivante :

Deterministe $\triangleq${Sequence "[]" ..}

9.4.1.4. Principales classes de PARSE

Les principales classes de PARSE sont données par la figure 3.2.

La classe CONSTRUCT est une classe abstraite ou différée. Elle favorise des caractéristiques communes aux quatre sortes de constructions supportées par PARSE : liste, agrégat, choix et terminale.

[6] Symbole défini.

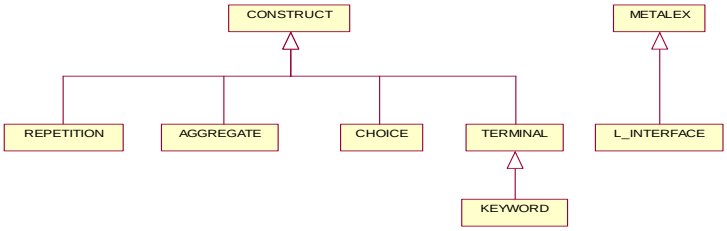


Figure 3.2. *Les principales classes de PARSE*

Les classes AGGREGATE, CHOICE et REPETITION représentent respectivement des constructions obtenues par des productions de type agrégat, choix et liste.

La classe TERMINAL modélise des constructions terminales ou lexicales.

La classe KEYWORD modélise des constructions terminales qui sont des mots-clés.

La classe L_INTERFACE assure l'interface entre l'analyseur syntaxique et l'analyseur lexical.

9.4.2. Une grammaire des expressions CSP adaptée à PARSE

La bibliothèque PARSE exige les deux contraintes suivantes :

- Une construction ne peut être définie que par une production ou plus.
- La grammaire syntaxique (ou la grammaire tout simplement) fournie ne doit pas être récursive à gauche. La bibliothèque PARSE offre un mécanisme permettant de vérifier l'absence de la récursivité à gauche dans la grammaire soumise.

Nous avons adapté la grammaire des expressions CSP pour prendre en considération les deux contraintes citées ci-dessus (cf. figure 3.4).

9.4.3. Réalisation

Le diagramme de classes illustré par la figure 3.5 modélise la grammaire des expressions CSP proposée dans la section précédente en réutilisant les classes REPETITION, CHOICE, AGGREGATE et TERMINAL offertes par la bibliothèque PARSE. Chaque construction introduite par cette grammaire est modélisée par une classe portant le même nom et dérivant directement ou indirectement impérativement en fonction de sa nature de REPETITION, CHOICE, AGGREGATE ou TERMINAL.

Dans la suite, nous allons décrire quelques classes issues de notre analyseur lexico-syntaxique (ou tout simplement syntaxique) des expressions CSP.

9.4.3.1. Classe Declaration

La classe Declaration modélise une construction syntaxique jouant le rôle de la racine de notre grammaire.

L'implémentation de la classe Declaration est donnée ci-dessous (cf. figure 3.3).

La classe Declaration implémente une construction non terminale définie par :

Declaration $\triangleq$ Def_nom "=" Exp_csp.

```
class Declaration
inherit
        AGGREGATE
                export
                        {PROCESS} all
                redefine
                        process,post_action
                end;
creation
        make
feature
        construct_name: STRING is
                once
                        Result := "Declaration"
                end; -- construct_name


production: LINKED_LIST [CONSTRUCT] is
                local
                        def_nom: Def_nom;
                        exp_csp: Exp_csp
                once
                        !!Result.make;
                        Result.forth;
                        !!def_nom.make;
                        put (def_nom);
                        keyword ("=");
                        !!exp_csp.make;
                        put (exp_csp)
                end; -- production
      post_action is
      do
        from
           child_start
        until
           child_after
        loop
            io.putstring(child.construct_name)
            child_forth
        end
        io.new_line
        io.new_line
        io.putstring("Fin action semantique.")
     end --post_action
     process is
     do
       parse
       if parsed
       then
          io.putstring("Expression reconnue:")
          io.new_line
          semantics
       else
         io.putstring("Expression non reconnue!!")
         io.new_line
      end
    end --process

end -- class Declaration
```

Figure 3.3. *Extrait de la classe Declaration*

```
Declaration     ≙ Def_nom "=" Exp_csp
Def_nom         ≙ Process_nom
Exp_csp         ≙ Choix | Parallel | Ex_where
Choix           ≙ Deterministe | Non_deterministe
Deterministe    ≙ {Sequence "[ ]".. }
Non_deterministe≙ {Sequence "|~|".. }
Sequence        ≙ {Facteur "->".. }
Facteur         ≙ Ev_exp | Exp_parenthesee | Nom_exp | Skip | Stop
Nom_exp         ≙ Identifier
Ev_exp          ≙ Ev_observe | Ev_initialise
Ev_initialise   ≙ Ev_simple | Ev_complexe
Ev_simple       ≙ "_" Ev_nom
Ev_complexe     ≙ "_" Ev_datalist
Ev_observe      ≙ Ev_nom | Ev_datalist
Ev_datalist     ≙ Exclamation | Interrogation
Exclamation     ≙ Ev_nom "!" Non_ev_data
Interrogation   ≙ Ev_nom "?" Non_ev_data
Ev_nom          ≙ Simple_nom
Non_ev_data     ≙ Simple_nom
Exp_parenthesee ≙ "(" Exp_csp ")"
Ex_where        ≙ Expression_w "Where" Declist
Expression_w    ≙ "(" Exp_csp ")"
Declist         ≙ Declaration
Skip            ≙ "Skip"
Stop            ≙ "Stop"
Parallel        ≙ {Process_nom "||" .. }
Process_nom     ≙ Simple_nom
Simple_nom      ≙ IDENTIFIER
```

Figure 3.4. *Grammaire PARSE des expressions CSP*

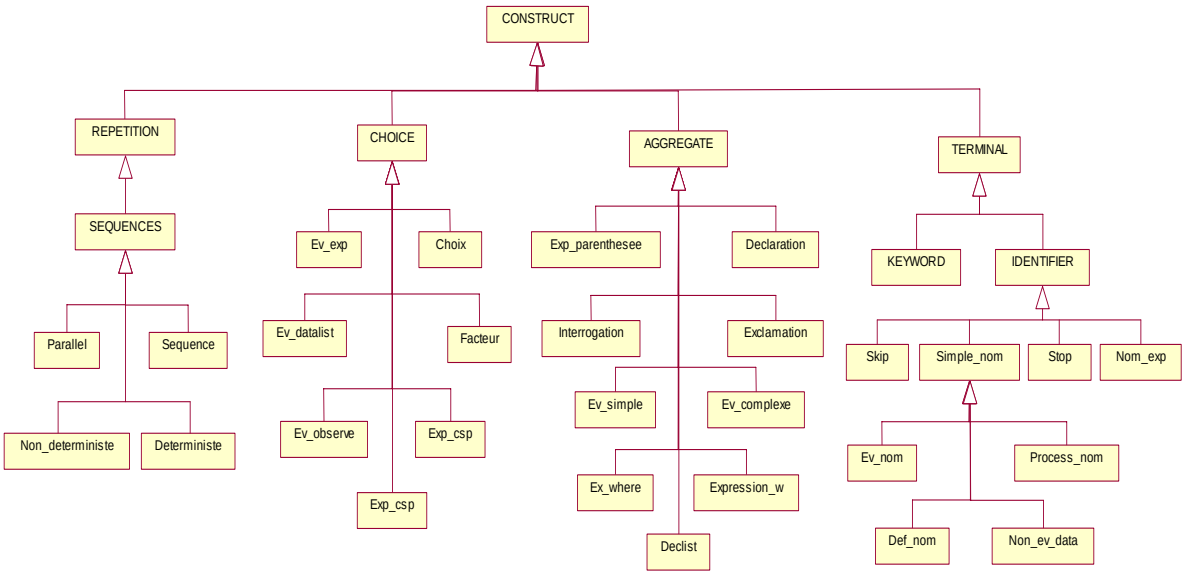


Figure 3.5. *Diagramme de classes des expressions CSP*

Cette règle est implémentée par la routine production. Celle-ci est introduite comme routine différée par la classe CONSTRUCT et elle demeure également différée dans les classes AGGREGATE, CHOICE et REPETITION. Toute classe modélisant une construction syntaxique non terminale doit proposer une implémentation à cette routine.

La routine post_action implémente l'action sémantique à effectuer une fois qu'un spécimen a été reconnu. L'implémentation actuelle de cette routine permet d'afficher les noms des constructions reconnues. Ultérieurement, une telle routine devrait être actualisée pour prendre en considération les fonctionnalités du module "Génération d'une machine à états de description de protocoles" (cf. section 2).

9.4.3.2. Classe Sequence

La classe Sequence implémente une construction non terminale définie par :

$$\text{Sequence} \triangleq \{\text{Facteur "->" .. }\}$$

L'implémentation de Sequence est donnée en annexe A. Les explications fournies en 9.4.3.1 sont valables pour cette classe. La seule particularité de cette classe est la routine production. En effet, Sequence hérite indirectement de REPETITION.

9.4.3.3. Classe Ev_exp

L'ADL Wright augmente la notation CSP pour faire la différence entre un événement initialisé recevant des données (événement avec une barre, suivi d'un point d'exclamation et d'une variable qui représente les données) et un événement observé fournissant des données (événement sans barre, suivi d'un point d'interrogation et d'une variable qui représente les données) (cf. chapitre 1).

La classe Ev_exp implémente une construction non terminale définie par :

$$\text{Ev\_exp} \triangleq \text{Ev\_observe} \mid \text{Ev\_initialise}.$$

Ceci traduit qu'un événement peut être un événement observé ou un événement initialisé. La classe Ev_exp hérite de la classe CHOICE. L'implémentation Ev_exp est donnée en annexe A.

9.4.3.4. Classe Simple_nom

Les spécimens de type Simple_nom sont formés par des lettres, des chiffres et des caractères « . » et « _ ». Cette classe hérite de la classe IDENTIFIER, qui hérite à son tour de la classe TERMINAL.

L'implémentation de la classe Simple_nom est donnée en annexe A.

9.4.3.5. Classe Racine

La classe PROCESS est la classe racine de notre bibliothèque au sens Eiffel. Cette classe hérite de la classe POLY_LEX qui définit tous les symboles qui peuvent apparaître dans une expression CSP. POLY_LEX hérite de la classe différée L_INTERFACE et de la classe CONSTANTS (cf. figure 3.6). Le contenu de la classe PROCESS est présenté en annexe A.

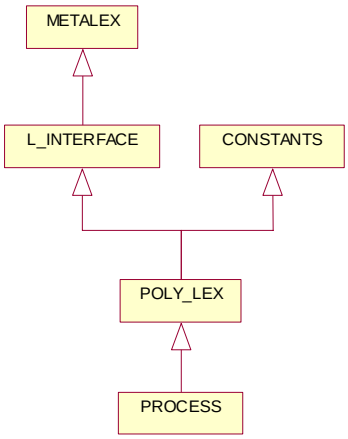


Figure 3.6. *Structure d'héritage de la classe PROCESS*

9.5 . Conclusion

Nous avons proposé un analyseur lexico-syntaxique des expressions CSP écrit en Eiffel en se servant des deux bibliothèques LEX et PARSE. Un tel analyseur forme le premier niveau d'un outil de transformation des expressions CSP en machines à états de description de protocoles d'UML2.0. Les deux autres niveaux "Génération d'une machine à états de description de protocoles UML2.0" et "Visualisation" sont en cours de réalisation.

Chapitre 10 : Vers une implémentation du langage de contraintes de l'ADL Wright

10.1 Introduction

L'ADL Wright définit une dizaine de propriétés (cf. chapitre 2), dites encore tests, souhaitables sur toute description architecturale dans ce langage. Parmi ces propriétés, à titre d'illustration, nous nous limiterons à la propriété 9, car cette propriété n'a encore fait l'objet d'aucune automatisation à notre connaissance.

Propriété 9 (contraintes de style) :

> *Les prédicats d'un style doivent être vrais pour une configuration déclarée être dans ce style.*

L'objectif recherché de ce chapitre est d'apporter une automatisation à cette propriété. Pour y parvenir, nous proposons un mécanisme d'évaluation des contraintes de Wright. Celles-ci sont assimilées à des expressions booléennes comportant des opérateurs logiques, des quantificateurs et des prédicats prédéfinis. Les contraintes Wright sont basées sur des ensembles prédéfinis.

Nous avons retenu le langage Eiffel (Meyer, 1994) en tant que langage de développement. Un tel choix peut être justifié par les aptitudes d'Eiffel vis-à-vis de la programmation contractuelle, la généricité, l'héritage multiple et des bibliothèques de classes liées aux algorithmes fondamentaux, aux structures de données universelles et à la génération d'analyseurs lexico-syntaxiques.

Ce chapitre comporte quatre sections. La première fait un rappel sur la notion de style d'architecture de Wright (cf. chapitre 1). La deuxième présente le langage de contraintes de Wright. La troisième présente les aspects conceptuels de l'évaluateur des contraintes de Wright proposé : EVALCWright. Enfin, la quatrième donne des indications sur l'implémentation et le test de notre évaluateur EVALCWright.

10.2 Notion de style d'architectures

Un style d'architectures permet de décrire un ensemble de propriétés communes à une famille de systèmes. Par conséquent, il décrit un vocabulaire commun en définissant d'une part un ensemble de types de composants et de connecteurs et d'autre part un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style.

Les styles d'architectures peuvent être vus comme des bibliothèques dans un langage de programmation qui sont mises à la disposition de l'architecte.

Ainsi Wright permet de définir des types de connecteurs et de composants pour une famille d'architectures. Les propriétés et les contraintes communes à une architecture peuvent être définies selon trois caractéristiques qui sont les types d'interfaces, les paramètres et les contraintes.

10.2.1. Composants, connecteurs et interfaces

Les styles permettent d'ajouter certains aspects dans la description des types composants et connecteurs. En effet on peut préciser que tous les composants d'un style doivent respecter les propriétés définies par la famille d'architectures à laquelle ils appartiennent.

Les types d'interfaces permettent d'identifier et factoriser certains traitements. On peut les utiliser dès qu'on a besoin d'un port appartenant à un composant ou d'un rôle appartenant à un connecteur. Ces types peuvent êtres vus comme des constantes dans un langage de programmation.

10.2.2. Paramètres

Les différents types dans un style peuvent supporter des paramètres qui seront précisés dans les configurations. Ils ont été introduits afin de fournir plus de flexibilité lors de la définition de types.

10.2.3. Contraintes

Les contraintes sont des prédicats logiques de premier ordre qui doivent être satisfaits pour tous les éléments appartenant au style. Dans la suite nous, allons décrire la notion de contrainte d'une façon détaillée.

10.3 Description du langage de contraintes Wright

Les contraintes sont l'un des aspects qui définissent un style dans Wright. Chaque contrainte déclarée dans un style représente un prédicat qui doit être satisfait par toutes les configurations implémentant ce style. Ces contraintes sont écrites par un langage formel mathématique en se basant sur la logique de premier ordre.

Les contraintes utilisent une syntaxe bien définie qui s'appuie sur des ensembles et des opérateurs spécifiques à l'ADL Wright. Nous décrivons :

- Components : L'ensemble des composants dans la configuration.
- Connectors : L'ensemble des connecteurs dans la configuration.
- Attachments : L'ensemble des liens dans la configuration. Chaque lien est représenté comme une paire de couple ((composant, port) ; (connecteur, rôle)).
- Name (e) : Le nom de l'élément e, où e est un composant, connecteur, port ou rôle.
- Type (e) : Le type de l'élément e.
- Ports (c) : L'ensemble des ports du composant c.

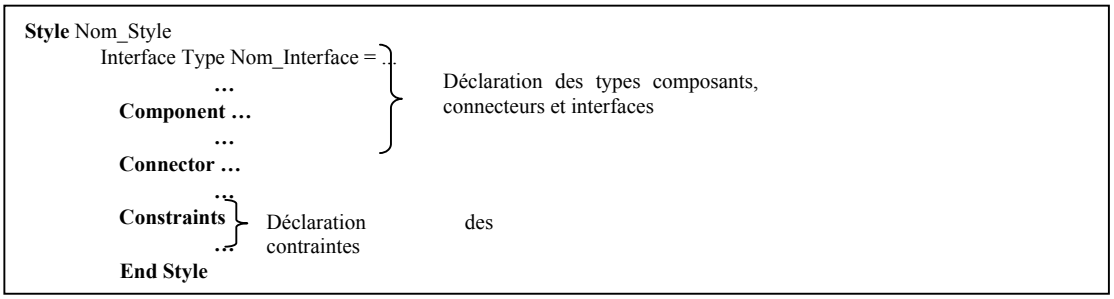


Figure 3.7. *Syntaxe du style*

- Computation (c) : Le calcul du composant c.
- Roles (c) : L'ensemble des rôles du connecteur c.
- Glue (c) : La glu du connecteur c.

$$\forall c : Components ; p : Port \setminus p \in Ports(c) \bullet Type(p) = DataInput \vee Type(p) = DataOutput$$

Figure 3.8. *Exemple de contrainte*

Les deux figure 3.7 et 3.8 donnent respectivement la syntaxe générale d'un style d'architecture et un exemple de contrainte.

10.4 Un évaluateur de contraintes de l'ADL Wright

Dans cette section, nous proposons un évaluateur de contraintes de Wright noté EVALCWright. Ce dernier est basé sur deux modèles conceptuels orientés objets. Le premier modèle permet de modéliser les ensembles et les prédicats prédéfinis du langage de contraintes de Wright. Et le second modèle permet de modéliser des expressions booléennes afin de les évaluer. Ces expressions peuvent utiliser avec profit des prédicats ou des ensembles prédéfinis fournis par le premier modèle. De plus, les deux modèles conceptuels exprimés en UML ont été réalisés en Eiffel sous forme d'une bibliothèque de classes. Enfin, notre évaluateur EVALCWright a été testé avec succès sur des exemples significatifs.

10.4.1. Structure macroscopique de l'évaluateur EVALCWright

10.4.1.1. Modélisation des ensembles et prédicats prédéfinis

Les contraintes Wright sont basées sur des ensembles et prédicats prédéfinis. La figure 3.9 décrit notre modélisation en UML sous forme d'un diagramme de classes de ces ensembles et prédicats prédéfinis. Pour modéliser un ensemble prédéfini, nous proposons, une classe UML (donc type) qui factorise les propriétés communes des objets qui seront créés et stockés dans cet ensemble. Par exemple, l'ensemble prédéfini Components - ensemble des composants dans la configuration - est modélisé par la classe Instance_Component. Cette classe va permettre de créer des instances de type component. Pour modéliser un prédicat prédéfini nous proposons soit une association UML, soit un attribut. Par exemple, le prédicat type -permettant de fournir le type d'une instance Wright telle que composant, port, rôle, et connecteur- est modélisé par une association entre les deux classes Instance_Component et Type_Component. Le prédicat name -permettant le nom de l'élément e, où e est un composant, connecteur, port et rôle- est modélisé par l'attribut name appartenant à la classe INSTANCE. Cette classe factorise des éléments communs aux classes Instance_Component, Instance_Connector, Port et Role.

10.4.1.2. Modélisation des contraintes

Une contrainte Wright est assimilée à un prédicat (ou fonction booléenne ou encore expression booléenne). Elle est modélisée comme un arbre dont les nœuds internes sont des opérateurs booléens (unaires, binaires et naires) et les nœuds externes sont des opérandes. Ces opérandes modélisent des sous-expressions booléennes basées sur des éléments

appartenant aux ensembles prédéfinis de Wright (figure 3.9). La figure 3.11 décrit notre modélisation des contraintes de Wright sous forme d'un diagramme de classes UML. Une telle modélisation est inspirée du pattern Composite de GoF (Gamma, 1999).

