



Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants

THÈSE

présentée et soutenue publiquement le 29 novembre 2005

pour l'obtention du

Doctorat de l'université des Sciences et Technologies de Lille
(spécialité informatique)

par

Olivier Barais

Composition du jury

Président : Pr Sophie Tison, Université de Lille I
Rapporteurs : Dr Jacky Estublier, Centre National de la Recherche Scientifique
Pr Jean-Marc Jezequel, Université de Rennes I
Examineurs : Dr Thomas Ledoux, École des Mines de Nantes
Dr Tom Mens, Université de Mons-Hainaut
Directeur de thèse : Pr Laurence Duchien, Université de Lille I

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022 — INRIA Futurs



à Claire et petit Luc,

Remerciements

Je voudrais tout d'abord remercier les membres du jury. Un très grand merci à Jacky Estublier et Jean-Marc Jézéquel d'avoir accepté de rapporter ce travail. J'ai apprécié la profondeur de la relecture et la pertinence de vos commentaires et de vos remarques qui ont permis d'améliorer la qualité de ce document. Merci à Thomas Ledoux et à Tom Mens d'avoir accepté d'examiner mon travail. Enfin, je remercie Sophie Tison de m'avoir accordé l'honneur d'être la présidente de mon jury.

Je tiens à remercier Jean-Marc Geib, directeur du laboratoire et responsable de l'équipe GOAL et du projet INRIA Jacquard, pour m'avoir accueilli dans son laboratoire. Merci pour les excellentes conditions de travail que tu m'as offertes pendant ces trois années de thèse.

Je tiens à remercier très chaleureusement Laurence Duchien sans qui rien de tout cela ne serait arrivé. J'ai énormément apprécié les trois années de travail sous ta direction. Merci pour ton soutien sans lequel je n'aurais jamais réussi à aller au bout, tes conseils toujours lumineux et ta patience face aux organisations hasardeuses du thésard ;-). Merci aussi pour le temps que tu m'as consacré au jour le jour pendant ces trois ans et ton amour contagieux de la recherche. Merci enfin pour l'amitié que tu m'as offerte.

Je remercie tout particulièrement Anne-Françoise Le Meur qui s'est attachée depuis son arrivée dans l'équipe à améliorer la qualité de ce travail. Merci aussi pour les encouragements tout particulièrement pendant la phase de rédaction. Enfin, merci d'avoir accepté de partager le bureau 234, partage qui a permis l'avènement de très nombreuses discussions mais aussi d'une nouvelle théorie révolutionnaire pour le rangement de bureau ;-).

Je souhaite remercier vivement Lionel Seinturier pour l'intérêt qu'il a porté à ces travaux. Un grand merci pour le temps que tu as consacré à la relecture de ce document alors que tu sortais aussi à peine d'une phase de rédaction intensive. Merci pour la pertinence de tes remarques et la qualité de tes pizzas les soirs de relectures ;-)¹.

Merci aussi à Julia Lawall qui, lors de son passage au sein du projet, a passé un mois (soirées et WE compris) pour améliorer la qualité du langage de transformation. J'espère avoir l'occasion de collaborer de nouveau avec toi tant j'ai appris pendant ce mois de mai.

Je tiens à remercier Missi. Merci pour ton calme, ta patience, ton amour des technologies et tes implémentations toujours incroyables.

Je tiens à exprimer mes plus vives remerciements à Arnaud Bailly pour les nombreux éclaircissements qu'il m'a apporté au cours de ces trois ans. Merci aussi pour la qualité de tes travaux et de tes développements. Je n'exprimerai qu'un seul regret : que nous n'ayons pas eu le temps de collaborer davantage...

Je tiens à exprimer une profonde amitié à Alexis Muller (²) qui a fait sa thèse en même temps que moi. J'ai particulièrement apprécié et appris lors de nos nombreuses discussions aux pauses café sur le logiciel libre. Je garde aussi d'excellents souvenirs de nos différents déplacements professionnels à la mer ;-) ou ailleurs.

Merci à Nicolas Pessemier qui m'a apporté beaucoup par sa disponibilité et son expertise sur Fractal ou sur le développement par aspects en général.

Il m'est difficile de ne pas remercier individuellement chaque membre ou ancien membre de l'équipe pour le cadre de travail amical et stimulant qu'ils m'ont offert. Il est tellement agréable de venir au labo dans ces conditions. Un grand merci donc à Raphaël Marvie, Renaud Pawlak, Romain Rouvoy, Emmanuel Renaux, Jérôme Moroy, Dolores Diaz, Olivier Caron, Christophe Contreras, Areski Flissi, Christophe Demarrez, Jérémy Dubus, Jean François Roos, Philippe Merle, Gilles Vanwormhout, Bernard Carré, Eric Cariou et tout ceux que j'ai oublié.

Enfin, par ces remerciements, je tiens à ne pas oublier l'école des Mines de Douai et tout particulièrement Nouri Bouraquadi et Jean-Loup Cordonnier pour m'avoir donné l'opportunité d'effectuer un DEA lors de ma dernière année d'école d'ingénieur.

¹ De telles soirées justifient à elles seules de faire une thèse

² *mon compagnon de galère*

De manière plus personnelle, un très grand merci à toute mon équipe de volley pour son accueil sans faille malgré mes absences répétées et mes coups de fatigue. Un très grand merci à ma famille pour son soutien permanent pendant toutes ces années d'études. Enfin, un grand merci à Luc et à Claire pour supporter au quotidien mes angoisses, mon manque de temps et mes plans galères. Vous me donnez tous les deux une réelle raison d'avancer.

Résumé

Dans une industrie du logiciel où la complexité des applications ne cesse de croître, un des objectifs du génie logiciel est de conserver l'intelligibilité du processus de construction et de maintenance de ces applications. Dans ce cadre, le concept de composant a été introduit comme une extension du concept d'objet pour faciliter la conception d'applications réparties complexes, c'est-à-dire formées de nombreux éléments en coopération, distribuées sur différents sites d'exploitation, et de nombreuses contraintes qui expriment des exigences variées (persistance, sécurité, tolérance aux pannes, etc.). Il existe à ce jour de nombreux modèles de composants avec des objectifs différents. Cependant, un des points communs de ces modèles est la mise en lumière de l'architecture de l'application, c'est-à-dire une description de haut niveau de cette application décomposée en différents composants interconnectés et définis par leurs fonctionnalités requises et offertes.

De par la complexité de certaines applications, la définition de l'architecture est elle-même une opération délicate et très propice aux erreurs. L'objectif de cette thèse vise à améliorer la maîtrise de la complexité de la construction de l'architecture d'une application définie par assemblage de composants. Nous proposons pour cela un modèle de composant abstrait, nommé SafArchie, qui permet d'assister l'architecte dans la construction d'une architecture logicielle cohérente. Ainsi, dans SafArchie, l'interface des différents composants d'une architecture est spécifiée à l'aide de contrats. Ces contrats permettent de définir la structure du composant, mais aussi son comportement vis-à-vis de son environnement. À partir de ces différentes informations et de la définition de l'assemblage de composants, nous proposons une notion de *compatibilité* entre composants interconnectés. Cette compatibilité permet de déterminer la cohérence de l'architecture.

Un deuxième axe de la thèse part du constat qu'il est difficile de construire un assemblage de composants qui adresse l'ensemble des préoccupations d'une application en une seule étape. L'idéal est alors de proposer une démarche de construction incrémentale où l'architecte peut à chaque étape ajouter une nouvelle préoccupation à son architecture. Malheureusement, les modèles de composants ne proposent en générale qu'une unique dimension de structuration d'une application, SafArchie n'échappe pas à cette règle. Dès lors, certaines préoccupations comme la sécurité, la persistance ou la traçabilité ne peuvent être correctement structurées au sein d'un seul composant et se retrouvent alors noyées au sein de nombreux éléments de l'architecture. Cette mauvaise structuration est très préjudiciable pour l'intégration de ces préoccupations dans une architecture existante. Nous proposons donc, dans une deuxième partie de cette thèse, un canevas de conception d'architecture nommé TranSAT qui permet l'intégration de nouvelles préoccupations dans une architecture par transformation de cette dernière. Ce canevas introduit la notion de *patron d'architecture* pour structurer les différentes préoccupations transverses d'une application. Ce patron comprend les éléments d'architecture à intégrer, les transformations à apporter sur l'architecture de base, mais aussi un ensemble de contraintes génériques sur les éléments d'une architecture cible sur laquelle le patron peut être intégré. De plus, TranSAT propose un langage dédié pour spécifier les modifications à apporter sur l'architecture de base afin d'intégrer la nouvelle préoccupation. Ce langage est concis et expressif, sa spécialisation permet de produire différentes analyses statiques afin de garantir la cohérence de l'architecture résultante sans analyser la totalité de cette architecture.

Mots-clés: Architecture logicielle, cohérence d'un assemblage de composants, construction incrémentale, développement de logiciel par aspects, évolution du logiciel



Abstract

In a software industry in which the application complexity increases more and more, one of the main goals of the software engineering consists of keeping the intelligibility of the design and the maintenance process of these applications. In this context, the concept of component was introduced as an extension of the object paradigm to improve the design of complex and distributed applications, i.e. applications composed of numerous cooperating elements, deployed on several sites, and submitted to lots of constraints that express different requirements such as persistency, fault tolerance, or security. Currently, there are lots of component model that have different goals and different features. However, they usually have a common characteristic. They highlight the software architecture : i.e. an abstract system specification composed of several interconnected components that are defined in terms of provided or required services.

Due to the extreme complexity of some applications, the design of the architecture is a complex and error-prone task. This PhD aims at improving the assistance of the architect when he designs its architecture defined as a component assembly. Consequently, we proposed an abstract component model, called SafArchie. In SafArchie, each component interface is defined with contracts. These contracts specify the structure but also the external behavior of the components. From this information, we defined the concept of compatibility between interconnected components. This compatibility helps architect to design consistent software architectures.

The second part of the PhD focuses on the difficulty to build an entire software architecture description in one step. The goal consists in proposing an incremental approach in which the architect integrates the different concerns into the software architecture stepwise. Unfortunately, the component models provide generally only one dimension to modularize the system. SafArchie obeys this rule. Consequently, some concerns such as the persistency or the security can not be correctly modularized and they are tangled into different elements of the architecture. This bad modularization is detrimental to the integration of new concerns into the software architecture. Thus, we propose in a second part of this PhD, a design framework calld TranSAT which allow the integration of new concerns into a software architecture. This framework performs the integration with a model transformation approach. It defines the concept of software architecture pattern to clearly modularize the tangled concerns. The software architecture pattern contains the architectural elements to integrate for a particular concern, the modification to perform on the software architecture in order to integrate the new architectural elements and also a set of generic conditions that constrain where the pattern can be integrated into the software architecture pattern. Furthermore, TranSAT introduces a specific transformation language to specify the modifications to perform to integrate a new concern. This language is small and expressive. Thanks to this framework, a set of static analysis are performed on the pattern to guarantee that its use does not break the software architecture consistency previously established.

Keywords: Software architecture, component assembly consistency, incremental design, aspect oriented design

Table des matières

Table des figures	xiii
Liste des tableaux	xvii
Chapitre 1 Introduction	1
1.1 Contexte	1
1.2 Problématiques	2
1.3 Contributions	3
1.4 Organisation du document	4
1.5 Liste des publications liées à cette thèse	6
Partie I État de l’art	9
Chapitre 2 Contexte de l’approche	11
2.1 Séparation des préoccupations	12
2.1.1 La modularité	13
2.1.2 L’abstraction	15
2.1.3 L’anticipation du changement	17
2.1.4 La construction incrémentale	19
2.2 La réutilisation	20
2.2.1 Conception par contrats	21
2.2.2 Taxonomie des contrats	22

2.2.3	Contractualisation des interactions entre composants	22
2.2.4	Contrats de qualité de service	23
2.2.5	Bilan sur la réutilisation	23
2.3	Architecture logicielle	24
2.3.1	Lien entre modèle de composant et architecture logicielle	24
2.3.2	Architectures orientées service ou architectures à base de composants ?	25
2.3.3	Conclusion	25
2.4	Défis de l'architecture logicielle	25
2.4.1	La modularité	25
2.4.2	L'abstraction	26
2.4.3	L'anticipation du changement	26
2.4.4	Une construction incrémentale : des composants métiers à l'intégration des préoccupations techniques	26
2.4.5	La réutilisation	27
Chapitre 3 La construction d'architecture à base de composants		29
3.1	Les langages de description d'architectures	31
3.1.1	Wright	31
3.1.2	Darwin	35
3.1.3	Bilan sur les ADLs	38
3.2	Les modèles académiques de construction d'applications à base de composants	39
3.2.1	ArchJava	39
3.2.2	Fractal	43
3.2.3	SOFA	48
3.2.4	Bilan sur les modèles de composant académiques	51
3.3	Les standards	54
3.3.1	AADL	54
3.3.2	UML 2.0	57

3.3.3	Bilan sur les normes associées à la description d'une architecture logicielle	61
3.4	Bilan sur les approches à base de composants	62
3.4.1	Anticipation des changements : prise en compte de la dynamique du système	62
3.4.2	Abstraction : indépendance vis-à-vis de la plate-forme d'exécution	63
3.4.3	Modularité : structuration multi-dimensionnelle de l'application	64
3.4.4	Réutilisation : définition étendue des interfaces de composant	64
3.4.5	Construction incrémentale de la description d'architecture logicielle	65
3.4.6	Proposition	65
Partie II	Construire une architecture logicielle à base de composants	67
Chapitre 4	Spécifier un type d'architecture logicielle cohérent	71
4.1	SafArchie ADL : vers une notion de type d'architecture logicielle	72
4.1.1	Définition du modèle de type d'architecture	72
4.1.2	Le méta-modèle de type d'architecture	74
4.1.3	Un méta-modèle - différentes représentations	75
4.1.4	Exemple d'architecture : le <i>DataPhoneCenter</i>	75
4.2	Une définition de composant étendue pour un assemblage cohérent	79
4.2.1	Notion de contrat dans SafArchie	79
4.2.2	Contrats structurels d'un composant	79
4.2.3	Contrat comportemental d'un composant	82
4.2.4	Conclusion sur les langages de contrats dans SafArchie	96
4.3	Construire un type d'architecture logicielle cohérent	97
4.3.1	Analyse structurelle de l'assemblage	97
4.3.2	Analyse autour du contrat de comportement	104
4.4	Bilan autour du modèle de type	109
Chapitre 5	Construire et déployer des architectures logicielles typées	111

5.1	Construction d'une architecture logicielle typée	112
5.1.1	Structure d'une architecture logique	112
5.1.2	Une architecture logicielle : instance d'un type d'architecture	114
5.1.3	Un méta-modèle - différentes représentations	114
5.1.4	Modèle logique et contrats d'interfaces de composants	114
5.1.5	Exemple d'architecture logique	115
5.1.6	Bilan sur la notion d'architecture logique	115
5.2	Déploiement d'une architecture logique sur une infrastructure physique	115
5.2.1	Un méta-modèle d'infrastructure physique	116
5.2.2	Exemple de modèle physique d'architecture	117
5.2.3	Placement des composants sur une infrastructure physique	117
5.2.4	Analyse du placement des composants	118
5.2.5	Exemple de description de placement de composants	119
5.2.6	Bilan sur le déploiement dans SafArchie	120
5.3	SafArchie face aux défis de l'architecture logicielle	120

Partie III Enrichir une architecture logicielle à base de composants 123

Chapitre 6 TranSAT, un canevas de conception pour l'intégration de nouvelles préoccupations 127

6.1	Introduction	128
6.1.1	Une approche par transformation	129
6.1.2	Une assistance nécessaire de l'architecte	129
6.1.3	Propriétés attendues du système de transformation d'architecture	129
6.2	Architecture de TranSAT	130
6.2.1	Le plan	131
6.2.2	Le masque de point de jonction	132
6.2.3	Un langage de transformation d'architecture avec une double personnalité	135

6.2.4	Le tisseur	139
6.3	Un processus itératif pour la construction d'architectures logicielles	143
6.3.1	Définition	143
6.3.2	Un processus d'intégration de nouvelles préoccupations	144
6.3.3	Synthèse	148
6.4	Sûreté de fonctionnement	149
6.4.1	Les risques	149
6.4.2	Vérifications statiques sur le patron d'architecture	151
6.4.3	Vérifications dynamiques	157
6.4.4	Conclusion sur l'analyse dynamique	159
6.5	Synthèse	159
Chapitre 7 Concrétisation de l'approche : un atelier de construction d'architectures logicielles		161
7.1	Introduction et motivations	161
7.2	SafArchie Studio : une architecture modulaire	162
7.2.1	Présentation générale de SafArchie Studio	162
7.2.2	Étendre ArgoUML	165
7.2.3	Vérification de la cohérence	174
7.2.4	Génération de code	177
7.3	TranSAT	180
7.3.1	Trois nouveaux modules pour la définition de nouveaux patrons d'architecture	181
7.3.2	Le tisseur, un moteur de transformation pour intégrer des nouvelles préoccupations dans une description d'architecture logicielle	183
7.4	Bilan et Perspectives sur SafArchie Studio	185
7.4.1	Bilan qualitatif	185
7.4.2	Bilan quantitatif	185
7.4.3	Perspectives	185

Partie IV Conclusion et perspectives	189
Chapitre 8 Conclusion et perspectives	191
8.1 Résumé des contributions	191
8.2 Perspectives	192
Annexe A Grammaire du langage textuel de description de type d'architecture	203
Annexe B Schéma XML du modèle de type d'architecture	205
Annexe C Grammaire du langage de contrat de port	209
Annexe D Grammaire du langage de contrat d'assertion	211
Annexe E Grammaire du langage SFSP permettant la description de comportement	215
Annexe F Grammaire du langage textuel de description de type d'architecture	217
Annexe G Schéma XML du modèle d'architecture logique de SafArchie	219
Annexe H Grammaire du langage textuel de description de masque de point de jonction	221
Annexe I Schéma XML du modèle de masque de point de jonction de TranSAT	225
Annexe J Grammaire du langage de transformation de TranSAT	227

Table des figures

1.1	Présentation globale de la thèse	5
2.1	Inversion des dépendances entre les modules métiers et les services techniques	19
3.1	Architecture d'un serveur Web décrit à l'aide de Wright	34
3.2	Architecture d'un serveur Web décrit à l'aide de Darwin	37
3.3	Mise en œuvre d'un serveur Web avec ArchJava	41
3.4	Modèle de composant Fractal	44
3.5	Mise en œuvre d'un serveur Web avec Fractal	47
3.6	Structure d'une classe prenant en charge le métier d'un composant Fractal, le contenu du composant	47
3.7	Mise en œuvre d'un serveur Web avec SOFA	52
3.8	Exemple simple AADL	55
3.9	Architecture d'un serveur Web décrit à l'aide de AADL	57
3.10	Architecture d'un serveur Web décrit à l'aide d'UML 2.0	61
3.11	Présentation générale de SafArchie	70
4.1	Méta-modèle de type d'architecture	74
4.2	Charte graphique du modèle de type de SafArchie	76
4.3	Type d'architecture pour l'exemple DataPhoneCenter	77
4.4	Définition des ports du type d'architecture <i>DataPhoneCenter</i>	78
4.5	Représentation graphique d'un système de transitions étiquetées	83
4.6	Représentation graphique d'un automate à entrées sorties	84
4.7	Identification des messages dans les contrats de comportement	89
4.8	Automate du comportement du type de composant <i>AA</i>	90
4.9	Automate du comportement du type de composant <i>MediaCenter</i>	92

