



A Hierarchical Component Model with Interaction Protocols

Sebastien Pavel

► To cite this version:

Sebastien Pavel. A Hierarchical Component Model with Interaction Protocols. Software Engineering [cs.SE]. Université de Nantes, 2008. English. NNT : . tel-00484788v2

HAL Id: tel-00484788

<https://theses.hal.science/tel-00484788v2>

Submitted on 26 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX

Année 2008

N° attribué par la bibliothèque

ED 503-037

Un modèle de composants hiérarchiques avec protocoles d'interaction

A Hierarchical Component Model with Interaction Protocols

THÈSE DE DOCTORAT

Spécialité Informatique
Présenté et soutenue publiquement par

Sebastian Pavel

Le 21 Octobre 2008
à l'École Nationale Supérieure des
Techniques Industrielles et des Mines de Nantes

devant la commission d'examen composée de :

Rapporteurs	:	Laurence Duchien	Professeur des Universités, Lille I
		Antoine Beugnard	Enseignant-Chercheur, HDR, ENST de Bretagne
Examineurs	:	Frédéric Besson	Chargé de Recherche, INRIA
	:	Mourad Oussalah	Professeur des Universités, Nantes
	:	Jacques Noyé	Maître-Assistant, École des Mines de Nantes
Directeur de thèse	:	Jean-Claude Royer	Professeur, École des Mines de Nantes

Laboratoire d'accueil : Laboratoire d'Informatique de Nantes-Atlantique (UMR 6241)

Équipe d'accueil : École des Mines de Nantes - Objets, Aspects et Composants (OBASCO)

Résumé

L'utilisation et la gestion des composants sont au coeur des nouvelles architectures logicielles. Les composants représentent les briques de bases des logiciels. Les efforts de recherche actuels se concentrent sur l'élaboration de modèles à base de composants qui intègrent des propriétés importantes comme, par exemple, la description et l'intégration des composants avec des comportements explicites (protocoles d'interaction). Ce sont ces descriptions plus complètes que les interfaces classiques (les points d'entrée et de sortie), qui ouvrent la voie vers la correction des assemblages.

Comme aboutissement des travaux de cette thèse, nous proposons un modèle de composants qui utilise des Systèmes de Transitions Symboliques (STSs) pour décrire les comportements des composants. Les composants de notre modèle sont des boîtes noires communiquant exclusivement par l'intermédiaire de leurs interfaces étendues avec des protocoles d'interaction. Le modèle spécifie aussi les règles de compatibilité, les algorithmes de vérification des assemblages des composants et de la substitution et un langage de description des composants.

Nous proposons une implémentation dans le langage Java en suivant une approche générative où le code Java est généré à partir des descriptions des composants de haut niveau. Le code est donc garanti à être conforme à la spécification.

Mots-clés : modèle de composants, protocoles d'interaction, systèmes de transitions symboliques (STS), génération de code, Java.

Abstract

Title : A Hierarchical Component Model with Interaction Protocols

The component-based Software Engineering (CBSE) represents an important trend in the development of software architectures. The components are the core of the software applications and the recent efforts concentrate on conceiving component models integrating important properties as for example explicit interaction protocols. Interaction protocols allow a component to publish its behavior in terms of message emission and receipt. Thus, component assemblages can be more effective than in the traditional approach where only static interfaces (input and output) are available.

In this thesis, we propose a component model considering black-box components integrating interaction protocols described as Symbolic Transition Systems (STSs). STSs allow the description and verification of complex protocols while dealing with the state explosion problem for example. The model also specifies the compatibility rules, the substitution and the assemblage verification algorithms and a component description language. We propose a generative approach on implementing our model in the Java programming language. The Java code is automatically generated starting from high level component descriptions assuring that the component code is conforming with its specification.

Keywords : component models, interaction protocols, symbolic transition systems (STS), code generation, Java.

Remerciements

Je remercie tout d'abord les membres du jury qui ont accepté d'évaluer mon travail. Merci aux rapporteurs de cette thèse, Laurence Duchien professeur à l'Universités de Lille I qui m'a soutenu pendant les quelques années d'interaction dans le cadre du projet ACI Dispo et à Antoine Beugnard, enseignant-chercheur à ENST de Bretagne qui, m'a beaucoup influencé par son travail dans le domaine des composants logiciels. Merci aux autres membres pour l'intérêt dans mon travail et pour l'honneur qui me font de participer dans ce jury.

Merci à Jean-Claude Royer, mon directeur de thèse. Sans toi Jean-Claude, je n'ai surement pas pu arriver si loin. Ta confiance et ton énergie m'ont beaucoup aidé à surmonter les difficultés et suivre le bon chemin. Merci aussi à Jacques Noyé, mon co-encadrant pour son rigueur et sa minutie. C'est la poursuite de la perfection qui te définit Jacques et je ne peut-être que reconnaissant pour tes conseils.

Merci aux membres du département informatique de l'Ecole de Mines de Nantes qui m'ont accueilli parmi eux avec un très grand merci à Pierre Cointe, ancien chef du service, qui a tout fait pour que je puisse intégrer l'EMN dans les meilleures conditions. Merci à Remy Douence et à Mario Südholt pour leur soutien et leur bonne parole dans des moments difficiles.

Merci aussi à toi Thierry Petit pour ton courage et ta volonté qui me donne la force de continuer à me battre et aussi à profiter des choses simples.

Merci aussi aux membres du projet ACI Dispo avec lesquels j'ai pu échanger beaucoup sur la recherche et qui ont représenté pour moi un soutien important tant sur le plan personnel que professionnel.

Merci aux anciens thésards parmi lesquels j'ai fait mes premiers pas dans le monde de la recherche : Yann-Gaël Guéhéneuc, Marc Ségura-Devillechaise, Gustavo Bobeff, Simon Denier et tous les autres.

Merci à ma famille et à mes amis qui ont su me soutenir et aussi me supporter dans beaucoup des moments très difficiles.

Sans toutes ces personnes, ce long travail n'aurait pas pu être amené au bout. Un grand merci à tous.

Table des matières

I	Résumé en Français	15
1	Résumé	17
1.1	Domaine d'étude	19
1.1.1	L'ingénierie des composants	19
1.1.2	Protocoles d'interaction : formalismes connus	23
1.2	Objectifs et propositions	24
1.2.1	Le modèle CwSTS	26
1.2.2	L'implémentation du modèle	33
1.2.3	Perspectives	44
1.3	Structure du mémoire de thèse	46
II	Work Context in English	49
2	Introduction	51
2.1	Objectives and Contributions	52
2.2	Document Structure	54
3	Component Models and Languages	57
3.1	From Objects and Modules to Software Components	58
3.2	Component Models	59
3.2.1	Component Model Characteristics	59
3.2.2	Academic Models	67
3.2.3	Industrial Models	71
3.3	From Components to Software Architecture	76
3.3.1	Software Architecture Definition	76
3.3.2	Software Connectors	76
3.3.3	Software Architecture Use	78
3.3.4	Architecture Description Languages (ADLs)	79
3.3.5	Service-Oriented Architectures (SOA)	83
3.3.6	Component-Oriented Programming (COP)	86
3.3.7	The Coordination Paradigm	89
3.4	Conclusions	91

4	Interaction Protocols	93
4.1	Introduction	93
4.2	Formalisms	94
4.2.1	Process Algebra	94
4.2.2	Behavioral Types	96
4.2.3	Finite State Machines	98
4.2.4	Temporal Logics	100
4.2.5	Other Approaches	101
4.3	Component Models and Interaction Protocols	101
4.3.1	Automata-Based Models	101
4.3.2	Regular Types	102
4.3.3	Coordination-Based Models	103
4.3.4	Other Approaches	104
4.4	Conclusions	105
III	Contribution in English	107
5	A Component Model with Explicit Interaction Protocols	109
5.1	Introduction	110
5.1.1	Components, a Generative Approach	111
5.2	Informal Presentation	112
5.2.1	Components	114
5.2.2	Interfaces	114
5.2.3	Composition	116
5.2.4	Life Cycle	121
5.3	Model Definition	122
5.3.1	Components	123
5.3.2	Interaction Protocols	123
5.3.3	Composition	125
5.3.4	Component Substitutability	126
5.4	CwSTS-Interface Description Language	128
5.4.1	Primitive Components	128
5.4.2	Composite Components	129
5.4.3	Symbolic Finite State Processes (SFSP) a process algebra for STSs	131
5.5	Conclusion	133
6	CwSTS Implementation	135
6.1	Component Implementation	135
6.1.1	Introduction	135
6.1.2	Java Packages and Component Entities	136
6.1.3	Primitive Component Implementation	137
6.1.4	Architectures Implementation	141
6.2	Behavioral Composition Implementation	146

6.2.1	Distributed Synchronization Mechanism	147
6.2.2	Distributed Synchronization Mechanism Integration in CwSTS-P . .	148
6.2.3	Distributed Synchronization Mechanism Evaluation	151
6.2.4	Centralized Synchronization Mechanism	152
6.2.5	Centralized Synchronization Mechanism Integration in CwSTS-P . .	153
6.2.6	Centralized Synchronization Mechanism Evaluation	155
6.3	Code Generation	155
6.4	Conclusion	157
7	Conclusions and Perspectives	159
7.1	Perspectives	163

Table des figures

1.1	Syntaxe graphique d'un composant CwSTS.	27
1.2	Le protocole du composant client du serveur de messagerie.	28
1.3	Le composant StatisticsEnabledServer	30
1.4	Le protocole du composant serveur de messagerie.	31
1.5	Le protocole du composant StatisticsReporter	31
1.6	Le protocole du composant adaptateur.	32
1.7	Le rôle du composant adaptateur dans l'architecture du système de messagerie.	32
1.8	Raffinement de la fonction f par la fonction g.	33
1.9	Grammaire du langage de description des interfaces CwSTS-IDL.	34
1.10	La grammaire abstraite du langage SFSP.	34
1.11	La définition du composant client du serveur de messagerie.	35
1.12	Scénarios de génération de composants.	36
1.13	Structuration d'un composant sous la forme d'un <i>package</i> Java.	37
1.14	Vue fonctionnelle sur la structure d'un composants primitif.	38
1.15	Vue de configuration sur la structure d'un composant primitif.	38
1.16	Création d'une instance d'un composant primitif.	39
1.17	Interaction des entités du composant primitif.	40
1.18	Vue fonctionnelle sur la structure d'un composant composite.	41
1.19	Vue de configuration sur la structure d'un composant composite.	42
1.20	Création d'une instance d'un composant composite.	43
1.21	Exécution parallèle de deux actions.	45
1.22	États mixtes dans l'interaction entre deux composants.	46
3.1	Architecture composite component.	64
3.2	Fractal Architecture.	68
3.3	CCM Component.	72
3.4	CORBA Component Model Execution Environment.	73
5.1	A Generative Approach to Components.	111
5.2	Graphical Component Syntax.	114
5.3	Message Client Protocol.	116
5.4	Composite Component Example.	117
5.5	The Synchronous Product.	117

5.6	StatisticsEnabledServer Protocol Representing the Synchronous Product of the Server and StatisticsReporter Protocols with Hidden Internal Actions.	118
5.7	The Server Protocol.	119
5.8	The StatisticsReporter Protocol.	119
5.9	Adaptor Component Protocol.	122
5.10	Complete Architecture of the Case Study.	122
5.11	Function refinement type relationships.	127
5.12	CwSTS Abstract EBNF Grammar.	129
5.13	Client Component Definition.	130
5.14	StatisticsEnabled composite component definition.	131
5.15	SFSP Abstract Grammar.	132
5.16	Client SFSP definition.	132
6.1	Component Package in Java.	136
6.2	Primitive Component Structure with a Partial View Over the Interactions.	138
6.3	Primitive Component Class Diagram.	138
6.4	Primitive Component Internals.	139
6.5	Component Structure.	140
6.6	Primitive Component Instance Creation.	140
6.7	Composite Functional Structure.	142
6.8	Composite Control Structure.	142
6.9	Composite Instance Creation.	144
6.10	Composite Execution.	145
6.11	ComponentController Java Class Excerpt.	149
6.12	Mixed State Situation.	151
6.13	Centralized Arbiter Entity	152
6.14	Component Protocols.	153
6.15	Synchronization Mechanism in Action.	154
6.16	Arbiter Implementation.	154
7.1	Parallel Execution of Two Actions.	164
7.2	Mixed State Situation.	164

Première partie

Résumé en Français

Chapitre 1

Résumé

Dans les années soixante il est devenu évident que la façon de produire les logiciels avait besoin d'être reconsidérée. Les logiciels étaient développés sur demande et très souvent sans rien capitaliser des développements précédents. Les premiers désavantages sont apparus au fur et à mesure que la demande a augmenté : la qualité était souvent médiocre et les coûts d'exploitation et de maintenance très élevés. De plus, la dynamique du marché a imposé que les applications logicielles soient rapidement modifiables pendant la phase de développement et aussi de plus en plus complexes, évolutives ou adaptables. Les pressions exercées par le marché et les clients avaient également un impact négatif sur la documentation : le temps alloué à celle-ci était insuffisant et le résultat souvent un produit incomplet voire même incorrect.