La classe PREDICAT joue le rôle d'une fondatrice d'une hiérarchie de classes. Il s'agit d'une classe différée. Elle modélise la notion d'une fonction booléenne au sens général, représentée par un arbre (figure 3.10)

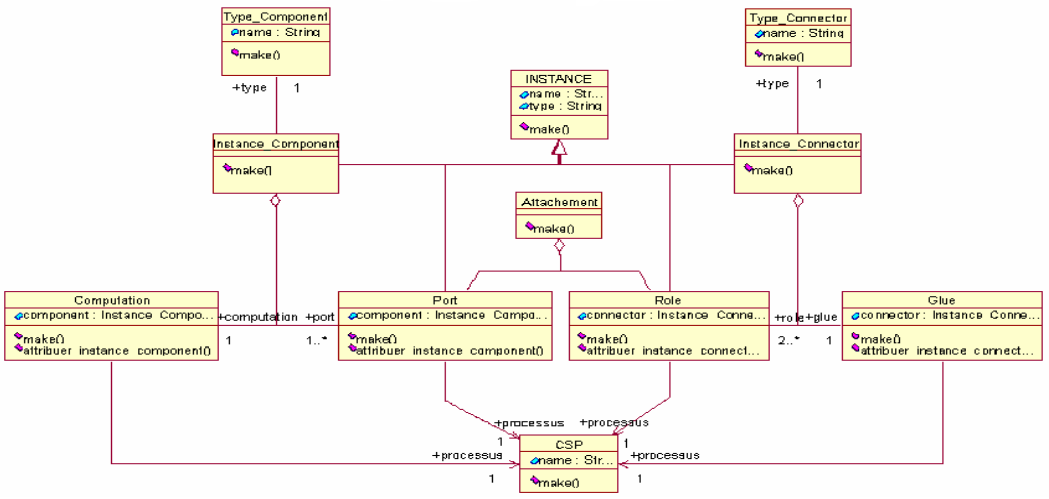


Figure 3.9. *Modélisation des ensembles et prédicats prédéfinis de Wright*

Il existe une relation d'héritage entre la classe PREDICAT et les classes UNAIRE, BINAIRE et NAIRE. En effet les trois classes modélisent respectivement un opérateur booléen unaire, binaire et naire au sens général. De plus il existe une association entre la classe PREDICAT et chacune de ces trois classes. Une telle relation est matérialisée dans le cas de la classe BINAIRE par deux attributs (gauche et droite) de type PREDICAT. Cette relation traduit le fait qu'un opérateur binaire exige deux opérandes : gauche et droite.

Concernant la classe EGAL, elle modélise la notion de comparaison entre deux chaînes de caractères pouvant provenir des prédicats prédéfinis tels que Type(e) ou Name(e) (où e désigne un élément de l'architecture).

Soit par exemple la contrainte suivante à vérifier où p1 est un port d'un certain composant.

$$\neg\ \text{type(p1)} = \text{« Data\_Input »} \vee \text{type(p1)} = \text{« Data\_Ouput »} \Leftrightarrow \text{name(p1.component)} = \text{« composant1 »}$$

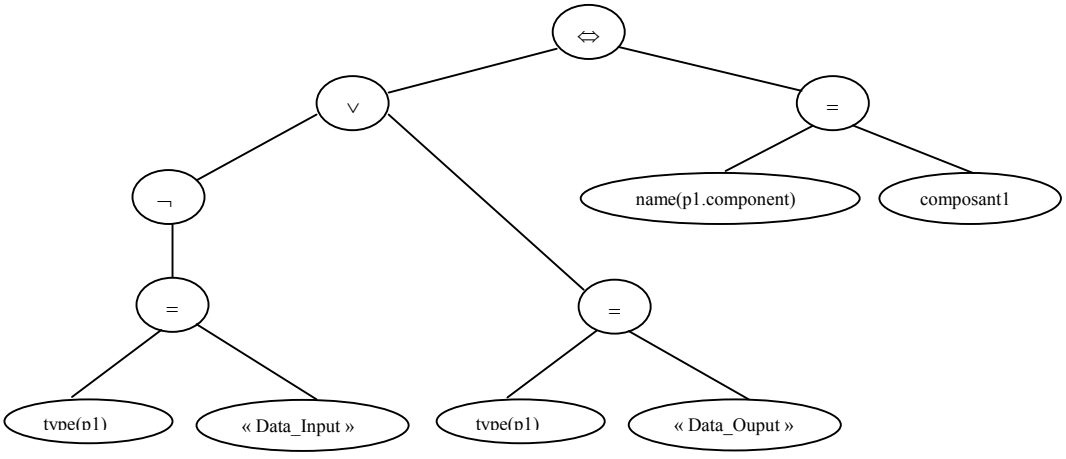


Figure 3.10. *Modélisation d'une contrainte*

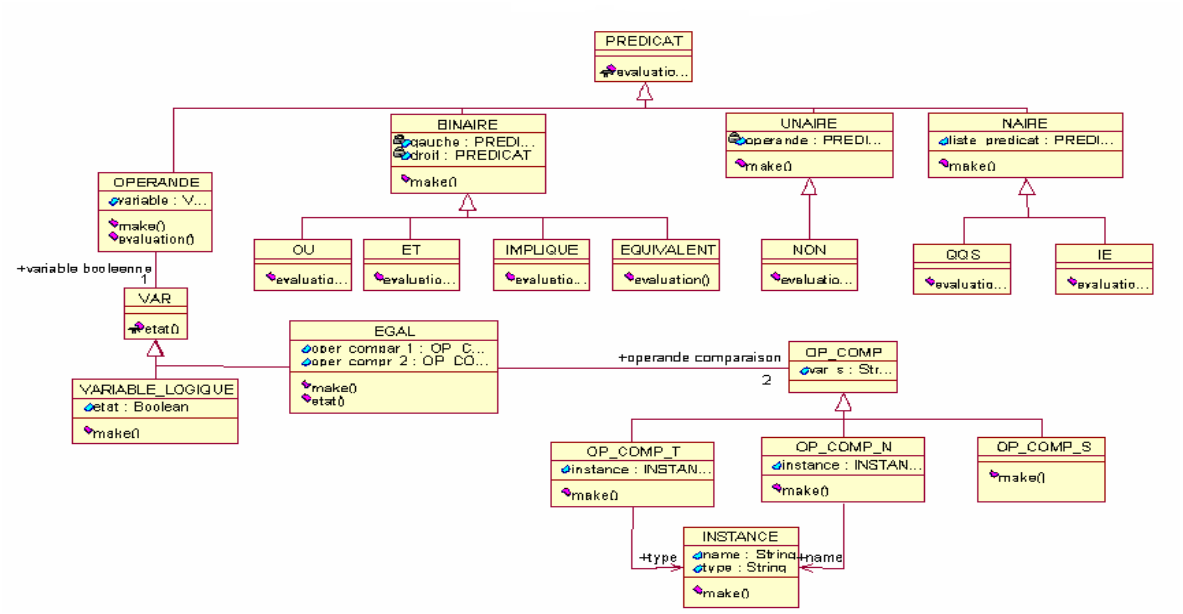


Figure 3.11. *Modélisation des contraintes*

10.5 Réalisation en Eiffel

L'évaluateur EVALCWright a été réalisé en Eiffel sous forme d'une bibliothèque de classes (cf. Annexe B). Ces classes sont considérées comme des types de données abstraits ayant des opérations bien définies grâce au contrats (ou assertions externes) supportés par Eiffel (précondition, postcondition et invariant). Nous avons utilisé avec profit les assertions externes et internes supportées par Eiffel pour tester soigneusement notre évaluateur EVALCWright.

10.6 Conclusion

Dans ce chapitre, nous avons conçu et réalisé en Eiffel un évaluateur de contraintes de l'ADL Wright : EVALCWright. Ce dernier permet d'automatiser la propriété 9 stipulant que les prédicats d'un style doivent être vrais pour une configuration déclarée être de ce style.

Notre évaluateur EVALCWright comporte deux modèles conceptuels orientés objets. Le premier a pour objectif de modéliser les ensembles et prédicats prédéfinis du langage de contraintes de Wright. Le second propose une modélisation des contraintes Wright considérées comme des expressions booléennes. L'évaluateur EVALCWright a été testé avec succès en utilisant les assertions externes et internes offertes par Eiffel.

Quant aux perspectives de ce travail, nous pourrions envisager les prolongements suivants :

- Identifier et intégrer des opérateurs fréquents dans la formulation des contraintes Wright telles que : $\leq$, $\geq$, $\neq$ …
- Concevoir et réaliser un analyseur lexico-syntaxique permettant de vérifier la validité d'une contrainte Wright,
- Générer la représentation interne basée sur les deux modèles proposés à partir d'une représentation externe des contraintes de Wright.

Chapitre 11 : La démarche Symphony

11.1 Introduction

Symphony (Hassine, 2005) est une méthode de développement logiciel élaborée par la société Umanis et l'équipe SIGMA (Laboratoire d'Informatique de Grenoble). Elle a pour but de spécifier les différentes phases d'un projet, de définir les tâches de chacun des intervenants, de contrôler les coûts, les délais et la qualité de l'application logicielle produite. Cette démarche se présente sous la forme d'un guide méthodologique offrant une solution basée sur l'utilisation de composants dès les phases amont du processus. Symphony s'appuie sur le langage unifié de modélisation UML et repose sur un certain nombre de pratiques de développement objet. Parmi les caractéristiques principales de cette démarche nous pouvons citer :

- une démarche itérative,
- un processus orienté utilisateur et piloté par les cas d'utilisation,
- une démarche orientée objets et composants métier,
- un modèle de développement en Y (André, 1994) (Larvet, 1994).

Symphony est adaptable selon la complexité des projets, leur durée ou leur coût de développement. Toutefois, elle comporte plusieurs points de passage obligés.

Elle assure quatre fonctions principales :

- déterminer les activités et les responsabilités des différents acteurs du développement,
- spécifier les produits à développer,
- guider la tâche des concepteurs, des développeurs et la coordination de l'équipe de développement,
- offrir des critères pour le contrôle et l'évaluation des activités du projet et des produits.

Ce chapitre est organisé en sept sections. La première section montre comment Symphony préconise le processus de développement en Y. La deuxième section présente la démarche itérative de Synphony. La troisième section montre que cette méthode est une démarche pilotée par les cas d'utilisation. La quatrième section présente l'orientation composant métier de la démarche. La cinquième section décrit les phases, les activités et les produits de la méthode Symphony. La sixième section présente le modèle conceptuel des composants métier Symphony et enfin on termine par une conclusion.

Ce chapitre est largement inspiré de la thèse d'Ibtissem Hassine (Hassine, 2005), doctorante de l'équipe SIGMA-Grenoble.

11.2 Processus de développement en Y

Le développement d'une application avec Symphony doit prendre en considération deux aspects fondamentaux et complémentaires :

- Les fonctions attendues de l'application pour répondre aux besoins du métier de l'entreprise.

– Les contraintes opérationnelles de l'application tel que les contraintes matérielles et logicielles, la qualité de service en temps de réponse, la tolérance aux pannes, etc.

Symphony sépare l'étude des besoins fonctionnels de celle des besoins techniques et ceci dès le début du cycle de vie des applications. Les deux aspect fonctionnels et techniques sont mis en évidence en adoptant un modèle de cycle de vie en Y (André, 1994) (Larvet, 1994). Cela permet une meilleure analyse du problème et des risques, mais aussi une meilleure réutilisation de l'existant. Menées en parallèle, ces deux activités se déroulent néanmoins en étroite collaboration afin d'assurer la cohérence. La figure 3.12, illustre le cycle en Y, dont nous pouvons observer trois branches : la branche fonctionnelle, la branche technique et la branche centrale de conception-réalisation.

– La branche fonctionnelle (gauche) correspond à la tâche traditionnelle de modélisation du domaine et des besoins des utilisateurs. Les résultats de l'analyse ne dépendent d'aucune technologie particulière et se résument à un modèle de composants métier (CM). La phase de spécification conceptuelle des besoins et la phase de spécification organisationnelle des besoins contribuent à une première identification des CM de haut niveau, et à un découpage ordonné (affectation de priorités) du processus de développement complet en itérations.

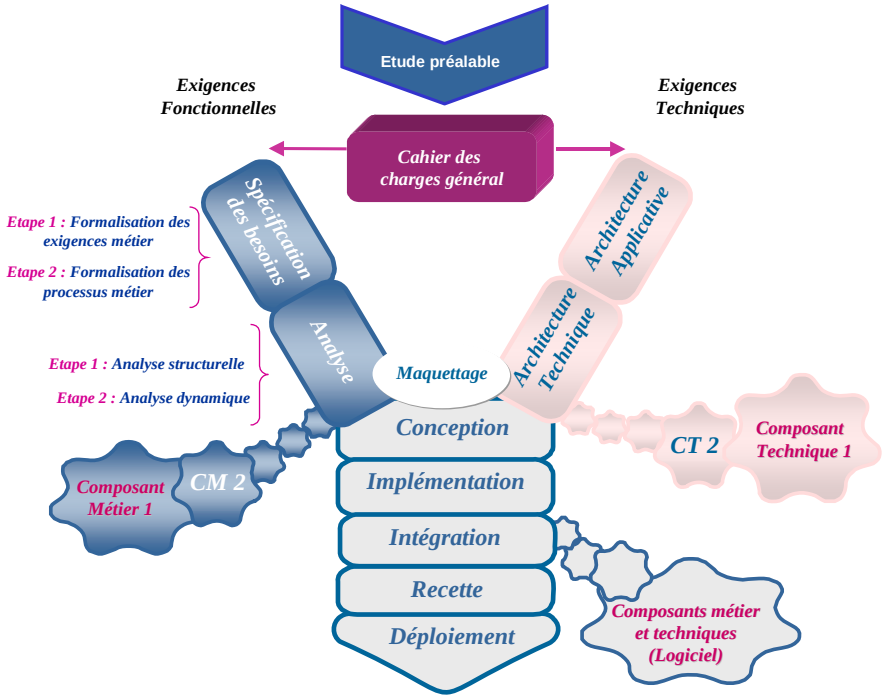


Figure 3.12. *Cycle de vie en Y de la démarche Symphony*

– La branche technique (droite) a pour objectif le recensement des contraintes et des choix techniques nécessaires pour la conception du système tels que la sécurité, la montée en charge, l'intégration à l'existant, etc. Ces éléments permettent, par la suite, l'élaboration des architectures applicatives et techniques de l'application.

– La branche centrale est une intégration des branches fonctionnelle et technique. Elle permet de réaliser une conception intégrant le modèle d'analyse dans l'architecture applicative de manière à obtenir un modèle de conception traçant les composants du système à développer. Ce modèle de conception est par la suite traduit dans un langage de programmation en utilisant les outils de développement choisis dans la branche technique. Les tests unitaires des composants sont réalisés au fur et à

mesure que leurs unités de code sont développées. La phase de recette permet de tester fonctionnellement l'application en suivant le cahier des charges préalablement rédigé. La phase de déploiement consiste enfin à mettre en production l'application testée techniquement et fonctionnellement.

Un tel modèle de cycle de vie apporte une réponse aux contraintes de changement continuel imposées aux systèmes d'information (selon les axes fonctionnel et technique) et permet ainsi une meilleure maîtrise des évolutions pour ces systèmes.

11.3 Démarche itérative

La démarche préconise un développement qui s'organise à travers un ensemble d'itérations. Chaque itération regroupe toutes les phases représentant l'enchaînement des différentes activités du processus. Chaque itération traduit donc un cycle de développement (cf. figure 3.13) et est centrée sur un sous-ensemble des exigences métiers (correspondant à un ou plusieurs processus métier).

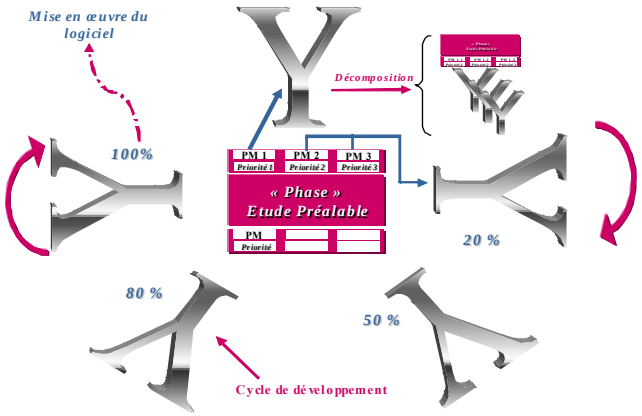


Figure 3.13. *Le processus itératif de Symphony*

Le résultat de chaque itération est un système testé, intégré et exécutable, mais incomplet et non encore prêt à être livré (un incrément). Il faudra une ou plusieurs itérations pour pouvoir le déployer dans un environnement de production. Le résultat n'est pas non plus un prototype[7]. En revanche, il représente un sous-système du système final et répond à un sous-ensemble des spécifications du système.

Pratiquement, pour chaque itération, il convient de choisir un ensemble d'exigences métier puis de les concevoir, de les implémenter et de les tester rapidement. Lors des premières itérations, les choix fonctionnels et techniques ne correspondent pas forcément au résultat final souhaité. Mais le fait de commencer la réalisation avant que les spécifications soient complètement finalisées ou que la totalité de la conception ait été pensée permet d'obtenir rapidement un retour d'évaluation des utilisateurs, des développeurs et aussi des testeurs. Ce retour d'information permet de remettre en cause l'exactitude des spécifications ou de la conception.

L'avancement du projet dépend des retours des évaluations des différents sous-systèmes résultats des différentes itérations. Avec ces retours d'évaluation, l'utilisateur final a la possibilité d'évaluer rapidement le système partiel et de donner ou non son approbation selon qu'il s'agisse ou non de fonctionnalités attendues du système final. Ainsi son avis

[7] Le développement itératif n'est pas une activité de prototypage.

intervenant en amont du processus constitue un bon moyen de découvrir en début de projet ce qui est réellement important pour lui.

11.4 Démarche pilotée par les cas d'utilisation

La démarche est orientée utilisateur et pilotée par les cas d'utilisation (Jacobson, 2000). En effet, elle intègre les besoins et les usages réels des utilisateurs dès les phases amont du développement. Ces besoins sont alors modélisés par des cas d'utilisation qui assurent la cohésion des activités et guident le processus de développement dans son ensemble.

Les modèles de cas d'utilisation décrivent les fonctionnalités complètes du système. A partir de ces modèles, les concepteurs créent une série de modèles de conception et de développement réalisant les cas d'utilisation. Les testeurs vérifient si les composants métier logiciels développés respectent correctement les cas d'utilisation. Les besoins des utilisateurs sont donc prioritairement traités dans la branche gauche du modèle en Y. Les cas d'utilisation constituent un mécanisme essentiel de traçabilité entre modèles (cf. figure 3.14).