Table des figures

4.10	Automate du comportement du type de composant <i>EmailReader</i>	93
4.11	Automate du comportement du type de composant <i>WebReader</i>	93
4.12	Automate du comportement du type de composant <i>FSAccess</i>	94
4.13	Automate du comportement du type de composite <i>DataPhoneCenter</i> réduit à ses seuls ports visibles	95
4.14	Propagation des règles de compatibilité	104
4.15	Exemple d'un automate présentant un interblocage	106
4.16	Automate résultant de la propriété de sûreté <i>PAIEMENT</i> (1)	108
5.1	Modèle logique de <i>SafArchie</i>	113
5.2	Modèle logique d'une configuration du <i>DataPhoneCenter</i>	116
5.3	Modèle physique de <i>SafArchie</i>	117
5.4	Exemple d'une infrastructure physique décrite à l'aide de <i>SafArchie</i>	117
5.5	Modèle de déploiement de <i>SafArchie</i>	118
5.6	Déploiement du <i>dataPhoneCenter</i> sur une infrastructure physique	120
6.1	TranSAT : tissage de plans d'architecture	128
6.2	Vue générale de TranSAT	131
6.3	Plan en charge de la préoccupation d'atomicité	131
6.4	Spécification du comportement du composant Coordinateur	132
6.5	Automate de comportement du coordinateur	133
6.6	Le modèle de masque de point de jonction	134
6.7	Exemple de masque de point de jonction pour le plan prenant en charge l'atomicité	135
6.8	Transformation du comportement	138
6.9	Règles de transformation pour la préoccupation d'atomicité	140
6.10	Plan de base	141
6.11	Intégration de l'atomicité au niveau du <i>DataPhoneCenter</i>	142
6.12	Vision externe du processus	145
6.13	Vision interne du processus	146
6.14	Diagrammes de flots et d'activités pour la définition d'un plan de base	146
6.15	Diagrammes de flots et d'activités pour la définition d'un nouveau plan	147
6.16	Diagrammes de flots et d'activités pour la définition d'un nouveau patron d'architecture	147
6.17	Diagrammes de flots et d'activités pour la spécification de l'intégration d'une nouvelle préoccupa- tion au sein d'une architecture logicielle métier	148

6.18	Diagramme de responsabilités	148
6.19	Exemple d'architecture	157
7.1	Trois nouveaux modules pour SafArchie Studio	163
7.2	Copie d'écran de SafArchie Studio sur l'exemple de la station	164
7.3	Modèle MVC de ArgoUML	165
7.4	Interface d'extension d'ArgoUML	166
7.5	Un mécanisme d'événements pour la notification de la couche « Contrôleur »	167
7.6	Architecture de l' <i>Explorer Framework</i>	169
7.7	Mise en œuvre de nouvelles couches métier	170
7.8	Exemple d'annotations pour définir la hiérarchie des objets à l'aide JAXB	171
7.9	Exemple d'annotations pour générer une fabrique prenant en compte des extensions des éléments du modèles	172
7.10	Hiérarchie de classes pour la couche Métier générée avec JAXB	173
7.11	Vérifications autour du contrat d'assertion	175
7.12	Vérifications autour du contrat de port	176
7.13	Façade pour le produit de synchronisation	177
7.14	Vérifications autour du contrat de comportement	178
7.15	Vue globale de l'architecture de la projection	178
7.16	Trois nouveaux modules pour la production de nouveaux patrons d'architecture	181
7.17	Un nouveau module pour l'intégration d'un nouveau plan au sein d'une architecture logicielle	181
7.18	Mise en œuvre de la recherche exhaustive de lieux d'intégration compatibles	184
7.19	Validation d'un lieu d'intégration spécifié manuellement	184

Liste des tableaux

- 3.1 Prise en compte de la dynamique 63
- 3.2 Indépendance vis-à-vis de la plate-forme d'exécution 64
- 3.3 Pouvoir d'expression disponible au niveau des interfaces selon la taxonomie de Beugnard *et al* . . 65
- 4.1 Lien entre la spécification du comportement à l'aide de messages et la définition structurelle du composant 86
- 6.1 Primitives d'introduction 137
- 6.2 Primitives de reconfiguration 138
- 7.1 Bilan quantitatif de SafArchie Studio 186

Chapitre 1

Introduction

1.1 Contexte

Le génie logiciel est défini comme l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser le développement du logiciel et son suivi. Discipline née de la première crise du logiciel dans les années 1970 [Dijkstra, 1972] et entretenue par une complexité des systèmes en croissance permanente, il trouve sa justification dans la difficulté intrinsèque à estimer correctement l'effort requis pour aboutir à un logiciel fiable. Les recherches dans ce domaine ont permis l'avènement de nouveaux paradigmes de conception comme par exemple la programmation par objets [Ducournau *et al.*, 1998, Meyer, 2000]. Fondée sur les notions d'encapsulation, d'héritage et de polymorphisme, cette dernière a apporté une contribution importante dans la manière de concevoir et de programmer une application. Néanmoins, les objets n'ont tenu que partiellement leurs promesses au niveau de la réutilisation et de la structuration des entités logicielles de grande taille. L'approche par composants (*Component Based Software Engineering - CBSE*) [Heineman and Councill, 2001] est une approche récente visant à résoudre les lacunes de l'approche par objets.

L'idée de composant logiciel date déjà d'un certain nombre d'années. On trouve une des premières utilisations explicites de ce terme en 1968 [McIlroy, 1968]. Cependant, il a fallu attendre la fin des années 90 et le constat d'échec du paradigme objet dans le domaine de la réutilisation pour voir apparaître un regain d'intérêt autour de la notion de composant logiciel comme unité de réutilisation de modules logiciels. Dans le domaine de la conception par objets, le principal mécanisme pour favoriser la réutilisation est le mécanisme d'héritage. Il introduit un couplage très fort entre la classe de base et ses classes dérivées, couplage responsable du problème identifié sous le nom de *Fragile Base Class* [Szyperski, 1996, Mikhajlov and Sekerinski, 1998]. La fragilité des classes de base vient du fait qu'une modification au sein de celles-ci peut introduire un problème au sein d'une classe dérivée [Holub, 2003]. Pour contrer ce problème, l'approche par composants propose une meilleure séparation entre la partie réalisation d'un module logiciel et son interface, c'est-à-dire la description de services qu'il fournit et qu'il requiert. Chaque composant est une boîte noire, indivisible, composable, déployable et identifiable par les services qu'elle offre et qu'elle requiert. Dès lors, la construction d'une application se fait par assemblage de composants. De nombreux modèles de composant sont actuellement disponibles [Marvie and Pellegrini, 2002]. Adopté dans la pratique pour le développement d'applications à large échelle, le paradigme composant améliore grandement la vitesse de production de logiciels.

Le découplage entre la partie réalisation d'un composant et son interface avec l'environnement met en avant deux informations : une description claire de l'interface des composants et une description de l'assemblage des composants, c'est-à-dire leurs interactions. Ces deux informations au cœur de l'approche par composants constituent les éléments de base de ce que l'on nomme l'*architecture logicielle*. Une description d'architecture logicielle possède quatre rôles centraux dans un projet informatique [Garlan and Shaw, 1993]. Elle facilite la compréhension de la structure d'un système. Elle permet son analyse. En tant que cadre pour la construction de l'application,

elle sert de base pour la génération de code et l'identification des opportunités de réutilisation. Enfin, elle définit les invariants du système et sert donc de pivot pour son évolution. Il en résulte qu'un langage de description d'architecture logicielle est un vocabulaire commun entre les différents acteurs d'un projet informatique. Pour illustrer ces concepts, de nombreux langages sont apparus pour permettre la description d'une architecture logicielle et son analyse tels que Rapide [Luckham *et al.*, 1995], UniCon [Zelesnik, 1996], Wright [Allen, 1997], ACME [Garlan *et al.*, 1997], Darwin [Magee *et al.*, 1995] ou plus récemment ArchJava [Aldrich *et al.*, 2002], XADL [Roshanak, 2000], AADL [Dissaux, 2004], ou Fractal ADL [Bruneton *et al.*, 2004b] avec chacun leurs spécificités, leurs avantages et leurs inconvénients [Medvidovic and Taylor, 2000].

Malgré les nombreux atouts des composants et des architectures logicielles, des questions très anciennes en génie logiciel se posent avec une acuité nouvelle, du fait de la complexité des systèmes concernés. Ainsi, la construction de systèmes de grande taille par composition de modules logiciels existants reste pour le moment une opération complexe, peu assistée et peu prise en compte dans les modèles de composant existants. Différents rapports récents ont montré la complexité de définir un assemblage de composants cohérent et de réutiliser un composant sur étagère dans un contexte différent de son contexte de création [Jézéquel and Meyer, 1997, Weyuker, 1998]. Cette complexité est souvent accrue du fait de l'intervention d'un grand nombre d'acteurs possédant des rôles différents dans un projet informatique.

1.2 Problématiques

La variabilité est une information peu prise en compte dans les descriptions d'architectures logicielles actuelles. Nous identifions deux origines distinctes de cette variabilité. Premièrement, la structure d'une architecture logicielle doit être modifiée pour rester adaptée à son contexte d'exécution. Nous appelons ces modifications des *évolutions internes*. Elles modélisent la dynamique de l'architecture, c'est-à-dire les modifications apportées à la configuration de l'architecture pour rester adaptée à son contexte d'exécution. En effet, la description de l'architecture logicielle en tant qu'abstraction de la structure de l'application, doit prendre en compte et permettre d'exprimer ces possibles évolutions de la structure de l'application afin de ne pas masquer cette propriété de l'architecture qu'est sa dynamique. Deuxièmement, l'architecture d'une application doit être constamment transformée afin de pouvoir prendre en compte les évolutions des exigences des utilisateurs. Nous appelons dans la suite de ce document ces transformations les *évolutions externes*, car elles permettent l'intégration de nouvelles préoccupations suite à une action de l'architecte. Une description d'architecture logicielle doit être capable de prendre en compte ces deux types d'évolution concernant le système qu'elle modélise.

De plus, afin d'être facilement compréhensible et maintenable, il est crucial de séparer convenablement les préoccupations au niveau d'une application [Parnas, 1972]. Dans ce domaine de la structuration, l'approche par composants possède de manière intrinsèque la même lacune que l'objet. Elle propose une seule et unique dimension de décomposition autour des fonctionnalités de l'application. Or différents travaux ont montré la difficulté de structurer une application à l'aide d'une unique dimension. Ce problème a été identifié comme la tyrannie de la décomposition dominante [Tarr *et al.*, 1999]. En conséquence, certaines préoccupations se retrouvent noyées et/ou répétées dans différents composants du système. Dès lors, de nombreuses préoccupations se retrouvent entrelacées au sein des composants ce qui rend leur définition difficile. Cet entrelacement des préoccupations nuit aussi à la réutilisation en diminuant la généricité des composants.

Le travail de cette thèse s'inscrit dans le projet Jacquard de l'INRIA Futurs qui porte sur la conception des applications réparties complexes c'est-à-dire formées de nombreux composants en coopération, distribuées sur de nombreux sites d'exploitation, et contraintes par des exigences variées (persistance, sécurité, tolérance aux pannes, *etc.*). Le projet Jacquard se base pour cela sur les notions de modèle de composants, de séparation des préoccupations et de tissage, en les utilisant dans les phases de modélisation, d'assemblage, de déploiement et d'exécution des applications. Au sein de ce projet, la problématique de la modélisation d'une application à base de composants logiciels est un point crucial. L'architecture logicielle permet d'obtenir une vision globale des composants intervenant au sein d'une application. De ce constat évident du besoin lié à la description d'une architecture logicielle, sont nées plusieurs problématiques :

- Comment spécifier un composant d’une architecture logicielle pour que sa définition soit suffisamment riche afin de garantir la cohérence de l’architecture logicielle tout en restant une information suffisamment abstraite afin de ne pas être dépendante des contraintes liées au choix de la plate-forme d’exécution ?
- Comment prendre en compte au niveau de la description de l’architecture son caractère dynamique ?
- Comment offrir un nouveau degré de structuration pour la notion de préoccupation qui ne peut être convenablement structurée à l’aide de la notion de composant ?
- Comment prendre en compte au niveau de la description d’une architecture logicielle le nouveau paradigme de composition que représente le tissage de préoccupations ?
- Lors de ce tissage de nouvelles préoccupations, comment garantir la cohérence de l’architecture résultante ?

Autant de problèmes qui sont pour le moment peu pris en compte dans les langages de description d’architecture ou les plates-formes à composants existants. En effet, il est souvent impossible d’obtenir une bonne séparation des préoccupations, les modèles existants souffrant des mêmes lacunes que l’objet dans ce domaine. Dès lors, l’intégration d’une nouvelle préoccupation, si elle modifie de nombreuses entités de l’architecture, est une tâche manuelle non assistée. Enfin, il n’y a aucune capitalisation du travail de l’architecte qui travaille sur cette opération d’intégration de nouvelles préoccupations.

Nous n’aborderons pas dans ce document les notions d’adaptabilité [Ledoux *et al.*, 2001] mais parlerons d’évolutivité. En effet, l’adaptabilité se concentre sur des problématiques de mise en œuvre au niveau des plates-formes d’exécution, quand nos travaux s’intéressent davantage à la phase de modélisation d’une architecture logicielle.

1.3 Contributions

Le sujet de la thèse s’intéresse aux étapes de construction et de modélisation des applications à base de composants. Pour cela nous définissons notre propre canevas de construction d’application appelé *SafArchie* (*Safe Architecture*). *SafArchie* s’appuie sur une description étendue, à la fois structurale et comportementale, des interfaces des composants afin de permettre une analyse de l’assemblage. Il est basé sur deux modèles de composant abstraits permettant de spécifier une architecture logicielle typée. Ces deux modèles nommés respectivement *modèle de type* et *modèle logique* permettent de séparer la spécification des invariants du système de sa dynamique.

Le deuxième axe de la thèse étudie la structuration des informations dans une description d’architecture dans un contexte d’évolution et, plus spécifiquement, dans un contexte d’intégration de nouvelles fonctionnalités. Nous proposons avec *TranSAT* (*Transformation for Software Architectures*) un cadre de conception pour l’intégration de nouvelles fonctionnalités au sein d’une description d’architecture. Inspiré par le développement de logiciels par aspects (*Aspect Oriented Software Development - AOSD*) [Kiczales *et al.*, 1997] qui vise à améliorer la séparation des préoccupations au sein d’une application, *TranSAT* intègre une dimension supplémentaire dans la description d’architecture par la définition du concept de *patron d’architecture*. Celui-ci représente une unité de structuration pour les préoccupations transverses au sein d’une architecture logicielle. Il contient un assemblage de composants, appelé *plan*, qui prend en charge les fonctionnalités liées à la préoccupation, une description des éléments devant être présents dans l’architecture de base afin de pouvoir tisser le plan, appelé *masque de point de jonction* et un ensemble de règles de transformation qui définissent les modifications à apporter sur le plan de base pour permettre l’intégration de la nouvelle préoccupation. Le patron d’architecture permet par un mécanisme de transformation de mêler plusieurs plans d’architecture. Inspiré par l’AOSD, il contient à la fois des primitives de reconfiguration qui modifient l’assemblage et des primitives de transformation plus intrusives qui modifient l’interface et le comportement des composants. *TranSAT* propose par l’intermédiaire du concept de patron d’architecture une nouvelle entité de structuration d’une architecture logicielle. Cet élément est réutilisable pour différents contextes d’intégration grâce à une spécification claire des attentes du patron vis-à-vis de son contexte d’intégration. En outre, son intégration se fait par un mécanisme de tissage dont la sémantique est explicitée par les règles de transformation.

Nous présenterons dans ce document comment le couple *SafArchie/TranSAT* est guidé par les grands principes du génie logiciel, à savoir la séparation des préoccupations, la modularité, l’abstraction, l’anticipation du change-

ment et la construction incrémentale. TranSAT permet la construction d'architectures logicielles dont l'architecte maîtrise la complexité et fournit un ensemble d'outils pour améliorer la productivité.

Enfin, la dernière contribution de cette thèse réside dans la production d'un atelier de génie logiciel mettant en œuvre les principes du couple SafArchie/TranSAT. Cet atelier fournit, outre la possibilité de construire une architecture par superposition de plans, des outils pour garantir la cohérence d'une architecture logicielle et la possibilité de générer du code vers des plates-formes à composants existantes comme Fractal ou ArchJava. La figure 1.1 présente la démarche globale de spécification d'une architecture logicielle au sein de notre projet. SafArchie permet la description d'une architecture logicielle, son analyse et la génération de code vers différentes plates-formes d'exécution. TranSAT permet, quant à lui, la spécification et l'analyse d'une évolution de cette architecture.

1.4 Organisation du document

Ce document est composé de trois parties principales. La première place le contexte de la thèse et dresse un état de l'art des langages de description d'architectures, modèles de composant issus du milieu académique ou des organismes de normalisation. La seconde partie présente notre canevas de description d'architecture *SafArchie* en détail. La troisième partie décrit TranSAT, notre proposition pour la structuration des fonctionnalités transverses au sein d'une architecture logicielle, son utilisation sur un exemple simple et l'atelier de génie logiciel mis en œuvre pour concrétiser les propositions autour du couple SafArchie/TranSAT. Enfin, une dernière partie regroupe les conclusions, contributions et perspectives de ce travail. L'ensemble des grammaires des différents langages utilisés dans ce document est disponible en annexe.

Partie 1 : « État de l'art »

Le chapitre 2 « Contexte de l'approche » rappelle les grands principes du génie logiciel dont les objectifs sont la maîtrise de la complexité et la réduction des coûts de développement d'une application. Ce chapitre aborde les grandes tendances actuelles du génie logiciel en montrant comment chacune d'elles se base majoritairement sur l'un de ces principes. L'énoncé de ces grands principes nous permet de mettre en lumière ce que nous appelons les grands défis de l'architecture logicielle qui motivent l'ensemble des travaux de cette thèse.

Le chapitre 3 « Langages de description d'architectures et modèles de composant » présente un ensemble de langages de description d'architecture, de modèles de composant et ainsi que des travaux de normalisation récents prenant en compte le concept d'architecture logicielle. L'étude de ces différents travaux se focalisera sur les niveaux de contrats disponibles selon la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999], la capacité d'abstraction par rapport aux plates-formes à composants existantes et enfin la prise en compte du cycle de vie du composant.