L'ingénierie des composants (*Component-Based Software Engineering* [Heineman 01a] en anglais) propose des réponses pour palier aux difficultés présentées ci-dessus. L'ingénierie des composants considère les composants logiciels comme la brique de base de toute construction logicielle. La notion de composant logiciel a été mentionnée pour la première fois par McIlroy [McIlroy 68] dans son discours à la conférence de l'OTAN en 1968. L'idée de brique logicielle comme solution aux problèmes de développement a émergé à cette époque. Les composants logiciels sont liés aux modules [Wirth 77, Mitchell 79] et à la programmation par objets (*Object-Oriented Programming* en anglais).

Les composants logiciels sont liés aux modules par le fait que la technologie des composants amène à mettre en place des solutions modulaires. Ceci dit, si la modularité est un prérequis, des règles supplémentaires (comme par exemple l'indépendance des composants et le contrôle explicite des dépendances entre les composants) sont nécessaires pour former des composants plutôt que des modules.

À la fin des années 90, le constat a été fait que l'approche de la programmation par objets était insuffisante pour construire des logiciels de grande taille et d'améliorer fortement la réutilisation. En contraste avec les mécanismes à objets qui introduisent un couplage fort entre les classes de base et les classes dérivées (problème connu sous le nom de classe de base fragile [Szyperski 96, Mikhajlov 98]), l'approche par composants fournit une meilleure séparation entre l'implémentation d'un composant et son interface. Chaque composant est une entité indivisible de type boîte noire qui peut être composée avec d'autres composants

et peut être déployée indépendamment des autres composants. La construction des logiciels est réalisée par assemblage de ces composants. Dans cette approche, l'interface du composant est d'une importance primordiale car elle définit un contrat entre le composant et son environnement. L'interface décrit d'une manière très explicite ce que le composant fournit et ce qu'il requiert de son environnement dans le but de réaliser sa finalité dans l'architecture de l'application.

Les caractéristiques particulières d'un composant sont décrites par les *modèles de composants*. Un *modèle de composants* décrit ce qu'un composant est (par définition de ses parties constitutives) et spécifie comment les composants peuvent être assemblés en suivant des règles de composition. Le modèle de composant décrit également le cycle de vie du composant et les rôles associés aux différents acteurs du développement logiciel et de l'exploitation des applications.

Les composants sont des entités qui collaborent en échangeant des messages pour coordonner des actions ou seulement échanger des données. Pour une interaction réussie, les composants doivent se conformer à une forme de contrat d'interaction. Les protocoles d'interaction décrivent généralement le comportement d'une entité (objet, acteur, agent ou composant) en termes de séquences de messages qui peuvent être échangés par les entités pour réaliser le comportement global attendu de l'application.

L'ingénierie des composants considère les protocoles d'interaction au niveau des interfaces des composants. Au début, les interfaces ont seulement décrit les signatures des services fournis et requis par le composant. Plus récemment, les protocoles d'interaction ont fait leur apparition dans des modèles de composants plus élaborés où le comportement dynamique est explicitement décrit. Ceci permet une meilleure documentation du comportement visible du composant et une meilleure intégration dans une architecture logicielle, mais ouvre également la voie à des outils et des techniques de vérification pour par exemple vérifier, *a priori*, la validité d'un assemblage.

En découplant l'interface de son implémentation, une question importante se pose : comment assurer que l'implémentation est *conforme* (ou *cohérente*) avec l'interface. Si dans une approche purement syntaxique où seulement les signatures des services sont spécifiées par une interface, le test de conformité est simple. Dans le cas où le comportement dynamique est également spécifié, l'implémentation se révèle plus difficile à réaliser. Beaucoup de modèles de composants de l'état de l'art actuel ne prennent pas en compte ce problème. D'autres se basent sur des techniques d'analyse de code pour assurer la conformité du code. Les langages de programmation de composants (*component programming language* en anglais) considèrent ce problème en permettant la programmation avec des composants plutôt que des objets. Le code est conforme à son interface car la description de l'interface se fait au même niveau que l'implémentation et le compilateur vérifie certaines propriétés importantes comme l'intégrité de la communication qui stipule que les composants ne peuvent communiquer que par des canaux bien spécifiés. Finalement, les techniques génératives assurent par construction que l'implémentation est conforme à la spécification de l'interface.

Les modèles et langages de composants actuels intègrent souvent des concepts trop nombreux ou ils sont trop spécifiques à certains contextes. Il est alors difficile d'analyser les conséquences de l'ajout de nouveaux concepts (comme les protocoles d'interaction) ou

de nouvelles fonctionnalités dans un modèle de composants existant. Les objectifs de cette thèse sont donc de proposer un modèle de composants qui intègre des protocoles d'interaction explicites donnés sous la forme des systèmes de transitions symboliques.

Suite à des expérimentations avec la programmation par composants (voir [Pavel 04] pour une proposition générative de développement des composants en Arch-Java [Aldrich 02b]) et avec l'élaboration d'un modèle de composants qui intègre des protocoles d'interaction et des communications asynchrones [Pavel 05a, Pavel 05b, Noyé 05], nous proposons un modèle de composants appelé CwSTS (*Components with Symbolic Transition Systems* en anglais). Des détails de notre proposition sont présentés dans la section 1.2 à la page 24.

1.1 Domaine d'étude

1.1.1 L'ingénierie des composants

Les caractéristiques particulières d'un composant sont décrites par les modèles de composants. Un modèle de composants décrit ce qu'est un composant (par la définition de ses parties constitutives) et spécifie la façon dont les composants peuvent être assemblés en suivant des règles de composition. Le modèle de composant décrit également le cycle de vie du composant et les rôles associés aux différents acteurs du développement logiciel et de l'exploitation des applications.

L'état de l'art actuel dans le domaine des modèles de composants n'a pas établi de consensus concernant une définition du terme composant. Plusieurs définitions co-existent [Brown 98, Larsson 00, Heineman 01b, Szyperski 02, Marvie 02]. Malgré ces différences, il est communément admis qu'un composant a au moins les propriétés suivantes :

- il est une unité indépendante de déploiement ;
- son état interne n'est pas visible de l'extérieur ;
- il est une unité de composition.

Szyperski [Szyperski 02] définit un composant logiciel comme une unité de composition avec des interfaces spécifiées contractuellement contenant uniquement des dépendances contextuelles explicites. Il peut être déployé d'une manière indépendante et il fait l'objet de composition par des tiers.

Cette définition a plusieurs implications. Premièrement, un composant a besoin d'encapsuler son implémentation et peut communiquer avec son environnement uniquement par l'intermédiaire des interfaces explicites. Ensuite, le composant est préparé pour être composé avec d'autres composants pour obtenir des applications logicielles. Cette définition implique aussi l'existence d'un cycle de vie associé au composant qui, d'une manière traditionnelle, consiste en quatre phases : la création, l'assemblage, le déploiement, et l'exécution. Une notion moins explicite est celle d'environnement où le composant peut se trouver dans ses différentes phases du cycle de vie.

1.1.1.1 Les contrats d'interfaces

La précédente définition des composants implique le fait que les dépendances contextuelles des composants soient explicites. Ceci est réalisé par l'intermédiaire des interfaces des composants. Les interfaces sont définies comme un ensemble de services que le composant fournit ou requiert de son environnement. Elles sont vues comme des contrats entre le composant et son environnement d'exécution. Antoine Beugnard [Beugnard 99] propose une taxonomie des contrats qui est liée aux interfaces de composants. Conformément à cette taxonomie, il y a quatre niveaux de contrats :

- *basique*;
- *comportemental*;
- *de synchronisation*;
- *quantitatif*;

Le niveau *basique* considère les propriétés syntaxiques : des noms, types de paramètres, types de retour et les exceptions des messages échangés par le composant. Le niveau *comportemental* considère des propriétés qui peuvent être spécifiées par des préconditions, postconditions et invariants. Ces propriétés sont liées à une opération et spécifient les types de paramètres et les dépendances entre les valeurs de ces paramètres. Le niveau *de synchronisation* considère les propriétés concernant les interactions des composants. Ces propriétés ne peuvent être exprimées seulement par l'intermédiaire de pré- et postconditions, car elles doivent décrire une séquence de pas à exécuter par le composant. Le niveau *quantitatif* correspond aux propriétés dites *non-fonctionnelles* comme la qualité de service, la gestion des ressources et le temps de réponse. Ces types de contrats sont les plus difficiles à exprimer et analyser car ils dépendent des propriétés qui ne sont pas connues au moment de la description de l'architecture.

1.1.1.2 Le mécanisme de composition

La *composition* est un mécanisme permettant la construction de composants complexes (appelés composants *composites*) à partir de composants plus simples (appelés composants *primitifs*). La composition est réalisée en mettant en correspondance les services des composants définis dans leurs interfaces. Plusieurs schémas de composition (*bind*, *export*, *import* et *glue*) sont utilisées pour obtenir des applications exécutables à partir des composants (voir la section 3.2.1.3 à la page 64).

La *compatibilité* est un concept très important qui intervient dans le mécanisme de composition. Indifféremment des schémas de composition utilisés, les services mis en correspondance doivent être *compatibles*. En fonction du niveau de contrats auquel nous nous plaçons, la compatibilité a un sens différent. Les services (et par extrapolation les interfaces) *incompatibles* ne devraient pas être connectés dans une architecture. Une adaptation peut être réalisée dans certaines conditions et par l'intermédiaire d'un *adaptateur*. L'adaptateur est appelé du *code glu* [Cherinka 98] et dans certains cas il faut plus de temps pour le développer que les composants concernés.

1.1.1.3 Modèles de composants dans la recherche et l'industrie

Aujourd'hui, il existe deux catégories de modèles de composants : les modèles académiques et les modèles industriels. Les modèles académiques (voir par exemple [Kenney 95, Bellissard 95, Magee 95, Medvidovic 96, Plašil 98, Flatt 98, Siegel 00, van Ommering 00, Cardone 00, Choppy 01, McDirmid 01, Seco 02, Sreedhar 02, Aldrich 02b, Coupaye 02]) se concentrent sur les concepts clés liés aux composants comme la définition des composants, de leurs interfaces et des mécanismes de composition. Différentes propriétés comme par exemple la substituabilité, la compatibilité et l'adaptation dynamique du comportement des composants sont étudiées. Les modèles industriels se concentrent sur la production, la distribution et l'exécution des applications industrielles. Ils fournissent des solutions comme par exemple la distribution, les transactions, la persistance et la sécurité dans le contexte des applications distribuées.

1.1.1.4 Architectures logicielles

Le standard ANSI/IEEE 1471 de 2000 définit l'*architecture logicielle* [Garlan 94, Bass 98] comme *l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec l'environnement et les principes qui guident sa conception et son évolution* [Shaw 96b, Szyperski 02]. Plus spécifiquement, l'architecture logicielle est l'organisation d'un système logiciel comme une collection de composants, de connexions entre les composants et des contraintes sur les modes d'interaction entre les composants.

L'utilisation d'une architecture logicielle peut être faite soit pour la conception et l'implémentation d'un système logiciel individuel, soit en tant que ligne de produits logiciels¹, soit comme une architecture standard utilisée pour un modèle public de composants [Bosch 00]. Les architectures logicielles sont généralement utilisées dans les but de :

1. Réduire le coût de développement.
2. Améliorer la qualité des logiciels en termes de fiabilité, maintenance et utilisation efficace des ressources.
3. Réduire le temps de mise sur le marché.
4. Réduire le coût de la maintenance.

Les *connecteurs logiciels* sont des éléments réutilisables de conception qui encapsulent un style particulier d'interaction entre les composants. Ils sont considérés comme des entités de première classe dans les architectures logicielles [Shaw 96a]. Les connecteurs régissent les règles d'interaction et précisent les mécanismes auxiliaires nécessaires dans ces interactions [Shaw 96b]. Plusieurs catégories de connecteurs existent, par exemple client-serveur, architecture en anneau, *publish-subscribe*, etc. [Mehta 00]. Alors que les connecteurs de communication assurent la transmission des données entre composants, ceux de coordination assurent le transfert de contrôle d'un composant à un autre. Des connecteurs spécifiques peuvent aussi convertir l'interaction fournie par un composant vers l'interaction requise par un autre composant. Finalement, les connecteurs peuvent aussi faciliter la médiation de

1. <http://www.sei.cmu.edu/productlines/>

l'interaction entre composants. Des mécanismes comme les optimisations des interactions sont typiquement réalisés par ce type de connecteurs.