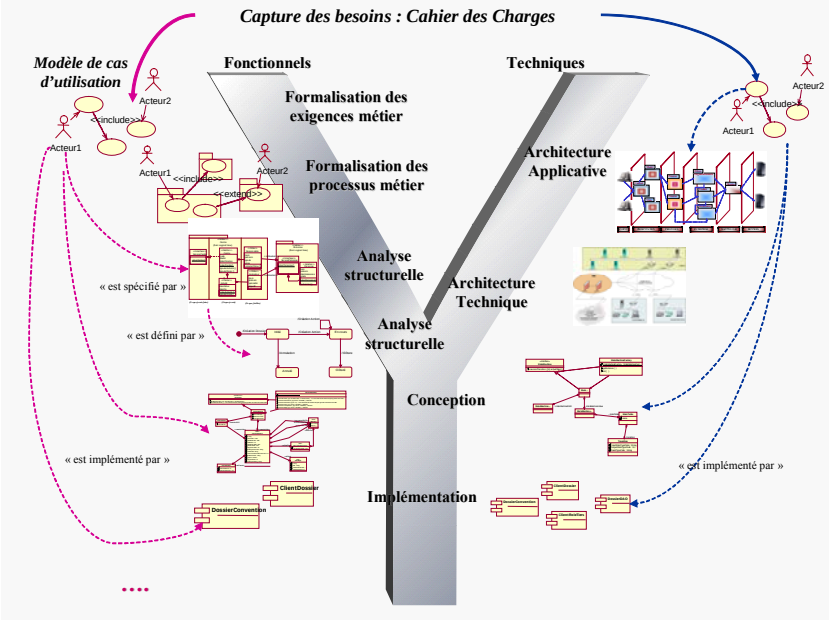


Figure 3.14. *Mécanisme de traçabilité basé sur les cas d'utilisation*

Les cas d'utilisation assurent donc une première forme de traçabilité entre les modèles représentant le système (modèle d'analyse, modèle de conception, modèle d'implantation, etc.). Les dépendances entre ces derniers et les modèles de cas d'utilisation sont formalisés par des relations de type dependency.trace (« est spécifié par », « est réalisé par », « est vérifié par », etc.).

Pour apporter des éléments de réponse aux problèmes et contraintes d'évolution du système et de ses besoins métier et technique, l'équipe de développement doit être en mesure de naviguer dans le système dans son ensemble et de pouvoir remonter d'un modèle à un autre pour faciliter la propagation des modifications. Cette deuxième forme de traçabilité est garantie par les liens entre les différents éléments des modèles.

La figure 3.15 montre une partie des liens de dépendance entre les éléments du modèle des besoins et ceux du modèle d'analyse (aux modèles de la branche fonctionnelle du Y).

La description des processus métier (PM) par des diagrammes d'activités permet l'identification des processus informatisés (PI) modélisés par des cas d'utilisation. Chaque cas d'utilisation informatisé affecté à un composant métier représente un service de ce

dernier et est spécifié dans le modèle d'analyse par une opération définie dans l'interface du composant métier. La traçabilité permet donc de préserver la cohérence du système et de l'actualiser dans son ensemble en fonction de l'évolution des besoins.

11.5 Une démarche orientée composant métier

La démarche est orientée composant métier car l'application est vue, tant au niveau conceptuel que logiciel, comme un assemblage de composants de type métier indépendants et interconnectés. Cette pratique garantit une bonne modularité des spécifications et facilite leur réutilisation.

La figure 3.12 met en évidence que les composants métier (resp. composants techniques) sont identifiés et analysés dans la branche fonctionnelle (resp. technique). Dans la branche centrale du Y, les composants métier conceptuels sont progressivement transformés en composants métier logiciels par intégration des choix techniques spécifiés dans les composants techniques.

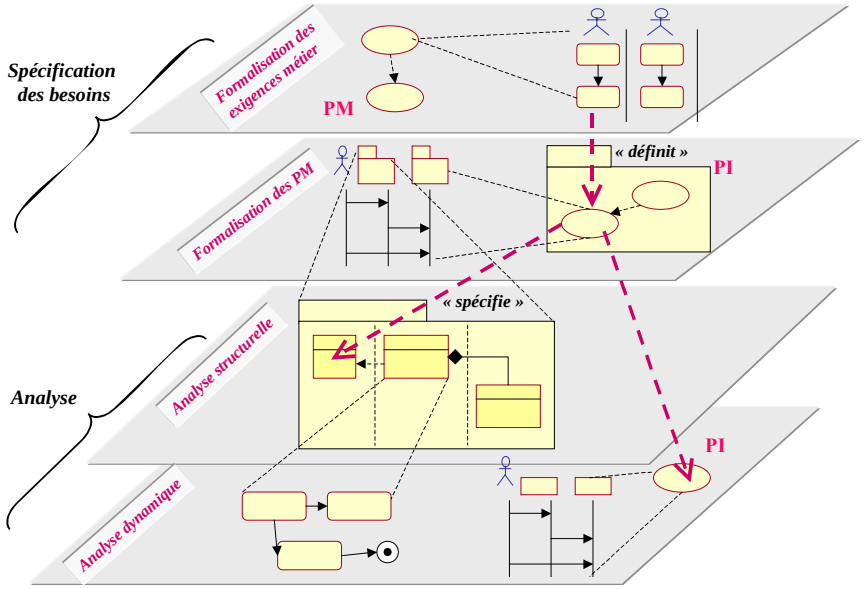


Figure 3.15. *Traçabilité par les cas d'utilisation dans la branche gauche du Y*

La démarche est basée sur un modèle de composants métier qui permet dès la phase de formalisation des exigences métier, l'identification des composants métier et de leur responsabilité dans le système d'information cible. Un composant métier représente un concept ou un processus du système d'information répondant à des besoins purement fonctionnels. Chaque composant métier est spécifié par une structure tri-partie mettant en évidence les services qu'il offre, sa structure interne et les composants auxquels il est lié.

Un composant métier pour un acteur extérieur correspond à une entité qu'il peut manipuler. Cette entité est apte à exécuter des opérations et maîtrise des informations qui lui permettent d'assumer des opérations qu'elle s'engage à réaliser. Les informations forment un réseau de concepts qui a un sens pour le métier.

Le modèle métier de la démarche spécifie trois types de composant métier :

– les composants « processus » permettent de décrire un processus applicatif (par exemple le processus Gestion des Agences, le processus Gestion du Personnel, etc.),

- les composants « entité » constituent la base des composants métier (par exemple des personnes, des agences, des points de vente, etc.),
- les composants « données » représentent les données de références (par exemple la codification des salaires).

De même que la démarche est basée sur les composants métier, elle l'est aussi sur les composants techniques. Ces derniers représentent des composants non-fonctionnels permettant de définir l'architecture technique de l'application et proposant des solutions à des problèmes techniques récurrents tels que la communication réseau, les connexions d'unités physiques, la persistance, etc.

11.6 Phases, activités et produits de la démarche Symphony

Le processus Symphony est constitué de plusieurs phases : spécification conceptuelle des besoins, analyse, architecture applicative, etc. Les phases sont structurées en activités. Ces activités concourent à la production d'un ensemble d'entités appelées ProduitSymphony ou « artefact » qui décrivent les différentes facettes du système final en fonction de celles fournies en entrée. Un Produit Symphony est d'un type donné, il peut s'agir d'un document texte, d'un diagramme UML, d'un exécutable, etc. Une activité Symphony est réalisée par un ou plusieurs ActeursSymphony, chacun pouvant être le responsable de la réalisation d'un certain nombre de produits du système.

La démarche se décompose en onze phases décrites dans le tableau 3.1. Cette décomposition inclut une phase préliminaire d'étude préalable du système d'information à développer. Chaque projet démarre après la spécification du cahier des charges général (CDCG). Ce dernier décrit les exigences métier du système à travers les processus métier qu'elles mettent en œuvre généralement décrits en langage naturel.

Le tableau 3.1 permet de visualiser les activités de chacune des phases de la branche gauche (et leurs étapes respectives). Dans la section suivante nous détaillerons le modèle conceptuel des composants métier Symphony (phase analyse), vis-à-vis de l'intérêt de notre travail.

Tableau 3.1. *Phases et activités de la méthode Symphony*

Phase/Etapes de la branche fonctionnelle	Description et Artefacts
Spécification conceptuelle des besoins	Modélise le domaine et les besoins d'utilisateurs **Artefacts** : modèle de cas d'utilisation.
Analyse	Permet de décrire l'architecture métier du système. **Artefacts** : Dossier d'architecture d'un composant métier.
	Branche commune
Maquettage	Son objectif est la mise en place de la cinématique de l'application, de la charte graphique, etc. La maquette résultat doit pouvoir refléter les besoins des clients et donc doit être testée et avalisée par ces derniers. Elle doit aussi pouvoir mettre en avant les difficultés majeures pour les développeurs, ce qui leur permettra de recentrer leur effort. **Artefacts** : Maquette (suite d'écrans montrant la navigation à travers l'application ainsi que son utilisation).
Conception	Décrit la transformation du modèle métier vers le modèle de conception par l'intégration des aspects techniques de l'application définis pendant la phase

	d'architecture applicative. **Artefacts** : Modèle de conception, Modèle de tables, Diagramme de classes de distribution, Diagrammes de séquences des objets métier (entité ou processus) enrichis par les choix techniques, DCT (Dossier de conception technique).
Développement	A pour objectif la traduction du modèle de conception dans un langage de programmation en utilisant les outils de développement. Cette phase inclut les tests unitaires des composants. **Artefacts** : Environnement de développement et un planning détaillé par développeur, Source de développements (codes sources des différents objets métier), Test unitaires et l'Application logicielle.
Intégration	Permet de « packager l'application », de tester techniquement l'application et de s'assurer de l'intégration globale des composants du système. **Artefacts** : Application testée techniquement.
Recette	A pour objectif de tester fonctionnellement l'application en suivant le cahier de recette préalablement rédigé. **Artefacts** : Planning des tests, Cahier de recette, fiches de test, Macros de tests, Suivi des résultats des tests, liste des fiches de non conformité.
Déploiement	Consiste à mettre en pilote ou en production l'application testée techniquement et fonctionnellement. **Artefacts** : Dossier d'intégration, Application de production déployée.

11.7 Modèle conceptuel des composants métier Symphony

Dans Symphony, l'architecture des CM est une structuration inspirée de la technique CRC (Classe-Responsabilité-Collaboration) (Wirfs-Brock, 1990). Un CM est modélisé par un paquetage composé de trois parties (cf. figure 3.16) et de quatre catégories de classes :

- une partie contrat avec l'extérieur (ce que je sais faire),
- une partie structurelle (ce que je suis),
- une partie collaboratrice (ce que j'utilise).

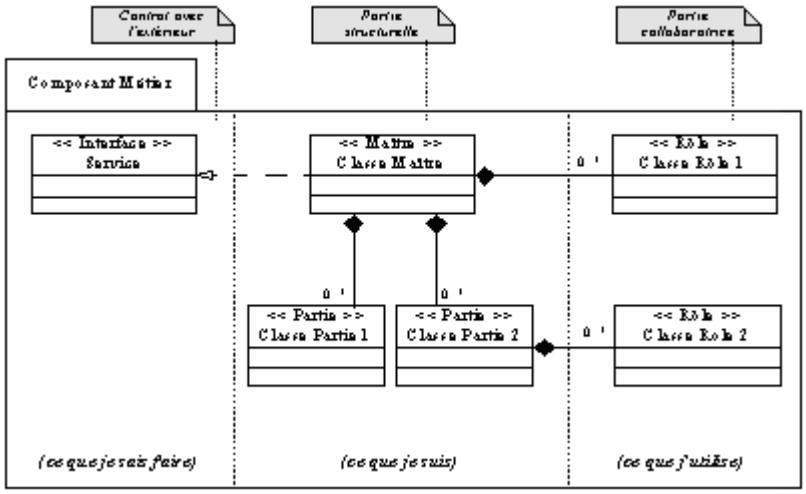


Figure 3.16. *Architecture d'un composant métier Symphony*

11.7.1 . Classe Interface

Cette classe correspond à la porte d'entrée du composant métier, elle définit les opérations réalisées par le composant métier dont l'exécution laisse le composant dans un état cohérent : partie contrat avec l'extérieur. La classe Interface permet d'accéder à la structure interne du composant métier et d'exécuter les services qu'il offre. La classe Interface définit la signature des opérations représentant les cas d'utilisations pour lesquels le composant métier a été déclaré responsable. Cette classe supporte également les opérations qui sont déclenchées par les messages des diagrammes de séquence (définis dans la phase d'expression des besoins). Il s'agit alors d'opérations nécessaires à l'exécution d'un cas d'utilisation.

11.7.2. Classe Maître

C'est la classe principale du composant métier pour lequel les opérations sont réalisées. Sa connaissance est nécessaire au préalable de toutes les manipulations du composant. Nous pouvons qualifier cette classe de hiérarchiquement supérieure. La classe Maître est autonome : une fois créée, elle ne dépend pas directement d'autres classes si ce n'est au travers de ses propres règles de gestion. Elle justifie son existence par la prise de responsabilité des finalités du système.

Pour être une classe Maître, la classe candidate doit vérifier les caractéristiques suivantes :

- être le centre du problème,
- exister en un seul exemplaire lorsque l'on traite le problème (être mono-instanciée).

La classe Maître décrit les éléments d'identification du composant métier qui permettent de rechercher le composant métier dans le cas où un composant métier client a besoin de l'un de ses services.

11.7.3. Classe Partie

C'est une classe complémentaire de la classe Maître ayant un intérêt dans son contexte. La classe Partie est reliée à la classe Maître par une relation de composition. Une classe Partie est une structuration des attributs de la classe Maître. Ce découpage fournit une structuration des données métier à partir de la classe Maître pour permettre soit une mise en valeur d'un aspect du composant métier, soit une multi-occurrences d'un concept ou un aspect optionnel d'une caractéristique.

11.7.4. Classe Rôle

Une classe Rôle représente un fournisseur de services auprès du composant client : partie collaboratrice. Un fournisseur est un autre composant qui rend un ou plusieurs services ; ces services sont appelés chez un client au travers d'une classe Rôle. En UML, cette dépendance est représentée par un lien de dépendance de stéréotype « utilisation ». Les attributs importés dans le rôle sont représentés par la notation d'attribut dérivé « / ». De plus, la classe Rôle peut aussi porter des attributs propres dont la pertinence n'est effective que dans le cadre du composant métier.

11.8 Conclusion

Nous avons présenté sommairement dans ce chapitre la méthode Synphony qui a pour but de spécifier les différentes phases d'un projet et de définir les tâches de chacun des intervenants. Ensuite on a mis l'accent sur la phase d'analyse (branche gauche) qui s'appuie

sur un modèle de composants organisé en de trois parties : interface, structure, collaboration.

D'une manière générale, les composants métier et leurs liaisons constituent l'architecture ou cartographie d'un système. Une telle architecture est compréhensible pour les acteurs du domaine d'application et elle est maintenue au niveau développement logiciel.

Dans le chapitre suivant nous utiliserons cette démarche en modélisant l'application «ferme» avec des composants métier Synphony.

Notre objectif est d'enrichir cette phase d'analyse de la démarche Symphony en faisant la projection du modèle conceptuel des composants métier Symphony vers le profil W-UML (cf chapitre 7), dans un but de vérifier des propriétés standards et des propriétés spécifiques d'une architecture logicielle décrite en Symphony.

Chapitre 12 : Traduction Symphony vers W-UML

12.1 . Introduction

Dans le but de pouvoir transformer une application décrite en Symphony vers le profil W-UML en vue d'une vérification des propriétés standards et spécifiques d'une architecture logicielle décrite en Symphony, nous adoptons une démarche de transformation qui se base sur des règles de correspondance entre les concepts de base du modèle conceptuel des composants métier Symphony (cf. figure 3.16) et les concepts de base du profil W-UML (cf. chapitre 7). Nous allons nous appuyer sur le modèle conceptuel des composants métier de Symphony et le profil W-UML en vue de spécifier une traçabilité entre les deux modèles et présenter ainsi un catalogue de schémas de transformation. Nous commençons à présenter une modélisation de l'application « ferme » en Symphony, application déjà modélisée en Wright (cf. chapitre 1-1.4).

12.2 . Modélisation de l'application «Ferme» en Symphony

12.2.1 . Démarche pour la description de l'application «Ferme» en Symphony

Pour pouvoir décrire l'application «Ferme» en Symphony nous suivons la démarche suivante largement inspirée des travaux d'Ibtissem Hassine (Hassine, 2005) :

- Identifier les composants métier de l'application,
- Pour chaque composant métier, modéliser en Symphony les deux parties (ce que je sais faire et ce que je suis),
- Etablir les relations Client-FournisseurClient entre les différents composants métier et modéliser ces relations en Symphony (partie ce que j'utilise),
- Associer la partie comportementale à la partie structurelle.

12.2.2 . Identification et modélisation des composants métier de l'application «Ferme»

12.2.2.1 . Identification des composants métiers

Le but de cette phase est de comprendre la finalité attendue par chaque acteur du système. Comprendre, c'est répondre aux questions suivantes : pour qui, pour quoi, dans quelles conditions, comment et quels résultats ?

Dans notre cas, un seul acteur principal est identifié : la ferme qui est sous la responsabilité d'un fermier. Pour gérer sa ferme le fermier emploie n contremaîtres et chaque contremaître est responsable de m ouvriers (cf. figure 3.16).

Le scénario décrivant ce cas d'utilisation est le suivant :

- la ferme est sous la responsabilité d'un fermier,
- un fermier emploie n contremaîtres,
- chaque contremaître est responsable de m ouvriers.

L'analyse du scénario permet de définir le diagramme des cas d'utilisation (cf. figure 3.17).

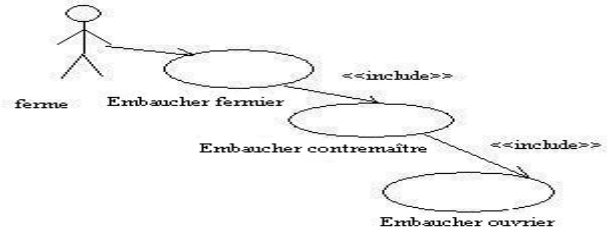


Figure 3.17. *Diagramme de Cas d'utilisation*

Une fois les cas d'utilisation précisés, le concepteur les affecte aux composants métier candidats les plus susceptibles de les assumer. Une technique simple pour déterminer le composant est d'appliquer une analyse syntaxique sur le nom du cas d'utilisation. Il faut choisir le complément du verbe nommant le cas d'utilisation. La cartographie finale des composants métier est proposée dans la figure 3.18.

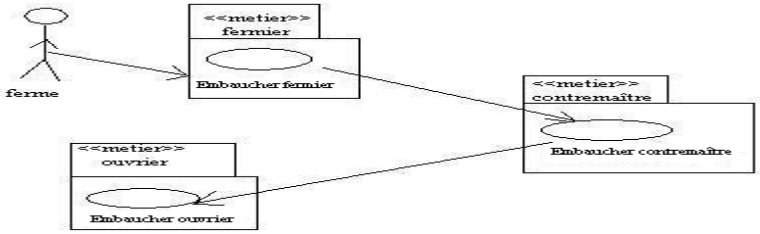


Figure 3.18. *Cartographie des composants métiers*

Ces représentations sont complétées par d'autres diagrammes UML. En particulier, des diagrammes d'activités permettent l'identification des cas d'utilisation informatisés et des diagrammes de séquences dits de haut niveau permettent d'exprimer pour chaque cas d'utilisation les demandes de services entre les composants métier.

La cartographie obtenue par regroupement de cas d'utilisation constitue une première représentation des composants métier. Elle est utilisée par la suite pour spécifier les composants métier, les adapter et les connecter.

12.2.2.2. Composant Fermier

Le composant métier Fermier offre un service ayant la signature suivante :

TraiterDemande(Contremaître c) permettant de traiter les demandes venant d'un contremaître donné. Il exige des services requis fournis par le composant métier. La multiplicité de la relation de composition entre les deux classes Fermier et RContremaître renseigne sur le nombre de contremaîtres sous la responsabilité du fermier (cf. figure 3.19).