Partie 2 : « Construire une architecture logicielle à base de composants »

Le chapitre 4 « Spécifier un type d'architecture logicielle cohérent » décrit le modèle de type de SafArchie. Il détaille la structure du modèle, les différents éléments (appelés contrats) ajoutés à cette description afin de préciser la définition de l'interface du composant et les mécanismes d'analyse d'un assemblage de composants.

Le chapitre 5 « Construire et déployer des architectures logicielles typées » propose le modèle logique de SafArchie et le modèle de déploiement qui permettent de modéliser un graphe d'instances de composant déployé sur une infrastructure physique.

Partie 3 : « Enrichir une architecture logicielle à base de composants »

Le chapitre 6 « TranSAT, un canevas de conception pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle » présente l'extension de SafArchie permettant la modélisation de l'évolution externe

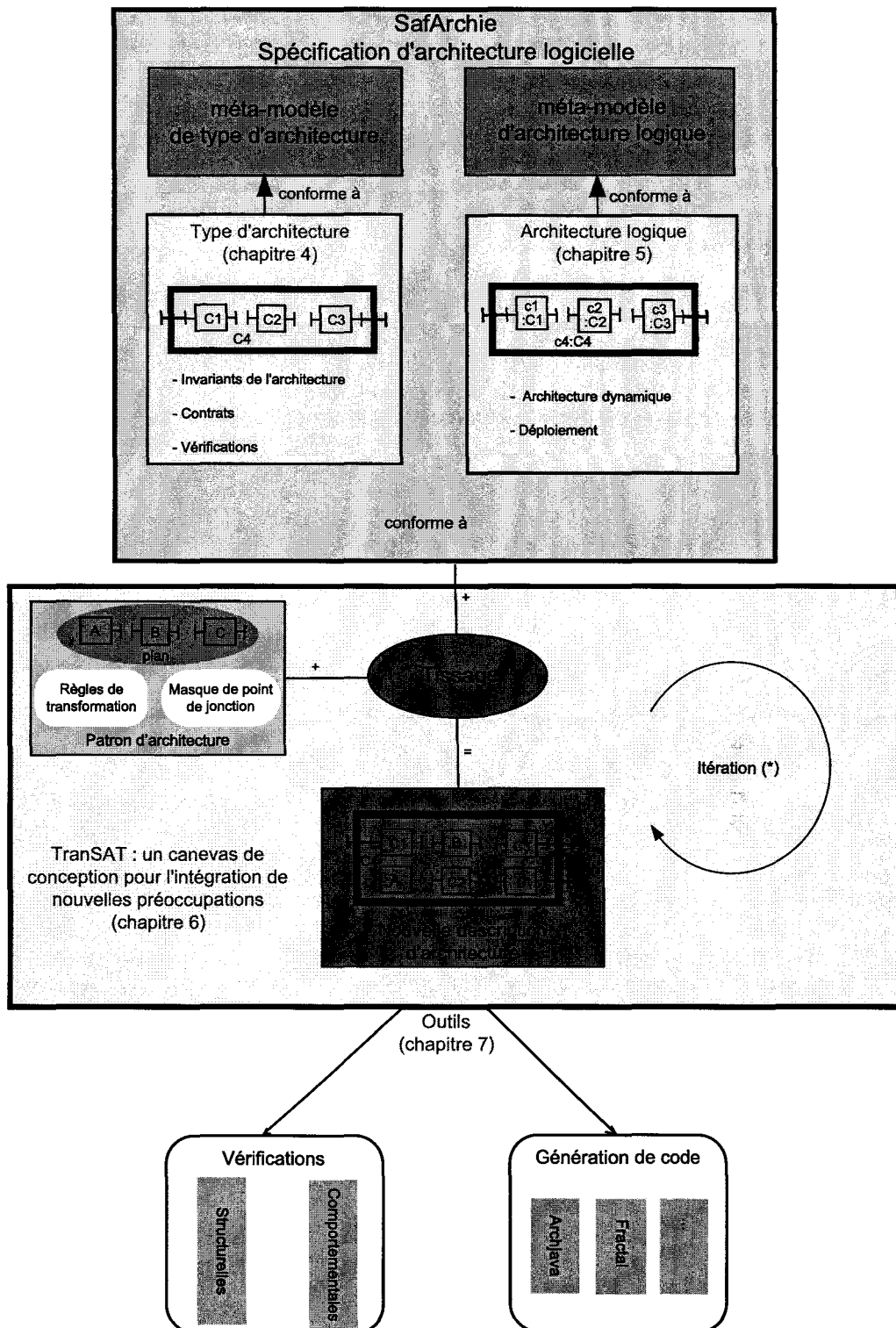


FIG. 1.1 – Présentation globale de la thèse

et la construction incrémentale d'une description d'architecture logicielle. Nous présentons les éléments clés du canevas, les mécanismes introduits par TranSAT pour permettre le tissage de nouvelles préoccupations et comment nous nous appuyons sur SafArchie pour garantir l'intégration correcte d'une nouvelle préoccupation.

Le chapitre 7 « Spécifier une architecture logicielle avec TranSAT » illustre les différents outils développés autour de SafArchie et TranSAT permettant la construction d'un atelier de génie logiciel, appelé SafArchie Studio, centré sur la description de nouvelles architectures logicielles.

Partie 4 : « Conclusion et perspectives »

Le chapitre 8 « Conclusion et perspectives » décrit les contributions de cette thèse et les perspectives de recherche associées.

1.5 Liste des publications liées à cette thèse

Revue nationale

- [1] Olivier Barais, Alexis Muller, et Nicolas Pessemier
Vers une Séparation Entités/Fonctions au sein d'une Architecture Logicielle à base de Composants
A paraître dans la Revue L'objet, Hermès Sciences, 25 pages

Chapitre d'ouvrage

- [2] Olivier Barais et Laurence Duchien
SafArchie Studio : An ArgoUML extension to build Safe Architecture
dans *Architecture Description Languages, 2005*, pages 85-100, Springer. ISBN : 0-387-24589-8.
version étendue de [6] (WADL 04)

Communications internationales

(Avec édition d'actes et comité de sélection)

- [3] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur et Laurence Duchien
Providing Support for Safe Software Architecture Transformations
5th IEEE/IFIP Working Conference on Software Architecture (WICSA), Pittsburgh, USA, November 6th-11th, 2005, 2 pages.
- [4] Olivier Barais, Laurence Duchien et Anne-Françoise Le Meur
A Framework to Specify Incremental Software Architecture Transformation
31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Porto, Portugal, August 30th - September 3rd, 2005, pages 62-70, IEEE Computer Society, ISBN : 0-7695-2431-1.
- [5] Nicolas Pessemier, Olivier Barais, Lionel Seinturier, Thierry Coupaye et Laurence Duchien
A Three Level Framework for Adapting Component-Based Systems
Proceedings of the ECOOP Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'05), Glasgow, Scotland, July 25, 2005, pages 71-77, Steffen Becker, Carlos Canal, Juan M. Murillo, Pascal Poizat and Massimo Tivoli (Eds.), ISBN : 84-689-2747-3.
- [6] Olivier Barais et Laurence Duchien
SafArchie Studio : An ArgoUML extension to build Safe Architecture
IFIP TC-2 Workshop on Architecture Description Languages (WADL 2004), Toulouse, France, August 22-27 2004, <http://www.laas.fr/FERIA/SVF/WADL04/>.
- [7] Olivier Barais, Eric Cariou, Laurence Duchien, Nicolas Pessemier et Lionel Seinturier
TransAT : A Framework for the Specification of Software Architecture Evolution
Proceedings of the ECOOP First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04). June 14, 2004. Oslo, Norway. <http://wcat04.unex.es/wcat04/index.htm>
Carlos Canal, Juan M. Murillo and Pascal Poizat (Eds.), ISBN : 84-688-6782-9.
- [8] Olivier Barais, Laurence Duchien et Renaud Pawlak
Separation of concerns in software modeling : A framework for software architecture transformation
IASTED International Conference on Software Engineering Applications (SEA), Los Angeles, USA, November 2003, pages 663-668, ACTA Press. ISBN : 0-88986-394-6.

Communications nationales

(Avec édition d'actes et comité de sélection)

- [9] Olivier Barais et Laurence Duchien
SafArchie : Maîtriser l'évolution d'une architecture logicielle
Conférence Langages, Modèles, Objets - Journées Composants (LMO-JC'04), Lille, France, Mars 2004, Hermès Sciences, volume 10 de la revue L'objet, pages 103-116, ISBN : 2-7462-0887-3.

- [10] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien et Olivier Barais
Une extension de Fractal pour l'AOP
Première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2004), Paris, France, Septembre 2004 pages 18-31.

- [11] Olivier Barais, Alexis Muller et Nicolas Pessemier
Extension de Fractal pour le Support des Vues au sein d'une Architecture Logicielle
Objets Composants et Modèles dans l'ingénierie des SI (OCM-SI 04), Biarritz, France, Mai 2004.
<http://inforsid2004.univ-pau.fr/AtelierOCMv1.htm>, pages 1-7.

- [12] Olivier Barais et Laurence Duchien
OPAD : Outils pour architectures dynamiques
Journées Composants Adaptables, Grenoble, France, Octobre 2002, pages 112-118.

Première partie

État de l'art

Chapitre 2

Contexte de l'approche

PRÉAMBULE

*Ce chapitre présente en détail le contexte de l'approche. Il revient sur quelques grands principes du génie logiciel motivés par la **maîtrise de la complexité du développement** d'une application et la **réduction des coûts**. Inspiré par ces principes et les paradigmes récents du génie logiciel, ce chapitre dégage les caractéristiques essentielles que l'on souhaite trouver dans un langage de description d'architecture logicielle : l'abstraction par rapport au système modélisé, différents niveaux de structuration, la prise en compte du caractère évolutif d'une architecture logicielle, la capacité à construire son système de manière incrémentale et le besoin de garantir l'utilisation correcte d'une brique logicielle qui est réutilisée dans un contexte différent de celui pour laquelle elle a été conçue. En effet, dans un contexte où l'architecture logicielle devient le modèle pivot pour comprendre la structure d'une application, la capacité du modèle de description d'architecture à être en adéquation avec ces principes du génie logiciel conditionne la maîtrise de la complexité et des coûts d'un projet de développement d'application.*

Sommaire

2.1 Séparation des préoccupations	12
2.1.1 La modularité	13
2.1.2 L'abstraction	15
2.1.3 L'anticipation du changement	17
2.1.4 La construction incrémentale	19
2.2 La réutilisation	20
2.2.1 Conception par contrats	21
2.2.2 Taxonomie des contrats	22
2.2.3 Contractualisation des interactions entre composants	22
2.2.4 Contrats de qualité de service	23
2.2.5 Bilan sur la réutilisation	23
2.3 Architecture logicielle	24
2.3.1 Lien entre modèle de composant et architecture logicielle	24
2.3.2 Architectures orientées service ou architectures à base de composants ?	25
2.3.3 Conclusion	25
2.4 Défis de l'architecture logicielle	25
2.4.1 La modularité	25
2.4.2 L'abstraction	26
2.4.3 L'anticipation du changement	26
2.4.4 Une construction incrémentale : des composants métiers à l'intégration des préoccupations techniques	26

L'objectif du génie logiciel est, comme tout processus industriel tendant à se rationaliser, de produire du logiciel de qualité en minimisant les coûts.

La notion de « qualité » du logiciel est une notion subjective qui dépend beaucoup de la relation que l'on entretient avec le logiciel. Si du point de vue de l'utilisateur final, les premières motivations sont avant tout les fonctionnalités du logiciel, sa robustesse, sa performance et son ergonomie, du point de vue du concepteur d'une application, la qualité du logiciel est aussi associée à la capacité à **maîtriser la complexité** du logiciel en vue de produire un logiciel robuste, évolutif, maintenable, réutilisable et interopérable. Sans refaire l'histoire du génie logiciel, on peut dire que la notion de séparation des idées ou séparation des préoccupations (*separation of concerns*) a très tôt été considérée comme un outil essentiel pour maîtriser cette complexité³. Ce principe est une règle de bon sens qui consiste à considérer séparément différentes facettes d'un problème afin d'en maîtriser la complexité. C'est une application de la stratégie générale du « diviser pour régner ».

D'un autre côté, la **maîtrise des coûts** a motivé le besoin d'être capable de développer des applications toujours plus rapidement. Dans ce domaine, beaucoup voient dans la réutilisation un moyen d'améliorer la productivité relative au développement et à l'entretien des applications [Biggerstaff and Perlis, 1989, Lanergan and Grasso, 1989, Caldiera and Basili, 1991]. Si le principe de réutiliser des modules logiciels est admis, la mise en pratique reste encore difficile.

Ce chapitre décrit le contexte général de nos travaux. Dans les sections 2.1 et 2.2, nous présentons les cinq grands principes du génie logiciel qui ont inspiré notre approche. Ainsi, la section 2.1 illustre quatre de ces principes que l'on peut rattacher à la séparation des préoccupations : la modularité, l'abstraction, l'anticipation du changement et la construction incrémentale. La section 2.2 présente le cinquième de ces principes, la réutilisation. Ce principe diffère des quatre premiers, car il n'est pas motivé par la maîtrise de la complexité des applications produites, mais par l'amélioration des coûts de production de ces applications. La section 2.3 introduit les concepts essentiels de l'architecture logicielle. Enfin, la section 2.4 positionne l'architecture logicielle face aux grands principes du génie logiciel présentés dans les deux premières sections et dégage les caractéristiques essentielles d'un langage de description d'architecture moderne.

2.1 Séparation des préoccupations

La séparation des préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et prend donc une multitude de formes. Il peut s'agir de la séparation dans le temps qui concerne le traitement de la conception à la réalisation des différentes facettes du logiciel qui sont alors abordées successivement. L'ordre de ces facettes est décrit dans le processus de construction du logiciel ou dans une modélisation de son cycle de vie. Un autre type de séparation concerne la séparation des caractéristiques que l'on cherche à optimiser à un stade donné. Par exemple, certaines approches proposent d'assurer la correction de l'application produite avant de se soucier de son efficacité. Une troisième illustration du principe de séparation des préoccupations est la séparation des « vues » que l'on peut avoir d'un système. On se concentre, par exemple, sur la facette « flots de données » avant de considérer la facette liée à l'ordonnancement des opérations ou « flot de contrôle ». La structuration du système en parties permet, elle aussi, de séparer les préoccupations d'une application. Par exemple, on trouve de nombreuses applications basées sur un noyau et des extensions.

La séparation des préoccupations propose un support méthodologique dans la conception d'une application. Entre autres, elle aide à l'identification des facettes à l'aide du principe d'orthogonalité. Selon ce principe, toute préoccupation idéalement séparée du reste de l'application doit être orthogonale aux autres préoccupations de l'application, c'est-à-dire elle ne doit plus avoir de relations fortes avec les autres préoccupations de l'application. La séparation des préoccupations doit en outre être associée à un processus d'intégration de ces différentes préoccu-

³ « Décomposer les difficultés en autant de parcelles qu'il se peut pour mieux les résoudre. » Descartes - Deuxième principe, Le Discours de la méthode.

pations pour produire le système final. C'est un processus que l'on peut qualifier, en reprenant la terminologie de la programmation par aspects, de tissage.

Les processus de génie logiciel aident à organiser le travail en guidant la décomposition de la tâche de conception d'une application, l'ordonnancement des questions à traiter et la composition des solutions apportées à chacun de ces sous-problèmes. Si le principe de la séparation des préoccupations reste l'un des principes fondamentaux du génie logiciel, on le retrouve au cœur d'autres principes très souvent cités en génie logiciel [Ghezzi *et al.*, 1991] tels que :

- la modularité,
- l'abstraction,
- l'anticipation du changement,
- la construction incrémentale.

Nous présentons dans cette section ces quatre grands principes, en quoi ils sont liés au principe de la séparation des préoccupations et comment les grandes tendances actuelles adressent plus spécifiquement certains de ces principes. La section 2.2 adressera le problème de la réutilisation.

2.1.1 La modularité

2.1.1.1 Définition

Un système modulaire se caractérise par le fait d'être composé de sous-systèmes plus simples, ou *modules*. La modularité est une mise en œuvre concrète du principe de séparation des préoccupations au niveau de la structure du programme assurant le partitionnement d'un système complexe en sous-systèmes plus simples. Un bon découpage modulaire se caractérise par une forte cohésion interne des modules (fonctionnelle, logique) et un faible couplage entre les modules (relations inter-modulaires en nombre limité et clairement décrites). Cette structuration permet à des équipes différentes de réaliser ces sous-systèmes et de les composer ensuite. Outre la maîtrise de la complexité, la modularité vise aussi à améliorer la productivité en fournissant des éléments de structure réutilisables.

Le principe de modularité ne définit pas précisément la nature du contenu des modules (code, données, modèle, *etc.*). Il insiste sur le fait que pour permettre une bonne séparation des préoccupations, il est nécessaire d'offrir un ensemble d'éléments de structure simples pour fractionner son application.

Le principe de modularité a été un des principes les plus étudiés par la communauté du génie logiciel pour structurer les programmes, les données, *etc.* Les premiers travaux sur la modularité remontent à la proposition du langage Modula [Wirth, 1977] par Niklaus Wirth. Cette section présente deux mises en œuvre récentes du principe de modularité : la construction d'applications à l'aide de composants logiciels et le développement par aspects.

2.1.1.2 Mise en œuvre de la modularité : l'approche composant

L'une des approches actuelles de la modularité est la notion de composant logiciel [Heineman and Council, 2001]. Paradigme introduit comme une suite à la notion d'objet, le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques.

Une des difficultés dans l'approche composant provient souvent de l'utilisation assez confuse du mot *composant*. En effet, la définition du concept de composant varie grandement en fonction du contexte d'utilisation et de la communauté scientifique qui l'utilise. Il est alors souvent difficile de savoir si nous parlons d'instances ou de types de composant, de binaires qui se déploient, qui se composent ou s'assemblent.

En outre, le terme de composant logiciel a été utilisé avec des sens parfois très vagues. Par exemple, Shaw et Garlan ont défini la notion de composant comme une unité de traitement [Garlan and Shaw, 1993], ce qui implique que tout élément logiciel est un composant logiciel. Brown et Short définissent le composant comme une unité indépendante qui fournit un ensemble de services réutilisables [Brown and Wallnan, 1996], ce qui par contre, n'implique pas la propriété de composition. Ce manque de précision a gêné l'établissement d'un discours précis autour des composants, les détracteurs jugeant que l'approche ne faisait que réinventer la roue.

Cependant, différentes propriétés permettent de caractériser le composant. La définition de Szypersky, globalement acceptée, illustre bien ces propriétés : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* » [Szyperski, 1998].