1.1.1.5 Les langages de description d'architectures

Les langages de description d'architecture (*Architecture Description Languages* en anglais), représentent une avancée dans le domaine des architectures logicielles. Leur objectif est d'aider dans la structuration et la composition des briques logicielles dans le but d'obtenir des architectures valides. Un langage de description d'architecture est défini comme une notation formelle ou informelle, textuelle ou graphique, qui permet la spécification des architectures logicielles et qui est accompagnée d'outils spécifiques comme présenté dans [Medvidovic 00a]. Le résultat de l'utilisation d'un langage de description d'architecture est une vision abstraite de l'application en termes de composants, connecteurs et configurations. Medvidovic et Taylor [Medvidovic 00b] donnent un cadre de classification et de comparaison des langages de description d'architecture existants. Parmi les langages de description d'architecture actuels, Darwin [Magee 95], Wright [Allen 97] et Rapide [Kenney 95] sont les plus connus.

1.1.1.6 Langages de programmation de composants

La plupart du temps, la vision concrète de l'implémentation est ignorée dans une approche de type langage de description d'architecture. L'analyse architecturale réalisée avec un langage de description d'architecture peut révéler des propriétés importantes, mais celles-ci ne sont pas en général garanties par l'implémentation. Pour permettre une analyse architecturale au niveau du code, le code doit être conforme à l'implémentation [Luckham 95a]. La programmation par composants propose le développement de logiciels en les programmant directement avec des composants plutôt que des objets. Parmi les langages de programmation par composants, ArchJava [Aldrich 02a, Aldrich 02b], Java/A [Baumeister 06] et Jiazzi [McDirmid 01] sont les plus avancés.

1.1.1.7 Le paradigme de la coordination

La coordination est une approche centrée sur la collaboration entre processus. Cette approche considère la programmation des systèmes distribués ou parallèles comme la combinaison de deux activités distinctes : l'activité de calcul proprement dite qui contient les processus impliqués dans la manipulation des données et l'activité de coordination qui est responsable de la communication et de la coopération des processus. Papadopoulos et Arbab [Papadopoulos 98] argumentent le fait que les modèles de coordination peuvent être rangés en deux catégories : les modèles orientés données et les modèles orientés contrôle. Dans la première approche, l'évolution des calculs est contrôlée par les types et les propriétés des données impliquées dans l'activité de coordination. Dans le cas des modèles orientés contrôle (ou orientés processus), les changements dans l'état des processus coordonnés sont réalisés par des événements spécifiques.

La coordination peut-être utilisée comme base pour des modèles de composants. Arbab et al. [Arbab 01], par exemple, propose un modèle de composants qui utilise des *canaux*

mobiles pour une communication anonyme, point à point et qui permette une configuration dynamique des connexions à l'exécution.

La coordination est largement présentée dans la section 3.3.7 à la page 89.

1.1.2 Protocoles d'interaction : formalismes connus

Les entités qui collaborent pour réaliser des actions utilisent très souvent l'échange de messages pour coordonner leurs actions ou pour simplement échanger des données. Pour réaliser une interaction, les entités (objets, agents ou composants) ont besoin de se conformer à un contrat d'interaction. Les protocoles d'interaction décrivent le comportement de l'entité en termes de séquences de messages qui doivent être échangés par l'entité pour participer à la réalisation du comportement global du système. Plusieurs approches coexistent pour formaliser, analyser et implémenter les interactions entre les composants. Les algèbres de processus, les types comportementaux, les machines à états finis et les logiques temporelles sont les formalismes les plus utilisés pour spécifier des protocoles d'interaction.

1.1.2.1 Algèbres de processus

Les algèbres de processus représentent une famille de techniques de spécification particulièrement adaptées à la spécification des systèmes de composants concurrents communicants. Elles décrivent les interactions d'un processus en termes de calcul. Les plus importantes algèbres de processus sont CCS [Milner 89], CSP [Hoare 85] et LOTOS [Brinksma 87] car elles servent aussi de base pour d'autres algèbres de processus comme Pi-calcul [Milner 92, Milner 99] et les agents mobiles [Cardelli 98]. Même si à première vue, CCS, CSP et LOTOS semblent être très similaires (elles se basent sur la notion de processus composé d'actions atomiques avec des sémantiques opérationnelles et beaucoup d'opérateurs communs), des différences subtiles existent notamment au niveau des sémantiques des opérateurs qui sont souvent identiques d'un point de vue syntaxique.

1.1.2.2 Types comportementaux

Des travaux récents présents dans [Nierstrasz 93, Puntigam 96] défendent la thèse de l'extension des théories de types classiques pour permettre la description des propriétés dynamiques des objets dans le cadre d'un modèle comportemental. Ces nouveaux types sont appelés *types comportementaux* car ils ne spécifient pas seulement l'ensemble de messages que les entités vont échanger mais aussi, le plus important, des contraintes sur les séquences acceptables de ces messages.

1.1.2.3 Machines à états finis

Les protocoles d'interaction peuvent être spécifiés comme des *processus réguliers*, c.-à-d. des processus avec un nombre fini d'états ou de comportements. Les formalismes basés sur les machines à états finis (comme les systèmes de transitions étiquetées [Keller 76], systèmes de transitions symboliques, automates à entrées et sorties [Lynch 87] et les diagrammes d'états UML [UML 03]) permettent la description des comportements des systèmes à un

certain niveau d'abstraction. Les machines à états finis décrivent l'ensemble des traces possibles qu'un composant peut produire en interagissant avec ses partenaires.

1.1.2.4 Les logiques temporelles

Les logiques temporelles sont utilisées pour décrire et raisonner sur les propositions qualifiées en termes de temps. En pratique, les logiques temporelles permettent la spécification et la vérification des propriétés comme la sûreté, la vivacité, l'équité et l'absence de blocage. Les plus connues des logiques temporelles sont PLTL [Clarke 81] et CTL [Pnueli 81]. L'expression des protocoles d'interaction est réalisée directement en utilisant les opérateurs de base spécifiés par [Clarke 00], mais compte tenu du fait que les logiques temporelles sont très coûteuses en termes de vérification, il y a très peu de logiques temporelles réellement utilisables.

1.1.2.5 Langages et modèles à composants avec protocoles d'interaction

Des nombreux modèles et langages de composants intègrent les formalismes présentés antérieurement pour décrire des protocoles d'interaction. Par exemple, Darwin [Magee 95] permet la description des protocoles d'interaction sous la forme de LTS et la vérification des propriétés est réalisée en utilisant FSP (*Finite State Processes* en anglais). Une autre approche basée sur des automates [Barros 05] permet la description des protocoles d'interaction dans le modèle de composants Fractal [Coupaye 02]. Au niveau des types comportementaux, les travaux de Yellin et Strom [Yellin 97] sont considérés comme une référence dans le domaine de l'intégration des descriptions des comportements au niveau des interfaces des composants. Le modèle de composants CwEP [Farias 03] et les travaux de Cyril Carrez [Carrez 03] sont d'autres avancées importantes dans ce domaine. Arbab *et al.* [Arbab 02] propose un modèle de coordination pour les systèmes à base de composants qui est basé sur la notion de *canaux mobiles*. Les canaux mobiles permettent des communications point à point anonymes avec la possibilité de reconfigurer dynamiquement les connexions. En dehors des types de formalismes présentés, le modèle de composants SOFA [Plašil 98], l'environnement visuel PACOSUITE [Vanderperren 03] et le langage de spécification des comportements MIDAS [Pryce 98] représentent des approches alternatives à la spécification et l'implémentation des protocoles d'interaction.

1.2 Objectifs et propositions

Les modèles et langages de composants actuels intègrent souvent des concepts trop nombreux ou ils sont trop spécifiques à certains contextes. Il est alors difficile d'analyser les conséquences de l'ajout de nouveaux concepts (comme les protocoles d'interaction) ou de nouvelles fonctionnalités dans un modèle de composants existant.

Dans cette thèse nos objectifs sont de proposer un nouveau modèle de composants qui intègre des protocoles d'interaction explicites donnés sous la forme de systèmes de transitions symboliques (STS). Les STS sont une extension des LTS (*Labelled Transition Systems*

en anglais) où les transitions sont dites *symboliques*. Le symbolisme se manifeste par la présence de données échangées, de complexité quelconque, et l'utilisation de conditions pour le déclenchement des transitions (concept classique de *garde*).

Les protocoles d'interaction, qu'ils soient basés sur des STS ou sur d'autres formalismes, ont comme but principal une meilleure intégration des composants logiciels dans une architecture. Les règles spécifiées par les protocoles d'interaction sont utilisées pour détecter de possibles incompatibilités entre les composants. Une fois la propriété de compatibilité prouvée, les architectures à base de composants bénéficient d'une meilleure qualité, indispensable surtout dans certains contextes ou domaines d'activité.

L'intérêt d'utiliser des formalismes basés sur les automates à états finis comme les LTS ou les STS réside dans les techniques automatiques de vérification et dans les outils correspondants. La lisibilité et la compacité de ce type de formalisme sont souvent avancées comme un des avantages pour leur utilisation. En contraste avec les LTSs, les transitions des STS décrivent des classes d'opérations possibles. Les transitions sont paramétrées avec des paramètres formels d'entrée et peuvent être gardées avec des conditions (opérations booléennes) paramétrées. Le principal avantage lié à l'utilisation des STS consiste dans leur format compact, la lisibilité et l'expressivité. Les descriptions STS sont plus faciles à utiliser par un ingénieur classique ce qui permet de les utiliser plus souvent en pratique.

Nous souhaitons également avoir un langage de description de composants qui facilite l'écriture et la compréhension des composants par les développeurs. Notre volonté est d'orienter nos travaux vers une intégration facile dans le monde du développement des applications à base de composants. Dans cette optique nous pensons que les STS présentent des avantages notables par rapport à d'autres formalismes : notamment leur lisibilité et compacité.

Un des problèmes soulevés par les modèles et langages de composants qui intègrent des protocoles d'interaction est la conformité entre la spécification et l'implémentation. Pour pallier à ce problème nous proposons une approche générative en ce qui concerne l'implémentation. En effet, la génération du code peut assurer *par construction* la conformité de l'implémentation à sa spécification conceptuelle.

Sur la base des travaux préliminaires présentés précédemment, nous avons développé un modèle de composants que nous appelons CwSTS (*Components with Symbolic Transition Systems* en anglais). CwSTS est conçu comme un modèle de composants où les communications sont réalisées exclusivement par l'intermédiaire d'une seule interface (composée de deux types d'interfaces). L'interface décrit les services fournis et les services requis par le composant. En plus de cette interface, que nous appelons *interface structurelle* car elle ne définit que les signatures des services, CwSTS propose une interface complémentaire décrivant le protocole d'interaction du composant. Cette interface est appelée *interface comportementale* car elle spécifie le comportement du composant en terme des messages envoyés et reçus et de leur ordonnancement logique. Notre modèle est basé sur une vision hiérarchique de la composition des composants. Deux ou plusieurs composants peuvent étre composés dans un composant unique appelé composant composite. Le composant composite adhère aux mêmes règles qu'un composant primitif et en plus il peut être utilisé comme s'il était un composant primitif dans des compositions futures. Ceci correspond à l'idée du patron de conception Composite [Gamma 95].

CwSTS-IDL est le langage de description d'interface (*Interface Description Language* en anglais) que nous proposons avec notre modèle. Il est utilisé pour décrire les interfaces structurelles des composants et aussi les interfaces comportementales. Les interfaces comportementales sont décrites avec un langage formel sous-jacent que nous appelons SFSP (*Symbolic Finite State Processes* en anglais). SFSP est inspiré du langage appelé FSP (*Finite State Processes* en anglais) [Magee 99] mais nous ne retenons que les transitions, le choix et la récursivité. De plus, les actions sont paramétrables avec des types de données quelconques dans SFSP, contrairement à FSP.

1.2.1 Le modèle CwSTS

Le modèle CwSTS est un modèle simple conçu dans le but de pouvoir analyser l'intégration des protocoles d'interaction donnés sous la forme des STS. De plus, nous adoptons une approche générative quant à la construction des composants et nous proposons une implémentation dite *distribuée* (en rapport avec la réalisation du comportement global) des architectures.

Dans ce but nous considérons que dans notre modèle :

- chaque composant expose une seule et unique interface (contenant les informations structurelles et comportementales) ;
- les communications entre les composants sont de type point à point ;
- l'envoi et la réception d'un message se réalisent d'une manière ; synchrone (le client qui envoie le message est bloqué tant que la réception n'a pas eu lieu) mais il n'y a pas de restriction quant à l'exécution proprement dite du message (donc, en fonction de l'implémentation du composant, l'exécution du message peut être synchrone ou asynchrone relatif au composant qui envoie le message) ;
- les architectures sont statiques, c.a.d. qu'il n'y a pas de possibilité de réconfiguration de l'architecture en cours d'exécution ;

Ces hypothèses peuvent paraître limitatives mais nous avons opté pour la simplicité dans notre approche. Néanmoins, certaines extensions de ce modèle pour permettre des communications de groupe, notamment, sont faciles à réaliser (voir la section 7.1 à la page 163 pour les perspectives de ce travail).