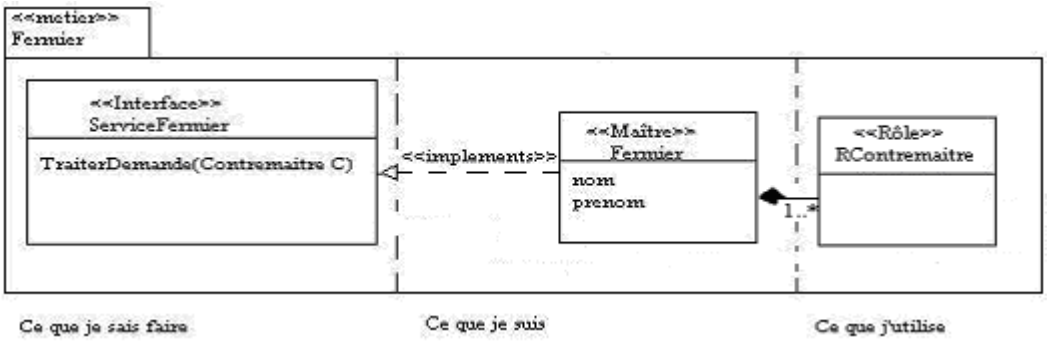


Figure 3.19. *Composant métier Fermier*

12.2.2.3. Composant Contremaître

Le composant métier Contremaître offre une interface ayant deux services *TraiterDemande(Ouvrier o)* et *AffecterTache(Tache T)* permettant respectivement de traiter les demandes venant du composant métier Ouvrier et de recevoir les tâches venant du composant métier Fermier. Il exige des services venant des composants métier Fermier et Ouvrier. Ceci est matérialisé par les deux classes RFermier et ROuvrier. La multiplicité (1) de la relation de composition entre Contremaître et RFermier traduit le fait qu'un contremaître est sous la responsabilité d'un seul fermier. Tandis que la multiplicité (1..*) de la relation de composition entre Contremaître et ROuvrier indique qu'un contremaître exerce son pouvoir sur plusieurs ouvriers (cf. figure 3.20).

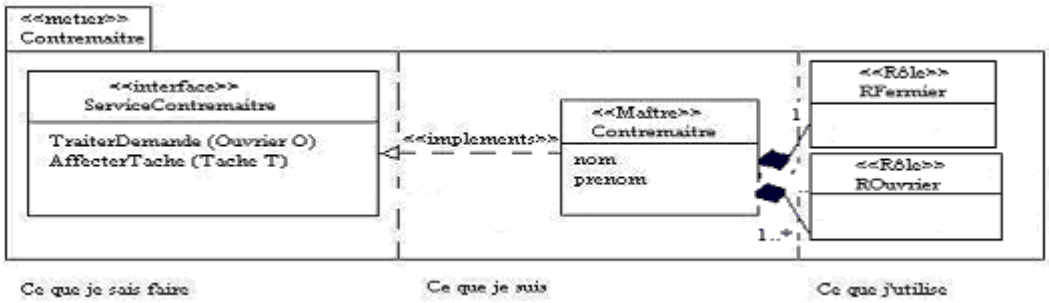


Figure 3.20. *Composant métier Contremaître*

12.2.2.4. Composant Ouvrier

Le composant métier Ouvrier offre une interface ayant un seul service *recevoirTache(Tache T)* permettant de traiter les tâches venant du composant Contremaître. Il exige des services venant du composant Contremaître. La multiplicité (1) de la relation de composition entre Ouvrier et RContremaître traduit le fait qu'un ouvrier s'adresse à un seul contremaître pour obtenir de nouvelles tâches à effectuer (cf. figure 3.21).

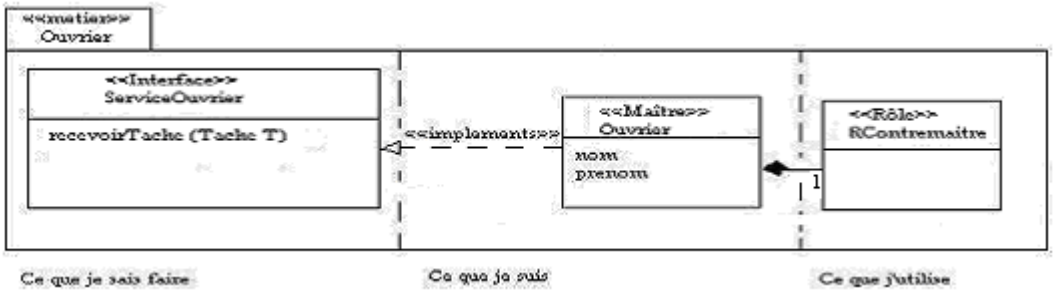


Figure 3.21. *Composant métier Ouvrier*

12.2.2.5. Les relations Client-Fournisseur entre les différents composants métier

Il y a une relation bilatérale entre les deux composants métier Fermier et Contremaître. Du côté Fermier (cf. figure 3.22), la classe Maître Fermier se compose de un à plusieurs Rôles (RContremaître) et la classe Rôle RContremaître utilise les services de la classe Interface ServiceContremaître. Du côté contremaître (cf. figure 3.23), la classe Maître Contremaître se compose de un à plusieurs Rôles (RFermier) et la classe Rôle RFermier utilise les services de la classe Interface ServiceFermier.

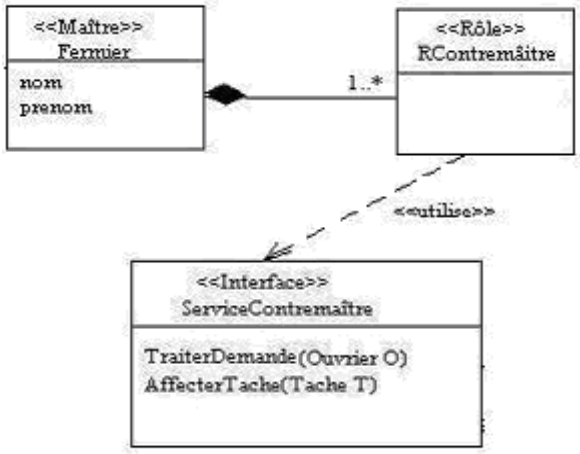


Figure 3.22. *Relation Client-Fournisseur entre les deux composants métier Contremaître et Fermier du côté Fermier*

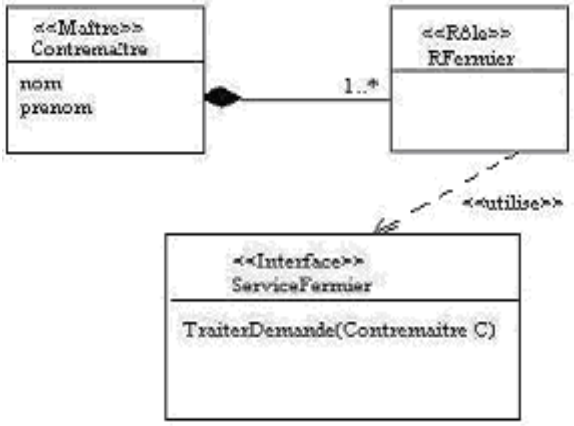


Figure 3.23. *Relation Client-Fournisseur entre les deux composants métier Contremaître et Fermier du côté Contremaître*

Il y a une autre relation bilatérale entre les deux composants métier Contremaître et Ouvrier. Du côté Contremaître (cf. figure 3.24), la classe Maître Contremaître se compose de un à plusieurs Rôles (ROuvrier) et la classe Rôle ROuvrier utilise les services de la classe Interface ServiceOuvrier. Du côté Ouvrier (cf. figure 3.25), la classe Maître Ouvrier se compose de un à plusieurs Rôles (RContremaître) et la classe Rôle RContremaître utilise les services de la classe Interface ServiceContremaître.

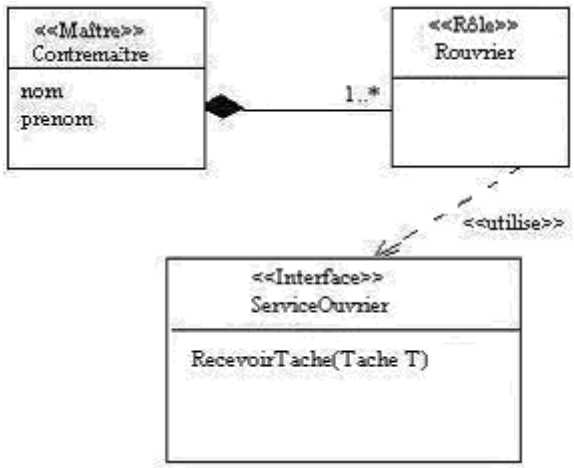


Figure 3.24. *Relation Client-Fournisseur entre les deux composants métier Contremaître et Ouvrier du côté Contremaître*

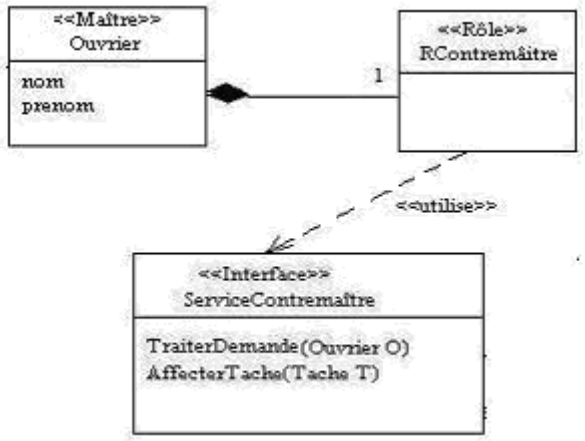


Figure 3.25. *Relation Client-Fournisseur entre les deux composants métier Contremaître et Ouvrier du côté Ouvrier*

En conclusion le composant métier Fermier a un seul type de rôle appelé RContremaître.

Le composant métier Contremaître a deux types de rôles : RFermier et ROuvrier.

Le composant métier Ouvrier a un seul type de rôles appelé RContremaître.

12.2.3. Scénario d'utilisation

Un scénario d'utilisation est donné dans la figure 3.26 pour une configuration dont n est égale à 1 et m est égale à 1 (un seul contremaître et un seul ouvrier) : f est une instance de Fermier, c1 est une instance de Contremaître et o1 est une instance d'Ouvrier.

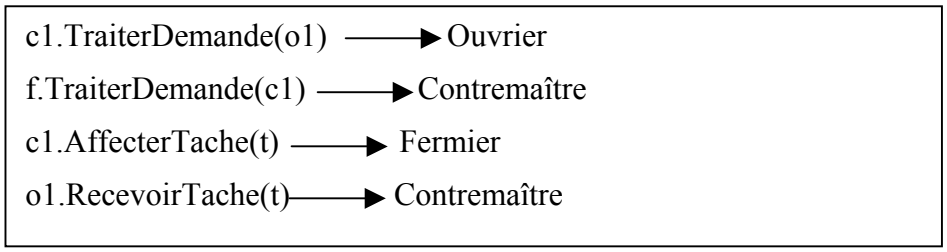


Figure 3.26. *Scénario d'utilisation*

12.2.4. Description de l'aspect architecture

La figure 3.27 décrit l'aspect structurel de l'application «Ferme» en Symphony en reliant les trois composants métier Symphony (cf. la section 12.2.2.) Fermier, Contremaître et Ouvrier par les relations client-fournisseur identifiées dans la section 12.2.5.

12.2.5. Description de l'aspect comportemental

Dans la méthode Symphony, les aspects comportementaux des composants métier ne sont pas traités d'une façon explicite. Pour y parvenir nous proposons :

- d'associer une machine d'états ou diagramme d'états-transitions à chaque composant métier permettant de décrire son comportement **global**. Conformément à UML1.x -le langage de modélisation utilisé par Symphony-, une telle machine d'états est associée à la classe <<Maître>> ;
- d'associer à chaque classe <<Rôle>>, une machine d'états permettant de décrire le comportement **partiel** du composant métier correspondant.

Par exemple le comportement partiel du composant métier Ouvrier via la classe RContremaître peut être décrit de la manière suivante :

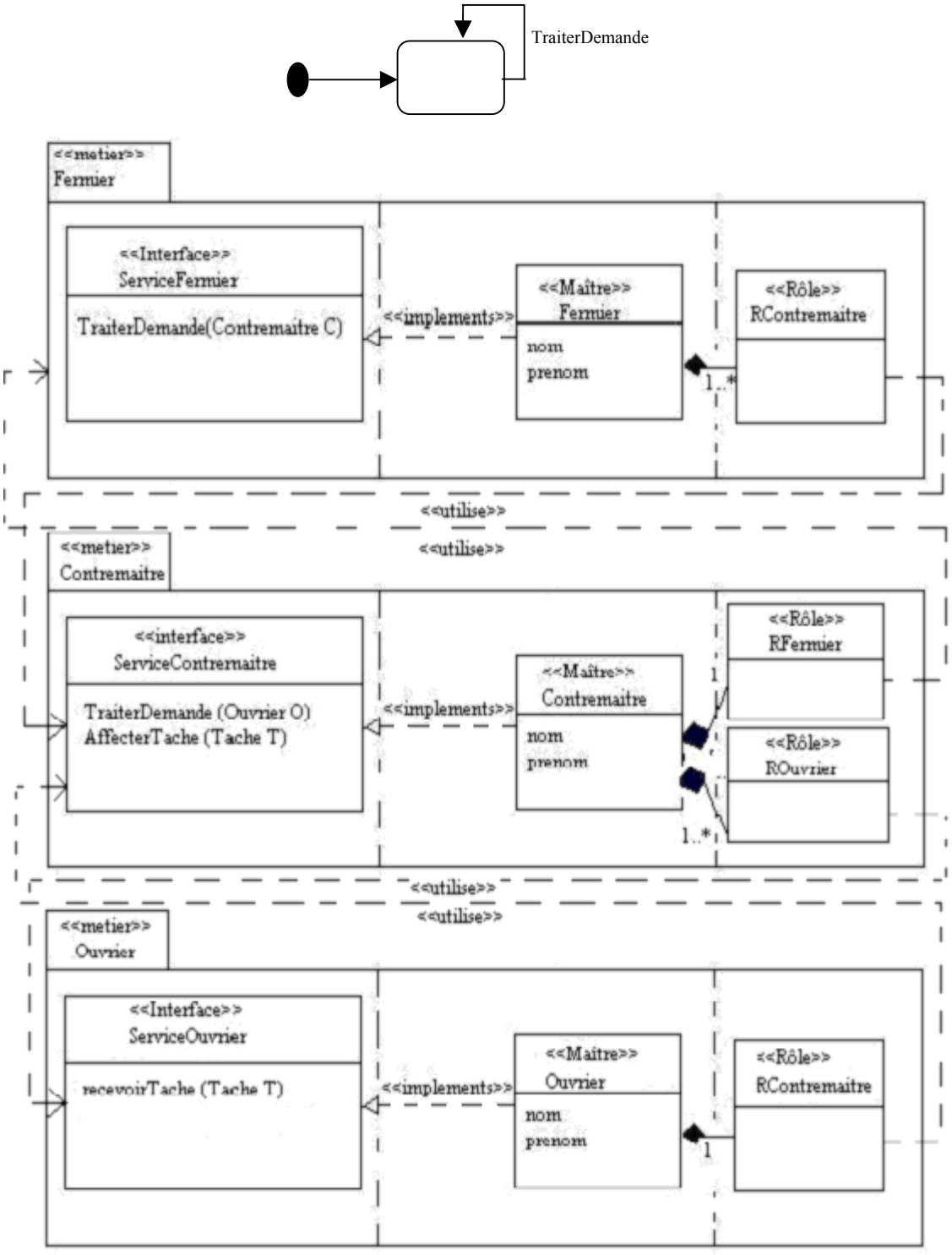


Figure 3.27. *Description de l'aspect structurel de l'application «Ferme» en Synphony*

12.3 . Transformation du modèle de composant métier Symphony vers le profil W-UML

12.3.1 . Traduction des aspects structuraux

Les concepts structurels de la méthode Symphony sont respectivement le Composant métier Symphony (Classe Maître et Rôle) et la relation Client-Fournisseur entre deux composants Symphony. Nous allons donner les règles de traduction en les illustrant par l'exemple de la «Ferme».

12.3.1.1. Traduction du Composant métier Symphony

Un composant métier Symphony est un package qui regroupe les classes Interface, Maître et Rôle.

- **Traduction 12.1 (Composant métier Symphony)**

Tout type de composant métier Symphony est traduit en <<WrightComponent>> du profil W-UML. Les ports attachés au <<Wright Component>> sont obtenus par la règle suivante : chaque classe Rôle attachée au composant métier est traduit par un <<WrightPort>> en tenant compte de la multiplicité de la relation de composition entre les deux classes <<Maître>> et <<Rôle>>.

La figure 3.28 donne la traduction du composant métier Symphony Contremaître de l'application <<Ferme>> vers le profil W-UML.

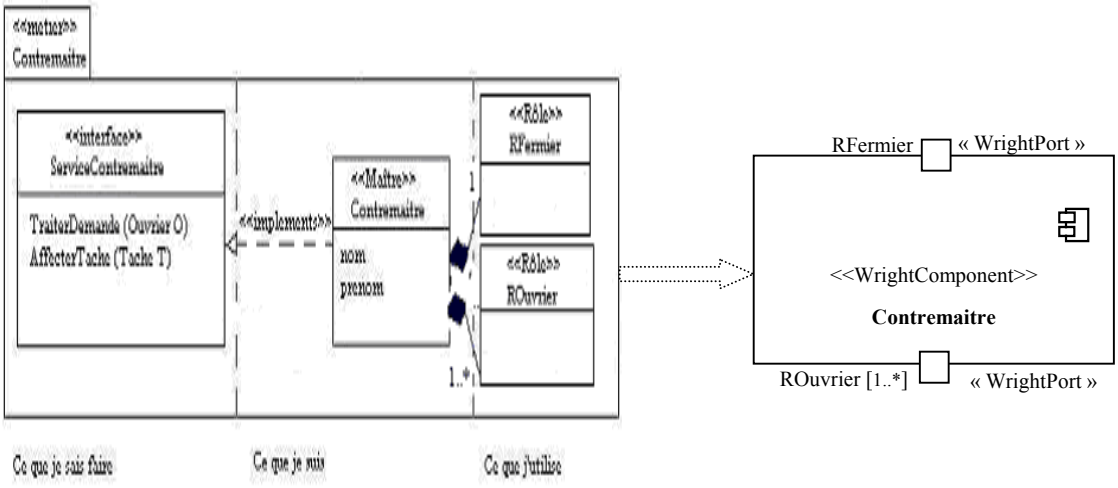


Figure 3.28. Traduction du composant métier Symphony Contremaître vers W-UML

12.3.1.2. Traduction de la relation Client-Fournisseur entre deux Composants métier Symphony

Une relation Client-Fournisseur bilatérale entre deux composants métier Symphony est matérialisée par deux relations de dépendance stéréotypées <<utilise>>. Chaque composant est considéré à la fois comme client et fournisseur au sein de cette relation.

- **Traduction 12.2 (Relation Client-Fournisseur)**

Toute relation Client-Fournisseur bilatérale entre deux composants métier Symphony est traduite par un <<WrightConnector>> ayant deux <<WrightPort>>.

La figure 3.29 donne la traduction de la relation Client-Fournisseur entre les deux composants métier Fermier et Contremaître vers le profil W-UML.