Cette définition met en lumière trois propriétés importantes du composant :

- Un composant est sujet à composition. Ce point met en avant la volonté de transformer l'ingénierie du logiciel en un modèle de Lego™ [Batory and O'Malley, 1992]. Dans ce modèle, une application est construite à partir d'un assemblage de composants de manière similaire à un assemblage de briques Lego.
- Un composant peut être déployé indépendamment. Cette affirmation définit en fait deux propriétés du composant. La première réside dans la prise en compte dans les modèles de composant de la tâche de déploiement. La deuxième rappelle le nécessaire découplage entre les composants. Contrairement à l'objet, le composant n'introduit pas de mécanisme d'héritage et cherche à minimiser les dépendances entre composants.
- Un composant explicite de manière contractuelle les services qu'il peut rendre et ses dépendances vis-à-vis de l'environnement. Il propose dans un langage défini les services et leurs règles de fonctionnement de façon à aider leur réutilisation au sein de différents assemblages.

Enfin, un dernier point, peu pris en compte dans cette définition et propice à engendrer des problèmes de compréhension, concerne le cycle de vie d'une application. En effet, le terme « composant logiciel » est utilisé depuis la phase de conception jusqu'à la phase d'exécution d'une application, il prend donc plusieurs formes. Dans ce document, nous parlerons de *type de composant* pour les informations relatives à la description des services requis et fournis par le composant. Nous appellerons *implantation de composant*, l'ensemble des sources et/ou du binaire permettant la réalisation des services. Enfin, nous parlerons d'*instance de composant* pour une entité existante et s'exécutant dans un système. Ainsi, une instance de composant est déployée et elle est caractérisée par une référence unique, un type de composant et une implantation particulière de ce type.

2.1.1.3 Mise en œuvre de la modularité : le développement d'application par aspects

Si les objets et les composants ont permis d'apporter de réelles avancées dans la construction de systèmes informatiques modulaires, il apparaît que ces approches proposent une unique dimension de structuration qui ne peut modulariser l'ensemble des préoccupations d'une application. En effet, quand l'application est décomposée à partir d'une unique dimension qui est la structuration du flot de données, il existe toujours des préoccupations dites transverses, qui ne peuvent pas être facilement modularisées. C'est le cas par exemple des services techniques (trace, persistance, etc.) qui sont utilisés par plusieurs modules. C'est ce que l'on appelle le phénomène de dispersion du code.

Dès lors, deux problèmes se posent :

- le code relatif à ces préoccupations se retrouve noyé dans l'ensemble de l'application. Il est par conséquent très difficile de les faire évoluer,
- le code relatif à ces préoccupations vient « polluer » les modules dans lesquels le code est noyé, diminuant ainsi la cohérence logique de ces modules et nuisant, par conséquent, à leur maintenance et à leur évolutivité.

L'objectif essentiel du développement par aspects est donc de fournir un nouveau paradigme de structuration permettant de regrouper dans une même entité du code qui serait autrement dispersé au sein d'une application. Un aspect peut donc se définir comme une *entité logicielle qui capture une fonctionnalité transversale à une application* [Pawlak et al., 2004]. L'aspect est toujours une nouvelle dimension de structuration ajoutée à une autre technique de structuration existante (l'objet, le composant, etc.).

Le domaine de l'AOP (*Aspect-Oriented Programming*) est issu des travaux de Gregor Kiczales et son équipe du centre de recherche Xerox Parc de Palo Alto en Californie en 1996 [Kiczales, 1996]. Le concept de la programmation par aspects a été mis en œuvre entre autres dans le langage AspectJ [AspectJ Team, 2002 2003].

2.1.1.4 Bilan sur le principe de modularité

Le principe de modularité permet de structurer une application en différentes entités. Cependant, la plupart des approches proposent une unique dimension de structuration d'une application autour des fonctionnalités. Cette unique dimension de structuration qui se révèle plus simple pour la composition de ces modules, pose dans en fait des problèmes de modularisation. Certaines préoccupations transverses à l'application ne peuvent pas, dans de telles approches, être clairement structurées dans des entités modulaires [Tarr *et al.*, 1999]. La programmation par aspects offre une première réponse grâce à l'introduction d'une deuxième dimension de structuration d'une application autour de la notion de préoccupation. Poussé à l'extrême, le principe de modularité devrait permettre de structurer son application selon différentes dimensions (fonctions, préoccupations, *etc.*).

2.1.2 L'abstraction

2.1.2.1 Définition

Le deuxième principe utilisé pour maîtriser la complexité de la conception de logiciels consiste à augmenter l'abstraction avec laquelle on aborde un problème. Le principe d'abstraction ne considère ainsi que les informations d'un système jugées importantes à un moment donné, en omettant les autres informations. Une même réalité peut souvent être décrite à différents niveaux d'abstraction. Par exemple, une application peut être décrite par un modèle mathématique abstrait (équation logique) et programmée dans un langage objet comme Java qui fait abstraction des caractéristiques du système d'exploitation et de l'infrastructure matérielle sur laquelle fonctionne l'application.

La caractéristique essentielle d'une abstraction est d'omettre un ensemble d'informations non directement lié au problème en cours de résolution. Par exemple, si l'on s'intéresse à la structure d'une application, il est gênant que cette information se retrouve noyée dans le code de l'application. Une bonne abstraction se caractérise par le fait de n'adresser qu'une facette de la construction d'une application. Cette facette doit être suffisamment orthogonale aux autres afin de pouvoir être isolée et composée.

Nous étudions dans cette section deux approches récentes du génie logiciel qui mettent en œuvre le principe d'abstraction : les langages dédiés et l'ingénierie dirigée par les modèles.

2.1.2.2 Mise en œuvre du principe d'abstraction : les langages dédiés

Un langage dédié (DSL pour *domain-specific language*) [van Deursen *et al.*, 2000, Consel and Marlet, 1998] est un langage de programmation conçu pour être particulièrement adapté à un domaine ou un champ d'application particulier. Un DSL utilise une notation ou une abstraction formelle, puissante et complète. Il existe actuellement de nombreux DSLs comme par exemple GraphViz [Gansner and North, 2000], un langage utilisé pour dessiner des graphes interconnectés, ou encore Bossa [Barreto, 2003], un langage pour la définition de politiques d'ordonnement dans un noyau Linux.

Ces dernières années, les DSLs ont été étudiés dans le domaine du génie logiciel afin d'améliorer la productivité et la qualité. Ils fournissent grâce à leur concision des langages à fort pouvoir d'expression. Leur précision sémantique permet de mettre en œuvre des analyses précises sur les programmes exprimés à l'aide du DSL.

Les avantages associés à la construction de langages dédiés sont nombreux. Ils manipulent uniquement les concepts du domaine visé. Par conséquent, les experts du domaine peuvent comprendre, valider, modifier, et même souvent développer eux-mêmes des programmes avec ces DSLs. Ces langages permettent la validation au niveau du

domaine visé et lève ainsi des messages d'erreur de haut niveau cohérents avec les entités manipulées par l'expert du domaine. Dès lors que les constructions du langage dédié sont déclarées sûres, n'importe quel programme écrit avec ces constructions peut être analysé et déclaré conforme à un certain degré de sûreté.

Néanmoins, la conception, la mise en œuvre et la maintenance d'un DSL peuvent se révéler des opérations onéreuses. La détermination du domaine visé, c'est-à-dire de les abstractions retenues, est une opération délicate et stratégique pour la pertinence du futur DSL. Le bon équilibre entre le domaine visé et les constructions classiques des langages de programmation généralistes est souvent difficile à trouver. Une appréhension classique par rapport aux approches à base de génération de code concerne la perte potentielle d'efficacité en comparaison avec la production du même logiciel codé à la main. Cette dernière remarque faite sur la plupart des compilateurs ou générateur de code est, quant à elle, très discutable étant donné le nombre d'études prouvant qu'en moyenne le code généré par ce type d'approche est au moins aussi efficace et surtout plus sûr qu'un logiciel codé à la main.

Les DSLs sont une très bonne illustration de l'application du principe d'abstraction pour maîtriser la complexité d'un domaine particulier. Ils proposent de maîtriser la complexité à l'aide d'une séparation par domaine où chaque domaine est muni d'un langage abstrait pour les informations liées à ce domaine. La composition de DSLs reste pour le moment un travail encore difficile et peu abordé.

2.1.2.3 Mise en œuvre du principe d'abstraction : l'ingénierie dirigée par les modèles

L'Ingénierie Dirigée par les Modèles (IDM) est une autre tendance récente du génie logiciel. Mise en avant par l'OMG (*Object Management Group*) en novembre 2000 avec l'introduction du concept de MDA (*Model Driven Architecture*) [Soley and the OMG staff, 2000], l'ingénierie dirigée par les modèles devait au départ s'appuyer sur UML pour décrire les parties métiers d'une application indépendamment des parties liées aux plates-formes. Toujours avec comme objectif premier de maîtriser la complexité importante des systèmes d'information, l'IDM consiste à se servir des *modèles* comme unité d'abstraction par rapport au système. L'abstraction par rapport à la plate-forme technologique offre alors l'avantage de la pérennité face à l'évolution technologique et permet ainsi de capitaliser le côté métier d'une application.

L'approche MDA de l'OMG préconise le développement de différents méta-modèles à l'aide d'un langage commun le MOF [OMG, 2002b], chaque méta-modèle permettant de définir des modèles liés aux préoccupations de son application. La composition de ses méta-modèles est assurée à l'aide de mécanismes de transformation spécifiés à l'aide du langage QVT [OMG, 2003a].

L'ingénierie dirigée par les modèles a maintenant largement dépassé la cadre de l'OMG [Estublier *et al.*, 2005a]. Ainsi, Microsoft fournit grâce à son outil « *Software Factories* » [Greenfield *et al.*, 2004] un environnement de construction de DSLs de petite taille, facilement manipulables, transformables, combinables. IBM, pour sa part, propose une alternative au MOF très outillée autour d'EMF [Budinsky *et al.*, 2003]. Dans ces deux approches, l'apport des modèles comme entités fournissant une capacité d'abstraction importante est indéniable. Si ce principe d'abstraction n'est pas une idée récente dans le domaine du génie logiciel, le placement de ces abstractions au cœur du processus de développement logiciel est en lui-même une révolution dans le domaine du génie logiciel.

2.1.2.4 Bilan sur le principe d'abstraction

Le principe d'abstraction offre un mécanisme efficace et attrayant pour maîtriser la complexité d'une application en fournissant une représentation de haut niveau de l'application pérenne et compréhensible par l'ensemble des acteurs d'un projet informatique. La construction de modèle métier à l'aide de techniques de méta-modélisation ou de définition de langages spécifiques à un domaine tend à se généraliser. Cette approche offre naturellement un mécanisme de structuration du développement centré autour des différents modèles. Par leur abstraction vis-à-vis du système d'information, ils permettent d'améliorer la pérennité de certaines facettes d'une application en les dégageant de l'obsolescence des plates-formes technologiques. Cependant, les mécanismes de composition

des langages métier ou des modèles restent un enjeu crucial encore peu abordé. La description de cette composition vise à permettre la cohabitation efficace de différents langages dédiés dans le développement d'une application [Estublier *et al.*, 2005b].

2.1.3 L'anticipation du changement

2.1.3.1 Définition

Le troisième principe important dans le génie logiciel est l'anticipation du changement. Un logiciel a comme caractéristique essentielle d'être continuellement soumis à des changements. Nous considérons dans nos travaux deux types de changement : les *évolutions internes* de la structure de l'application au cours de son exécution, c'est-à-dire sa dynamique et les *évolutions externes* d'une application pouvant être liées à des corrections d'imperfections, mais étant plus généralement liées à l'évolution des besoins des utilisateurs finaux. Les évolutions internes caractérisent le fait qu'une application peut, de manière autonome, changer en cours de fonctionnement pour s'adapter à son contexte d'exécution. Les évolutions externes considèrent les modifications apportées à une application pour y ajouter de nouvelles préoccupations. Ces modifications, qui concernent la phase de conception d'une application, caractérisent sa capacité à être construite par intégrations successives de nouvelles préoccupations. Ainsi, les propriétés requises d'une anticipation au changement sont :

- la prise en compte de la dynamique d'une application. Elle se concentre sur les modifications apportées à la structure de l'application de manière autonome pour prendre en charge les évolutions de son contexte d'exécution,
- la maîtrise de l'impact d'un changement. Cette caractéristique est alors souvent étroitement liée au caractère modulaire de l'application.

Nous proposons dans cette section, l'étude de deux mises en œuvre du principe d'anticipation du changement permettant la prise en compte des évolutions externes : les *design patterns* et l'inversion des dépendances. Les *design patterns* ou motifs de conception permettent de limiter le couplage entre les éléments constitutifs d'une application afin de limiter l'impact d'un changement. La programmation par aspects se base sur un principe d'inversion des dépendances qui permet, pour sa part, de diminuer le couplage entre le code technique et le code métier favorisant l'évolution de l'un sans impact sur l'autre.

2.1.3.2 Mise en œuvre de l'anticipation des changements : les *design patterns*

Une des solutions pour permettre une bonne anticipation des changements consiste à offrir une modularité « correcte », c'est-à-dire une forte cohésion interne des modules et un faible couplage entre ces modules. Ce faible couplage permet de conscrire l'évolution dans un minimum de modules et limite l'impact des changements d'un module sur son environnement.

Les motifs de conception, appelés plus généralement *design patterns* et largement utilisés dans la conception par objets, sont des formalisations de bonnes pratiques, validées par l'expérience, permettant de résoudre des problèmes récurrents dans la conception de systèmes [Gamma *et al.*, 1995]. L'objectif des motifs de conceptions est de concevoir l'application en prenant en compte son évolution. Pour cela, les motifs de conception offrent des mécanismes pour minimiser les interactions entre les différentes classes d'un même programme. Ils contribuent à la bonne modularité de l'application. Ils fournissent aussi différents moyens d'étendre une application sans utiliser directement les mécanismes d'héritage qui sont coûteux et qui introduisent un couplage fort entre une classe de base et ses classes dérivées.

De nombreuses autres habitudes de conception introduisent un couplage fort entre différentes classes d'une application et limite par conséquent les possibilités d'extensions de ce système. Par exemple, la création d'un objet en référençant sa classe explicitement engendre une dépendance forte avec une mise en œuvre particulière. Pour diminuer ce couplage, les motifs de conception proposent l'utilisation d'une fabrique abstraite (*AbstractFactory*) à qui l'on passe un paramètre à la création qui définit l'objet que l'on va créer. Un autre exemple considère

le couplage fort introduit par la dépendance engendrée par l'appel à une méthode spécifique. Le motif « chaîne de conception » permet alors de déléguer le traitement à une chaîne d'objets par l'intermédiaire d'une interface générique ; il sera alors beaucoup plus facile d'introduire un nouveau type d'objet pouvant prendre en charge le traitement. Ainsi, d'une manière générale, les motifs de conception *AbstractFactory*, *Bridge*, *Chain Of Responsibility*, *Command*, *Facade*, *Mediator* ou *Observer* sont quelques exemples de bonnes pratiques permettant de limiter le couplage entre entités logicielles et donc de faciliter l'évolution d'une application⁴.

Lors de l'évolution d'un système, il peut être nécessaire d'étendre les classes ou les modules d'un système. Là aussi, les motifs de conception *Bridge*, *Chain Of Responsibility*, *Composite*, *Decorator*, *Observer*, *Strategy* ou *Proxy* offrent des solutions élégantes et plus efficace que l'héritage pour étendre le comportement d'un objet. De même, l'extension d'une classe peut dans certains cas se révéler tout simplement impossible du fait d'une absence de source, d'un trop grand nombre de répercussions sur le système ou d'un langage offrant uniquement l'héritage simple. Là encore, les motifs de conceptions *Adapter*, *Decorator*, *Visitor* offrent une formalisation de bonnes pratiques permettant de ne pas utiliser l'héritage et de minimiser le couplage entre les objets .

2.1.3.3 Mise en œuvre de l'anticipation des changements : l'inversion des dépendances

Afin de limiter l'impact d'un changement, le principe d'anticipation des changements est aussi décliné en un principe appelé l'*inversion des dépendances* [Martin, 1996]. Ce principe part du constat que dans la plupart des applications, les modules qui portent la logique fonctionnelle de l'application, appelée aussi logique « métier », sont construits directement au dessus des modules de bas niveau appelés aussi modules « techniques ». Si de telles constructions paraissent naturelles au premier abord, elles posent en réalité deux problèmes essentiels :

- les modules de haut niveau doivent être modifiés lorsque les modules de bas niveau sont modifiés,
- il n'est pas possible de réutiliser les modules de haut niveau indépendamment de ceux de bas niveau. En d'autres termes, il n'est pas possible de réutiliser la logique d'une application en dehors du contexte technique dans lequel elle a été développée.

Pour limiter l'impact de l'évolution d'un module logiciel par rapport à son environnement, l'inversion des dépendances permet de diminuer le couplage entre les modules métier et les modules techniques. Selon ce principe, la relation de dépendance doit être inversée : les modules de bas niveau doivent être conformés à des interfaces définies et utilisées par les modules de haut niveau.

Ce principe conduit à des applications dans lesquelles la logique « métier » est parfaitement réutilisable dans des contextes techniques différents.

Si le problème de l'inversion des dépendances propose de ne pas utiliser du code dit technique depuis du code dit métier afin de ne pas nuire à l'évolutivité, la maintenance et à la réutilisation du code métier d'une application, une approche naïve proposant uniquement d'appeler des parties de code métier depuis des modules techniques pose évidemment les mêmes difficultés. Le problème consiste donc à être capable de structurer son application sans introduire une dépendance forte du code métier vers le code technique ou du code technique vers le code métier.

L'aspect fournit une bonne illustration de l'intérêt de l'inversion des dépendances pour anticiper le changement dans le développement d'une application. En effet, l'aspect ne prend pas en charge le service technique, il l'utilise et l'intègre dans l'application comme le présente la figure 2.1. Il permet de supprimer les dépendances directes entre les modules métiers et les services techniques. Dès lors, une évolution de la couche métier engendrera une modification de la façon dont l'aspect y est intégré. Une modification d'un service technique engendrera uniquement l'évolution de l'aspect qui utilise ce service technique.

⁴Ces motifs de conceptions sont présentés dans l'ouvrage [Gamma *et al.*, 1995]

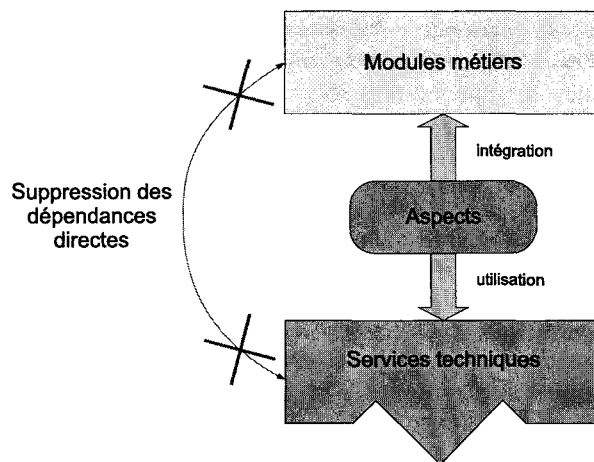


FIG. 2.1 – Inversion des dépendances entre les modules métiers et les services techniques

2.1.3.4 Bilan sur l'anticipation du changement

L'anticipation du changement est un des principes de conception qui est généralement le moins pris en compte dans la construction d'applications. Le développement étant conditionné par les exigences des utilisateurs, il est souvent difficile de ne pas considérer uniquement ces exigences comme critères d'évaluation de l'avancement d'un projet informatique. Cependant, la prise en compte dès la conception du caractère évolutif d'une application est un critère très important pour la pérennité d'une application. En outre, cette prise en compte au plus tôt dans la phase de conception permet de maîtriser la complexité associée à l'apparition de nouvelles exigences.