1.2.1.1 Les composants CwSTS

La figure 1.1 représente la représentation graphique d'un composant CwSTS. Le composant implémente une interface unique. L'interface est composée de l'ensemble des services que le composant fournit et requiert de son environnement. En plus l'interface inclut un protocole d'interaction (donné sous la forme d'un STS) qui spécifie les règles qui régissent l'ordonnancement des messages que le composant échange avec son environnement.

L'interaction entre deux composants est réalisée par l'envoi d'une requête d'exécution de service de la part du composant qui requiert le service envers le composant qui fournit (et implémente) le service correspondant. D'un point de vue opérationnel, le composant qui requiert un service envoie un message qui va être reçu et finalement exécuté par le composant qui fournit le service. En effet, nous ne spécifions pas si l'exécution effective

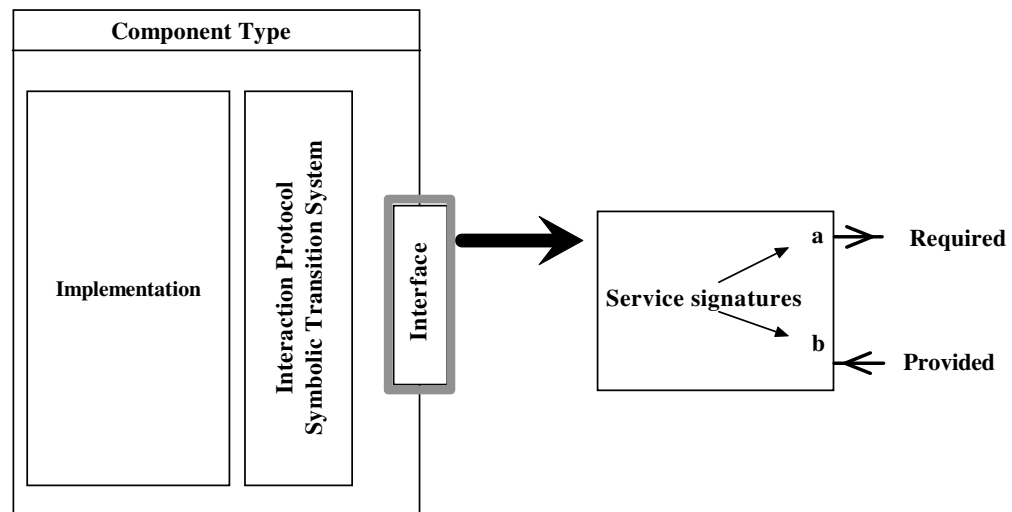


FIGURE 1.1 – Syntaxe graphique d'un composant CwSTS.

du message reçu se fasse en synchrone ou en asynchrone avec l'appel du client. Il peut être exécuté immédiatement après son arrivée (exécution synchrone) et tout en bloquant le client ou ultérieurement à son arrivée (exécution asynchrone) à un moment non spécifié.

Dans le modèle CwSTS, nous parlons des *types de composant* quand nous considérons la définition du composant. L'*implémentation du composant* est représentée par la somme des binaires qui réalisent l'interface spécifiée dans le type du composant. Une *instance du composant* est une instance du type du composant qui est utilisée dans une configuration spécifique à l'exécution.

1.2.1.2 Les interfaces des composants

Comme indiqué dans la figure 1.1, l'interface d'un composant CwSTS décrit les signatures des services qui sont requis ou fournis par le composant (l'interface structurelle). L'interface structurelle correspond au premier niveau des contrats dans la taxonomie proposé par Antoine Beugnard [Beugnard 99]. L'interface comportementale spécifie les instants où un message spécifique peut être reçu par le composant mais aussi les moments où le composant lui-même est susceptible d'envoyer un message. Les protocoles dans le modèle CwSTS correspondent au troisième niveau des contrats de la taxonomie citée ci-dessus.

Les systèmes de transitions symboliques (STS) ont été développés initialement comme une solution aux problèmes d'explosion des états et des transitions dans les algèbres de processus avec passage de valeurs entre les processus. Ce formalisme étend les systèmes de transitions étiquetées avec l'introduction des paramètres et gardes sur les actions des transitions. Nous avons choisi de décrire les protocoles d'interaction au niveau de CwSTS avec une généralisation du formalisme STS. Notre proposition associe un système d'états et transitions symboliques avec une description du type de données qui est représenté par le code d'implémentation du composant. Cette approche n'est pas nouvelle et a prouvé son utilité comme décrit dans [Choppy 00, Royer 03b, Attiogbé 03].

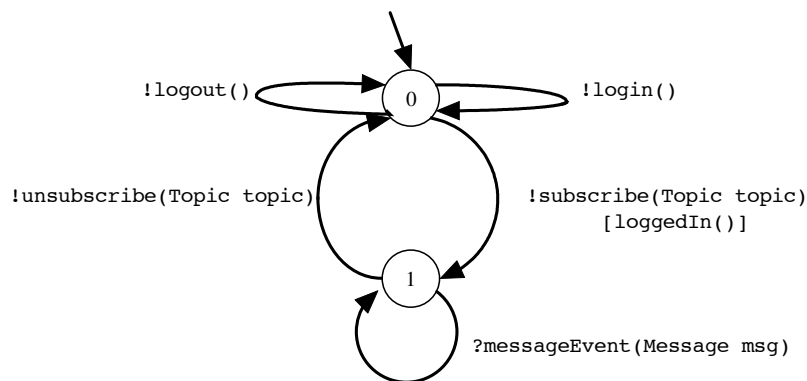


FIGURE 1.2 – Le protocole du composant client du serveur de messagerie.

D'un point de vue opérationnel, l'interface structurelle est une *interface statique*, car les informations qu'elle décrit ne changent pas pendant la phase d'exécution du composant. L'interface comportementale est une *interface dynamique* car elle spécifie les services *disponibles* à des moments précis de l'exécution du composant.

La figure 1.2 présente le protocole d'un composant qui joue le rôle du client d'un serveur de messagerie. Le client va s'identifier auprès du serveur et va s'enregistrer pour recevoir un type spécifique de message. L'interface statique de ce composant est constituée des services requis `login`, `logout`, `subscribe` et `unsubscribe`. Le service fourni `messageEvent` est appelé par le serveur pour signifier l'arrivée d'un message. Le protocole est composé de deux états (les états 0 et 1) qui représentent des états logiques dans l'exécution du composant. Tous les services définis dans l'interface structurelle sont représentés comme des actions forçant la transition du protocole d'un état source à un état destination. La transition qui réalise l'action `subscribe` est gardée par l'évaluation de l'opération booléenne `loggedIn()` qui assure que cette action ne sera pas réalisée tant que la garde n'est pas vraie.

1.2.1.3 La composition

La composition est hiérarchique dans le sens décrit dans le patron de conception *Composite* [Gamma 95]. En composant deux ou plusieurs composants primitifs nous obtenons des composants composites qui à leur tour pourront être utilisés comme des composants dans de futures compositions. Dans notre modèle les composants composites ne contiennent pas de code métier. Ils représentent seulement des agrégats de composants primitifs qui implémentent la logique métier. Ce sont les composants primitifs qui génèrent les messages de sortie ou qui consomment les messages reçus par le composant composite.

La composition est réalisée tant au niveau des interfaces structurelles qu'au niveau des interfaces comportementales. Au niveau structurel, le modèle propose des schémas de composition *binding*, *import* et *export* qui permettent respectivement de connecter deux services correspondants requis/fourni, d'importer un service requis par un sous-composant au niveau de l'interface du composite et d'exporter un service fourni d'un sous-composant au niveau de l'interface du composite. L'interface comportementale du composant compo-

site est représentée par le *produit synchronisé* [Arnold 94, Godefroid 91] des protocoles des sous-composants. Le produit synchronisé est l'automate produit libre qui ne contient que les transitions qui sont autorisées par les règles de synchronisation (et en conformité avec le vecteur de synchronisation). D'une manière générale le produit synchronisé ne contient que les états et les transitions globaux qui sont atteignables dans le cas d'une exécution du protocole du produit. Le vecteur de synchronisation est déterminé par les connexions dans l'architecture (la mise en correspondance des services compatibles conformément à la section 1.2.1.4).

La figure 1.3 présente le composant composite `StatisticsEnabledSever` représentant un serveur de messagerie qui est capable de réaliser des statistiques. Dans cette figure les services fournis et requis du composant composite et les connexions entre sous-composants sont représentées. Le protocole du composant `StatisticsEnabledSever` correspond au produit synchronisé des protocoles de ses sous-composants (leurs protocoles sont représentés dans la figure 1.4 et respectivement dans la figure 1.5 à la page 31).

1.2.1.4 La compatibilité

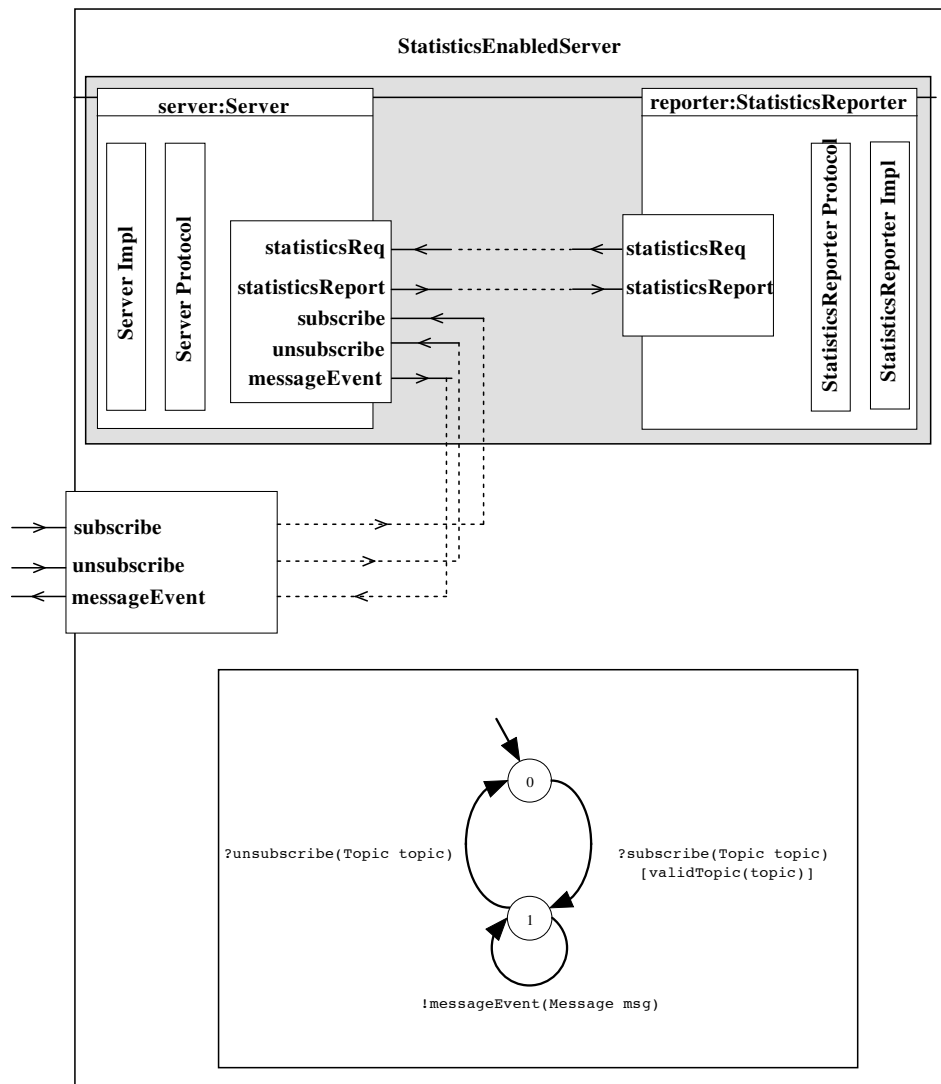
D'un point de vue informel deux ou plusieurs composants sont compatibles dans une architecture si :

1. Les services qui sont mis en relation sont compatibles (compatibilité structurelle).
2. Les protocoles d'interaction des tous les composants impliqués sont compatibles (compatibilité comportementale).
3. L'architecture est valide.