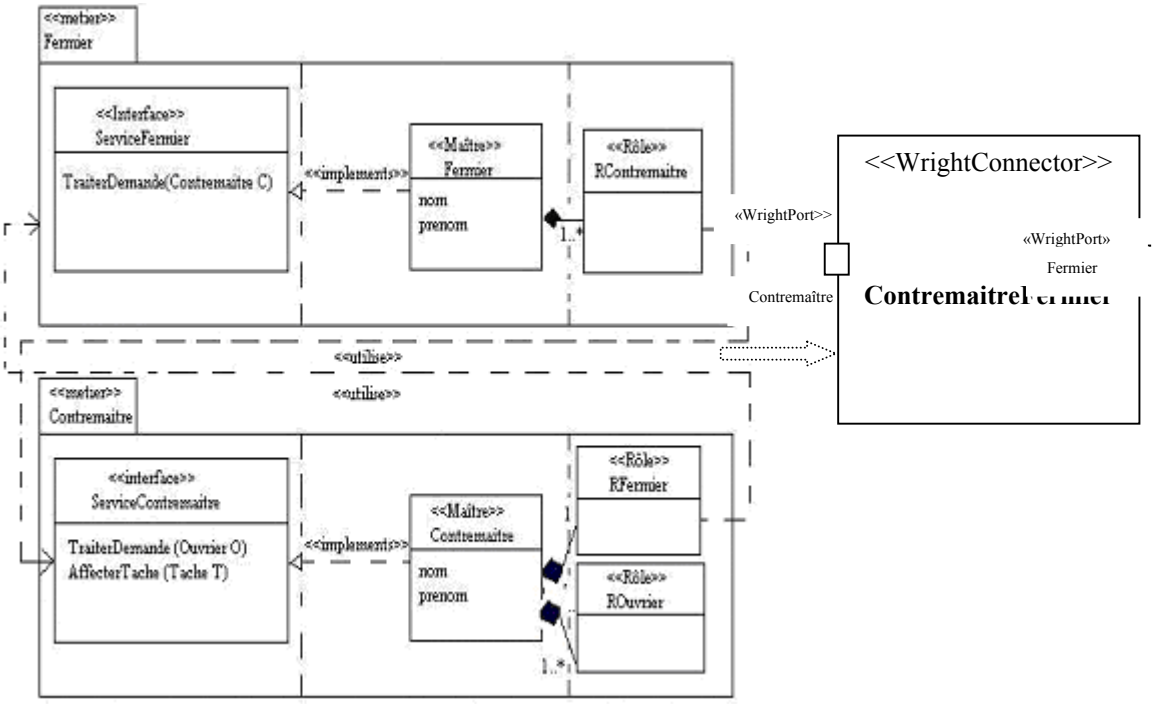


Figure 3.29. *Traduction de la relation Client-Fournisseur entre deux composants métier Symphony vers W-UML*

12.3.1.3. Traduction d'un diagramme d'instances de composants métier Symphony

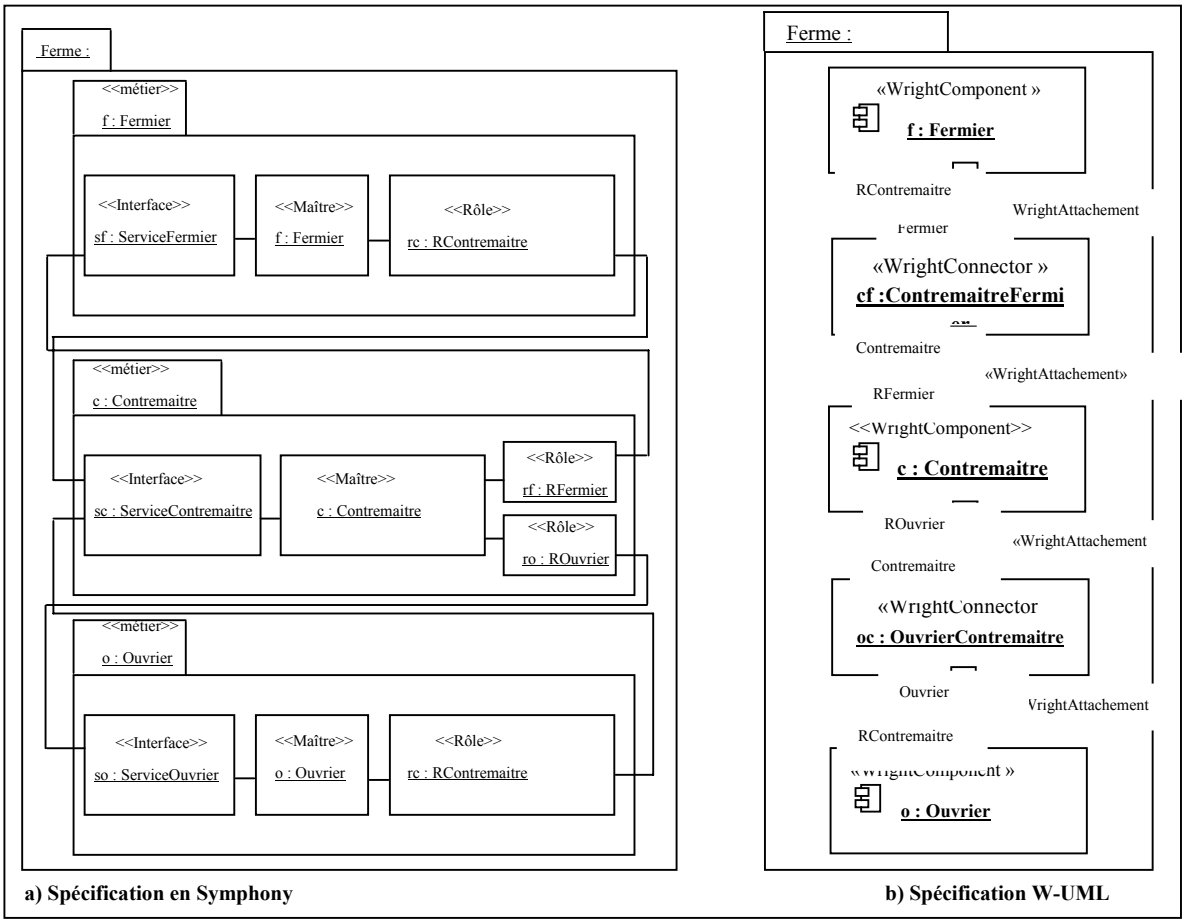


Figure 3.30. *Traduction d'un diagramme d'instances Symphony vers W-UML*

Un diagramme d'instances de composants métier Symphony est matérialisé par un diagramme d'objets UML (cf. exemple 3.30 de l'application <<Ferme>> pour n = 1 et m = 1).

- **Traduction 12.3 (Diagramme d'instances de composants métier)**

Un diagramme d'instances des composants métier Symphony est traduit par une configuration Wright. Sachant que celle-ci est matérialisée par un diagramme d'objets du profil W-UML.

Nous appliquons les règles suivantes :

R1- Toute instance à base de la classe <<Maître>> est traduite par une instance de type <<WrightComponent>>.

R2- Deux liens matérialisant une relation Client-Fournisseur bilatérale entre deux instances de composants métier sont traduits par une instance de type <<WrightConnector>>. De plus deux liens de type <<WrightAttachement>> sont créés entre une instance de type <<WrightComponent>> et une instance de type <<WrightConnector>>.

La figure 3.30 donne la traduction du diagramme d'instances Symphony vers le profil W-UML pour l'application « ferme ».

12.3.2 . Traduction des aspects comportementaux

La traduction des aspects comportementaux attachés aux composants métier de Symphony (cf. 12.2.5) se heurte aux deux problèmes majeurs suivants :

- L'interface d'un composant W-UML combine via le concept port les services offerts et requis (cf. 1.2.1). Par contre, un composant métier Symphony les sépare nettement : classe <<Interface>> et classe <<Rôle>>.
- Contrairement au profil W-UML, la méthode Symphony ne propose pas un modèle de connecteur explicite.

La résolution de ces deux problèmes exige des connaissances spécifiques liées à l'application. Ceci peut être assuré par l'architecte en attachant aux constructions structurales, issues des schémas de transformation proposés dans 12.3.1, les aspects comportementaux correspondants en tenant compte du profil W-UML et précisément du concept <<WrightProtocolStateMachine>>.

12.4 . Conclusion

Pour mettre à la disposition des utilisateurs de la méthode Symphony, les concepts et les mécanismes sous-jacents issus de notre profil W-UML, nous avons préconisé une approche de traduction des concepts structuraux et comportementaux de la méthode Symphony vers le profil W-UML. Ainsi nous avons proposé des règles permettant de traduire une architecture décrite en Symphony en une architecture décrite en W-UML. Ceci ouvre des perspectives liées à la vérification formelle des architectures Symphony en projetant ensuite l'architecture W-UML résultante de la traduction de Symphony vers Wright en appliquant les règles définies dans le chapitre 8. En effet, l'architecture Wright issue de l'architecture W-UML peut être ensuite traduite en une spécification CSP contenant des processus CSP décrivant les aspects comportementaux de l'architecture et des assertions exprimant les propriétés standards définies par Wright. Une telle spécification CSP peut être analysée par l'outil de vérification formelle FDR (FDR2, 2003). Par la suite on peut vérifier des propriétés spécifiques à cette architecture en suivant la démarche DVFAL (cf. chapitre 3).

Conclusion générale

1. Problématique

La problématique abordée dans cette thèse peut être résumée par l'interrogation : Comment peut-on vérifier la correction d'une architecture logicielle ?

Nous avons apporté notre réponse à cette question en tenant compte du fait que les ingénieurs d'applications sont confrontés à :

- une pluralité des formalismes de description d'architectures logicielles : langage de description d'architectures (ADL), langage de modélisation standard (UML), méthodes des composants métier (par exemple Symphony),
- une pluralité des classes des propriétés à vérifier : propriétés d'atteignabilité, de sûreté, de vivacité, de blocage et d'équité,
- une pluralité des techniques et outils de vérification applicables aussi bien sur les modèles que les programmes : simulation, preuve de théorème, model checking, test et analyse statique par interprétation abstraite.

Ces formalismes, techniques et outils sont utilisés à différents niveaux d'abstraction selon des cycles de développement rarement bien définis si ce n'est par quelques métaphores (V, Y, etc.). Il est difficile de limiter les formalismes pour différentes raisons (adéquation, culture d'entreprise, etc.) et les outils sont souvent inaccessibles pour des raisons de prix ou de compatibilité d'environnement. Dans cette thèse, en partant d'éléments bien connus dans le monde académique comme Wright, CSP et Eiffel, ou dans le monde professionnel comme UML et Ada, et d'éléments plus originaux comme Symphony, nous avons souhaité montrer qu'il est possible de mettre en pratique l'approche MDA-MDE pour améliorer sensiblement la qualité du processus d'ingénierie d'architectures de systèmes et plus précisément d'architectures logicielles.

Avant de résumer le bilan de nos contributions et de proposer des perspectives de notre recherche, nous souhaitons revenir sur quelques travaux étudiés. Ces travaux permettent de positionner nos contributions et ils ont fortement nourri notre réflexion sur ce domaine de recherche.

2. Travaux de référence

Nous distinguerons des travaux sur des formalismes qu'ils soient formels ou semi-formels et des travaux plus axés sur des outils de vérification.

2.1. Formalismes de description d'architectures

La communauté scientifique a développé plusieurs formalismes permettant de décrire des architectures logicielles. Nous classerons ces formalismes en trois catégories.

- **Les langages de description d'architectures (ADL)**

Un ADL se concentre plus sur la structure de haut niveau de l'ensemble de l'application, que sur les détails d'implémentation (Vestal, 1993). Parmi les ADL les plus connus, nous citons : Wright (Allen, 1997a), ACME (Garlan, 2000), Rapide (Luckham, 1995), Unicon (Shaw, 1995), C2 (Taylor, 1996), Darwin (Magee, 1995) et AESOP (Garlan, 1994). Les

ADL partagent au moins trois concepts fondamentaux : composant, connecteur et configuration. Rares sont les ADL qui permettent de raisonner formellement sur les architectures logicielles tels que Wright (cf. chapitre 1 et 2) et Rapide. Mais contrairement à Wright, Rapide ne supporte pas un modèle de connecteurs explicite.

L'ADL Wright permet l'étude de deux critères fondamentaux sur une architecture logicielle qui sont la **cohérence** et la **complétude**. Pour y parvenir, il définit des propriétés architecturales standards. Certaines propriétés sont formalisées grâce au raffinement CSP de Hoare (Hoare, 1985) (Hoare, 1995) (Hoare, 2004) et par conséquent elles peuvent être vérifiées par des outils supportant CSP comme FDR2 (FDR2, 2003). Mais Wright en tant que langage «généraliste» ne peut pas proposer des propriétés architecturales spécifiques c'est-à-dire liées à un domaine d'application ou à des applications particulières. L'architecte qui désire vérifier des propriétés architecturales spécifiques doit travailler sur des spécifications CSP produites par l'outil Wr2fdr. De plus, il faut que les propriétés à vérifier soient naturellement exprimables sous forme de raffinement CSP. Mais les propriétés spécifiques potentiellement vérifiables dans le cadre de CSP ne couvrent pas toutes les classes de propriétés. En effet, CSP ne peut pas traiter naturellement les propriétés d'équité, les propriétés orientées état et les propriétés qui incluent des événements et/ou en excluent d'autres.

- **UML**

UML est un langage moins formel et moins rigoureux que les ADL. UML2.0 a introduit des nouvelles constructions telles que composant, interface offerte, interface requise, port, connecteur, structure composite, machine de description de protocoles. Ces constructions rendent UML2.0 plus adapté au développement à base de composants et à la spécification des descriptions architecturales (Ivers, 2004). Les gains apportés par l'utilisation d'UML -surtout UML2.0- comme langage de description d'architectures sont :

- permettre à plusieurs intervenants qui ont peu de connaissances dans le domaine des spécifications formelles de comprendre et manipuler une description architecturale ;
- tirer profit des outils supportant le standard UML.

Comme inconvénient majeur, une description architecturale en UML ne peut pas être analysée d'une façon rigoureuse et formelle dans le cadre d'UML.

- **Formalismes orientés composant**

Certains formalismes orientés composant comme la méthode Symphony (Hassine, 2005) et Ugatze (Seyler, 2004) offrent des possibilités permettant de modéliser une architecture. Mais ces formalismes comme UML ne favorisent pas une analyse rigoureuse d'une description architecturale.

2.2. Vérification formelle

L'activité de vérification formelle est l'étape qui permet de s'assurer que le système est développé correctement c'est-à-dire possède certaines propriétés. Dans le domaine des systèmes séquentiels, la communauté de vérification formelle a développé des notations formelles comme VDM, Z, B et Perfect Developer. Egalement, pour des systèmes concurrents, la même communauté a développé des notations formelles CSP, CCS et réseaux de Petri. En outre, la communauté de vérification formelle a développé des approches supportées par des outils permettant de raisonner formellement sur des programmes écrits dans divers langages de programmation. Dans ce cadre, nous pouvons citer les outils d'analyse statique tels que FLAVERS (Dwyer, 1994) (Dwyer, 1998) (Dwyer, 1999) (Cobleigh, 2002) et INCA (Corbette, 1995) qui analysent des programmes

concurrents écrits en Ada ou en Java. De plus le model checking est applicable sur des programmes écrits dans divers langages de programmation en passant par les outils comme SPIN (Holzmann, 1991) et SMV (McMillan, 1993).

Les formalismes d'architectures logicielles (ADL) sont mieux adaptés qu'UML pour tirer profit de la richesse et de la puissance des outils de vérification formelle existants.

Dans le tableau ci-dessous, nous résumons les points forts et les points faibles de quelques travaux de référence qui ont guidé nos choix. Quelques éléments de ce tableau sont issus des travaux de Mhamed Saidane (Saidane, 2005).

Tableau. *Comparaison de travaux de référence*

	Points forts	**Points faibles**
Unicon	- usage souple - modèle de composants et connecteurs - générateur de code	- absence de spécification dynamique - absence de propriétés non fonctionnelles
Rapide	- spécifications statiques et dynamiques - simulation	- absence de représentation explicite des connecteurs
Wright	- spécification formelle des éléments d'une architecture (composant, connecteur, configuration) - utilisation de CSP	- utilisation par des « spécialistes » - absence de génération de code - absence de séparation des propriétés fonctionnelles et non fonctionnelles
ACME	- formalisme pivot pour intégrer des ADL	- absence de spécification dynamique - absence de séparation des propriétés fonctionnelles et non fonctionnelles
UML	- formalisme normalisé et répandu - représentation d'architectures avec des composants, des connecteurs et des interfaces - automates de description de protocoles	- plus orienté objet que composant - semi-formel - sémantique des éléments d'architecture (composants, connecteurs, interfaces) dispersée
Symphony	- modèle de composant métier orienté architecture : connecteur explicite par une conjugaison rôle-interface - démarche intégrant préoccupations fonctionnelles et non fonctionnelles	- semi-formel - nécessite la connaissance d'UML - absence d'atelier de référence
B	- formel - démarche de raffinement de spécifications - preuves	- utilisation par des « spécialistes » - difficile à combiner avec d'autres formalismes et d'autres démarches
Ada	- structuration de données et de traitements - langage modulaire et concurrent - outils d'analyse statique (FLAVERS, INCA)	- utilisation par des « spécialistes »
CSP	- formel	- utilisation par des « spécialistes »

	- outil de model checking FDR	

3. Bilan de nos contributions

Notre apport principal est de proposer et d'expérimenter un cadre méthodologique permettant de vérifier la correction d'architectures logicielles : c'est la démarche DVFAL basée sur des formalismes et des outils.

La démarche DVFAL permet d'intégrer des formalismes formels de description d'architectures logicielles comme les ADL et moins formels comme UML et Symphony. Notre démarche s'appuie concrètement sur une variété d'outils issus de plusieurs domaines (ADL, programmation générale, programmation spécifique de systèmes concurrents, etc.).

Pour pouvoir réutiliser les divers outils de vérification formelle applicables aussi bien sur les modèles que les programmes, la démarche DVFAL propose deux types de représentation d'architectures logicielles : représentation comme modèle (CSP de Hoare) et représentation comme programme concurrent (Ada).

L'ADL Wright est le langage pivot de la démarche DVFAL. Ce choix est justifié par les aptitudes de ce langage vis-à-vis de la vérification formelle des propriétés standards d'architectures logicielles.

La démarche DVFAL réutilise le traducteur Wr2fdr qui accompagne Wright et permet de traduire les expressions Wright vers CSP pour appliquer ensuite l'outil de model checking FDR. De plus, notre démarche propose un traducteur de Wright vers Ada pour inclure le traitement de propriétés spécifiques. En outre, pour faciliter la traduction des formalismes à base d'UML vers Wright, la démarche DVFAL propose un langage intermédiaire sous forme d'un profil UML : W-UML permettant d'adapter UML à l'ADL Wright. Plus généralement W-UML peut servir de modèle pivot pour des spécifications semi-formelles voisines d'UML.

La démarche DVFAL propose deux outils complémentaires supportant le profil W-UML et l'ADL Wright : traduction d'expressions CSP vers des machines à états d'UML et implantation du langage des contraintes Wright.

4. Perspectives

La version actuelle de l'outil Wr2fdr comporte des limites : absence de distinction entre les événements initialisés et observés, évènements ne portant pas d'informations ni d'entrée ni de sortie, absence d'interface. De plus, en se basant sur notre expérience avec cet outil, il contient des erreurs liées à la génération du code CSP et au traitement syntaxique du concept composant. En outre, Wr2fdr génère du CSP dédié à l'outil FDR. Nous avons lancé une action pour apporter des améliorations significatives à cet outil et l'ouvrir sur les autres outils de vérification des modèles CSP tels que satchecker (Martin, 2000) et Deadlock Checker (Martin, 1997a).