2.1.4 La construction incrémentale

2.1.4.1 Définition

Le quatrième principe du génie logiciel lié à la séparation des préoccupations est le principe de la construction incrémentale. Ce principe propose de ne pas construire l'ensemble d'une application en une seule étape, mais de ne s'occuper que d'une difficulté à la fois. Un procédé incrémental atteint son but par étapes en s'en approchant de plus en plus ; chaque résultat est construit en étendant le précédent.

2.1.4.2 Mise en œuvre

Le principe de construction incrémentale est généralement associé à la notion de « raffinement »⁵. Dans le domaine du génie logiciel, on considère généralement deux types de raffinement : le raffinement d'une application par ajout de fonctionnalités ou le raffinement sans modification des fonctionnalités de l'application.

Le raffinement par ajout de fonctionnalités consiste à modifier le code, les modèles ou tout autre artefact associé à une application afin d'intégrer de nouvelles fonctionnalités. Le développement d'une application à l'aide du Processus Unifié (UP pour *Unified Process*) [Kruchten, 2000] est un bon exemple de démarche de conception basée sur le raffinement de fonctionnalités. En effet, avec le processus unifié, pour réduire la complexité en la maîtrisant, un projet informatique se trouve décomposé en sous projets qui représentent chacun une *itération* qui donne lieu à un *incrément*. Les itérations désignent des étapes de l'enchaînement d'activités, tandis que les incréments correspondent à des stades de développement du produit. Le développement d'une application à l'aide

⁵ dans le sens étymologique du terme qui consiste à rendre plus fin ou plus précis

du processus unifié est piloté à l'aide des cas d'utilisation. Une itération prend en compte un certain nombre de ces cas d'utilisation décrivant chacun une fonctionnalité de l'application. Les itérations successives exploitent les livrables de développement (code, modèles, documentation) dans l'état où les a laissés l'itération précédente et les enrichissent progressivement. Un ensemble d'itérations est regroupé dans une phase. Une phase est ponctué par un jalon qui marque la décision que les objectifs ont été remplis.

Au contraire, le Model Driven Architecture (MDA) propose une démarche de conception d'une application basée sur le raffinement de modèles par ajout de nouvelles propriétés sans intégration de nouvelles fonctionnalités. Elle permet de séparer les spécifications fonctionnelles d'un système des spécifications liées à la mise en œuvre sur une plate-forme donnée. Le MDA définit une architecture de spécifications structurée en modèles indépendants des plates-formes (PIM pour *Platform Independent Model*) et en modèles spécifiques (PSM pour *Platform Specific Model*). La démarche MDA permet de réaliser le même modèle sur plusieurs plates-formes. Elle supporte l'évolution des technologies liées aux plates-formes d'exécution en définissant le modèle métier de manière indépendamment de toute plate-forme technique.

Ces deux démarches de conception illustrent l'intérêt de construire une application de manière incrémentale.

2.1.4.3 Bilan sur le principe de construction incrémentale

La construction incrémentale est le principe le plus complémentaire à la notion de séparation des préoccupations. Elle met en avant le fait que quel que soit le critère choisi pour guider la décomposition du système, celui-ci résulte d'un processus de raffinement : soit un raffinement de fonctionnalités qui correspond à la composition des différents sous-systèmes identifiés par la démarche de conception, soit un raffinement de propriétés visant à enrichir une spécification existante avec de nouvelles informations par exemple celles liées aux choix de la plate-forme technologique de mise en œuvre.

2.2 La réutilisation

Le deuxième objectif fondamental du génie logiciel consiste à réduire toujours davantage les coûts de production. La réutilisation est un moyen efficace pour diminuer ces coûts. Cependant, si l'attrait économique est évident, la réutilisation peut parfois nuire à la qualité de l'application. L'exemple du premier décollage manqué du lanceur Ariane 5 est pour cela tristement célèbre [Jézéquel and Meyer, 1997]. Une plate-forme de référence inertielle, achetée clés en main (le matériel comme le logiciel), et servant notamment à orienter la fusée, qui avait déjà fait 23 fois ses preuves sur Ariane 4 a dysfonctionné causant l'autodestruction de la fusée trente-neuf secondes après son décollage. L'analyse des causes de l'accident a montré que cet échec qui coûta la bagatelle de 5 millions d'euros n'était ni un problème de gestion de projets, ni un problème de choix de langage de programmation, ni un problème de conception, ni un problème de mise en œuvre, ni un problème de test tant l'espace de test pour ce genre de projet est immense, mais bien un problème de réutilisation, le composant issu d'Ariane 4 ayant une contrainte « cachée ».

Ce cas d'étude a montré que la réutilisation et l'intégration d'un composant logiciel existant impliquent la connaissance du rôle du composant au sein d'un assemblage, c'est-à-dire la façon dont il va interagir avec son environnement et la façon dont son environnement peut interagir avec lui. Ainsi pour réutiliser un composant, il est donc nécessaire de lui associer des spécifications précises de son interface. Cette spécification permet de définir de manière contractuelle les services qu'il peut rendre et ses dépendances vis-à-vis de l'environnement.

La section 2.2.1 présente l'approche « conception par contrats » qui a proposée l'utilisation de la notion de contrat pour favoriser la réutilisation de composants logiciels. Nous décrivons dans la section 2.2.2 la taxonomie de Beugnard *et al* qui structure l'ensemble des informations que l'on peut trouver au niveau de l'interface d'un composant. Nous revenons enfin dans les sections 2.2.3 et 2.2.4 sur les niveaux trois et quatre de cette taxonomie. Elle propose une généralisation de la notion de contrat d'interface en y ajoutant les informations concernant les interactions d'un composant avec son environnement et la qualité de service.

2.2.1 Conception par contrats

La conception par contrats (DbC pour *Design by Contrat*) [Meyer, 1992a] est une méthode de conception logicielle. Elle est basée sur la théorie des types de données abstraits [Ruane, 1984] et sur la métaphore du contrat au sens juridique qui lie deux parties. Le DbC met en avant l'intérêt de spécifier précisément les interfaces d'un module logiciel en termes de pré-conditions, post-conditions et invariants.

Si le terme DbC a été proposé par Bertrand Meyer en 1992 [Meyer, 1992a], les premiers travaux sur les contrats remontent à Floyd [Floyd, 1967] et Hoare [Hoare, 1969]. Ces travaux montrent l'intérêt de l'utilisation des assertions dans la spécification de programmes. Une assertion est définie comme une expression booléenne qui doit être vraie pour qu'un code spécifié s'exécute correctement. Reprises dans la spécification des interfaces de classe ou de composant, les assertions permettent le raffinement de contraintes de typage. D'un point de vue conceptuel, les contrats vont au-delà des mécanismes d'assertions. Ces dernières ne sont qu'un mécanisme de base qui n'a que peu en commun avec les aspects méthodologiques et l'intégration au modèle objet qu'offrent les contrats. Cependant, plusieurs langages, qui ne proposent pas de programmation par contrats dans leur version de base, utilisent le mécanisme d'assertion, lorsqu'il est disponible, pour mettre en oeuvre les contrats avec des bibliothèques, des préprocesseurs ou d'autres moyens spécialisés.

L'idée centrale du DbC est que les entités logicielles ont des responsabilités envers les autres entités avec lesquelles elles interagissent. Ces responsabilités sont alors basées sur des règles formalisées entre ces entités. Des spécifications fonctionnelles, ou « contrats », sont créées pour chaque module dans le système avant que ces modules ne soient codés. Les interactions entre les divers modules de l'application sont alors bornées par ces contrats.

Le travail le plus connu dans ce domaine est probablement celui autour d'Eiffel [Meyer, 1992b]. Eiffel utilise des assertions dans des pré et post-conditions et dans des invariants de classes pour décrire des contrats entre l'utilisateur et une classe. D'autres approches plus récentes comme le langage OCL [OMG, 2003b] pour UML ou JML [Leavens *et al.*, 2000] pour Java fournissent maintenant des mises en oeuvre étendues de la conception par contrats.

L'utilisation des concepts du DbC dans le développement d'une application entraîne de très nombreux bénéfices :

- possibilité d'outillage : grâce à une formalisation précise des langages de contrats, il est possible d'outiller convenablement l'approche afin de garantir statiquement certaines propriétés, de générer du code pour garantir à l'exécution le respect des contrats,
- documentation : le contrat est une documentation naturelle, car il répond à la question « *qu'est-ce que cet élément requérant un service est supposé faire ?* ». Plutôt que de se contenter de l'interface, on peut donc utiliser l'interface et le contrat (invariants et pre/post-conditions) comme élément de base de la documentation d'un élément logiciel. Ceci est utile lors des spécifications, mais aussi lors de la génération automatique de la documentation à partir du code et de ses commentaires, comme le permettent des outils disponibles dans de nombreux langages,
- une meilleure séparation des préoccupations : si les approches par contrats cherchent avant tout à favoriser la réutilisation de modules logiciels, la déclaration explicite d'un certain nombre d'informations relatives à l'interaction permet de clarifier le code des méthodes. Ce code contient alors uniquement le code fonctionnel lié aux méthodes et n'est pas envahi par des parties de code vérifiant la correction des paramètres envoyés ou retournés au cours d'une interaction.

Les approches par composants visent à favoriser la réutilisation d'entités logicielles. Dans ce sens, il est très intéressant de porter ces concepts au niveau des interfaces de composant. Cependant, la conception par contrats a, à l'origine, été développée pour les approches de conception ou de programmation par objets. Dans les approches à composant, l'hétérogénéité des composants et des infrastructures rend l'évaluation des contrats à l'exécution plus difficile. Les systèmes peuvent souvent être composés à l'exécution, ce qui impose de conserver les tests tout au long du cycle de vie du composant. Finalement, les spécifications des interfaces des composants sont généralement plus fermées qu'une spécification objet, il n'est pas toujours possible d'avoir accès aux champs caractérisant l'état du composant. Par conséquent, si la notion de contrat permet d'améliorer la réutilisabilité d'un module logiciel, on

peut s'interroger sur les opportunités de transposition dans le monde des composants ? Différents travaux comme ceux de Collet *et al* [Collet *et al.*, 2005] ou les travaux autour du modèle de composant du projet RNTL ACCORD [Legond-Aubry *et al.*, 2003, Legond-Aubry, 2005] ont déjà exploré cette piste.

2.2.2 Taxonomie des contrats

La notion de conception par contrat introduit une première catégorie d'informations qui peut être représentée au niveau de l'interface d'un module logiciel. Beugnard *et al* [Beugnard *et al.*, 1999] proposent une taxonomie plus générale qui identifie quatre niveaux de contrat qui sont :

- *basique* : les propriétés syntaxiques des noms des méthodes et le type des paramètres, des exceptions et des retours (par exemple, les langages de définition d'interfaces avec les IDL),
- *comportemental* : les propriétés qui peuvent être spécifiées avec des pré-conditions, des post-conditions et des invariants. Ces propriétés sont liées à une méthode, qui est vue comme une unité atomique. Elles précisent le typage des paramètres et les éventuelles dépendances entre les valeurs de ces paramètres. Les travaux sur la conception par contrats se placent à ce niveau,
- *synchronisation* : il s'agit ici des propriétés concernant les interactions des composants. Ces propriétés ne peuvent pas s'exprimer uniquement à l'aide de pré et post-conditions sur une méthode. Elles correspondent à la prise en compte des interactions du composant avec son environnement, ces interactions influençant le comportement interne du composant. On peut décrire leur exécution sous forme de suite d'étapes. Le principe de l'atomicité, ici, ne peut être conservé, l'indéterminisme des appels et la concurrence de l'exécution des opérations est à prendre en compte dans la description de ces propriétés,
- *quantitatif* : toutes les propriétés non fonctionnelles, telles que la qualité de service, la gestion des ressources ou le temps de réponse correspondent au quatrième niveau de contrat. On parlera ici de contrats de Qualité de Service. Ces derniers sont les plus complexes à exprimer et à analyser, car le plus souvent dépendant des propriétés que l'on cherche à vérifier, de l'environnement et du contexte d'exécution du composant qui ne sont pas, a priori, connus au moment de la description de l'architecture.

Nous revenons dans les sections 2.2.3 et 2.2.4 sur les deux derniers niveaux de cette taxonomie. Le premier niveau n'est pas présenté car il correspond à un système de typage de données que l'on trouve dans la plupart des langages de programmation. Le deuxième niveau regroupe, quant à lui, les éléments présentés dans la section 2.2.1.

2.2.3 Contractualisation des interactions entre composants

La conception par contrat permet de préciser les types de données en définissant une partie de leur mise en œuvre ou simplement en faisant apparaître des contraintes masquées liées à cette mise en œuvre (voir Section 2.2.1). Cependant, cette approche ne précise pas l'ordonnancement des messages au sein d'un composant et ne détecte pas les problèmes de synchronisation qui peuvent en résulter.

Toujours dans un objectif d'améliorer la réutilisation des composants, il est nécessaire de connaître le plus précisément possible l'interface du composant. En effet, les composants logiciels sont des processus qui s'exécutent simultanément et s'échangent des informations. Ils sont souvent distribués sur plusieurs machines et ne possèdent pas d'horloge globale. Chaque composant est alors libre d'évoluer à son propre rythme, mais doit se synchroniser avec d'autres composants lorsqu'il veut communiquer, on parle de parallélisme asynchrone [Garavel, 2003].

Cet asynchronisme pose de grosses difficultés pour la réutilisation de composants. En effet, un système construit par assemblage de sous-systèmes plus simples et prouvés corrects n'est pas nécessairement correct. Par exemple, l'interaction entre deux composants qui ne comportent pas de blocage peut cependant créer un interblocage. C'est le cas de deux composants qui se mettent tous les deux en attente d'un message de l'autre partie. Frein énorme pour la réutilisation de composants logiciels, il est nécessaire de préciser l'interface des composants avec

un contrat de niveau trois d'après la taxonomie proposée par Beugnard *et al* afin de définir les caractéristiques essentielles liées à l'asynchronisme des composants et ainsi garantir la correction de leur assemblage.

Ce problème de la conception de systèmes complexes parallèles et asynchrones n'est pas un problème réservé uniquement à la conception d'applications par assemblage de composants, il se retrouve dans le domaine des télécommunications pour la conception de nouveaux protocoles, dans le domaine du matériel pour la conception d'architectures multiprocesseurs (arbitrage de bus, cohérence de caches.) De ce fait, de très nombreux formalismes ont été proposés allant des modèles théoriques comme les réseaux de Petri [Peterson, 1981] ou les automates communicants [Harel, 1987, Lynch and Tuttle, 1989]), en passant par les algèbres des processus (CCS [Milner, 1980], CSP [Hoare, 1985], LOTOS [ISO, 1989] ou FSP [Magee and Kramer, 1999]) ou les langages dédiés (SDL [ITU, 1999a] dans le domaine des télécommunications ou systemC [Syn, 2002] dans la conception de matériel).

L'intérêt d'enrichir la description des interfaces de composant avec une description précise du comportement est un des moyens de détecter les problèmes de synchronisation. Cette information contribue fortement à la réutilisation d'un composant en fournissant des informations de haut niveau sur le fonctionnement du composant.

2.2.4 Contrats de qualité de service

Les trois premiers niveaux de contrat définissent l'interface d'un composant d'un point de vue fonctionnel. Cependant, d'autres propriétés, plus quantitatives, peuvent permettre de caractériser un composant et son fonctionnement au sein d'un assemblage. Par exemple, si l'on prend un composant qui distribue du contenu multimédia, la latence, la qualité du résultat ou la synchronisation entre l'image et le son sont des caractéristiques qui affectent le degré de satisfaction d'un utilisateur de ce service. Ces caractéristiques non fonctionnelles sont généralement associées à la notion de qualité de service.

Il existe maintenant de nombreux travaux pour la spécification de propriétés de qualité de service au niveau de l'interface des composants [Aagedal, 2001] comme SMIL (*Synchronized Multimedia Integration Language*) [Hoschka, 1998], TINA Object Definition Language [Mercouroff and Parhar, 1997] une extension de CORBA IDL pour la spécification de la qualité de service au niveau des interfaces CORBA, ou le plus connu QML (*Quality of service Modeling Language*) [Frølund and Koistinen, 1998].

L'expression de la qualité de service au niveau de l'interface d'un composant est rendue difficile généralement pour deux raisons. Premièrement, la qualité de service regroupe un nombre de facettes de l'application qu'il est difficile de borner. En effet, la synchronisation, le temps d'exécution ou la latence sont autant de facettes de la qualité de service d'une application. En outre, les propriétés liées à la qualité de service d'un composant sont généralement difficiles à exprimer au niveau de l'interface d'un composant. Soit ces propriétés sont fortement dépendantes de la mise en œuvre du composant. Il est alors nécessaire de pouvoir extraire les informations de la mise en œuvre pertinentes pour une propriété de QoS donnée. Soit ces propriétés sont fortement dépendantes de l'environnement d'exécution, là aussi il est alors difficile d'attacher un contrat de qualité de service à une interface de composant si ce composant n'est pas défini dans un contexte d'utilisation particulier.

2.2.5 Bilan sur la réutilisation

Il est nécessaire, pour permettre la réutilisation de composants de type boîte noire, de préciser l'interface du composant d'un point de vue syntaxique et sémantique. Ainsi, un langage de description d'interface moderne ne peut se limiter à une description syntaxique des opérations fournies et requises par le composant. L'ajout d'informations sur le comportement du composant avec son environnement permet une meilleure compréhension du fonctionnement du composant et l'analyse de son assemblage avec d'autres composants. Cette vérification garantit la réutilisation correcte d'un composant, mais fournit aussi un cadre pour la réutilisation des composants. Cette réutilisation de l'étape de spécification à l'étape de mise en œuvre est une des solutions intéressantes pour réduire les coûts de production d'une nouvelle application.

2.3 Architecture logicielle

De nombreuses approches par composants industrielles ou académiques existent pour la conception ou la mise en œuvre des applications. L'hétérogénéité de ces approches et de leurs objectifs rend difficile une définition de la notion de composant logiciel. Cependant, un point commun que l'on retrouve dans la plupart de ces approches réside dans la notion d'architecture logicielle définie comme une description abstraite du système, de sa décomposition en composants, de l'interface de ces composants et de leurs interactions.

Nous présentons dans la section 2.3.1 les concepts essentiels de l'architecture logicielle à base de composants. Au travers de la section 2.3.2, nous proposons notre point de vue sur les architectures orientées service. Enfin, la section 2.3.3 présente les bénéfices liés au fait de disposer d'une description d'architecture et les problèmes encore à résoudre.