Deux services sont compatibles s'ils ont le même nom et signature. La compatibilité comportementale fait référence à l'absence de blocage dans le produit synchronisé des protocoles des composants impliqués. Une architecture est considérée comme *valide* si toutes les connexions nécessaires à l'exécution de l'application sont réalisées. En effet, le modèle CwSTS ne requière pas que tous les services des composants soient connectés dans une architecture particulière. Certains services peuvent être connectés dans une architecture alors qu'ils ne le sont pas dans une autre architecture. Le fait qu'il n'y ait pas de dépendance du contexte environnemental est d'une importance majeure au moment de la construction des composants, car ceci assure le fait qu'un composant peut être assemblé et déployé dans plusieurs architectures et environnements différents. Une architecture exécutable est une architecture qui ne requiert pas de services externes, et qui est donc *complète*.

1.2.1.5 Adaptation des composants

Dans le cas où deux ou plusieurs composants ne sont pas compatibles dans une architecture, le modèle CwSTS propose l'utilisation de composants d'adaptation que nous appelons des *adaptateurs*. Au niveau des interfaces structurelles les adaptateurs permettent de changer les noms et signatures des services (ils implémentent le patron de conception *Adapter* [Gamma 95]). Au niveau des comportements, l'adaptateur peut être introduit pour éliminer le blocage qui apparaît au niveau du produit synchronisé. La figure 1.7 présente

FIGURE 1.3 – Le composant **StatisticsEnabledServer**.

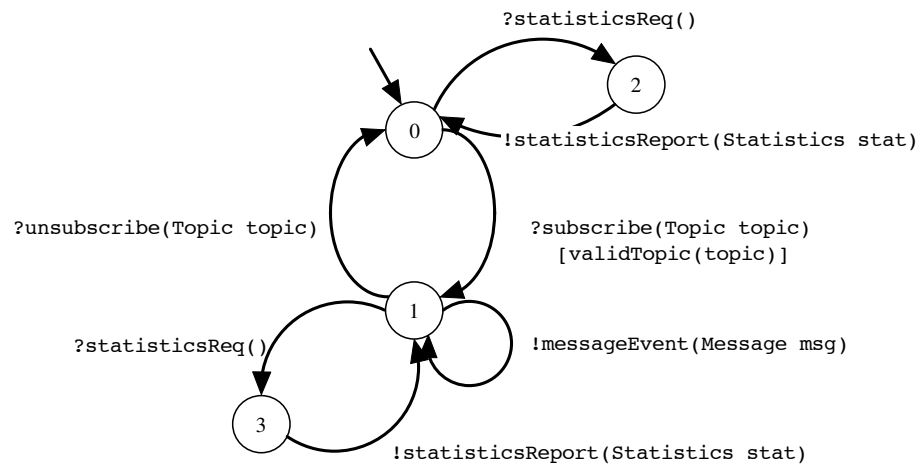


FIGURE 1.4 – Le protocole du composant serveur de messagerie.

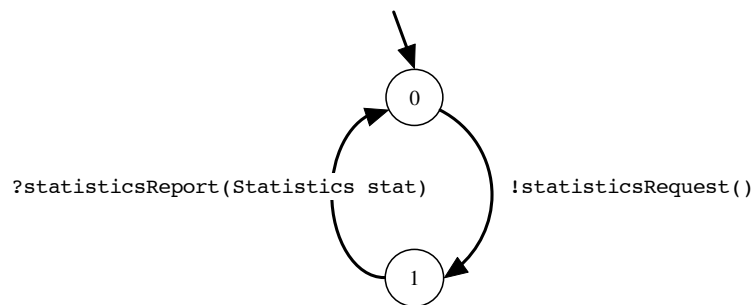


FIGURE 1.5 – Le protocole du composant StatisticsReporter.

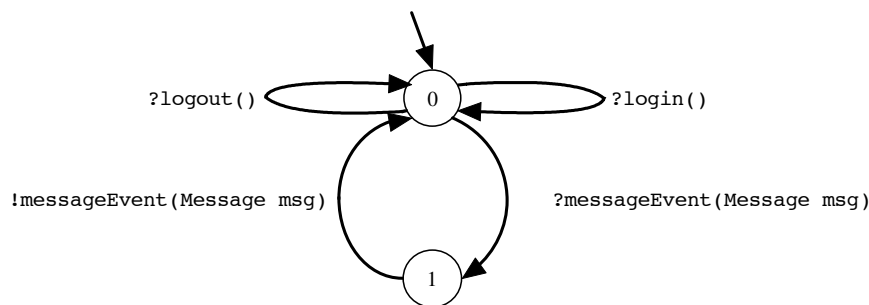


FIGURE 1.6 – Le protocole du composant adaptateur.

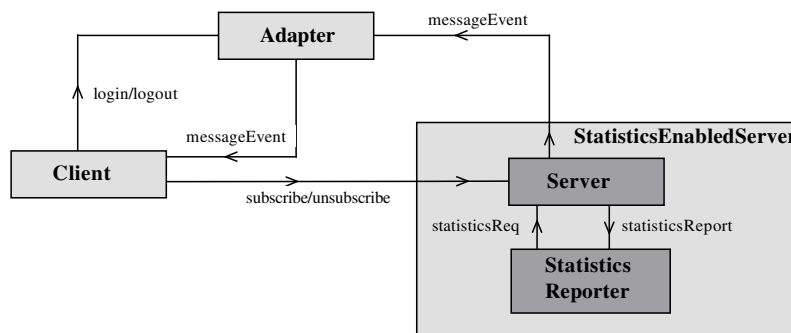


FIGURE 1.7 – Le rôle du composant adaptateur dans l’architecture du système de messagerie.

le cas où un composant adaptateur doit être utilisé pour connecter le client du système de messagerie au serveur de messagerie dû au fait que les interfaces des deux composants ne sont pas compatibles. Le composant client implémente le fait qu’il faut se connecter et se déconnecter avant de souscrire et recevoir des messages de la part du serveur qui lui ne requiert pas ces étapes. Le protocole du composant adaptateur est présenté dans la figure 1.6 et décrit le fait que le client peut faire `login/logout` sans que ces messages arrivent au serveur. Les messages de type `messageEvent` sont transférés du serveur vers le client.

L’adaptateur n’as pas besoin d’intercepter toutes les messages échangés par les composants. Il intercepte que ce que doit être adapté. Dans cet exemple, l’adaptateur doit seulement faire en sorte que les messages `login/logout` soit absorbés sans que le serveur les reçoivent. Évidemment, les messages `subscribe/unsubscribe` pourront aussi passer par l’adaptateur mais ceci ne servirait à rien dans ce cas. Avec les adaptateurs nous fournissons un mécanisme d’adaptation très configurable et adaptable aux différentes cas d’utilisation.

1.2.1.6 Substitution des composants

La substitution définit une notion de raffinement sur le type des entités considéré (fonctions, objets ou composants). Les langages de programmation à objets tels que Java

$$\begin{array}{c}
 f (\quad T_1 \quad) : R_1 \\
 \text{IA} \quad \text{VI} \\
 g (\quad T_2 \quad) : R_2
 \end{array}$$

FIGURE 1.8 – Raffinement de la fonction f par la fonction g .

ont déjà abordé ce problème en introduisant une relation de sous-typage (l'héritage). La substitution des composants est plus difficile à réaliser car elle doit considérer les services requis et fournis et aussi les protocoles d'interaction.

Dans le modèle CwSTS, l'intuition que nous avons est que nous pouvons faire une analogie entre la substitution d'un composant et la substitution d'une fonction. En effet, si nous considérons l'ensemble des signatures des services fournis par le composant comme son type d'entrée et l'ensemble des signatures des services requises comme son type de sortie, le parallèle avec le raffinement d'une fonction (comme présenté dans la figure 1.8 où les notions de variance et covariance sont représentées explicitement) est immédiat. Un composant peut remplacer un autre composant au niveau des interfaces structurelles s'il fournit au moins ce que le composant remplacé fournit. De plus, le composant remplaçant requiert au plus ce que le composant remplacé requiert. Nous nous retrouvons dans le cas classique des relations de covariance/contravariance entre les types des deux composants.

Le raffinement comportemental implique le fait que si nous considérons une sémantique des traces associée aux protocoles STS, le protocole du composant remplaçant doit accepter au moins les mêmes traces (envois et réceptions des messages) que le composant remplacé. De plus, le composant remplaçant doit refuser les mêmes requêtes que le composant remplacé. Nierstratz [Nierstrasz 93] définit ce type de substitution en fonction des traces et défaillances. Alors que les traces définissent l'ensemble des actions à exécuter, les défaillances définissent les requêtes qu'un composant ne peut pas accepter après qu'une trace spécifique ait été exécutée.

1.2.1.7 Le langage de description de composants CwSTS-IDL

Nous avons développé un langage de description des interfaces des composants CwSTS. Ce langage permet la description des interfaces structurelles et comportementales. La syntaxe abstraite de ce langage est présentée dans la figure 1.9. Le protocole est défini en utilisant un langage de processus appelé SFSP (*Symbolic Finite State Processes*). SFSP est inspiré de FSP (*Finite State Processes*) [Magee 99] mais retient seulement les transitions, le choix et la récursivité. De plus les actions sont paramétrables avec des types de données quelconques. La syntaxe abstraite du langage SFSP est présentée dans la figure 1.10 à la page 34. À titre d'exemple, la figure 1.11 à la page 35 présente la description structurelle et celle comportementale, du composant client du système de messagerie.

1.2.2 L'implémentation du modèle

CwSTS-P est l'implémentation préliminaire de notre modèle dans le langage de programmation à objets Java. Le rôle de cette implémentation est de permettre une validation

```

component_def      ::= primitive_def | composite_def
primitive_def      ::= component_type
                    primitive_struct_def
                    guards_def
                    primitive_protocol
primitive_struct_def ::= service_def+
service_def        ::= service_type op_id formal_parameter_list? | op_id formal_parameter_list?
guards_def         ::= op_id formal_parameter_list?
service_type       ::= 'required' | 'provided'
composite_def      ::= component_type
                    subcomponent_decl+
                    composite_structural_def?
                    connection_def+
subcomponent_decl  ::= component_type subcomponent_id
composite_struct_def ::= service_def+
connection_def     ::= bind_exp | export_exp | import_exp
bind_exp           ::= subcomponent_op_id 'to' subcomponent_op_id
export_exp         ::= subcomponent_op_id 'as' op_id
import_exp         ::= op_id 'to' subcomponent_op_id

subcomponent_op_id ::= subcomponent_id '.' op_id

```

FIGURE 1.9 – Grammaire du langage de description des interfaces CwSTS-IDL.

```

specification      ::= process_def+
process_def        ::= process_id formal_param_list process_body

process_body       ::= choice+
choice             ::= action+ process_inst
process_inst       ::= process_id actual_param_list | special_process

special_process    ::= STOP

action            ::= receive_act | send_act | internal_act
receive_act       ::= action_id formal_param_list
send_act          ::= action_id actual_param_list
internal_act      ::= action_id actual_param_list

formal_param_list  ::= param_type param_id | formal_param_list | ε
actual_param_list  ::= param_id | actual_param_list | ε

```

FIGURE 1.10 – La grammaire abstraite du langage SFSP.

```

primitive component Client {
  interface
    provided messageEvent(Message msg);
    required login();
    required logout();
    required subscribe(Topic topic);
    required unsubscribe();

  guards
    loggedIn();

  protocol
    P=!login -> P | !logout[!loggedIn()] -> P | !subscribe[!loggedIn()] -> Q,
    Q=?messageEvent -> Q | !unsubscribe() -> P.
}

```

FIGURE 1.11 – La définition du composant client du serveur de messagerie.

des notions conceptuelles décrites dans le modèle de composants CwSTS et de fournir des outils de génération de code, d'analyse des propriétés des architectures (validité, compatibilité) et d'adaptation des composants à l'origine non compatibles tel que décrit dans le modèle CwSTS.

Nous nous basons sur une approche générative dans la construction des composants CwSTS. La figure 1.12 présente les deux scénarios que nous considérons :

- **Scénario A.** Nous partons d'une ou plusieurs descriptions de composants (descriptions des interfaces structurelles et comportementales) qui, avec des instructions d'instrumentation, sont passées à un moteur de génération de code. Le résultat est un composant CwSTS valide pour chaque description de composant fournie en entrée. Cette implémentation représente soit une implémentation par défaut du code métier du composant soit une implémentation orientée code d'adaptation (code d'implémentation du composant adaptateur). Le code métier par défaut ne contient pas de comportement particulier et est préparé pour être personnalisé afin de répondre à des besoins spécifiques. La construction du code personnalisé doit ensuite utiliser un certain nombre de règles simples afin de garantir la conformité de l'implémentation vis-à-vis de la spécification.
Les *composants adaptateurs* générés sont utilisés dans une architecture qui requiert l'adaptation structurelle et/ou comportementale des composants la constituant. Le code des adaptateurs n'a pas besoin d'être personnalisé pour répondre aux besoins pour lesquels il a été généré.