Les perspectives plus immédiates liées à l'outillage de la démarche DVFAL sont :

- l'automatisation de l'approche de traduction de Wright vers Ada préconisée dans le chapitre 4,
- l'automatisation des schémas de transformation du Profil W-UML vers Wright introduits dans le chapitre 8,

– l'automatisation des schémas de transformation de Symphony vers le profil W-UML introduits dans le chapitre 12.

Des perspectives plus globales sont aussi envisagées :

- transformation (W-UML vers Wright, Symphony vers W-UML), l'enrichissement du métamodèle Wright -introduit dans le chapitre 5- par des règles de bonne utilisation (Well-Formedness Rules : WFR) en OCL s'impose.
- L'outil EVALCWright, introduit dans le chapitre 10, permet l'implantation du langage de contraintes de Wright. Un tel outil permet la vérification de la propriété 9, à savoir contraintes pour les styles, définie par Wright. Mais notre outil EVALCWright suppose que les contraintes assimilées à des prédicats soient cohérentes. Une ouverture de Wright sur la méthode formelle avec preuve B pourrait résoudre ce problème. De plus, une telle ouverture serait bénéfique pour vérifier les propriétés orientées état d'architectures logicielles.

Nous devons préciser que notre recherche centrée sur une démarche générale de coordination d'utilisation de formalismes, de techniques et d'outils de représentation et de vérification, correspond à une sorte de « coupe horizontale » des spécifications d'un système : son abstraction architecturale. Une telle coupe horizontale doit être combinée avec des « coupes verticales » représentant des processus métier de l'application. Nous avons exploré ce type de combinaisons, coupe horizontale et coupe verticale, avec la notion de composant métier issue de la méthode Symphony.

En combinant démarche et boîte à outils, il faut noter que tout travail dans ce domaine de la vérification et du raffinement d'architectures logicielles doit tenir compte d'impératifs liés à la taille et à la complexité des systèmes d'aujourd'hui (systèmes d'entreprises ouvertes ou étendues, systèmes mondiaux, etc.). Ainsi, l'évolution des systèmes et la réutilisation de fragments de systèmes doit être possible tout en maintenant la traçabilité entre les différentes formes d'architectures qu'elles soient de niveau système d'information ou logiciel.

Bibliographie

A

(Abrial, 1996) Abrial J.R., *The B Book : Assingning Programs to meanings*, Cambridge University Press, 1996.

(Abrial., 2003) Abrial J.R., *B : passé présent futur*, Technique et Science Informatiques, Vol.22, n°1, 2003.

(ACCORD, 2002) Projet ACCORD, *Etat de l'art sur les Langages de Description d'Architecture (ADL)*, Rapport technique INRIA, France, Juin 2002. http:// www.infres.enst.fr.projet/accord/

(Agedis. 2003) Agedis Consortium, http://www.agedis.de/

(Allen, 1996) Allen R., Garlan D., *The Wright architectural specification langage*, Rapport technique, CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pillsburgh, PA, Septembre 1996.

(Allen, 1997a) Allen R., *A Formal Approach to Software Architecture*, Phd Thesis, Carnegie Mellon University, School of Computer Science, May 1997.

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

(Alti, 2006) Alti A., Khammaci T., Smeda A., *Building a UML Profil for COSA Software Architecture,* Proceedings of the Second IEEE International Conference on Information & Communication Technologies : From Theory to Applications (ICTTA'06), Volume: 2, On page(s): 2849- 2854, April 2006.

(André, 1994) André J., *Moving From Merise to Schlaer and Mellor*, SIG-Publication vol n° 3, 1994.

(André, 2006) André P., Ardourel G., Attiogbé C., *Spécification d'architectures en Kmelia : hiérarchie de connexion et composition*, Actes du 1ère Conférence francophonne sur les Architectures Logicielles, CAL, Septembre 2006.

B

(Bézivin, 2002a) Bézivin J., Blanc X., *Vers un nouveau paradigme du génie logiciel*, Développeur Référence, V2.16/15 Juillet 2002.

(Bézivin, 2002b) Bézivin J., Blanc X., *Promesses et Interrogations de l'Approche MDA*, Développeur Référence, Septembre 2002.

(Bjørner, 1978) Bjørner D., Jones C., *The Vienna Development Method: The Meta-Language*, volume 61 de Lecture Notes in Computer Science. Springer-Verlag, 1978.

(Blanc, 2005) Blanc X., *MDA en action ingénierie logicielle guidée par les modèles*, Eyrolles 2005.

(Blanchet, 2006) Blanchet B., Cousot P. et al., *Abstract Interprétation in a Nutshell*, January 2006, http://www.astree.ens.fr/IntroAbsInt.html.

(Booch, 1991) Booch G., *Ingénierie du logiciel avec Ada*, interEditions, 3e tirage, Juillet 1991.

(Bryant, 1986) Bryant R.E., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on computers, Vol., C-35, n° 8, Page : 677-691, August 1986.

(Burns, 1998) Burns A., Wellings A., *Concurrency in Ada* , Second Edition, Cambridge University Press, Cambridge, 1995, 1998.

C

(Cansell, 2003) Cansell D., *Assistance au développement incrémental et à sa preuve*, Habilitation à diriger des recherches, Université Henri Poincaré, 3 avril 2003.

(Chamillard, 1996) Chamillard A.T., L. Clarke L.A, Avrunin G.S., *Experimental Design for Comparing Static Concurrency Analysis*, University of Massachusetts, Amherst, MA, 1996.

(Cobleigh, 2002) Cobleigh J.M., Clarke L.A., Osterweil L.J., *FLAVERS: A finite state verification technique for software systems*, IBM Systems Journal, Volume 41, Number 1, 2002 Software Testing and Verification.

(Cok, 2004) Cok D.R., Kiniry J.R., : *ESC/Java2: Uniting ESC/Java and JML,* CASSIS 2004: 108-128.

(Corbette, 1995) Corbette J.C., Avrunin G.S., *Using integer programming to verify general safety and liveners properties*, Formal Methods in system Design, 6(1) : 97-123, 1995.

(Cousot, 2000) Cousot P., *Interprétation abstraite*. In Numéro spécial de Technique et science informatiques, Informatiques, enjeux, tendances et évolutions, sous la direction de René Jacquart, Vol. 19, Nb 1-2-3, January 2000, pp. 155-164. Éditions Hermès sciences, Paris.

(Cousot, 2005) Cousot P., *The Verification Grand Challenge and Abstract Interpretation*, In Verified Software: Theories, Tools, Experiments (VSTTE), ETH Zürich, Switzerland, October 10th-13th, 2005.

(Cousot, 2007) Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival D., *Varieties of Static Analyzers: A Comparison with ASTRÉE*, invited paper. First IEEE & IFIP International Symposium on ``Theoretical Aspects of Software Engineering", TASE'07, Shanghai, China, 6-8 June 2007, pp. 3-17.

(Crocker, 2003) Crocker D., *Perfect Developer: A tool for object oriented formal specification and refinement*. In FME 2003, Tools Exhibition Notes, http://www.eschertech.com/papers/fme 2003 tools paper.pdf, 2003.

D

(DE Bondeli, 1998) DE Bondeli P., *Ada 95, langage temps réel normalisé par l'ISO,* Technique et science informatiques, volume 17-n°1/1998, pages 11-38.

(Dwyer, 1994) Dwyer M.B., and L. A. Clarke, *Data Flow Analysis for Verifying Properties of Concurrent Programs*, Proceedings, Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, New Orleans, LA (December 6-9, 1994), pp. 62-75.

(Dwyer, 1998) Dwyer M.B., Pasareanu S.C., Corbett J.C., *Translating Ada programs for Model checking*: A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.

(Dwyer, 1999) Dwyer M.B., and L. A. Clarke, *Flow Analysis for Verifying Specifications of Concurrent and Distributed Software*, Technical Report 99-52, University of Massachusetts, Department of Computer Science, Amherst, MA (August 1999).

E

(El Boussaidi, 2006) El Boussaidi G., Mili H., *Les langages de description d'architectures*, LATECE Technical Report, October 2006.

(Engels, 2001) Engels C., Heckel R., Küster J-M., *Rule-based Specification of Behavior Consistency based on the UML Meta-Model*, proceedings of the 4th International Conference on the Unified Modeling Language (UML 2001), pages 272-287, LNCS 2185, October 2001.

(Engels, 2002) Engels G., Küster M., *Consistent Interaction of Software Components*, University of Paderbon, Integgrated Design and Process Technology, IDPT-2002.

(Enrique, 2003) Enrique J., Martinez P., *Heavyweight extensions to the UML metamodel to describe the C3 architectural style*, ACM SIGSOFT Software Engineering notes volume 28, Issue 3, May 2003

F

(FDR2, 2003) Ltd, Formal System (Europe) Failure-Divergence Refinement, FDR2 user Manual, May 2003.

G

(Gamma, 1999) Gamma E., Helm R., Johnson R., Vlissides J., *Design patterns, Catalogue des modèles de conception réutilisables,* Vuibert 1999.

(Garlan, 1993) Garlan D., Shaw M., *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, 1993.

(Garlan, 1994) Garlan D., Allen R., Ockerbloom J., *Exploiting style in architectural design environments*. In Proc. 2nd. Symposium on the Foundations of Software Engineering, pages 75-188, New York, USA, Dec. 1994. ACM Press.

(Garlan, 1997a) Garlan D., Monroe R. Wile D., *Acme: An Architecture Description Interchange Language,* In proceedings of CASCON 97, Toronto, Canada, November 1997.

(Garlan, 1997b) Garlan D., Monroe R. Wile D., ACME : *Architectural Description of Component-Based Systems*, Computer Science Department, Carnegie Mellon University, Pittsburg et USC/information science institute, 1997.

(Garlan, 2000) Garlan D., Cheng S.W., Kompanek A., *Acme : Architectural Description of Component-Based Systems, Leavens Gary and Sitaraman Murali, Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 47-68.

(Garlan, 2001) Garlan D., Cheng S.W., Kompanek A., *Reconciling the Needs of Architectural Description with Object-Modeling Notations*, Science of Computer Programming Journal, Special UML Edition Elsevier Science, 2001.

(Garlan, 2002) Garlan D., Cheng S.W., Kompanek A., Reconciling the Needs of Architectural Description with Object-Modeling Notations, Science of Computer Programming Journal, 44 (1), July 2002, p. 23-49.

(Goulão, 2003) Goulao M., Abreu F.B., *Bridging the gap between Acme and UML 2.0 for CBD*, Workshop at ESEC/FSE 2003-Septembre 1-2, 2003.

(Graiet, 2005a) Graiet M., Bhiri M.-T., Giraudin J.-P., Ben Hamadou A., *Architecture logicielle : d'UML2.0 vers Wright*, Actes du 4ème atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, OCM-SI, Mai, 2005.

(Graiet, 2005b) Graiet M., Bhiri M.-T., Giraudin J.-P., Ben Hamadou A., *Vérification d'une architecture UML 2.0 avec l'ADL Wright*, Actes 3ème Manifestation des Jeunes Chercheurs francophones dans les domaines des STIC, Novembre, 2005.

(Graiet, 2006a) Graiet M., Bhiri M.-T., Giraudin J.-P., Belkhatir N., *Architecture des systèmes avec la norme UML2.0 et l'ADL Wright*, Actes du XXIVème Congrés Informatique des organisations et systèmes d'information et de décision, INFORSID, Juin, 2006.

(Graiet, 2006b) Graiet M., Bhiri M.-T., Damak F., Giraudin J.-P., *Adaptation d'UML2.0 à l'ADL Wright*, Actes du 1ère Conférence francophonne sur les Architectures Logicielles, CAL, Septembre 2006.

(Guyomard, 1999) Guyomard M., *Spécification et programmation : Le cas de la construction de boucles*, support de cours Université de Rennes 1, Décembre 1999.

H

(Hassine, 2005) Hassine I., *Spécification et formalisation des démarches de développement à base de composants métier : La démarche Symphony*, rapport de thèse, Septembre 2005.

(Hoare, 1985) Hoare C.A.R., *Communicating Sequential Processes,* Prentice Hall, Englewood Califfs NJ, 1985.

(Hoare, 1995) Hoare C.A.R., *Communicating Sequential Processes,* Prentice Hall, 1995.

(Hoare, 2004) Hoare C.A.R., *Communicating Sequential Processes,* 2004. http://www.usingcsp.com/cspbook.pdf

(Holzmann, 1991) Holzmann G.J., *Design and Validation of Computer Protocol*. Prentice Hall, Englewood Cliffs, NJ, 1991.

(Holzmann, 2003) Holzmann G.J, *The SPIN Model checker*, Addison-Wesley, 2003.

I

(Ivers, 2004) Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J.R.O., *Documenting Component and Connector Views with UML 2.0*. Technical report CMU/SEI-2004-TR-008, April 2004.

J

(Jackson, 1996a) Jackson D., Nitpick: *A Checkable Specification Language*, In Proc. Workshop on Formal Methods in Software Practice, San Diego, CA, January 1996.

(Jackson, 1996b) Jackson D., Damon G., *Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector*, IEEE Transactions on software Engineering. 22(7): 484-495 (1996).

(Jacobson, 2000) Jacobson I., Booch G., Rumbaugh J., *Le Processus Unifié de développement logiciel*, Eyrolles, Juin 2000.

(Jeffrey, 1999) Jeffrey J.P.T., Kuang X., *An empirical evaluation of deadlock detection in software architecture specifications*, Annals of Software Engineering, v.7 n.1-4, p.95-126, 1999

(Jéron, 2002) Jéron T., *TGV : théorie, principes et algorithmes,* Technique et science informatiques- numéro spécial Test de Logiciels, (21), 2002.

K

(Kandé, 2000) Kandé M. M., Strohmeier A., *Towards a UML Profile for Software architecture description*, Proceedings of UML'2000, Third International Conference, York, UK, October 2-6, 2000.

(Küster, 2004) Küster J.M., *Consistency Management of Object-Oriented Behavioral Models*, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn, March 2004.

L

(Larvet, 1994) Larvet P,. *Analyse des systèmes : de l'approche fonctionnelle à l'approche objet*, InterEditions, 1994.

(Lawrence, 2005) Lawrence J., *Pratical Application of CSP and FDR to Software Design*, CSP25, LNSC 3525, pp. 151-174, Springer-Verlag Berlin Heidelberg, 2005.

(Le Verrand, 1982) Le Verrand D., *le langage Ada*, manuel d'évaluation, Dunod 1982.

(Luckham, 1995) Luckham D.C., Kenney J.J., Augustin L.M, Vera J., Bryan D., Mann W., : *Specification and Analysis of System Architecture Using* Rapide. IEEE Trans. Software Eng. 21(4): 336-355 (1995).

M

(Magee, 1995) Magee J., Eisenbach, S., Kramer,J., Modelling Darwin in π-Calculus in "Theory and Practice in Distributed Systems", K.P. Birman, F. Mattern, A. Schiper (Eds), Springer Verlag LNCS 938, July 1995, 133-152.

(Martin, 1997a) Martin J. M. R., Jassim A., *A tool for proving deadlock freedom*, in Proc. WoTUG20: Parallel Programming and Java, ed. A. Bakkers, IOS Press, Amsterdam, April 1997, pp.1-16.

(Martin, 1997b) Martin J.M.R., Jassim S.A., *How to Design DeadlockFree Networks Using CSP and Verification Tools -- A Tutorial Introduction*, Proceedings of the WoTUG-20 conference, 13-16 April 1997, Enschede, NL, pp 326-338.

(Martin, 2000) Martin J. M. R., *A Tool for Checking the CSP sat Property*, The computer Journal, vol. 43, No, 1, 2000.

(McMillan, 1993) McMillan K.L., *Symbolic Model checking*, Kluwer Academic Publishers,Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993

(Medvidovic, 2000) Medvidovic N., Taylor R.N., *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Vol 26, no1, pp 70-93, Janvier 2000.

(Medvidovic, 2002) Medvidovic N., Rosenblum D.S., Redmiles D.F., Robbins J.E., *Modeling Software Architectures in the Unified Modeling Language*, ACM Transactions on Software Engineering and Methodology, Vol. 11, N0 .1, January 2002.

(Meyer, 1992) Meyer B., *Applying Design by Contract,* IEEE Computer, vol, 25, pp 40-51, 1992.

(Meyer, 1994) Meyer B., Eiffel, Reusable Software: *The Base Object-Oriented Libraries*, Prentice-Hall, Englewood Cliffs, USA, 1994.

(Meyer, 2000) Meyer B., *Conception et programmation orientées objet*, Eryolles 2000.

(Meyer, 2003) Meyer B., *Towards Practical Proofs of Class Correctness*, in ZB 2003: Formal Specification and Development in Z and B, Proceedings of 3rd International Conference, Turku, Finland, June 2003.

(Merz, 2006) Merz S., *Model checking: Éléments de base*. Dans: N. Navet (éd.): Systèmes temps réel 1 : techniques de description et de vérification, pp. 89-120, Hermès Lavoisier 2006.

(Milner, 1980) Milner R., *A Calculus of Communicating Systems*, vol. 92 de Lecture Notes in Computer Science. Springer Verlag, 1980.

N

(Naumovich, 1997) Naumovich G., Avrunin G.S., Clarke L.A, Osterweil, L.J *Applying static analysis to software architectures*, ACM SIGSOFT Software Engineering Notes, v.22 n.6, p.77-93, Novembre. 1997

O

(Occam, 2000) Occam Langage, http://wotug.kent.ac.uk/parallel/occam/, 2000.

(OMG, 2000) OMG (Object Management Group), *Meta Object Facility (MOF) Specification, Version 1.3*, Object Management Group, Mars 2000. OMG TC Document formal/00-04-03.

(OMG, 2003a) OMG (Object Management Group), *Unified Modeling Language (UML) Specification 1.5*, March 2003.

(OMG, 2003b) Object Management Group. UML2.0 OCL Specification: Final Adopted Specification. http://www.omg.org/docs/ptc /03-10-14.pdf (October 2003).

(OMG, 2003c) MDA Guide, version 1.0.1. http://www.omg.org, Juin 2003.

(OMG, 2005) OMG (Object Management Group), UML 2.0 *superstructure final adopted specification*, 4 July 2005.

(Oussalah, 2005) Oussalah M., *Ingénierie des composants : concepts, techniques et outils*, (chapitre 4), Editions Vuibert Informatique, Paris, 2005.

P

(Peterson, 1981) Peterson J., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

(ProBE, 2003) L'outil ProBE; Fomal Systems (Europe) Ltd http://www.fsel.com/>, *Process Behaviour Explorer*, ProBE user Manual, Juin 2003.

R

(Richters, 2000) Richters M., Gogolla M., *Validating UML Model and OCL Constraints*, Third International Conference on the Unified Modeling Language : UML 2000, 2-6th October, 2000, York, UK.

(Roh, 2004) Roh S., Kim K., Jeon T., *Architecture Modeling Language based on UML2.0*, Proceedings of the 11 th Asia-Pacific Software Engineering Conference (APSEC'04).

(Rousseau, 1994) Rousseau R., *Fallait-il concevoir Ada 9x ?* Numéro spécial sur le langage Ada, Technique et Science Informatiques, 13(5):723--729, Octobre 1994.

(Roscoe, 1994) Roscoe A.W., *The Theory and Pratice of Conccurency,* Prentice-Hall, 1997.