2.3.1 Lien entre modèle de composant et architecture logicielle

Bien qu'issus de communautés au départ peu liées, les travaux relatifs aux modèles de composant et aux architectures logicielles sont complémentaires et tendent à se rapprocher. Les premiers visent à fournir les briques de base pour la production d'entités logicielles réutilisables. Les seconds se sont concentrés sur la description de l'assemblage de ces briques et donc sur leurs interactions afin de construire des applications de qualité. De par leur objectif premier d'offrir des langages de haut niveau pour décrire un assemblage de composants, les travaux sur l'architecture logicielle ont introduit les concepts de connecteurs et de configuration. Ainsi, malgré une très grande diversité, les trois concepts de composant, connecteur et configuration sont généralement acceptés comme essentiels [Medvidovic and Taylor, 2000].

La notion de *composant* a pris de nombreuses formes dans les différentes approches de l'architecture logicielle, souvent plus générales que la seule notion de composant logiciel, présentée dans la section 2.1.1.2, car le domaine pris en compte était généralement plus large que celui du logiciel. Ainsi, certaines approches autorisent par exemple la définition de composant matériel.

Le *connecteur* correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions. Les connecteurs sont donc des éléments essentiels pour la maintenance et la réutilisation des composants. Ils permettent de séparer la phase de développement en deux niveaux : la programmation classique des modules logiciels et le codage des interactions entre ces modules. Ils contiennent les informations concernant les règles d'interconnexion entre les composants, telles que le protocole, la spécification des échanges de données ainsi que les transformations éventuelles du format d'échange. Par exemple, un connecteur peut décrire des interactions simples de type : appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux. Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur, c'est-à-dire son interface, qui permet la description des rôles des participants à une interaction. La seconde partie correspond à la description de son implantation. Il s'agit de la mise en œuvre du protocole associé à l'interaction. Dans certaines approches, le connecteur est parfois appelé adaptateur, car il permet de mettre en relation deux composants non prévus initialement pour interagir ensemble.

Composants et connecteurs peuvent être assemblés à partir de leurs interfaces pour former une *configuration*. Celle-ci décrit l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que leurs interactions. Elle permet de gérer la complexité de l'application en raffinant peu à peu l'architecture en un ensemble hiérarchique de modules logiciels. La configuration peut aussi modéliser le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles comme les propriétés spatio-temporelles ou la qualité de service. On la nomme alors « configuration comportementale ». Un langage de contraintes accompagne souvent le langage de description d'architecture dans le but de décrire les règles de composition. D'une manière générale, la manipulation de la configuration permet de simplifier la création d'une nouvelle application.

2.3.2 Architectures orientées service ou architectures à base de composants ?

La notion d'architecture orientée service (SOA pour *Service Oriented Architecture*) définit un modèle d'interaction applicative mettant en œuvre des connexions en couplage lâche entre diverses entités logicielles [SOA, 2004]. Un *service* désigne une action exécutée par un « fournisseur » à l'attention d'un « consommateur », basée éventuellement sur un autre système. C'est une fonction qui reçoit des messages et les restitue après un traitement. Un service peut être considéré comme une façade qui se positionne devant le composant ou l'objet pour le standardiser. Le service s'apparente alors à une interface de localisation, d'accès et d'exécution d'un traitement applicatif modulaire. De ce fait, les SOA peuvent être perçues comme des architectures basées sur un ensemble d'entités modulaires dont on ne précise pas la nature. Ces entités peuvent être des fonctions, des objets ou des composants. L'objectif de l'architecture orientée service est alors d'homogénéiser la façon dont ces entités coopèrent au travers d'interfaces qui facilitent leur localisation et leur exécution.

Les architectures orientées service offrent un fort découplage entre les interfaces des services disponibles et leur mise en œuvre. Cependant, leur principale lacune réside dans le manque d'expression des dépendances entre services, dépendances qui sont exprimées par des interfaces requises au niveau des modèles de composant. De plus, l'absence au niveau de la spécification de la notion d'encapsulation est aussi gênante, elle n'offre aucun mécanisme pour structurer les services selon leur granularité. Il n'y a donc pas d'antagonisme entre les approches de l'architecture logicielle orientée service ou à base de composants. L'architecture orientée service offre une vision des services disponibles au niveau d'un système d'information en faisant abstraction de la mise en œuvre de ces services. L'architecture à base de composants fait le choix de représenter les entités logicielles offrant ces services sous forme de composants explicitant leurs dépendances avec leur environnement.

2.3.3 Conclusion

Pour améliorer la modularité des applications, les modèles de composant permettent de structurer les différentes entités logicielles en briques réutilisables. L'architecture logicielle constitue alors une information importante, car elle identifie la configuration de ces briques de base, c'est-à-dire leur assemblage. C'est une abstraction de haut niveau par rapport à la réalisation sur une plate-forme d'exécution. Cependant, la construction de telles architectures reste une tâche difficile. La seule construction de langages dédiés permettant de spécifier précisément les interfaces de composant et leurs interactions ne peut suffire à maîtriser la complexité de cette opération. En outre, ces descriptions d'architecture doivent être capables de prendre en compte le caractère évolutif de l'application :

- évolution interne de l'application afin de pouvoir anticiper cette dynamique et d'en évaluer l'impact sur la cohérence de l'architecture,
- évolution externe liée aux fonctionnalités de l'application afin de pouvoir rester en adéquation avec les besoins des utilisateurs.

2.4 Défis de l'architecture logicielle

Si la description de l'architecture logicielle prend maintenant une place prépondérante dans la phase de conception d'un logiciel, cette tâche reste pour le moment une opération complexe pour l'architecte. En effet, la description d'une architecture logicielle se trouve confrontée aux mêmes défis que ceux inhérents à la production du logiciel à savoir la modularité, l'abstraction, l'anticipation du changement, la construction incrémentale et la réutilisation (voir Sections 2.1 et 2.2).

2.4.1 La modularité

Les notions de composant, connecteur et configuration offrent un premier niveau de modularité dans la description d'une application. Si ces notions permettent une bonne séparation entre la mise en œuvre des entités de base

du système et leurs interactions, elles proposent une unique dimension de décomposition, à savoir le composant. Or différents travaux ont montré la difficulté de structurer une application à l'aide d'une unique dimension. Ce problème a été identifié comme la tyrannie de la décomposition dominante [Tarr *et al.*, 1999].

Un des problèmes résultant de cette unique dimension de modularité consiste à introduire par exemple de nombreuses dépendances entre les composants et les composants techniques rendant la spécification des composants métiers très difficile à établir. En effet, le travail de conception d'une description d'architecture logicielle demande la spécification des interfaces des composants de l'architecture et la spécification de l'assemblage de ces composants. Même si cette description propose un formalisme abstrait permettant de simplifier la conception d'une telle description, le nombre de préoccupations à prendre en compte dès la spécification des interfaces des composants rend ce travail très complexe et propice aux erreurs.

Premier défi

La conception d'un ADL moderne moderne doit s'appuyer sur différentes dimensions de structuration afin de simplifier la définition des composants métiers.

2.4.2 L'abstraction

L'ingénierie dirigée par les modèles montre le besoin de définir les fonctionnalités des entités de son système d'information indépendamment de toute plate-forme d'exécution pour garantir la pérennité et l'interopérabilité des entités du système d'information. Les approches par composants offrent de leur côté une structure pour la définition de l'architecture logicielle séparant clairement les interactions entre les entités logicielles de la mise en œuvre de ces entités. Ainsi, la notion d'architecture logicielle véhicule principalement une distinction nette entre la structure conceptuelle d'une application (modèle abstrait) et ses possibles réalisations techniques (implantations) sur des cibles particulières (langage et système d'exploitation).

Deuxième défi

La description de l'architecture du système doit être définie à partir d'un modèle abstrait afin de permettre à l'architecte de manipuler les constructions essentielles de l'architecture et de rendre ces descriptions indépendantes des évolutions technologiques.

2.4.3 L'anticipation du changement

L'anticipation du changement consiste dans le domaine de l'architecture logicielle à construire un modèle conscient du besoin permanent de changement du système, capable de spécifier et de « cadrer » ces changements, c'est-à-dire de préciser la variabilité de l'architecture. Nous considérons deux principaux types de changement associés à une description d'architecture logicielle. Le premier concerne la dynamique de l'architecture appelée aussi évolution interne. Le deuxième concerne l'adaptation de l'architecture aux nouveaux besoins de l'utilisateur appelée aussi évolution externe.

Troisième défi

La description de l'architecture logicielle doit donc prendre en compte ces deux types d'évolution au niveau du modèle pour permettre à l'architecte d'intégrer dans sa spécification ces possibilités d'évolution.

2.4.4 Une construction incrémentale : des composants métiers à l'intégration des préoccupations techniques

La description d'une architecture logicielle résulte directement de l'étape d'analyse des besoins avec les futurs utilisateurs d'une application. Il est donc plus facile pour l'architecte de raisonner à partir d'une approche partant de considérations métiers en y intégrant au fur et à mesure des préoccupations plus techniques. Or, une telle démarche incrémentale est pour le moment difficile à mettre en œuvre à partir des langages de description d'architectures

existants, les modifications liées à l'intégration d'une nouvelle préoccupation sur les composants présents dans l'architecture logicielle devant être effectuées manuellement par l'architecte.

Quatrième défi

La description d'architecture logicielle doit permettre la construction du système par ajout successif de nouvelles préoccupations.

2.4.5 La réutilisation

L'architecture logicielle décrit, outre la structure du système en un assemblage de composants, l'interface de chaque composant. Or, les composants cherchent à être, en plus d'une unité de structuration, une unité de réutilisation. Dès lors, il est indispensable de disposer au niveau de l'interface d'un composant de suffisamment d'informations sur celui-ci pour détecter certaines incompatibilités entre composants liés, ceci dès la phase de description de l'architecture. Dès lors, la taxonomie proposée par Beugnard et les différents travaux sur la conception par contrats démontrent qu'une interface de composant ne peut se limiter à la description syntaxique des services offerts et requis, mais doit contenir des informations précises sur la façon dont le composant interagit avec son environnement.

Cinquième défi

La description d'architecture logicielle doit permettre de valider l'assemblage de composant à la vue de certaines propriétés.

Chapitre 3

La construction d'architecture à base de composants

PRÉAMBULE

Dans ce chapitre nous étudions les caractéristiques de la définition d'une architecture logicielle à base de composants en s'appuyant sur différents travaux autour des langages de description d'architecture, des modèles de composant académiques et des travaux en cours issus des organismes de normalisation. Parmi ces caractéristiques, ce chapitre étudie les éléments structurels du modèle de composant, la définition des interfaces des composants, l'expression de la dynamique et la prise en compte du déploiement.

Sommaire

3.1	Les langages de description d'architectures	31
3.1.1	Wright	31
3.1.2	Darwin	35
3.1.3	Bilan sur les ADLs	38
3.2	Les modèles académiques de construction d'applications à base de composants	39
3.2.1	ArchJava	39
3.2.2	Fractal	43
3.2.3	SOFA	48
3.2.4	Bilan sur les modèles de composant académiques	51
3.3	Les standards	54
3.3.1	AADL	54
3.3.2	UML 2.0	57
3.3.3	Bilan sur les normes associées à la description d'une architecture logicielle	61
3.4	Bilan sur les approches à base de composants	62
3.4.1	Anticipation des changements : prise en compte de la dynamique du système	62
3.4.2	Abstraction : indépendance vis-à-vis de la plate-forme d'exécution	63
3.4.3	Modularité : structuration multi-dimensionnelle de l'application	64
3.4.4	Réutilisation : définition étendue des interfaces de composant	64
3.4.5	Construction incrémentale de la description d'architecture logicielle	65
3.4.6	Proposition	65

Décrire précisément une architecture logicielle est un problème qui a été souvent étudié dans la communauté de recherche en génie logiciel avec une effervescence importante dans les années 90 autour des langages de description

d'architecture. La définition du modèle de composant abstrait SafArchie (voir la deuxième partie) repose sur l'étude des différents ADLs, modèles de composant abstraits académiques ou issus des organismes de normalisation. Pour illustrer cette étude, ce chapitre présente quelques travaux reconnus sur ces langages ou ces modèles abstraits en se focalisant plus précisément sur leurs objectifs et les caractéristiques essentielles liées à leur modèle. L'évaluation de ces approches se focalise sur le niveau d'abstraction du langage et son indépendance vis-à-vis d'une plate-forme d'exécution, la prise en compte de la dynamique du système, la prise en compte de l'infrastructure physique sur laquelle est déployé le logiciel et enfin le pouvoir d'expression au niveau de l'interface du composant en suivant la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999].

Abstraction du langage par rapport à une plate-forme d'exécution

Le premier critère d'évaluation étudie la présence d'une description claire de l'architecture, le niveau d'abstraction du formalisme utilisé pour effectuer cette description et son indépendance vis-à-vis d'une plate-forme d'exécution. L'idéal consiste à proposer à l'architecte un modèle de haut niveau de l'architecture qui se concentre sur les éléments caractéristiques d'une description d'architecture. Cette abstraction doit lui permettre de travailler avec un formalisme simple, facilement compréhensible par l'ensemble des acteurs d'un projet informatique.

Prise en compte la dynamique de l'architecture

Le deuxième critère d'évaluation correspond à la prise en compte de la dynamique de l'architecture dans les approches abordées dans ce chapitre. Cette dynamique caractérise le caractère évolutif et changeant d'une description d'architecture logicielle. Le travail de cette thèse considère deux types d'évolution (voir Section 2.1.3). La dynamique de l'architecture est associée aux évolutions dites internes. Celles-ci caractérisent le fait qu'une architecture logicielle peut, de manière autonome, changer en cours de fonctionnement pour s'adapter à son contexte d'exécution. Ce premier type d'évolution peut être abordé soit de manière explicite en prévoyant l'ensemble des configurations admissibles pour une architecture logicielle soit, de manière implicite, en créant un cadre pour cette évolution définissant un certain nombre d'invariants que l'architecture logicielle doit respecter. Ce premier type d'évolution concerne principalement l'évolution de la configuration d'une architecture logicielle, à savoir la création ou la destruction d'instances de composant ou de connecteur.

Contrats d'interface et description du comportement des composants

Le troisième critère d'évaluation se concentre sur le pouvoir d'expression associé à la spécification des interfaces d'un composant dans les différentes approches de l'architecture logicielle abordées dans ce chapitre. Ce pouvoir d'expression est très important, car il conditionne la vérification de la cohérence d'un assemblage de composants par rapport à certaines propriétés structurelles ou comportementales. La richesse de description des interfaces d'un composant améliore la réutilisabilité de ce composant en garantissant sa bonne intégration dans un contexte différent de celui pour lequel il a été conçu.

Déploiement sur une infrastructure physique

Enfin, le dernier critère de ce chapitre concerne la prise en compte du déploiement d'une architecture logicielle sur une infrastructure physique. Ce critère souvent peu pris en compte dans les modèles de composant est une information importante pour permettre l'extension des modèles de composant vers l'expression de la qualité de service.

L'étude de ces quatre critères ajoutée à une caractérisation des éléments du modèle de composant pour chaque approche permet de confronter chacun de ces travaux aux défis de l'architecture logicielle énoncés dans la section 2.4.

Ce chapitre est organisé de la manière suivante. Les deux premières sections présentent deux ADLs issus des approches formelles Wright [Allen, 1997] et Darwin [Magee *et al.*, 1995] et trois modèles de composants académiques récents profitant du rapprochement de la communauté des ADLs et des approches par composants à savoir ArchJava [Aldrich *et al.*, 2002], Fractal [Bruneton *et al.*, 2004b] et SOFA [Kalibera and Tuma, 2002]. La section 3.3 s'intéresse à deux approches issues des organismes de normalisation : AADL [SAE, 2004] et UML 2.0 [OMG, 2003c] montrant l'intérêt des industriels de disposer d'un standard pour la description d'une architecture logicielle.

3.1 Les langages de description d'architectures

Les langages de description d'architectures (ADL pour *Architecture Description Language*) constituent l'un des aboutissements des travaux sur l'architecture logicielle. Ils ont pour objectif premier d'aider les concepteurs à structurer et composer leurs éléments logiciels pour former des applications. Un ADL est défini comme une notation formelle ou semi-formelle représentée textuellement ou graphiquement qui permet de spécifier des architectures logicielles, accompagnée d'outils spécifiques [Medvidovic and Taylor, 2000].

Pour illustrer ces travaux, cette section se focalise sur deux langages de description d'architecture qui ont largement inspiré nos travaux : Wright [Allen, 1997] un des précurseurs et Darwin et son extension Tracta [Giannakopoulou, 1999].

3.1.1 Wright

Wright [Allen, 1997] a été une des premières approches à permettre la description du comportement de composants. Il se centre sur la spécification de l'architecture et de ses éléments. Il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application comme pour Rapide [Luckham *et al.*, 1995].

3.1.1.1 Présentation du modèle

Les trois notions de base d'un ADL à savoir le composant, le connecteur et la configuration sont présentes dans Wright.

Le composant Wright

Un composant en Wright est une unité de traitement abstraite localisée et indépendante. C'est d'après la caractérisation des composants présentée dans la section 2.1.1.2 une instance. La sémantique associée au composant est le processus. La description d'un composant contient deux parties importantes : l'interface et la partie calcul (*computation*). L'interface consiste à décrire les ports, c'est-à-dire les interactions auxquelles le composant peut participer. Par exemple, un composant représentant une station service peut avoir deux ports, un pour les requêtes du client et un autre pour la banque. À chaque port est associée une description formelle par le langage CSP (*Communicating Sequential Processes*) [Hoare, 1985] spécifiant son comportement par rapport à l'environnement. La partie calcul, quant à elle, consiste à décrire le comportement du composant 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.

Le connecteur Wright

Un connecteur représente une interaction entre une collection de composants. Il spécifie le patron d'une interaction de manière explicite et abstraite. Ce patron peut être réutilisé dans différentes architectures. Par exemple, un protocole de validation à deux phases (*two-phase-commit*) peut être un connecteur.

Le connecteur contient deux parties importantes qui sont un ensemble de rôles et la glue. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. La sémantique associée au connecteur est donc aussi le processus. Le comportement du rôle est décrit par une spécification CSP. La glue décrit comment les participants (c'est-à-dire les rôles) interagissent entre eux pour former une interaction. Par exemple, la glue d'un connecteur « appel de procédure » indiquera que l'appelant doit initialiser l'appel et que l'appelé doit envoyer une réponse en retour. La notion de glue et de rôle sont à rapprocher de la notion de port et de partie calcul du composant. La principale différence concerne l'absence de typage explicite des messages au niveau d'un connecteur d'où le nom glue.

La configuration Wright

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composant et des instances de connecteur. 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 composant et de connecteurs, les descriptions des liens entre les instances de composant par les connecteurs.

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.

3.1.1.2 Expression du déploiement

Wright ne s'intéresse pas au déploiement d'une architecture logicielle à base de composants. Il se concentre sur la description des modules logiciels et leurs interactions. En effet, l'objectif de Wright est orienté vers la vérification formelle d'une architecture composée de composants et de connecteurs. Si des composants sont sur des sites différents, Wright considère qu'il s'agit de composants dans des processus différents.