- **Scénario B.**

Ce scénario est différent du premier scénario par le fait qu'au moment de la génération, nous fournissons aussi le code représentant le code métier du composant généré. Nous partons donc du code qui a besoin d'être instrumenté pour obtenir

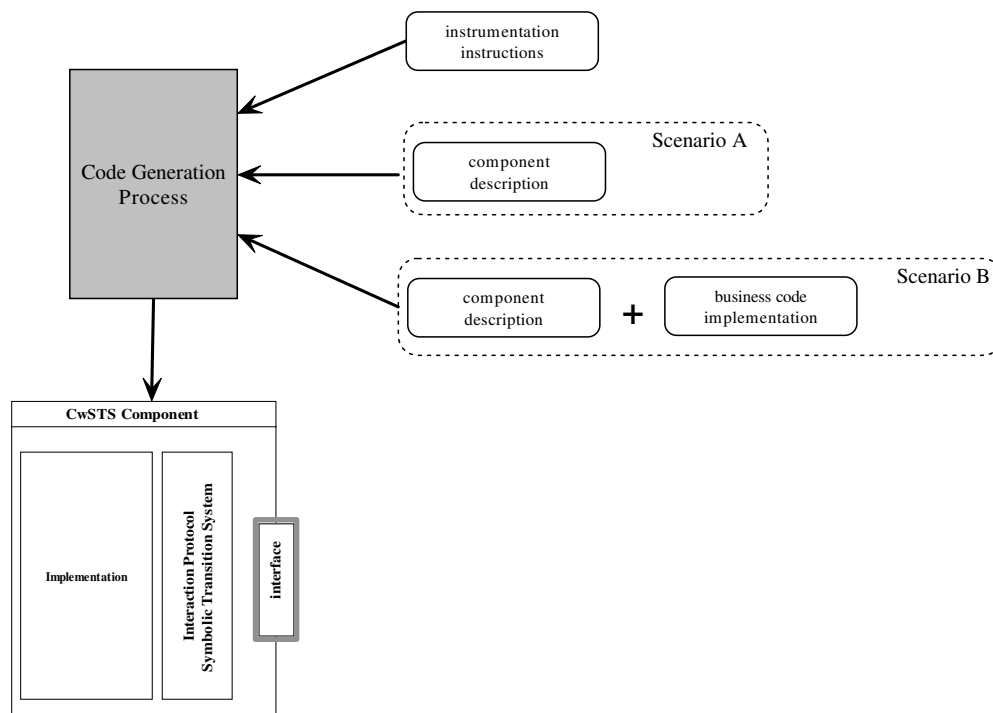


FIGURE 1.12 – Scénarios de génération de composants.

un composant conforme au modèle CwSTS. Dans ce scénario, la description du composant doit être *conforme* au code fourni, et ceci aux niveaux structurel et comportemental.

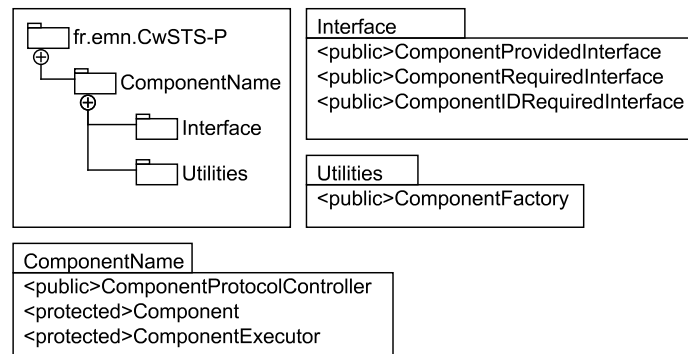
Notre approche générative répond au besoin d’assurer que l’implémentation est *conforme* aux spécifications. En suivant n’importe lequel des deux scénarios présentés ci-dessus, nous assurons *par construction* que le code d’implémentation du composant CwSTS est conforme tant en terme de structure que de comportement aux spécifications.

Dans ce qui suit nous allons présenter des points significatifs de l’implémentation des composants : l’encapsulation du code Java pour répondre aux exigences du modèle CwSTS, la structure d’un composant (primitif ou composite) et l’implémentation du contrôle du comportement d’un composant.

1.2.2.1 Structure des composants

Dans le prototype CwSTS-P, chaque composant est structuré sous la forme d’un *package* de classes Java classique. Ce *package* contient les classes d’implémentation du composant. La figure 1.13 présente la structure du *package* où les classes et les interfaces constituant le code du composant sont structurées dans des sous-*packages*.

Le nom du *package* est le nom du composant pendant la phase de développement. Le nom est conforme au standard proposé qui est : `fr.emn.CwSTS-P.NomDuComposant`. La

FIGURE 1.13 – Structuration d’un composant sous la forme d’un *package* Java.

racine de ce package contient les classes qui seront instanciées pour obtenir une instance de composant exécutable. Les deux sous-packages *Interface* et *Utilities* contiennent les interfaces déclarées du composant et une classe *factory*. Les interfaces définissent les services fournis et requis par le composant et la *factory* est utilisée pour obtenir de nouvelles instances du composant de ce type.

Notre modèle de composants définit des composants de type boîte noire. Ceci signifie que la structure interne du composant n’est pas visible de l’extérieur du composant à l’exécution. Le seul moyen d’interagir avec le composant est d’utiliser ses interfaces. L’encapsulation du code d’exécution du composant sous la forme d’un package Java va dans la direction de répondre à cette contrainte. Mais le package Java tout seul ne suffit pas. Afin de parfaitement isoler les classes du package de l’environnement d’exécution du composant, les seules entités publiques du package sont les interfaces, le contrôleur de protocole (qui représente le *front-end* du composant) et le *factory* correspondant au composant.

1.2.2.2 Implémentation d’un composant primitif

Un composant primitif CwSTS-P est constitué de trois types d’entités : une ou plusieurs classes contenant le *code métier* du composant, un *exécuteur* du composant et une classe qui joue le rôle de *contrôleur* du protocole implémenté par le composant. La figure 1.14 présente une vue fonctionnelle de la structure du composant.

Le code métier

Le code métier (*business logic* en anglais), est le code qui réalise la fonctionnalité du composant. Il implémente les interfaces fournies par le composant et est donc capable d’exécuter les messages reçus par le composant. C’est aussi cette entité qui génère des messages à destination d’autres composants.

L’exécuteur

L’exécuteur du composant fonctionne comme un mandataire (*proxy* en anglais) au nom du code métier. Son rôle principal est d’intercepter les fils d’exécution (*threads* en anglais) qui appellent les méthodes du code métier et d’exécuter ces appels mais dans des fils

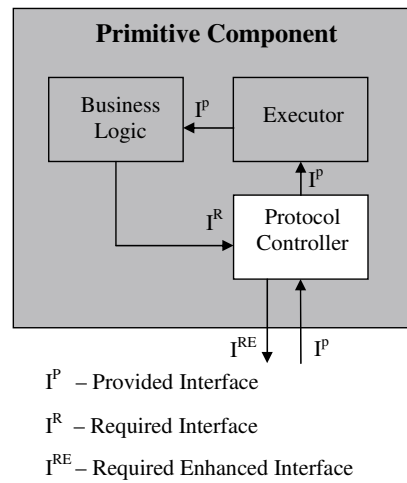


FIGURE 1.14 – Vue fonctionnelle sur la structure d'un composants primitif.

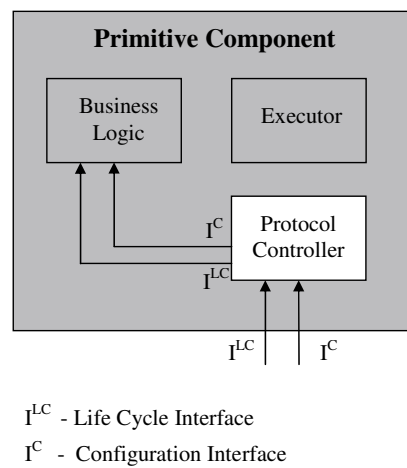


FIGURE 1.15 – Vue de configuration sur la structure d'un composant primitif.

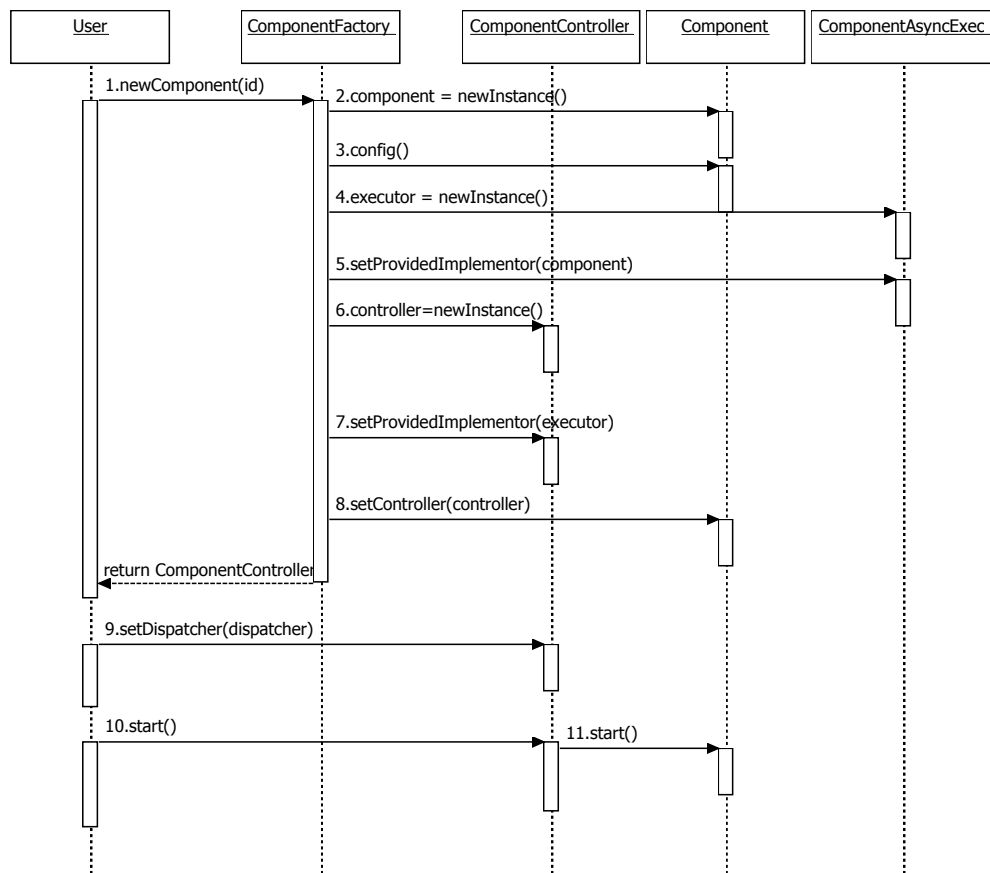


FIGURE 1.16 – Création d’une instance d’un composant primitif.

différents de ceux qui contiennent l’appel d’origine. Ceci est utile pour implémenter les spécifications du modèle CwSTS qui stipule que la réception d’un message est toujours synchrone avec son envoi par un autre composant mais que l’exécution proprement dite du message (de l’appel encapsulant le message) est soit synchrone soit asynchrone avec l’envoi. En effet, le fil d’exécution démarré par l’exécuteur découple l’appel de l’exécution du message.

Le contrôleur

Le contrôleur du protocole du composant contrôle les appels qui entrent et qui sortent du composant. Le protocole implémenté par le composant spécifie la séquence des messages que le composant doit recevoir ou émettre. Le contrôleur assure que l’ordre des appels encapsulant les messages va être conforme au protocole. Le contrôleur implémente l’interface fourni par le composant et l’interface requise augmentée d’une information qui est l’identifiant du composant lui même. Ce mécanisme est nécessaire pour permettre la composition du composant dans une architecture (voir les explications sur le répartiteur à la page 41).

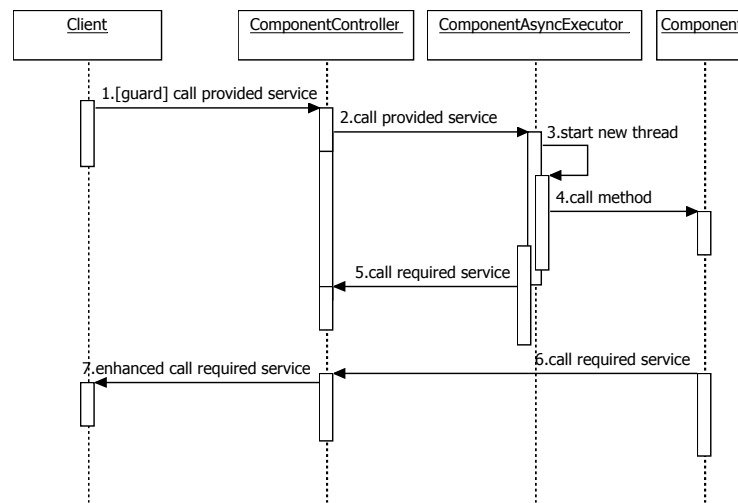


FIGURE 1.17 – Interaction des entités du composant primitif.