(Roscoe, 1997) Roscoe A.W., *Model-checking CSP*, A Classical Mind: Essays in Honour of CARHoare Prentice-Hall, 1997.

S

(Saidane, 2005) Saidane M., *Formalisation de Familles d'Architectures Logicielles Coopératives : Démarches, Modèles et Outils*, Thèse de l'Université Joseph Fourier, Grenoble, 13 décembre 2005.

(Spivey, 1989) Spivey M., *An Introduction to Z and Formal Specification*, IEEE Software Engineering Journal, 4(1):40–50, 1989.

(Sanlaville, 1997) Sanlaville R,. *Description d'architectures logicielles: utilisation du formalisme Wright pour l'interconnexion de machines abstraites B*, DEA, LSR, IMAG, Grenoble, 1997.

(Schnoebelen, 1995) Schnoebelen P., *Vérification de logiciels, techniques et outils du model-checking*, Vuibert, Paris, 1995.

(Seyler, 2004) Seyler F., *Ugatze : métamodélisation pour la réutilisation de composants hétérogènes distribués*, Thèse de l'Université de Pau et des Pays de l'Adour, 16 décembre 2004.

(Shaw, 1995) Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M., Zelesnik G., *Abstractions for Software Architecture and Tools to Support Them*, IEEE Trans. Software Eng., vol.21, no.4, p.314-335, April 1995.

(Shaw, 1996) Shaw M., Garlan D., *Software Architecture : Perspective of an Emerging Discipline*, Prentice Hall, Avril 1996.

T

(Taylor, 1996) Taylor R.N., Medvidovic N., Anderson K.M., Whitehead Jr.E.J., Robbins J.E., Nies K.A., Oreizy P., Dubrow D.L., *A component- and message-based architectural style for GUI software*, IEEE Transactions on Software Engineering, Vol.22, N°.6, June 1996.

V

(Vestal, 1993) Vestal S., *A Cursory overview and comparaison of Four Architecture Description Lanaguages*, Honeywell Technical Report, February 1993.

W

(Warmer, 2003) Warmer J., Kleppe A., *The Object Constraint Language*, Addison-Wesley, Août 2003.

(Wirfs-Brock, 1990)	Wirfs-Brock R., Wilkerson B., Weiner L., *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
(Wr2fdr, 2005)	Wr2fdr, http://www.cs.cmu.edu/~able/wright/wr2fdr_bin.tar.gz, Update December 2005.

X

(Xanthakis, 2000)	Xanthakis S., Réginier P., Karapulios C., *Le test des logiciels*, Paris, Hermes, 2000.

Z

(Zarras, 2001)	Zarras A., Issarny V., Kloukinas Ch., Nguyen V. K., *Towards a Base UML Profile For Architecture Description*, INRIA Rocquencourt, www.soi.city.ac.uk/~kloukin/pubs/icse01.pdf.
(Zaffalon, 1999)	Zaffalon. L., Breguet. P., *Programmation concurrente et temps réel en ADA 95*, Presses polytechniques et universitaires romandes, Décembre 1999.
(Ziemann, 2003)	Ziemann P., Gogolla M., *Validating OCL Specifications with the USE Tool*, An Example Based on the BART Case study, Workshop on Formal Methods for Industrial Critical Systems (FMICS'03),vol.80, ENTCS, Elseiver, 2003.

Annexe A : Les classes en Eiffel de l'outil de transformation d'expressions CSP en machine à états de description de protocoles d'UML2.0

1. Classe Sequence

La classe Sequence implémente une construction non terminale définie comme suit :

Sequence {Facteur "->" .. }

L'implémentation de Sequence est donnée ci-dessous

```
class Sequence
inherit
        SEQUENCES
                redefine
                        post_action
                end
creation
        make
feature
        construct_name: STRING is "Sequence"
feature {NONE}
separator: STRING is "->"
feature
        production: LINKED_LIST [CONSTRUCT] is
                local
                        base: Facteur
                once
                        !!Result.make;
                        Result.forth;
                        !!base.make;
                        put (base)
                end; -- production

    post_action is
        do
        from
            child_start
        until
            child_after
        loop
            io.putstring(child.construct_name)
            child_forth
            if not child_after
            then
                io.putstring(separator)
            end
          child_back
            child_forth
            end;
```

```
        io.new_line
        io.new_line
    end -- post_action
end -- class Sequence
```

2. Classe Ev_exp

La classe Ev_exp implémente une construction non terminale définie comme suit :

Ev_exp≙Ev_observe | Ev_initialise. Ceci traduit qu'un événement peut être un événement observé ou un événement initialisé. La classe Ev_exp hérite de la classe CHOICE. L'implémentation Ev_exp est donnée ci-dessous.

```
class Ev_exp
inherit
        CHOICE
                redefine
                        post_action
                end;
creation
        make
feature
        construct_name: STRING is
                once
                        Result := "Ev_exp"
                end; -- construct_name
        production: LINKED_LIST [CONSTRUCT] is
                local
                        ev_ob: Ev_observe;
                        ev_init: Ev_initialise
                once
                        !!Result.make;
                        Result.forth;
                        !!ev_init.make;
                        put (ev_init);
                        !!ev_ob.make;
                        put (ev_ob);
                end; -- production
   post_action is
     do
       from
          child_start
       until
          child_after
       loop
           io.putstring(child.construct_name)
           io.new_line
           child_forth
       end
   end -- post_action
end -- class Ev_exp
```

3. Classe Simple_nom

Les spécimens de type Simple_nom sont formés par des lettres, des chiffres et des caractères « . » et « _ ». Cette classe hérite de la classe IDENTIFIER, qui hérite à son tour de la classe TERMINAL.

L'implémentation de la classe Simple_nom est donnée ci-dessous.

```
class Simple_nom
inherit
        IDENTIFIER
                redefine
                action, construct_name
                end
creation
        make
feature {NONE}
        construct_name: STRING is
                once
                        Result := "Simple_nom"
                end; -- construct_name
        action is
        do
                 io.putstring(token.string_value)
        end -- action
end -- class Simple  nom
```

4. Classe Racine

La classe PROCESS est la classe racine de notre bibliothèque au sens d'Eiffel. Cette classe hérite de la classe POLY_LEX qui définit tous les symboles qui peuvent apparaître dans une expression CSP. POLY_LEX hérite de la classe différée L_INTERFACE et de la classe CONSTANTS.

```
class PROCESS
inherit
        POLY_LEX
creation
        make
feature
        exp_csp: Declaration;
    make is
        local
          nb_try: INTEGER;

t_b: BOOLEAN;
                text_name: STRING
        do
                !!exp_csp.make;
                build (exp_csp.document);
                io.putstring ("Vous voulez tester la récursivité à gauche?");
                io.readline;
                io.next_line;
if io.laststring.is_equal ("o") then
                        exp_csp.print_mode.put (true);
                        exp_csp.expand_all;
                        t_b := not exp_csp.left_recursion;
                        exp_csp.check_recursion;
                        if not exp_csp.left_recursion.item then
                        io.putstring ("Aucune récursivité à gauche n'a été
détectée.%N");
end
                else
                        from
                        until
                          nb_try < 5 and
                          text_name /= Void and then text_name.count > 0
```

```
loop
        io.putstring ("Nom du fichier avec description de l'expression CSP:");
                        io.readline;
text_name := io.laststring;
                        nb_try := nb_try + 1
                end;
                        if nb_try = 5 then
                                io.putstring ("*** Demo stopped%N");
                        else
                                exp_csp.document.set_input_file (text_name);
                                exp_csp.document.get_token;
                                exp_csp.process


                        end
                end
        end -- make
```

5. Exemple d'application

```
Vous voulez tester la récursivité à gauche? o
Declaration :    Def_nom  = Exp_csp
Exp_csp :    Choix | Parallel |  Ex_where
Choix :    Deterministe | Non_deterministe
Deterministe :  Sequence .. "[]"
Sequence :   Facteur .. "->"
Facteur :    Ev_Exp | Nom_exp  | Exp_parenthesee | Skip
Ev_exp :    Ev_initialise | Ev_observe
Ev_initialise :    Ev_complexe  | Ev_simple
Ev_complexe  :    _ Ev_datalist
Ev_datalist  :    Exclamation  | Interrogation
Exclamation  :    Ev_nom  ! N_E_Data
Interrogation :    Ev_nom  ? N_E_Data
Ev_simple :    _ Ev_nom
Ev_observe  :    Ev_datalist  | Ev_nom
Exp_parenthesee :  ( Exp_csp )
Non_deterministe : Sequence .. "|~|"
Parallel :  Process_nom  .. "||"
Ex_where :    Exp_w  Where Declist
Exp_w  :    ( Exp_csp )
Declist  :    Declaration
Aucune récursivité à gauche n'a été détectée.
```

Pour l'expression simple : Consultation = requete -> Consultation, nous avons comme résultat :

```
Vous voulez tester la récursivité à gauche? n
Nom du fichier avec description de l'expression CSP: testfile
Expression reconnue
Consultation = requete -> Consultation
Facteur-> Facteur
Sequence
Def_nom=Exp_csp
Fin action semantique.
```

Pour l'expression : Consultation = requete?u -> _reponse!r -> Consultation [] skip, nous obtenons le résultat suivant:

```
Vous voulez tester la récursivité à gauche? n
Nom du fichier avec description de l'expression CSP: testfile
Expression reconnue:
Consultation = requete?u
Ev_nom ? Non_ev_data
reponse!r
Ev_nom ! Non_ev_data
E_datalist
Consultation
Facteur -> Facteur -> Facteur
skip
Facteur
Sequence []Sequence
Def_nom = Exp_csp
Fin action semantique.
```

Si nous introduisons une expression incorrecte, nous avons comme résultat :

```
Vous voulez tester la récursivité à gauche? n
Nom du fichier avec description de l'expression CSP: testfile
Expression non reconnue!!
```

Annexe B : Les classes en Eiffel de l'outil de gestion du langage de contraintes de l'ADL Wright

1. Réalisation en Eiffel de la modélisation des ensembles et prédicats prédéfinis

1.1.. Type composant

```
class T_COMP                        -- Type_Composant de la modelisation
creation
  make
feature
  name: STRING                      -- Matérialise l'attribut name
feature
  make (new_name: STRING) is
    require
      name_existe: new_name /= void and then not new_name.empty
    do
      name:= clone(new_name)        -- Affectation du nom du type
    end
invariant
  not name.empty
end
```

1.2.. Type connecteur

```
  class T_CONN                      -- Type_Connecteur de la modelisation
creation
make
feature
  name: STRING                      -- Matérialise l'attribut name
feature
  make (new_name: STRING) is
 require
      name_existe: new_name /= void and then not new_name.empty
    do
      name:= clone(new_name)        -- affectation du nom du type
    end
invariant
  not name.empty
end
```

1.3.. Instance

C'est la généralisation des classes possédant les attributs name et type telles que : composant, connecteur, port et rôle.

```
  class INSTANCE     -- Generalisation des classes qui ont en commun les attributs
                      -- name et type
creation
  make
feature
  name: STRING                        --L'attribut name
  type: STRING                        --L'attribut type
feature
  make (n: STRING ; t: STRING) is
    require
      name_existe : n /= void and then not n.empty
      type_existe : t /= void and then not t.empty
    do
      name:= clone(n)
      type:= clone(t)
    end       -- make
invariant
      not name.empty and then not type.empty
end
```

1.4.. CSP

Cette classe est conçue pour qu'elle puisse contenir les caractéristiques d'une expression CSP.

```
class CSP                 -- Interface de la modélisation
creation
  make
feature
  name: STRING
feature
  make (new_name: STRING) is
    require
      name_existe: new_name /= void and then not new_name.empty
    do
      name:= clone(new_name)
    end
invariant
  name_existe: not name.empty
end
```

1.5.. Instance composant

```
class I_COMP                                  -- Instance_Composant de la modelisation
inherit
  INSTANCE
    rename
      make as make_instance
    end
  creation
  make
feature
computation: COMPU              --  Matérialise l'association composant, computaion
  l_port: LINKED_LIST [PORT] --  Matérialise la composition de composant en ports
feature
```

```
  make (n: STRING; t: STRING; c: COMPU; L: LINKED_LIST [PORT]) is
    require
name_existe: n /= void and then not n.empty
      type_existe: t /= void and then not t.empty
    do
      make_instance(n,t)                -- Affectation le nom et le type du composant
      !!l_port.make                     --  Création d'une liste de ports vide
      l_port:= L
      computation:= c
    end
invariant
  au_moin_un_port: l_port.count >=1
  computation_existe: computation /= void
end
```

1.6.. Port

```
class PORT
inherit
  INSTANCE
    Rename              -- Renomme les caractéristiques venant des classes ascendantes
      make as make_instance
    end
creation
make
feature
  component: I_COMP
  interface: CSP
feature
  attribuer_instance_component(comp: I_COMP) is
    require
      composant_existe: comp /= void
    do
      component:= comp                          -- Affectation du composant qui utilise le
port
    end
  make (name1: STRING; processus: CSP) is
    require
name_existe: name1/= void and then not name1.empty
      type_existe: processus /= void and then not processus.name.empty
    do
      make_instance(name1,processus.name)  -- Affectation le nom et le type du port
      interface:= processus
    end end
```

1.7.. Computation

```
class COMPU                                          -- Computation de la modélisation
creation
  make
feature
component: I_COMP
  csp: CSP
feature
  attribuer_instance_component (comp: I_COMP) is--  donne une computation a un
composant
```

```
 require
     composant_existe: comp /= void
    do
     component:= comp
    end
  make (processus: CSP) is
    require
     interface_existe: processus /= void
    do
     csp:= processus
    end
```

1.8.. Instance connecteur

```
 class I_CONN                    -- Instance_Connecteur de la modélisation
inherit
  INSTANCE
 rename
      make as make_instance
    end
creation
make
feature
  glue: GLUE                     --  Matérialise l'association connecteur,glue
  l_role: LINKED_LIST [ROLE] -- Matérialise la composition du connector en role
feature
  make (n: STRING; t: STRING; g: GLUE; L: LINKED_LIST [ROLE]) is
    require
      name_existe: n /= void and then not n.empty
      type_existe: t /= void and then not t.empty
    do
      make_instance(n,t)         -- Affectation le nom et le type du connecteur
      !!l_role.make              --  Création d'une liste de role vide
      l_role:= L
      glue:= g
    end
invariant
  au_moin_deux_roles: l_role.count >=2
  glu_existe: glue /= void
end
```

1.9.. Rôle

```
class ROLE
inherit
  INSTANCE
    Rename
      make as make_instance
    end
  creation
make
feature
connector: I_CONN
 interface: CSP
feature
  attribuer_instance_connector(conn: I_CONN) is
require                          -- Précondition
composant_existe: conn /= void
    do
 connector:= conn                 -- Affectation du connecteur qui utilise le role
    end
```

```
    make (name1: STRING; processus: CSP) is
  require
    name_existe: name1 /= void and then not name1.empty
    type_existe: processus /= void and then not processus.name.empty
  do
    make_instance(name1,processus.name)  -- Affectation le nom et le type du port
    interface:= processus
  end
end
```

1.10.. Glu

```
class GLUE
creation
make
feature
  connector: I_CONN
  csp: CSP
feature
  attribuer_instance_connector (conn: I_CONN) is  -- Attribue une glu a un connecteur
    require
      connecteur_existe: conn /= void
    do
      connector:= conn
    end
  make (p: CSP) is
    require
      interface_existe: p /= void
    do
      csp:= p
    end
end
```

1.11.. Attachement

Il représente l'ensemble des liens entre les ports et les rôles. Ces liens décrivent la configuration du système en indiquant quel composant participe à quelle interaction.

```
 class ATTA                                    -- classe Attachements de la modélisation
creation
  make
feature
  port: PORT
  role: ROLE
feature
  make (p: PORT; r: ROLE) is
    require
      le_port_et_le_role_existent: p /= void and then r /= void
do
      port:= p
     role:= r
    end
invariant
port_existe: port /= void
  role_existe: role /= void
end
```

2. Réalisation en Eiffel de la modélisation des contraintes

2.1.. L'abstraction PREDICAT

```
deferred class PREDICAT     -- La classe PREDICAT est différée
feature
  evaluation: BOOLEAN is
deferred
  end
end
```

2.2.. L'abstraction BINAIRE

La classe BINAIRE modélise les opérateurs binaires logiques. Elle hérite de la classe PREDICAT. Cette classe apporte les caractéristiques suivantes :

- Un attribut gauche permettant de mémoriser l'opérande gauche,
- Un attribut droite permettant de référencer l'opérande de droite,
- Une routine de création permettant de créer l'opérateur binaire.

La classe BINAIRE est une classe retardée car elle ne peut pas concrétiser l'opération evaluation héritée de la classe PREDICAT.

2.2.1. Réalisation en Eiffel

```
 deferred class BINAIRE
inherit
  PREDICAT
Feature {NONE}                  -- Attributs privé
  gauche: PREDICAT
  droite: PREDICAT
feature
  make(gau,dro: PREDICAT) is
    require
      gauche_et_droite_existent: gau /= void and then dro /= void
    do
      gauche := gau
      droite := dro
    end             -- make
end
```

2.2.2. Les classes effectives descendantes de BINAIRE

La classe BINAIRE admet plusieurs classes effectives. Dans la suite nous présentons successivement les classes : OU, ET, IMP et EQU permettant de modéliser respectivement les opérateurs binaires logiques : $\vee$, $\wedge$, $\Rightarrow$ et $\Leftrightarrow$.

2.2.2.1. La classe OU

```
class OU
inherit
  BINAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      Result:= gauche.evaluation or else droite.evaluation
  end
end
```

2.2.2.2 . La classe ET

```
class ET
inherit
  BINAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      Result:= gauche.evaluation and then droite.evaluation
    end
end
```

2.2.2.3 . La classe IMPLIQUE

```
class IMP  -- class Implique
inherit
  BINAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
Result:= gauche.evaluation implies droite.evaluation
    end
end
```

2.2.2.4 . La classe EQUIVALENT

```
class EQU  -- class Equivalent
inherit
  BINAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      Result:= gauche.evaluation = droite.evaluation
    end
end
```

2.3.. L'abstraction UNAIRE

La classe abstraite UNAIRE modélise les opérateurs unaires logiques. Elle est retardée car elle ne peut pas concrétiser la routine evaluation héritée de la clase PREDICAT.

```
deferred class UNAIRE
inherit
  PREDICAT
feature {NONE}
  operande: PREDICAT
feature -- creation
  make(op: PREDICAT) is
    require
      operande_exixte: op /= void
    do
      operande:= op
    end -- make
end
```

2.3.2 . La classe effective descendante de UNAIRE

Elle admet une classe effective très connue : c'est la classe NON.

2.3.2.1. La classe NON

```
class NON
inherit
  UNAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      Result:= not(operande.evaluation)
    end
end
```

2.4.. L'abstraction NAIRE

La classe NAIRE modélise les opérateurs logiques exigeant n opérandes (avec n>2). Elle hérite de la classe PREDICAT. Cette classe apporte la caractéristique suivante : un attribut L_predicat qui regroupe dans une liste les opérandes exigées par l'opérateur NAIRE.

2.4.1. Réalisation en Eiffel

```
deferred class NAIRE
inherit
  PREDICAT
feature
 L_predicat: LINKED_LIST[PREDICAT]
feature -- creation
  make(L_pred: LINKED_LIST[PREDICAT]) is
    require
      liste_exixtent: L_pred /= void and then L_pred.count>=1
    do
      L_predicat:= clone(L_pred)
    end -- make
end
```

2.4.2. Les classes effectives descendantes de la classe NAIRE

La classe NAIRE admet deux classes effectives qui représentent les deux quantificateurs.