3.1.1.3 Définition de la dynamique

Wright n'a pas été construit au départ pour spécifier et analyser la dynamique d'une architecture logicielle. Cependant, une approche proposée dans [Allen *et al.*, 1998], permet de spécifier et d'analyser une partie de la dynamique de l'architecture à l'aide de Wright. Ce projet implique l'ajout d'un composant « *configurateur* » à une architecture dynamique. L'information liée à la politique de reconfiguration est définie dans ce composant à l'aide d'un langage *ad hoc* basé sur la notion d'actions et de règles, les règles définissant les actions à effectuer en fonction de la réception de certains événements. Les actions, quant-à-elles, peuvent être la création ou la destruction d'un composant, la création ou la destruction d'une liaison. Cette approche offre deux points très intéressants. Elle intègre dans la spécification de l'architecture, la description de la dynamique de l'architecture à savoir ses politiques de reconfiguration. Elle identifie dans un langage minimaliste les primitives de base de la reconfiguration pour Wright, à savoir l'ajout et la destruction d'instances et la création et la destruction de liaisons.

3.1.1.4 Description du comportement

La mise en œuvre du composant et du connecteur Wright est décrite à l'aide du langage CSP. Ce langage a été créé en 1978 par Hoare dans un souci de formalisation théorique. Il est basé sur une logique de composition des actions (Logique de Hoare) [Hoare, 1969].

Les caractéristiques principales de CSP sont les suivantes :

- le support du comportement est le processus,
- leur nombre est invariable, Wright permet uniquement la description d'architecture statique,
- ils ne possèdent pas de variables communes,
- ils s'exécutent en parallèle.

L'absence de variables communes n'autorise pas le type de communication par mémoire partagée. Les processus communiquent par messages. Ils utilisent des canaux de communication, par lesquels ils s'échangent les messages. Il n'y a pas de tampon sur les canaux de communication : un message émis est perdu si le destinataire n'est pas prêt à le recevoir. Pour éviter cela, il faut donc que les deux processus, émetteur et récepteur, se synchronisent pour communiquer directement. C'est le mécanisme du rendez-vous.

Le rendez-vous se réalise entre un processus émetteur, qui souhaite envoyer un message sur un canal de communication, et un processus récepteur, qui attend le message. L'émetteur se bloque en attendant que le récepteur soit disponible. De même, le récepteur doit attendre que l'émetteur soit prêt pour récupérer un message. Lorsque les deux processus sont prêts pour l'échange, la communication est effectuée, puis les deux processus poursuivent leur exécution.

Le processus émetteur désigne le processus récepteur qu'il désire atteindre. De même, le récepteur indique de quel processus il attend un message. Dans Wright, cette désignation est décrite au niveau de l'assemblage qui indique quel composant communique avec quel connecteur.

La description du comportement dans Wright permet une analyse de l'assemblage de composants. Cette analyse détecte deux types de propriétés au niveau de l'architecture :

- les propriétés de sûreté qui garantissent que rien de mauvais ne va arriver au niveau d'un assemblage de composant. Ces propriétés permettent de vérifier que l'assemblage ne produit pas d'interblocage. Ces propriétés permettent en outre de vérifier des propriétés, définies par l'utilisateur, sur l'ordonnancement des messages,
- les propriétés de vivacité qui garantissent que quelque chose de bon peut toujours arriver.

3.1.1.5 Exemple

Pour illustrer les principaux mécanismes de Wright, nous prenons l'exemple d'un serveur Web possédant trois composants : un composant prenant en charge la lecture/écriture sur un système de fichier, différents composants prenant en charge le traitement des requêtes et un composant servant de distributeur des différentes requêtes (voir Figure 3.1).

```
Configuration System
  Connector ...
  Component ClientType
  Component WebServerType
    Port ServiceHttp
    Computation
      Configuration WebServer
        Component Coordinator
          Port ServiceHttp (lire les données jusqu'à la fin des données)
          Port Dispatch1 (sortir les données de manière permanente)
          ...
          Computation (Attendre les requêtes sur le port ServiceHttp
            et les envoyé à un des trois ports Dispatch)
        Component FileSystem
```

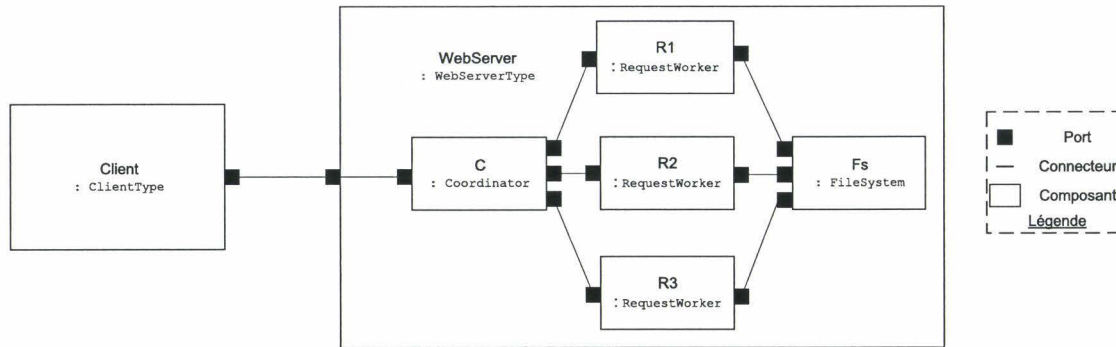


FIG. 3.1 – Architecture d'un serveur Web décrit à l'aide de Wright

```

Component RequestWorker
Connector CSConn
Role Client
Role Server
...
Instances
  C : Coordinator
  Fs : FileSystem
  R1 : RequestWorker
  R2 : RequestWorker
  R3 : RequestWorker
  S1 : CSConn
  S2 : CSConn
  S3 : CSConn
  S4 : CSConn

Attachments
  C.Dispatch1 as S1.Client
  R1.Service as S1.Server
  ...
End WebServer
Bindings
  C.ServiceHttp = ServiceHttp
End Bindings
Instances
...
Attachments
...
End System

```

Les liens entre les composants et les connecteurs sont définis dans la clause *Attachement*. Le mot-clé *Bindings* permet de spécifier les liens entre les ports du composite et ceux des composants faisant partie du composite. Dans notre exemple, le port *ServiceHttp* du composant *Coordinator* est lié au port *ServiceHttp* du composant *WebServerType*.

3.1.1.6 Évaluation de Wright

Le principal avantage de Wright est de fournir un langage formel (CSP) pour la spécification des composants et des connecteurs. Ainsi, la description d'une architecture peut être analysée. En outre, les différents acteurs (concepteur, architecte, développeur) d'un projet peuvent communiquer sans ambiguïté leurs points de vue sur l'architecture d'une application. Dans ce sens, Wright est un des premiers ADLs à fournir une description des interfaces de niveau trois selon la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999]. Cependant, la description des deux premiers niveaux est assez pauvre et éloignée des habitudes des développeurs. Ainsi, un des inconvénients majeurs de Wright est qu'il est difficile à assimiler. En effet, l'outil de spécification (CSP couplé à Wright) n'est pas facile à comprendre pour un développeur débutant. Il peut donc s'avérer inefficace lorsque les délais d'un projet sont courts ou lorsque les compétences font défaut. De plus, Wright ne possède pas d'environ-

nement d'utilisation ou d'exécution. Il ne possède pas par exemple de générateur de code. En conclusion, Wright est issu du monde des spécifications formelles. Un travail important a été mené afin d'analyser un assemblage de composants, mais les problématiques liées à la génération de code ou à la prise en main du modèle par les développeurs ont été peu prises en compte.

3.1.2 Darwin

Darwin [Magee *et al.*, 1995] a été développé au Distributed Software Engineering Group de l'Imperial College de Londres. Il propose un modèle de composant pour la construction d'applications distribuées. Sa particularité est de permettre la spécification d'une partie de la dynamique d'une application en termes de schéma de création de composants logiciels avant, pendant ou après l'exécution d'une application.

3.1.2.1 Présentation du modèle

Le modèle de Darwin reprend les trois concepts de base des modèles à composants à savoir, le composant, le connecteur et la configuration. Cependant le connecteur n'a pas une sémantique de processus comme dans Wright.

Le composant Darwin

Comme pour la plupart des ADLs, un composant est une instance définie par une interface qui décrit les services qu'il fournit et qu'il requiert pour lui permettre de fonctionner et de s'intégrer dans une architecture complexe. La sémantique associée au composant est le processus, chaque composant correspond à un processus créé.

Le connecteur Darwin

L'interaction entre les composants se fait par l'intermédiaire des services offerts et requis par le composant. Le concept de connecteur n'est pas explicitement présent dans Darwin. En effet, chaque interaction est représentée par un lien entre un service fourni et un service requis de composants différents. Les services n'ont aucune connotation fonctionnelle, ils désignent seulement le type d'objet de communication utilisé et autorisé à utiliser une fonction du composant. Ces types d'objets sont définis par le support d'exécution répartie appelé *Regis* et sont limités. Parmi les types d'objets, le port est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre composants répartis ou non.

Dans une description d'architecture exprimée avec Darwin, la section « port » caractérise le type d'objet de communication, la section « signature » décrit la signature de la fonction du composant. Darwin par défaut effectue un simple test d'équivalence des noms et des types avant la connexion, vérifiant que le service requis est compatible avec le service fourni.

La configuration Darwin

Afin d'offrir une hiérarchie au sein d'une application, Darwin contient deux sortes de composant :

- les composants primitifs sont avant tout des entités d'encapsulation de fonctions et de données d'un module logiciel. Ils sont définis par leur nom et leur interface qui déclarent les services requis et fournis par le composant métier,
- le composant composite contient des composants primitifs et des composants composites. C'est donc une unité de configuration. Il décrit les interconnexions entre les composants. Ainsi, la structure finale d'une application est représentée par un composant composite.

Regis : une plate-forme d'exécution pour Darwin

Darwin possède une plate-forme d'exécution *Regis*. *Regis* fournit un cadre écrit en C++ pour construire et exécuter des programmes distribués et spécifiés par Darwin. Les composants primitifs sont des classes C++ qui héritent de la classe *Process*. Pour la communication entre les composants, l'environnement *Regis* fournit aux programmeurs plusieurs types d'objets de communication tels que le port ou la diffusion multiple d'événements.

3.1.2.2 Le déploiement des composants

Darwin fournit un modèle de déploiement en rapport avec sa plate-forme d'exécution *Regis*. Chaque composant Darwin est alors associé à un site de déploiement de l'environnement Darwin. Le site de création des composants est déterminé de manière absolue. Elle se fait lors de la création d'instances de composant en y ajoutant un numéro désignant le site de création. Ce numéro correspond à un site disponible au niveau de la plate-forme d'exécution. Ceci pose un problème, car il existe toujours un risque que le déploiement ne puisse être effectué, par exemple, si une machine est momentanément hors service, ou si une machine a été remplacée.

3.1.2.3 L'expression de la dynamique

Une caractéristique importante de Darwin est sa capacité à décrire la création dynamique de composants par l'application. Cette caractéristique apporte une amélioration notable puisque l'architecture d'une application n'est plus considérée comme un ensemble de composants prévus dès la phase de conception. Darwin fournit aux architectes les principaux mécanismes pour la description de structures dynamiques :

- l'instanciation paresseuse est une pré-déclaration des instances qui seront effectivement créées non pas lors de la phase d'initialisation, mais dès qu'un premier appel vers l'instance est effectué. L'instanciation paresseuse permet par une spécification de structures dynamiques, de décrire la configuration potentielle à l'exécution en déclarant des instances qui seront créées par des appels à leurs services. Ce type d'instanciation permet notamment de décrire des structures dont la taille ne peut être déterminée que dynamiquement. Malheureusement, ce mécanisme ne permet d'instancier qu'un seul composant par clause d'interconnexion, contrairement à l'instanciation dynamique,
- l'instanciation dynamique permet de décrire la création d'une instance de composant en lui fournissant des paramètres d'initialisation. D'une manière générale, la création dynamique d'un composant est réalisée au sein d'un composant englobant (composant composite). L'accès à ce composant dynamiquement créé est géré par le composant englobant selon un système de nommage indirect. Toutes ces déclarations sont réalisées à l'aide du langage de configuration de Darwin.

3.1.2.4 Description du comportement

Une extension de Darwin, le projet Tracta [Giannakopoulou, 1999], a permis d'ajouter une description du comportement des composants au sein d'une description d'architecture. Les informations liées à l'architecture peuvent être directement exploitées pour l'analyse du système. Ainsi dans Tracta, l'architecte ajoute une description de comportement des composants primitifs et un ensemble de propriétés que le système doit satisfaire. Darwin utilise un langage de haut niveau dérivé de CSP appelé FSP pour la description des comportements des composants. Ce langage fournit une syntaxe concise pour la définition de LTS (*Labelled Transition Systems*) qui modélise le comportement d'un composant primitif.

Tracta utilise les informations issues de la description de l'assemblage de composants afin de construire le comportement des composites et obtenir le comportement global du système à partir des descriptions de comportement des composants primitifs.

L'association de la définition des services, de la dynamique de l'architecture et du comportement des composants fait de Darwin un modèle d'architecture précurseur, très intéressant pour la spécification et l'analyse d'applications distribuées.

La description du comportement dans Darwin permet comme dans Wright de détecter la présence d'interblocage et de vérifier les propriétés de sûreté ou de vivacité. Au niveau des propriétés de vivacité, Darwin définit la notion de propriété de progression. Cette propriété permet dans le cas où l'on donne une priorité d'ordonnement des processus de garantir que pour une infinité d'exécutions d'un processus, on est sûr qu'au moins une action définie au niveau de la propriété va s'exécuter une infinité de fois. Ce type de propriété qui est un sous-ensemble des propriétés de vivacité permet de mettre en évidence, dans les cas où la priorité des composants est définie, que certaines actions n'ont jamais l'occasion de s'exécuter.

3.1.2.5 Exemple

L'exemple de la figure 3.2 présente l'architecture du serveur web avec répartition de charge décrit avec Darwin. On constate que contrairement à Wright, il est possible de créer plusieurs liaisons sur un même port. Le serveur web est alors constitué de 3 types de composant : le coordinateur (*Coordinator*), le gestionnaire de requêtes (*RequestWorker*) et le système de fichiers *FileSystem* et de cinq instances de composant : $R[1]$, $R[2]$, $R[3]$, FS et C . Nous utilisons le mécanisme d'instanciation statique pour créer la configuration du serveur web.

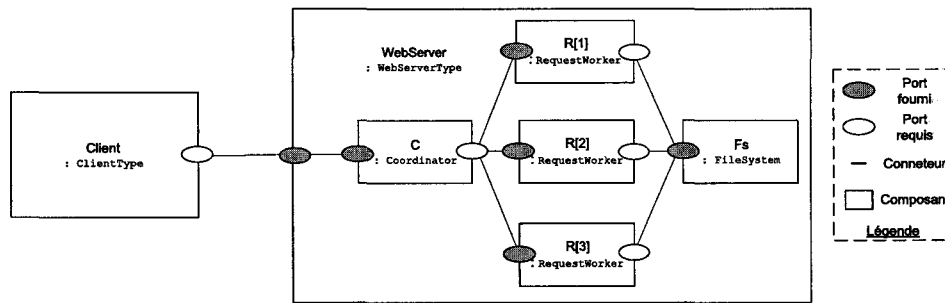


FIG. 3.2 – Architecture d'un serveur Web décrit à l'aide de Darwin

```
Component WebServer {
  const int nbMaxWorker = 3;
  array R[0..nbMaxWorker-1] : RequestWorker; // ensemble de composants de traitement
  inst C : Coordinator;
  inst FS : FileSystem;
  forall k:[0..nbMaxWorker-1] // itérateur, crée les RequestWorker
  // et les interconnexions entre le coordinateur
  // et les RequestWorker
  inst R[k] : RequestWorker(k);
  bind -- Coordinator.serviceHttp -- RequestWorker[k].Service;
  bind -- FS.Access -- RequestWorker[k].Accessfs;
}
```

3.1.2.6 Évaluation de Darwin

Darwin propose une vision de l'architecture claire et fonctionnelle. L'apport majeur de Darwin aux ADLs, à nos yeux, est sans aucun doute sa prise en compte de la dynamique de l'architecture logicielle. En outre, Darwin offre par l'intermédiaire du projet Tracta la possibilité de modéliser le comportement du système à l'aide du langage d'algèbre de processus FSP. Darwin offre ainsi la possibilité de décrire des interfaces de niveau trois selon la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999]. Le niveau un est proposé par un système de nommage est de typage au niveau des interfaces du composant. Les deux autres niveaux ne sont pas adressés par le langage de description d'architecture Darwin.

Parmi les inconvénients de Darwin, on peut citer le fait qu'un composant n'est associé qu'à une seule sémantique, le processus. Ainsi, un composant ne permet pas d'exprimer un autre élément comme un fichier ou une mémoire partagée.

Le code d'utilisation de l'environnement Darwin n'est pas transparent pour le programmeur. Il existe une unique implantation de Darwin, le développeur d'application ne peut faire le choix de sa plate-forme d'exécution, il doit explicitement utiliser les objets de type port pour réaliser ses communications et effectuer les créations de composants lorsqu'ils les sollicitent.

Enfin, la description de la dynamique de l'application reste tout de même limitée. Par exemple, il n'est pas possible de supprimer des composants dynamiquement ni de permettre à un composant primitif de communiquer avec des composants dynamiquement créés.

3.1.3 Bilan sur les ADLs

La présentation de ces deux ADLs permet de mettre en lumière les avantages des langages de description d'architecture. Conçus spécifiquement pour la définition des interfaces de composant et leur assemblage, ces langages ont les avantages des DSLs (voir Section 2.1.2.2). L'architecte manipule en effet directement les concepts qui lui sont propres en faisant abstraction de toute connaissance particulière d'un langage de programmation. L'analyse travaillant uniquement à partir des concepts présents au niveau du DSL remonte une information compréhensible pour l'architecte. La pertinence du domaine ciblé se justifie par le besoin de séparer clairement la déclaration des interactions entre entités logicielles de leur mise en œuvre afin de pouvoir modifier l'un sans modifier l'autre. Le fait de disposer d'un plan global de l'application avec une abstraction suffisante fournit un modèle pivot de la modélisation d'une application permettant à différents acteurs d'un projet de disposer d'un vecteur de communication commun. En outre, elle reste un modèle suffisamment précis pour servir de base pour l'analyse de la cohérence de l'application. Cependant, le manque d'effort de la communauté pour définir un langage commun a limité l'utilisation des ADLs dans le domaine industriel. Dans le même ordre d'idée, si les langages ont fourni des mécanismes de spécification puissants, ces langages semblent difficiles à prendre en main par l'ensemble des acteurs d'un projet informatique. Enfin, si les plates-formes à composants ont beaucoup évolué pour permettre la création d'applications dont la structure évolue à l'exécution, les langages de description d'architecture disposent de peu de mécanismes pour décrire ou contraindre cette dynamique.