La configuration, l'instanciation et l'exécution des composants

La figure 1.15 présente la vue de configuration sur les relations entre les différentes entités d'un composant primitif. Plusieurs phases ont lieu avant l'exécution proprement dite du composant (voir la figure 1.16). Les instances Java du code métier (*Component* dans la figure), du code exécuteur (*ComponentAsyncExec* dans la figure) et du contrôleur (*ComponentController* dans la figure) sont créées. Après une phase de paramétrage des trois entités, la référence au contrôleur du protocole est retournée au client ayant demandé l'instanciation du composant. Le client peut aussi configurer des informations environnementales avant de démarrer l'exécution proprement dite du composant en appelant la méthode `start()`, appel qui sera propagée au code métier du composant.

Une fois démarré, le composant se comporte comme indiqué dans la figure 1.17. La garde sur la ligne 1 du diagramme de séquence indique la présence d'un mécanisme qui en utilisant la notion de *monitor* Java, permet la synchronisation de l'appel du client avec la réception au niveau du *ComponentController*. Dans le cas d'un appel émis par le composant lui-même (appel provenant du code métier), le contrôleur du composant transfère l'appel après une phase de *décoration* de l'appel avec des informations d'identification du composant. Ces informations sont utilisées par le composant composite (comme présenté ci-dessus) pour diriger l'appel vers son composant destinataire (et ceci conformément aux liens structurels définis dans l'architecture).

1.2.2.3 Implémentation d'un composant composite

Conformément aux spécifications du modèle CwSTS, le composant composite ne contient pas de code métier : la fonctionnalité du composite est réalisée à l'exécution par l'interaction entre les sous-composants qui le composent. De plus, le composant composite comporte un contrôleur (similaire au contrôleur des composants primitifs) et une entité répartiteur (*dispatcher* en anglais). Le répartiteur implémente une stratégie de transfert

d'appels qui réalise à l'exécution les relations structurelles entre les sous-composants du composant composite. La figure 1.18 présente la structure d'un composant composite.

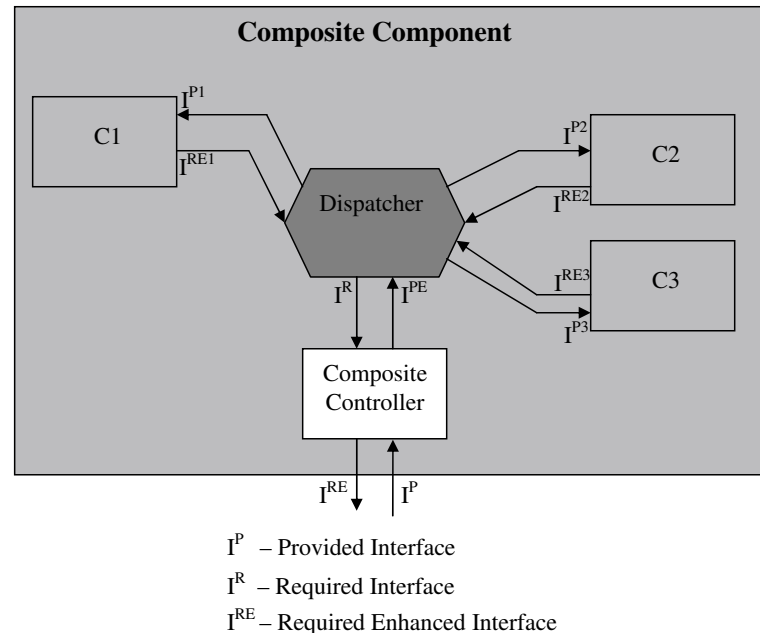


FIGURE 1.18 – Vue fonctionnelle sur la structure d'un composant composite.

Sous-composant

Un composant composite contient plusieurs sous-composants. Les sous-composants sont soit des composants primitifs soit des composants composites (car la composition est hiérarchique dans CwSTS). Les sous-composants ne sont pas directement interconnectés dans un composite. Ils sont tous connectés à l'entité répartiteur.

Le répartiteur

Le répartiteur est responsable du transfert des appels qui proviennent des sous-composants vers les composants spécifiés dans l'architecture du composite. Le répartiteur est basé sur une table de routage qui représente à l'exécution les relations structurelles entre les sous-composants du composant composite. Le répartiteur ne contient pas de code de synchronisation Java et donc aucun blocage n'est possible à son niveau. La synchronisation effective entre les composants se réalise au niveau des sous-composants qui contrôlent localement les réceptions et les émissions des appels. Pour réaliser le routage des appels, le répartiteur doit connaître l'identité des composants qui sont la source des appels. Comme spécifié dans la figure 1.18, le répartiteur implémente toutes les interfaces requises des sous-composants du composant composite. De plus, ces interfaces sont personnalisées avec un paramètre qui est l'identificateur du sous-composant à l'origine de l'appel.

Le contrôleur

Le contrôleur est le *front-end* du composant composite. Tous les appels qui sortent ou qui entrent dans le composant composite sont interceptés par cette entité. Dans ce but, le contrôleur implémente les interfaces fournies et requises par le composant composite. À l'opposé du contrôleur d'un composant primitif, le contrôleur du composite ne réalise pas de vérification du protocole. En effet, le contrôleur ne connaît même pas le protocole du composite, car celui-ci est réalisé à l'exécution par les interactions entre les sous-composants qui réalisent le produit synchronisé de leurs protocoles. C'est le contrôle réalisé au niveau de chaque composant primitif qui permet la réalisation des synchronisations nécessaires au bon comportement du composant composite.

La configuration, l'instanciation et l'exécution des composants

La configuration, l'instanciation et l'exécution des composants composites sont réalisées en utilisant les relations statiques présentées dans la figure 1.19.

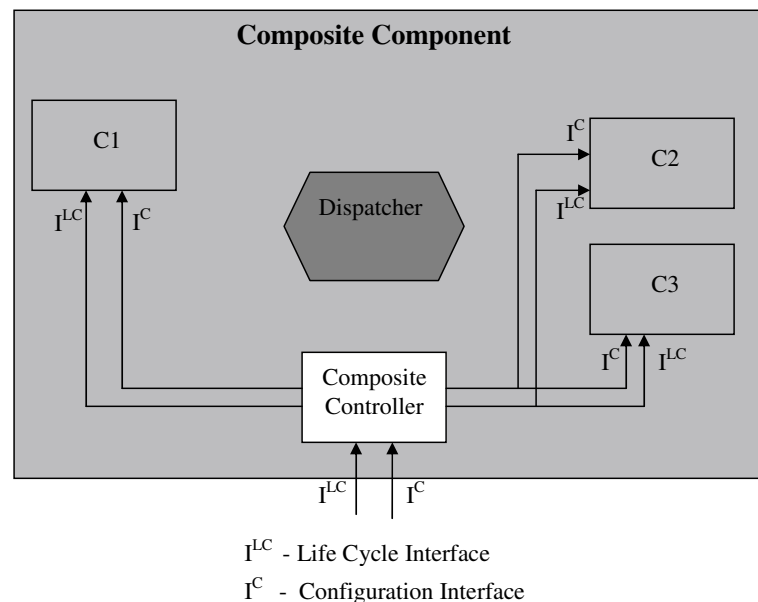


FIGURE 1.19 – Vue de configuration sur la structure d'un composant composite.

La phase de configuration a lieu entre les actions de création d'instances de sous-composants et de démarrage d'exécution (voir figure 1.20). Le client d'un composant composite configure les données nécessaires au composite pour s'exécuter. Le contrôleur du composite se charge de transmettre ces informations au niveau des sous-composants. Habituellement, ces informations contiennent les valeurs de paramètres d'initialisation nécessaires aux composants. D'autres informations qui sont complètement transparentes pour les clients du composite incluent la référence du répartiteur qui est transmise aux sous-composants et les relations structurelles entre les sous-composants. Ces relations structurelles sont codées dans la table de routage au niveau du répartiteur.

Une fois la phase de configuration terminée et les composants démarrés les appels qui arrivent au niveau du composite sont interceptés par le contrôleur qui les transfère au niveau du répartiteur. En fonction de l'identité de l'appelant (les sous-composants ou le contrôleur), le répartiteur route les appels vers leur destinataire. Les appels émis par le composant sont traités de la même manière.

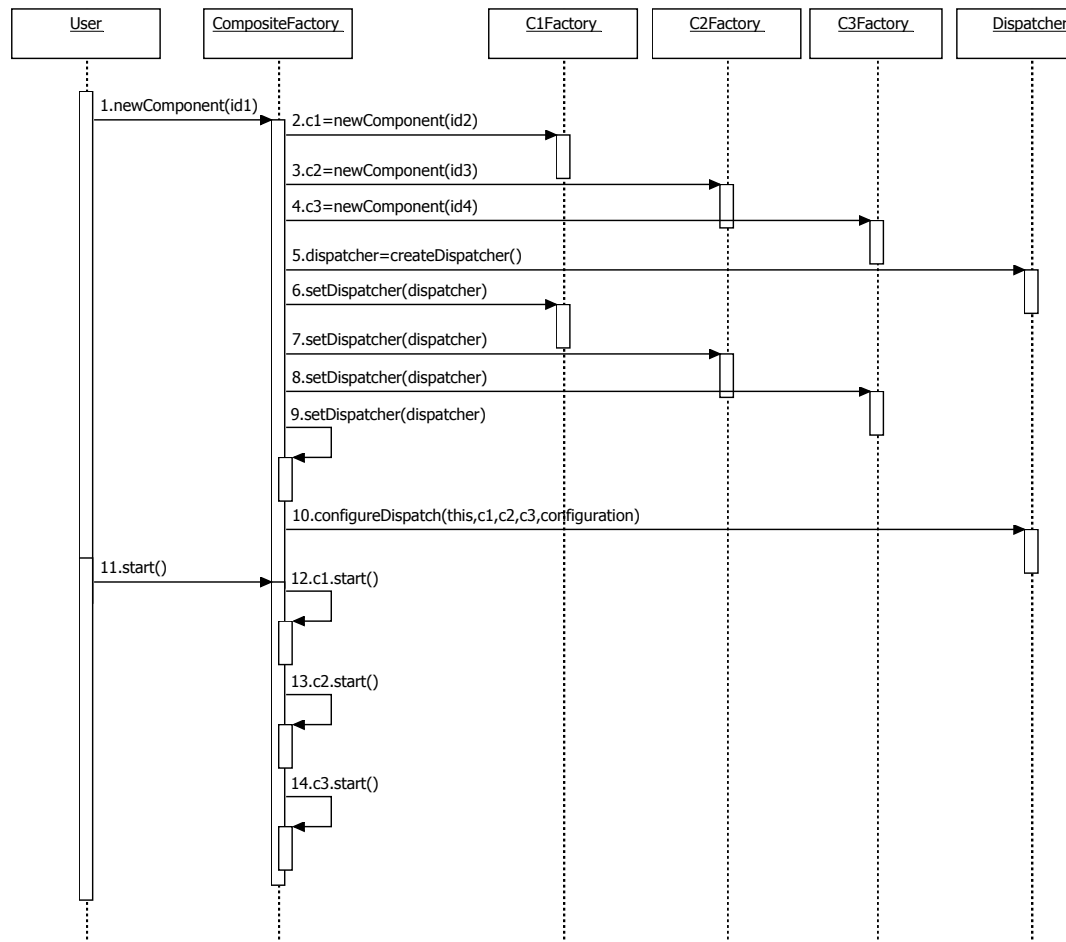


FIGURE 1.20 – Création d'une instance d'un composant composite.

1.2.2.4 La composition comportementale

Comme indiqué auparavant, un composant composite n'implémente pas le protocole représentant le produit synchronisé de ses sous-protocoles. Le produit synchronisé n'est donc pas contrôlé par le contrôleur du composite. C'est l'exécution des sous-composants qui réalise le produit synchronisé global de leurs protocoles respectifs.