2.4.2.1. La classe QQS

```
class QQS   -- class Quelque Soit ∀
inherit
  NAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      from
        L_predicat.start
      until
         L_predicat.off or else not L_predicat.item.evaluation
      loop
        L_predicat.forth
      end
      Result:= L_predicat.off
    end
end
```

2.4.2.2 . La classe IE

```
class IE  -- class il existe ∃
inherit
  NAIRE
creation
  make
feature
  evaluation: BOOLEAN is
    do
      from
        L_predicat.start
until
        L_predicat.off or else L_predicat.item.evaluation
      loop
        L_predicat.forth
      end
      result:= not L_predicat.off
    end
end
```

2.5.. L'abstraction OPERANDE

L'abstraction OPERANDE permet de modéliser la notion d'une fonction booléenne définie sur une seule variable.

```
class OPERANDE
inherit
 PREDICAT
creation
  make
feature
variable: VAR
feature
  evaluation: BOOLEAN is
```

```
    do
      Result:= variable.etat
    end
feature
  make (var: VAR) is
    require
      var_existe: var /= void
    do
      variable:= var
    end
end
```

2.6.. L'abstraction VAR

Elle modélise une variable booléenne au sens général. Elle propose une routine abstraite permettant de connaître l'état courant de la variable.

```
deferred class VAR
feature
  etat: BOOLEAN is
    deferred
    end
end
```

2.7.. L'abstraction VAR_L

La classe VAR_L modélise la notion d'une variable logique.

```
class VAR_L  -- classe variable logique
inherit
  var
feature
  etat: BOOLEAN       -- etat implémenté par mémorisation
feature
  make (bool: BOOLEAN) is
    do
      etat:= bool
    end
end
```

2.8.. La classe EGAL

```
class EGAL
inherit
  VAR
creation
  make
feature
  op_c1: OP_COMP
op_c2: OP_COMP
feature
  etat: BOOLEAN is                    -- etat implémenté par calcul
    do
      Result:= op_c1.var_s.is_equal (op_c2.var_s)
    end
feature
   make (op1,op2: OP_COMP) is
    require
      op_comp_existe: op1 /= void and then op2 /= void
    do
      op_c1:= op1
      op_c2:= op2
    end
end
```

2.9.. La classe OPERANDE _COMPARAISON

La classe Operande_Comparaison comporte un attribut var_s qui mémorise une valeur de type chaîne de caractères. En effet les contraintes du langage Wright comportent souvent des comparaisons sur les types et les noms – considérés comme des identificateurs – des éléments utilisés dans une configuration Wright.

```
class OP_COMP
feature
  var_s: STRING
end
```

2.10.. La classe OPERANDE _COMPARAISON_NAME

Elle modélise des opérandes relatives aux noms des instances utilisées dans une configuration Wright.

```
class OP_COMP_N
inherit
  OP_COMP
creation
  make
feature
  instance: INSTANCE
feature
  make (inst: INSTANCE) is
    require
      instance_existe: i /= void
    do
      instance:= inst
      var_s:= instance.name
end
end
```

2.11.. La classe OPERANDE _COMPARAISON_Type

Elle modélise des opérandes relatives aux types des instances utilisés dans une configuration Wright.

```
class OP_COMP_T
inherit
  OP_COMP
creation
  make
feature
instance_existe: inst /= void
    do
instance:= inst
var_s:= instance.type
    end
end
```

2.12.. La classe OPERANDE _COMPARAISON_STRING

Elle modélise des opérandes considérées comme des chaînes de caractères.

```
class OP_COMP_S
inherit
  OP_COMP
creation
  make
feature
  make (valeur: STRING) is
    do
   var_s:= valeur
    end
end
```

3. Expérimentation

Dans cette partie, nous présentons une expérimentation de notre outil EVALCWright sur un exemple significatif.

3.1.. Présentation du problème

Il s'agit d'une application permettant de générer un index relatif à un ensemble de couples (mot, numéro(s) de ligne) (Sanlaville, 1997). Une telle application comporte les caractéristiques suivantes :

- Un mot correspond à une séquence de lettres et/ou de chiffres. Ces mots sont délimités par des séparateurs. Dès qu'un mot a été trouvé, il est transcrit en miniscule,
- La numérotation des lignes commence à 1 et son incrémentation a lieu lorsque nous rencontrons le caractère spécial CR, (retour chariot),
- Les mots identiques sont fusionnés et leur numéro de ligne sont soit fusionnés s'ils sont identiques, soit stockés par ordre croissant s'ils sont différents. Ainsi, à la sortie de notre index, il ne peut pas exister deux couples ayant le même mot,
- Les couples doivent être ordonnés par un tri alphabétique au niveau des mots.

3.2.. Présentation de l'architecture de la solution

L'architecture de cette application comporte trois composants : Filtre_Texte, Filtre_Tri et Filtre_Fusion ; et par deux connecteurs Pipe1 et Pipe2.

On présente par la suite le composant Filtre_Texte : A partir d'une séquence de caractères (y compris le caractère espace et retour chariot) il détermine les mots et leurs affecte leur numéro de ligne. (Voir figure B.1).

Component Filtre_Texte
Port Input = Lire les données, s'arrêter au signal end-of-data.
Port Output = Envoyer les données, signaler la terminaison par close.
Computation = Lire les données du port Input, suivant les cas
continuer à lire les données du port Input ou envoyer
le couple (mot, numéro ligne) sur le port Output.

Figure B.1. *Le composant Filtre_Texte*

On identifie pour l'application index deux connecteurs de type Pipe. Ce dernier reçoit les données par l'intermédiaire du rôle source et les transmet par le rôle destination. (voir figure B.2)

Connector Pipe
Role Source = Lire les données, s'arrêter au signal end-of-data.
Role Sink = Envoyer les données, signaler la terminaison par close.
Glue = Sink va recevoir les données dans l'ordre délivré par Source.

Figure B.2. *Le connecteur Pipe*

Ainsi on présente toutes les caractéristiques de l'architecture de l'application index en Wright sans spécifier les différentes expressions écrites en CSP (voir figure B.3 et B.4). Le

traitement_i représente une expression en CSP.

```
Style PipeFilter
Interface Type DataOutput = Traitement_1
Interface Type DataInput = Traitement_2
Connector Pipe
Role Source = DataOutput
Role Sink = DataInput
Glue = Traitement_3
Constraints
∀ c : Connector • Type(c) = Pipe
∀ c : Connector ; p : Port \ p ∈ ports(c) • Type(p) = DataInput ∨ Type(p) = DataOuput
End Style
```

Figure B.3. *Le style Pipe_Filtre*

```
Configuration Index
Component Filtre_Texte
                Port Input = DataInput
                Port Output = DataOutput
                Computation = Traitement_4
        Component Filtre_Tri
                Port Input = DataInput
                Port Output = DataOutput
                Computation = Traitement_5
Component Filtre_Fusion
                Port Input = DataInput
                Port Output = DataOutput
                Computation = Traitement_6
Instances
Texte     :   Filtre_Texte
Tri       :   Filtre_Tri
Fusion    :   Filtre_Fusion
P1, P2    :   Pipe
Attachments
Text.Output     as     P1.Source
Tri.Input       as     P1.Sink
Tri.Output      as     P2.Source
Fusion.Input    as     P2.Sink
 End Configuration
```

Figure B.4. *La cofiguration de l'application index*

Nous avons conçu et réalisé une classe Eiffel (voir 3.3 classe Test) permettant de créer une configuration de l'application index. De plus, nous avons vérifié que cette configuration respecte les deux contraintes exprimées au niveau du style (voir figure B.3). Nous avons réalisé la configuration de l'application index et l'exécution de deux contraintes par notre évaluateur EVALCWright.

```
class TEST
creation
  make
feature
  attribuer_inst_comp(L_port: LINKED_LiST[PORT];inst_comp: I_COMP) is
   do
     from
      L_port.start
     until
      L_port.off
     loop
       L_port.item.attribuer_instance_component(inst_comp)
       L_port.forth
     end
   end
  attribuer_inst_conn(L_role: LINKED_LiST[ROLE];inst_conn: I_CONN) is
     do
     from
      L  role.start
```

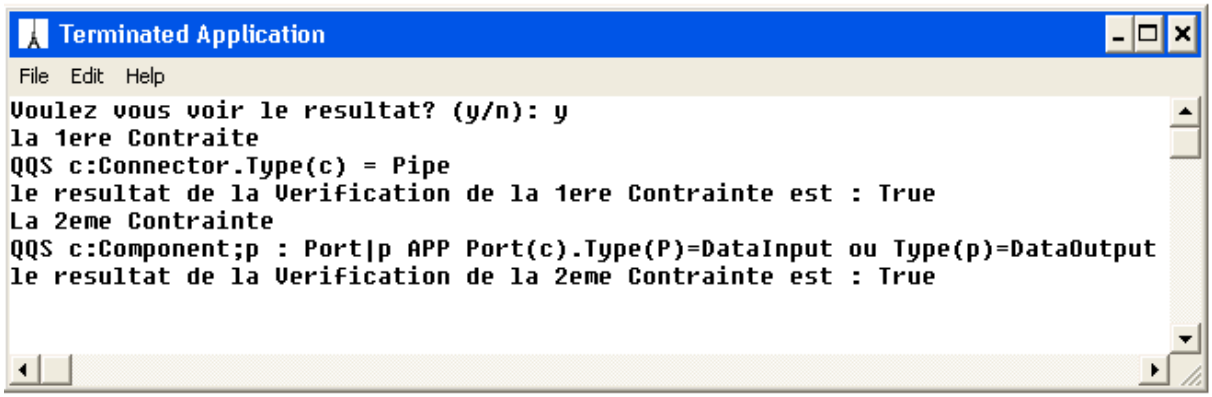
```
 until
     L_role.off
    loop
      L_role.item.attribuer_instance_connector(inst_conn)
      L_role.forth
  end
  end
feature
  make is
local
       chaine: STRING
        t_comp1, t_comp2, t_comp3: T_COMP              -- Type_Composant
        t_conn: T_CONN                                 -- Type_Connector
        i_comp1, i_comp2, i_comp3: I_COMP              -- Instance_Composant
        i_conn1, i_conn2, i_conn3: I_CONN              -- Instance_Conecteur
        p1,p2,p3,p4,p5,p6: PORT
        r1,r2,r3,r4: ROLE
        comp1, comp2, comp3: COMPU                     -- Computation
        g1,g2: GLUE                                    -- glu
        a1,a2,a3,a4: ATTA                              -- Attachements
        csp1,csp2,csp3: CSP
        L_port_comp1, L_port_comp2,L_port_comp3: LINKED_LIST [PORT]
        L_role_conn1, L_role_conn2: LINKED_LIST [ROLE]
        L_component: LINKED_LIST [I_COMP]
        L_connector: LINKED_LIST [I_CONN]
        L_qqs1: LINKED_LIST [PREDICAT]
        L_qqs2: LINKED_LIST [PREDICAT]
qqs1,qqs2: QQS
        ou1,ou2,ou3,ou4,ou5,ou6: OU
        op1,op2,op3,op4,op5,op6,op7,op8,op9,op10,op11,op12,op13,op14:
OPERANDE
        eg1,eg2,eg3,eg4,eg5,eg6,eg7,eg8,eg9,eg10,eg11,eg12,eg13,eg14: EGAL
        type1,type2,type3,type4,type5,type6,type7,type8: OP_COMP_T
        chaine1,chaine2,chaine3: OP_COMP_S
do
         !!L_component.make
         !!L_connector.make
         !!L_port_comp1.make
         !!L_port_comp2.make
         !!L_port_comp3.make
         !!L_role_conn1.make
         !!L_role_conn2.make
         !!L_qqs1.make
         !!L_qqs2.make
          !!t_comp1.make("Filtre_Texte")               -- creation Type composant
          !!t_comp2.make("Filtre_Tri")
         !!t_comp3.make("Filtre_Fusion")
           !!t_conn.make("PIPE")                       -- creation Connecteur
         !!csp1.make("DataInput")                     -- creation Interface
         !!csp2.make("DataOutput")
         !!csp3.make("Traitement")
          !!p1.make("Input",csp1)                     --creation Port
         !!p2.make("Output",csp2)
         !!p3.make("Intput",csp1)
         !!p4.make("Output",csp2)
         !!p5.make("Input",csp1)
         !!p6.make("Output",csp2)
         L_port_comp1.extend(p1)
         L_port_comp1.extend(p2)
         L_port_comp2.extend(p3)
         L_port_comp2.extend(p4)
         L_port_comp3.extend(p5)
         L_port_comp3.extend(p6)
```

```
    !!comp1.make(csp3)                    -- creation computation
      !!comp2.make(csp3)
      !!comp3.make(csp3)

      !!r1.make("Source",csp2)              -- creation Role
      !!r2.make("Sink",csp1)
      !!r3.make("Source",csp2)
      !!r4.make("Sink",csp1)

      L_role_conn1.extend(r1)
      L_role_conn1.extend(r2)
      L_role_conn2.extend(r3)
      L_role_conn2.extend(r4)

      !!g1.make(csp3)                     -- creation Glue
      !!g2.make(csp3)
!!i_comp1.make("Text",t_comp1.name,comp1,L_port_comp1)     --creation Instance
Component
      !!i_comp2.make("Tri",t_comp2.name,comp2,L_port_comp2)
      !!i_comp3.make("Fusion",t_comp3.name,comp3,L_port_comp3)

      !!i_conn1.make("P1",t_conn.name,g1,L_role_conn1)          --creation Instance
Connector
      !!i_conn2.make("P2",t_conn.name,g2,L_role_conn2)

      --attribuer instance
        attribuer_inst_comp(L_port_comp1,i_comp1)
        attribuer_inst_comp(L_port_comp2,i_comp2)
        attribuer_inst_comp(L_port_comp3,i_comp3)
        attribuer_inst_conn(L_role_conn1,i_conn1)
        attribuer_inst_conn(L_role_conn1,i_conn1)
comp1.attribuer_instance_component(i_comp1)          --attribuer Computation
        comp2.attribuer_instance_component(i_comp2)
        comp2.attribuer_instance_component(i_comp3)


        g1.attribuer_instance_connector(i_conn1)            -- attribuer Glue
        g2.attribuer_instance_connector(i_conn2)
!!a1.make(p2,r1)                          --creation attachements
        !!a2.make(p3,r2)
        !!a3.make(p4,r3)
        !!a4.make(p5,r4)
        L_component.extend(i_comp1)
        L_component.extend(i_comp2)
        L_component.extend(i_comp3)
        L_connector.extend(i_conn1)
        L_connector.extend(i_conn2)
--Verification Des Contraintes
        --1ere Contraintes
        !!chaine1.make("Pipe")
!!type1.make(i_conn1)
        !!type2.make(i_conn2)
        !!eg1.make(type1,chaine1)
        !!eg2.make(type2,chaine1)
        !!op1.make(eg1)
        !!op2.make(eg2)

        L_qqs1.extend(op1)
        L_qqs1.extend(op2)
```

```
            !!qqs1.make(L_qqs1)
           --2eme Contraintes
           !!chaine2.make("DataInput")!!chaine3.make("DataOutput")
           !!type3.make(p1)
           !!type4.make(p2)
           !!type5.make(p3)
           !!type6.make(p4)
           !!type7.make(p5)
           !!type8.make(p6)
           !!eg3.make(type3,chaine2)
           !!eg4.make(type3,chaine3)
           !!eg5.make(type4,chaine2)
            !!eg6.make(type4,chaine3)
           !!eg7.make(type5,chaine2)
           !!eg8.make(type5,chaine3)
           !!eg9.make(type6,chaine2)
           !!eg10.make(type6,chaine3)
           !!eg11.make(type7,chaine2)
           !!eg12.make(type7,chaine3)
           !!eg13.make(type8,chaine2)
           !!eg14.make(type8,chaine3)
           !!op3.make(eg3)
           !!op4.make(eg4)
           !!ou1.make(op3,op4)
           !!op5.make(eg5)
           !!op6.make(eg6)
           !!ou2.make(op5,op6)
            !!op7.make(eg7)
           !!op8.make(eg8)
           !!ou3.make(op7,op8)
           !!op9.make(eg9)
           !!op10.make(eg10)
           !!ou4.make(op9,op10)
           !!op11.make(eg11)
           !!op12.make(eg2)
           !!ou5.make(op11,op12)
            !!op13.make(eg13)
           !!op14.make(eg14)
           !!ou6.make(op13,op14)
           L_qqs2.extend(ou1)
           L_qqs2.extend(ou2)
           L_qqs2.extend(ou3)
           L_qqs2.extend(ou4)
           L_qqs2.extend(ou5)
           L_qqs2.extend(ou6)
           !!qqs2.make(L_qqs2)
--test
    io.putstring("Voulez vous voir le resultat? (y/n): y")
    io.new_line
    io.putstring("la 1ere Contraite")
    io.new_line
io.putstring("QQS c:Connector.Type(c) = Pipe")
io.new_line
    if(qqs1.evaluation) then
      io.putstring("le resultat de la Verification de la 1ere Contrainte est : True")
    else
      io.putstring("le resultat de la Verification de la 1ere Contrainte est : True")
    end
io.new_line
    io.putstring("La 2eme Contrainte")
```

```
    io.new_line
    io.putstring("QQS c:Component;p : Port|p APP Port(c).Type(P)=DataInput ou
Type(p)=DataOutput")
    io.new_line
    if(qqs2.evaluation) then
      io.putstring("le resultat de la Verification de la 2eme Contrainte est : True")
    else
      io.putstring("le resultat de la Verification de la 2eme Contrainte est : False")
    end
  end --make
end --TEST
```

3.4.. Exécution des deux contraintes



Résumé :

Cette thèse propose une Démarche de Vérification Formelle d'Architectures Logicielles : DVFAL. La démarche DVFAL supporte divers formalismes de description d'architectures logicielles tels que : les ADL (langages de description d'architectures), UML2.0, Symphony et des profils UML2.0 dédiés au domaine des architectures logicielles. La démarche DVFAL préconise l'ADL Wright en tant que langage formel pivot permettant de représenter des architectures logicielles décrites dans les divers formalismes. En outre, elle propose des transformations de modèles sous forme des traducteurs (Wright vers CSP de Hoare et Wright vers Ada) pour bénéficier des outils de vérification des propriétés supportant CSP et Ada tels que FDR et FLAVERS. Enfin, la démarche DVFAL propose un profil UML2.0-Wright jouant le rôle d'un langage intermédiaire entre les formalismes à base d'UML et Wright.

Mots clés : Langage de Description d'Architectures, architecture logicielle, UML, vérification formelle, Wright

---

Abstract :

This thesis proposes a process for Formal Verification of Software Architectures: DVFAL. The DVFAL process supports various formalisms of description of software architectures as: the ADL (Architecture Description Language), UML2.0, Symphony and of the UML2.0 profiles dedicated to the the software architectures domain. The DVFAL process recommends the Wright ADL as a pivot formal language permitting to represent software architectures described in the various above stated formalisms. Besides, he proposes transformations of models into the translators (Wright toward CSP of Hoare and Wright toward Ada) to benefit from tools of verification of the properties supporting CSP and Ada as FDR and FLAVERS. Finally, the DVFAL process proposes a UML2.0-Wright profile playing the role of an intermediate language between the formalisms basis of UML and Wright.

Keywords : Architecture Description Language, software architecture, UML, formal verification, Wright