Ceci représente une solution dite *distribuée* de contrôle du protocole. Les synchronisations entre l'envoi et la réception des messages encapsulés sous la forme des appels d'un composant envers un autre composant se réalisent au niveau de chaque composant (au

niveau du contrôleur des composants primitifs plus exactement). L'envoi et la réception du message sont exécutés d'une manière *synchrone*, c.à.d. que le composant émetteur est bloqué une fois l'émission démarrée et tant que le composant receveur n'a pas reçu le message. Une fois la réception acquittée, le composant receveur débloquent l'émetteur et procède à l'exécution immédiate ou différée du message reçu. C'est dans ce mécanisme que réside l'implémentation de la synchronisation nécessaire à la réalisation du produit synchronisé des composants qui interagissent. Un composant émetteur va émettre un message uniquement si les conditions spécifiées par la garde indiquée dans son protocole d'interaction sont vérifiées. Sinon, aucun message n'est émis. Un composant receveur peut recevoir le message uniquement si, par rapport à son protocole d'interaction, il se trouve dans un état où il peut recevoir ce message. Sinon, le message est mis en attente de réception, ce qui bloque son émetteur, qui n'est pas autorisé à exécuter d'autres actions pendant toute la période de blocage. Au moment où le composant receveur se trouve dans un état permettant la réception du message bloqué, la réception a lieu et l'émetteur est débloquent. Même si le protocole du receveur est dans un état où il peut recevoir un message particulier, la garde spécifique à la transition doit être vraie. Si ce n'est pas le cas, le message n'est pas reçu et mis en attente. Le receveur n'est jamais bloqué par un message en attente d'être reçu ce qui fait que d'autres messages reçus peuvent changer l'état interne du composant en permettant à une nouvelle évaluation des gardes de débloquent un message en attente de réception dûe aux évaluations des gardes. L'implémentation est non déterministe dans le sens où si deux ou plusieurs gardes deviennent franchissables le message reçu n'est pas déterminé d'une manière statique. Ceci dépend des mécanismes de contrôle des fils d'exécution au niveau de la machine virtuelle Java.

Ce mécanisme a prouvé son efficacité dans l'implémentation du produit synchronisé, sans avoir à recourir à un mécanisme dit *centralisé* ou à une entité spécifique (comme par exemple un contrôleur du protocole global d'une application) qui fera le contrôle de chaque paire d'actions émission/réception. Une approche centralisée pose des problèmes importants tant en terme d'efficacité que de passage à l'échelle car complètement contre-indiquée dans un environnement réparti.

Pour l'implémentation effective de notre mécanisme *distribué* de synchronisation nous avons recours au mécanisme de *synchronisation* du langage Java ou les fils d'exécution des applications Java sont instrumentés pour arriver à une exécution chorégraphiée. Le prototype CwSTS-P étant implémenté en utilisant la version Java 5, nous avons utilisé des structures de synchronisation très optimisées comme par exemple les files d'attente, les verrous explicites (les *locks* en anglais), etc pour réaliser la synchronisation des fils d'exécution qui encapsulent les appels d'un composant à un autre.

1.2.3 Perspectives

Nous avons opté pour une approche générative en ce qui concerne la construction des composants. Ceci est en phase avec l'approche MDA actuellement en pleine reconnaissance dans le monde du développement des applications logicielles. La génération permet de construire plus rapidement des applications plus sûres et peut être intégrée avec des

techniques de vérification et d'analyse sophistiquées. Les outils proposés avec l'implémentation du modèle CwSTS représentent les briques de base pour pouvoir réaliser des logiciels de génération de code qui intègrent des protocoles d'interaction sous la forme de systèmes de transitions symboliques.

Le modèle actuel prend en charge uniquement des configurations statiques. Une fois les composants de l'architecture configurés et démarrés, aucune modification structurelle ou comportementale des composants n'est plus possible. La tendance actuelle dans le développement des logiciels est de fournir aux applications la possibilité de se reconfigurer rapidement *à chaud*. Ceci signifie qu'une application est capable de se reconfigurer tout en continuant son exécution, ou du moins sans avoir à tout recompiler et redéployer. Une des possibles évolutions du modèle CwSTS est donc de permettre aux composants de se reconfigurer (surtout en termes de connexions à l'intérieur d'un composite) sans avoir à arrêter complètement l'exécution de l'application. Ceci sera possible en concevant des composants qui fonctionnent suivant plusieurs phases (*démarré, en configuration, en pause, en arrêt*, par exemple) après le démarrage de l'application.

Pour des raisons de simplicité, nous n'avons considéré que le cas où les communications entre les composants sont point à point. Ceci signifie que le service fourni par un composant ne peut être appelé qu'à partir d'un seul et unique autre composant pendant toute la vie de l'application. Dans la réalité des applications logicielles d'aujourd'hui, la communication point à point est très restrictive surtout si nous considérons des applications client-serveur où un seul composant serveur peut recevoir des appels d'exécution de services en provenance de plusieurs clients. D'autres schémas d'interaction peuvent être considérés par une extension du modèle CwSTS. Ces schémas peuvent inclure la communication n à 1 ou 1 à n permettant de concevoir des architectures complexes en termes d'interaction entre les composants. Le mécanisme de synchronisation permettant la réalisation du produit synchronisé est, en partie, équipé pour faire face à de tels schémas de communication mais des études complémentaires et leurs implémentations sont à prévoir.

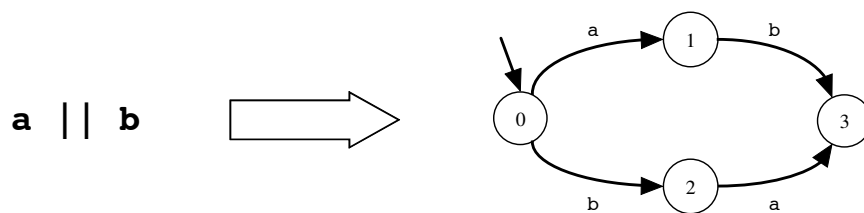


FIGURE 1.21 – Exécution parallèle de deux actions.

Le langage SFSP (basée sur le langage FSP) est un langage de processus qui décrit les protocoles d'interactions des composants. Le langage est très précis dans la description des transitions des protocoles mais pour le cas où nous voulons décrire deux transitions qui peuvent s'exécuter en parallèle, nous sommes obligés de transcrire cette situation comme présenté dans la figure 1.21. Les langages FSP et SFSP décrivent les actions parallèles en les représentant d'une manière séquentielle plutôt qu'en restant *indépendantes* et en faisant abstraction de la façon dont l'exécution des actions aura lieu. Une possible évolution du langage SFSP sera de considérer ce cas précis de description des transitions en parallèle

qui fera abstraction de la manière exacte de l'exécution pourvu que le comportement décrit soit réalisé. Dans un environnement distribué, l'exécution des actions se réalise rarement d'une manière séquentielle ce qui indique une fois de plus l'utilité d'une telle extension.

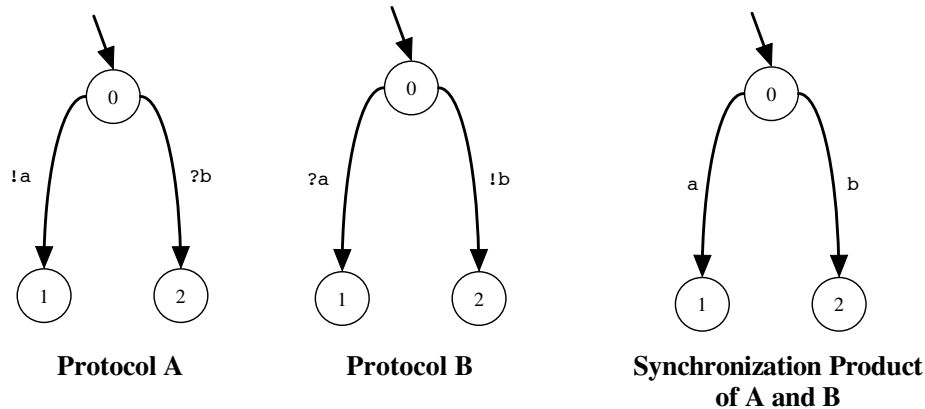


FIGURE 1.22 – États mixtes dans l'interaction entre deux composants.

En ce qui concerne l'implémentation du modèle et plus particulièrement l'implémentation de la réalisation du produit synchronisé à l'exécution, le cas des *états mixtes* [Yellin 97] n'est pas complètement pris en compte. La figure 1.22 présente un exemple de ce cas. Si deux composants qui communiquent entre eux se trouvent chacun dans un état où ils peuvent tous les deux émettre ou recevoir un message, notre implémentation ne garantit pas qu'il n'y aura pas d'interblocage applicatif (dû au fait que les deux composants essaient d'envoyer un message « au même moment »). Comme l'implémentation actuelle bloque l'émetteur tant que le receveur n'a pas reçu le message, chacun des deux composants peut être bloqué en attendant que son partenaire de communication reçoive son message. Une évolution de l'implémentation devrait donc éliminer cette limitation qui, en pratique, est généralement imprévisible et dépend exclusivement des entrelacements des actions des composants. Ceci pourra être basé sur l'intégration de la synchronisation distribué avec une forme de contrôle centralisé (basé sur l'idée de moniteur centralisé présenté dans la section 6.2).

Toujours en rapport avec SFSP, le langage FSP propose aujourd'hui des outils de visualisation et de vérification des propriétés comme par exemple l'interblocage. Une évolution future de l'implémentation de notre modèle pourrait proposer des outils similaires qui permettront une meilleure description et vérification des protocoles d'interaction.

1.3 Structure du mémoire de thèse

Le corps de ce mémoire en anglais est structuré en deux parties. La première partie présente le contexte de notre travail. La deuxième partie de ce document présente la contribution de notre travail : un modèle de composants intégrant un formalisme lisible, expressif et compact basé sur des systèmes de transitions symboliques.

La première partie présente le contexte de notre travail. Dans la section 3.1 nous présentons l'état de l'art des caractéristiques des modèles de composants actuels et nous l'illustrons avec des propositions du domaine de la recherche et de l'industrie. Les langages de description d'architectures [Medvidovic 00b] sont présentés dans la section 3.3.4. Un ADL est défini comme une notation formelle ou informelle, textuelle ou graphique qui permet la spécification des architectures logicielles et qui est accompagnée d'outils spécifiques comme présenté dans [Medvidovic 00a]. Le résultat de l'utilisation d'un langage de description d'architecture est une vision abstraite en termes de composants, connecteurs et configurations sur l'application.

La plupart du temps, la vision concrète de l'implémentation est ignorée dans une approche de type langage de description d'architecture. L'analyse architecturale réalisée avec un ADL peut révéler des propriétés importantes, mais celles-ci ne sont pas en général garanties par l'implémentation. Pour permettre une analyse architecturale au niveau du code, le code doit être conforme à l'implémentation. L'approche de la programmation à base de composants (COP pour *Component-Oriented Programming*) dans la section 3.3.6) propose de développer des logiciels en les programmant directement avec des composants plutôt que des objets.

Le paradigme de coordination est présenté dans la section 3.3.7. La coordination est une approche académique centrée sur la collaboration entre processus. Cette approche considère la programmation des systèmes distribués ou parallèles comme la combinaison de deux activités distinctes : l'activité de calcul proprement dite, qui contient un nombre des processus impliqués dans la manipulation des données et l'activité de coordination qui est responsable de la communication et de la coopération des processus.

Après cela, nous considérons l'état de l'art dans le domaine de la spécification des protocoles d'interaction. Différents formalismes et modèles de composants intégrant ces formalismes sont présentés. Les algèbres de processus sont présentées dans la section 4.2.1. Le terme algèbre de processus fait référence à une famille de techniques de spécification particulièrement adaptées à la description des systèmes concurrents. Le processus d'interaction est décrit en termes de règles de calcul sur des expressions représentant des processus.

La section 4.2.2 et la section 4.2.3 présentent les types comportementaux et les formalismes basées sur les machines à états finis. Les logiques temporelles sont présentées en dernier dans la section 4.2.4, pour clore cette partie sur les formalismes de description des protocoles d'interaction.

La deuxième partie de ce document présente la contribution de notre travail : un modèle de composants intégrant un formalisme lisible, expressif et compact basé sur des systèmes de transitions symboliques.

Nous commençons par décrire notre modèle de composants d'un point de vue informel dans la section 5.2. Les composants, leurs interfaces, les règles de composition et les détails sur les protocoles STS sont illustrés avec un exemple qui nous sert de fil conducteur.

Les notions de composant, d'interface structurelle et comportementale, ainsi que la compatibilité et la substitution des composants sont définies formellement dans la section 5.3. Le langage de description d'interfaces de notre modèle, CwSTS-IDL (*Components with STS - Interface Description Language*), et est présenté dans la section 5.4. SFSP le

langage de description des protocoles d'interaction de composants CwSTS est détaillé dans la section 5.4.3.

Le prototype (CwSTS-P) de notre modèle et son implémentation sont présentés dans la section 6.1. L'encapsulation de l'implémentation d'un composant sous la forme d'un paquetage Java sur lequel nous imposons des restrictions est présentée dans la section 6.1.2. Dans la section 6.1.3 nous présentons la structure standard d'un composant primitif avec ses entités constitutantes et le mécanisme de configuration, d'instanciation et d'exécution. La structure d'un composant composite est indiquée dans la section 6.1.4 et le mécanisme de composition des protocoles d'interaction est présenté dans la section 6.2.

Nous formulons les conclusions et les perspectives de notre travail dans le chapitre 7.