3.2 Les modèles académiques de construction d'applications à base de composants

À l'instar des travaux sur l'ingénierie des composants, de nombreux modèles académiques sont apparus pour le développement d'applications à base de composants logiciels. Ces modèles académiques représentent des expérimentations sur des fonctionnalités précises dans le but de mieux comprendre les composants et de contribuer par la suite à la définition de modèles industriels. Cette section étudie trois modèles de composant académiques récents : ArchJava, Fractal et Sofa qui expérimentent respectivement l'intégration des concepts architecturaux dans Java, la construction d'un modèle d'assemblage modulaire et extensible et l'intégration d'informations liées au comportement des composants. D'autres modèles de composant académiques récents comme OpenCOM [Coulson *et al.*, 2002] ou K-Component [Dowling and Cahill, 2001] ne sont pas abordés dans cette section car leurs principales problématiques, la construction d'applications adaptables à base de composants, n'est pas une caractéristique retenue pour notre étude. Cette caractéristique étant de notre point de vue, davantage une problématique associée aux fonctionnalités de la plate-forme d'exécution.

3.2.1 ArchJava

ArchJava [Aldrich *et al.*, 2002] se situe à la frontière entre le langage de description d'architectures et les modèles de construction d'applications à composants. Il a été créé à l'université de Washington par Jonathan Aldrich et Craig Chambers. ArchJava a pour objectif d'améliorer la compréhension des programmes, de garantir l'architecture de l'application, de permettre une meilleure évolutivité des applications et d'encourager les développeurs à se servir des avantages offerts par les concepts de composant, connecteur et configuration.

Alors que de nombreux ADLs introduisent une syntaxe ad-hoc, ArchJava étend le langage Java afin d'incorporer les éléments d'architectures à l'intérieur du code. Un travail important a été réalisé sur le compilateur ArchJava afin de respecter l'intégrité des communications entre les composants. La prise en compte des concepts architecturaux au niveau des langages de programmation permet une traçabilité tout au long de la phase de développement des concepts architecturaux.

3.2.1.1 Présentation du modèle de composant

Le modèle de composant d'ArchJava définit, comme la plupart des ADLs, les trois concepts de composant, connecteur et configuration. ArchJava reprend ces concepts, comme élément de structure du code de l'application. Il ajoute de nouveaux éléments de syntaxe au langage Java pour gérer les composants, les connecteurs et les ports.

Le composant ArchJava

Le concept de base est le composant qui est une unité de traitement. Un composant est décrit par une interface qui contient les services fournis et requis, s'apparentant aux entrées et sorties de flots de communication. Deux types de composant existent : les primitifs et les composites. Les composites qui contiennent la description d'un assemblage de composants peuvent contrairement au modèle Darwin contenir du code logiciel.

Le connecteur ArchJava

La communication entre les composants se fait à l'aide d'un appel de méthode. ArchJava garantit que les communications entre composants respecteront les connexions définies au niveau de l'architecture et qu'aucune autre interaction entre composants ne sera possible. La vérification de cette propriété est réalisée à la compilation, évitant l'apparition impromptue de messages d'erreur à l'exécution. Ainsi, les interactions entre composants

connectés et entre un parent et ses sous-composants immédiats sont autorisées. Par contre, les appels externes à un sous-composant et les appels entre composants non connectés sont interdits. ArchJava interdit aussi les appels violant la structure hiérarchique de l'architecture.

ArchJava permet enfin de définir des classes de connexion appelées plus communément connecteurs complexes. En effet, lorsque le programmeur désire découpler l'implémentation des composants de la façon dont ils communiquent, il peut transformer les connexions entre les composants en objets à part entière. Des objets dont le seul but est de gérer la communication en tant que telle. Ces classes de connexion offrent un mécanisme d'extension simple permettant à l'architecte de transformer le simple appel de méthode (sémantique de base de la connexion en ArchJava) en un appel de procédure à distance ou en une interaction chiffrée. Au niveau langage, quand l'architecte utilise de tels connecteurs, il doit utiliser le mot clé *with* à la suite du mot clé *connect* pour préciser la classe qui permettra le nouveau type de connexion, cette classe étendant nécessairement la classe *Connector* (voir l'exemple de la section 3.2.1.5).

La configuration ArchJava

Les composants composites sont les unités de configuration. Ils contiennent la description des instances de composant qui composent une application ainsi que leurs interconnexions. Une application est donc un composant composite.

3.2.1.2 Le déploiement

Le modèle de composant ArchJava n'introduit pas pour le moment de modèle de déploiement. Les éléments de l'architecture ne sont en effet pas localisés. Ceci n'implique pas l'impossibilité de déployer une application ArchJava sur plusieurs machines, simplement, ce déploiement n'est pas spécifié. La définition de nouveaux connecteurs permet ainsi de placer les composants au sein de différentes JVM (*Java Virtual Machine*). L'opération d'instanciation et de placement de ces composants devra être réalisée manuellement.

3.2.1.3 Définition de la dynamique

ArchJava offre la possibilité de travailler sur des architectures dynamiques. En effet, il est possible de créer des composants et des liaisons au sein d'un composite. Pour conserver la capacité d'analyse à la compilation des interactions entre composants, toute nouvelle connexion à l'exécution devra avoir été préalablement déclarée. Le mot clé *pattern* permet ainsi de définir au niveau d'un composite des types de connexions acceptables entre deux types de port. Par exemple, dans l'exemple de la figure 3.3, si l'expression **connect pattern** *RequestWorker.AcessFS*, *FileSystem.AcessFS* est définie, elle décrit un type de connexion possible entre le port *AccessFS* du type de composant *RequestWorker* et le port *AccessFS* du type de composant *FileSystem*. Les connexions seront alors instanciées dynamiquement sur appel de la fonction *connect()* dans le code des composants.

Cependant, cette prise en charge de la dynamique reste néanmoins très limitée. En effet, il n'y a pas de possibilité de destruction d'instances de composant ni de destruction de liaisons. Il n'est pas possible de déplacer une instance au sein d'un autre composite. Dans ce sens, les définitions d'architecture logicielle réalisées avec ArchJava sont relativement statiques. Ceci reste cependant cohérent avec la volonté de garantir à la compilation l'intégrité des communications entre composants. La deuxième lacune concerne l'impossibilité de sortir la spécification de cette dynamique à l'extérieur du code métier. En effet, de par la philosophie d'ArchJava, le code lié à la gestion de la dynamique d'une architecture est mêlé au code lié au métier de chaque composant, ce qui complique le développement d'outils d'analyse autour de la dynamique.

3.2.1.4 Description de comportement

ArchJava ne propose pas de modèle abstrait de description du comportement des composants. Celui-ci est directement exprimé en Java.

3.2.1.5 Exemple

Pour illustrer les concepts d'ArchJava, nous reprenons l'exemple du serveur Web avec service de répartition de charge.

ArchJava permet de définir des connecteurs complexes. Nous utilisons cette fonctionnalité dans l'exemple pour créer un connecteur réalisant le protocole de répartition de charge *Round Robin*. Le reste de l'architecture reste assez classique avec quatre types de composant : *Client*, *Coordinator*, *RequestWorker* et *FileSystem*.

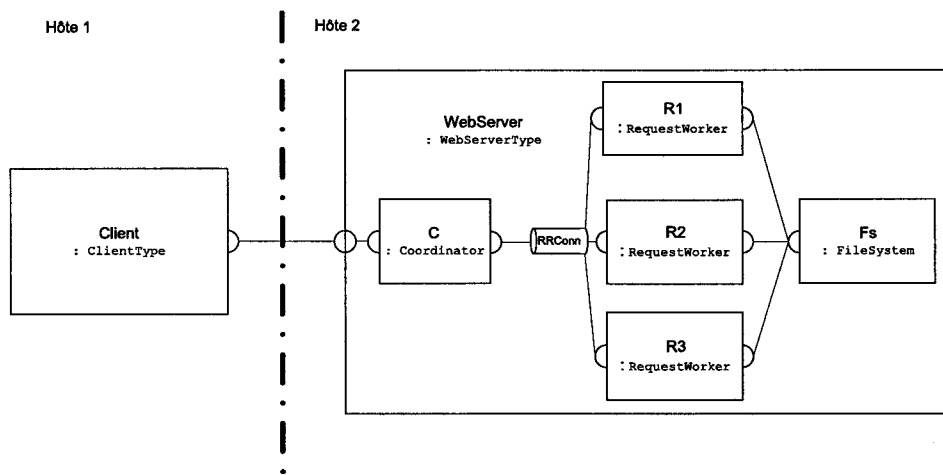


FIG. 3.3 – Mise en œuvre d'un serveur Web avec ArchJava

```

1 public component class WebServer {
2   private final Coordinator c = new Coordinator ();
3   private final RequestWorker R1 = new RequestWorker ( "R1" );
4   private final RequestWorker R2 = new RequestWorker ( "R2" );
5   private final RequestWorker R3 = new RequestWorker ( "R3" );
6   private final FileSystem fs = new FileSystem ( new File ( "/home/barais/test" ) );
7
8   // Déclaration des liaisons
9   connect c.serviceHttp, rserviceHttp;
10  connect R1.handleRequest, c.handleRequest with RoundRobinConnector.getConnector ( connection );
11  connect R2.handleRequest, c.handleRequest with RoundRobinConnector.getConnector ( connection );
12  connect R3.handleRequest, c.handleRequest with RoundRobinConnector.getConnector ( connection );
13  connect R1.AccessFS, fs.AccessFS;
14  connect R2.AccessFS, fs.AccessFS;
15  connect R3.AccessFS, fs.AccessFS;
16
17  public port serviceHttp {
18    provides String get ( URL url ){ ... }
19  }
20  private port rserviceHttp {
21    requires String get ( URL url );
22  }
23 }
24
25 public component class Coordinator {
26   public port serviceHttp {
27     provides String get ( URL url ){ ... }
28   }
29   public port handleRequest {
30     requires String handleRequest ( URL url );

```

```

31 }
32 }
33
34 public component class RequestWorker {
35     private String _name;
36     public RequestWorker (String name ){ this ._name=name; }
37
38     public port HandleRequest {
39         provides String handleRequest ( URL url ){ ... }
40     }
41     public port AccessFS {
42         requires boolean open ( File f ) ;
43         requires String read ( File f ) ;
44         requires void close ( File f ) ;
45     }
46 }
47
48 component class FileSystem {
49     public port AccessFS {
50         provides boolean open ( File f ){ ... }
51         provides String read ( File f ){ ... }
52         provides void close ( File f ){ ... }
53     }
54 }
55
56 public class RoundRobinConnector extends Connector {
57
58     private port otherPort;
59     private ArrayList ports = new ArrayList () ;
60     private int k = 0 ;
61     private static RoundRobinConnector instance = null ;
62
63     public static RoundRobinConnector getConnector ( archjava.reflect.Connection c ){
64         if ( instance== null )
65             return new RoundRobinConnector ( c ) ;
66         if ( !instance.ports.contains ( c.getPorts ()[ 0 ] ) )
67             instance.ports.add ( c.getPorts ()[ 0 ] ) ;
68         return instance;
69     }
70     private RoundRobinConnector ( archjava.reflect.Connection c ){
71         super ( c ) ;
72         otherPort = c.getPorts ()[ 1 ] ;
73         ports.add ( c.getPorts ()[ 0 ] ) ;
74     }
75
76     public Object invoke ( Call call ) throws Throwable {
77         System.out.println ( "invoke was called" ) ;
78         Method meth = call.getMethod () ;
79         Method methToCall = findMethod ( meth.getName () , meth.getParameterTypes () ) ;
80         Object o = methToCall.invoke ( call.getArguments () ) ;
81         k++;
82         if ( k== this .ports.size () ) k= 0 ;
83         return o;
84     }
85
86     protected Method findMethod ( String methodName, Type parameterTypes [] ) {
87         System.out.println ( "looking for method " + methodName ) ;
88         Method methods [] = ((Port ) ports.get ( k ) ).getProvidedMethods () ;
89         loop: for ( int i = 0 ; i < methods.length ; ++i ) {
90             Method m = methods [ i ] ;
91             System.out.println ( "considering method " + m.getName () ) ;
92             if ( !m.getName () .equals ( methodName )
93                 || m.getParameterTypes () .length != parameterTypes.length )
94                 continue ;
95             for ( int j = 0 ; j < parameterTypes.length ; ++j ) {
96                 if ( !parameterTypes [ j ] .equals ( m.getParameterTypes () [ j ] ) )
97                     continue loop;
98             }
99             return m;
100         }
101         throw new RuntimeException ( "Method " + methodName + " not understood" ) ;
102     }
103 }

```

3.2.1.6 Évaluation d'ArchJava

L'approche ArchJava est une approche très orientée « langage » de programmation. Le compilateur ArchJava arrive maintenant à maturité. Il a été utilisé sur des applications de taille importante et a prouvé sa robustesse. Les principaux avantages d'ArchJava sont les suivants. L'utilisation du langage garantit l'intégrité de l'architecture durant la phase de développement. Le compilateur s'assure qu'aucune communication ne vient contourner la structuration de l'application imposée par la description de l'architecture. On peut détecter ainsi facilement toute violation de l'encapsulation d'un composant.

La prise en compte au niveau du développement de l'application des concepts architecturaux permet une meilleure compréhension et une meilleure lisibilité du programme. En outre, des programmes comme *archjavadoc* permettent d'obtenir une vision abstraite de l'architecture.

La robustesse du compilateur ajoutée au fait que Java fournit un modèle d'exécution multi-plates-formes transforme ArchJava en une plate-forme d'exécution attrayante.

Cependant, la principale faiblesse d'ArchJava est de rester très proche de l'implantation. Pour pallier à cet inconvénient, des travaux sont en cours autour d'ACME [Garlan *et al.*, 1997] pour fournir une couche d'abstraction supplémentaire.

ArchJava spécifie uniquement la structure d'une application. Son comportement est implanté. Dans ce sens, ArchJava ne propose qu'une description des interfaces de niveau un dans la taxonomie de Beugnard *et al* [Beugnard *et al.*, 1999]. Aucun mécanisme ne permet de préciser la sémantique d'une opération ou d'un port, il n'y a pas d'abstraction du comportement d'un composant et aucune analyse ne permet de garantir la compatibilité comportementale entre des composants qui interagissent.

Enfin, le mélange des constructions architecturales et des constructions objets engendre une complexité du code importante. Ainsi, trouver le code lié à la mise en œuvre au cœur de la description de l'architecture semble être un point négatif du point de vue de la structuration du code. Certes, trouver les concepts architecturaux au niveau de la mise en œuvre est très intéressant pour la traçabilité. Cependant, une meilleure structuration entre la description de l'architecture et la mise en œuvre des services des composants permettrait de simplifier la tâche du développeur tout en gardant la possibilité de faire intervenir plusieurs acteurs dans le développement de l'application, ce qui est assez difficile avec une approche comme ArchJava. Dans ce sens, ArchJava présente de sérieuses contre-indications pour être utilisé comme langage de description d'architecture, les arguments en faveur de l'intérêt d'une bonne séparation entre la description de l'architecture et sa mise en œuvre n'étant pas facile à garantir avec ArchJava.

3.2.2 Fractal

Malgré l'adoption des modèles à composants industriels comme EJB [DeMichiel, 2003] ou CCM [OMG, 2002a], la nécessité d'avoir un modèle plus léger et plus proche des concepts des langages de programmation s'est fait ressentir. Le modèle de composant Fractal [Bruneton *et al.*, 2004b] réalisé dans le cadre du consortium *ObjectWeb* par France Telecom R&D et par l'INRIA cherche à combler ce besoin. Fractal vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants, ainsi qu'une séparation claire des préoccupations fonctionnelles et non fonctionnelles.

3.2.2.1 Présentation du modèle de composant

Le modèle de composant Fractal possède deux niveaux d'abstractions. Le modèle général aborde de manière abstraite la notion de système à base de composants. Ce modèle comporte trois notions principales, la notion de cellule (kell), qui possède un contenu (plasm) et une membrane (membrane). Ce modèle est inspiré par une métaphore entre un composant et une cellule biologique. Le modèle concret spécialise ce modèle général pour obtenir un modèle proche des langages de programmation. Cette spécialisation introduit un système de type inspiré du langage de programmation ciblé.

On retrouve au niveau du modèle concret de composant Fractal les concepts de composant, connecteur et

configuration.

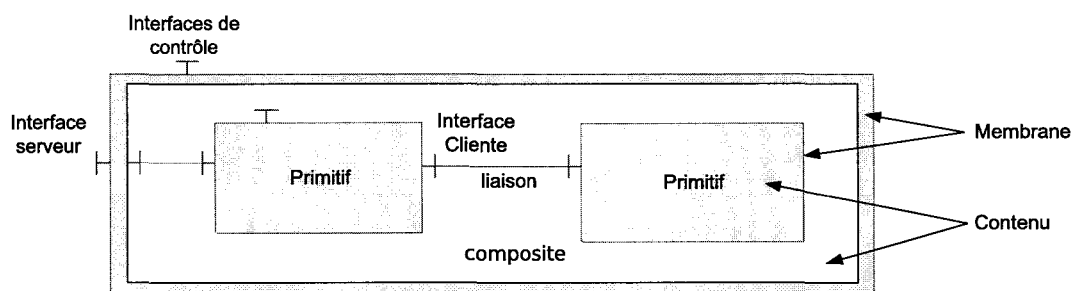


FIG. 3.4 – Modèle de composant Fractal

Le composant Fractal

Un composant Fractal est formé d'une membrane et d'un contenu. La membrane définit, par l'intermédiaire d'un ensemble d'interfaces, les services requis et fournis par le composant. Le contenu d'un composant Fractal permet de différencier deux sortes de composant : les primitifs et les composites. Le contenu d'un primitif identifie un module logiciel (classe, ensemble de fonctions, *etc.*) permettant de réaliser les services du composant primitif. Le contenu d'un composite définit un assemblage de composants permettant de mettre en œuvre les services du composite (voir Figure 3.4).

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métier et les interfaces de contrôle (la membrane correspond à l'ensemble de ces interfaces de contrôle). L'ensemble des interfaces métier et de contrôle d'un composant définit son type. Les interfaces métier sont les points d'accès externes au composant alors que les interfaces de contrôle prennent en charge des propriétés non fonctionnelles du composant comme la gestion de son cycle de vie ou de ses liaisons. Une interface Fractal est composée d'un nom, d'une signature et d'un type. Dans la projection du modèle en Java, la signature d'une interface est une interface Java.

Fractal met à disposition un ensemble d'interfaces de contrôle qui sont la gestion du cycle de vie (*life-cycle controller*), du nom (*name-controller*), des liaisons (*binding-controller*), des attributs (*attribute-controller*), des parents (*super-controller*) et du contenu (*content-controller*) pour les composites. Le modèle étant extensible, il est possible de définir ses propres interfaces de contrôle.

Une interface Fractal métier est du type client ou serveur. Une interface serveur identifie les services offerts par un composant alors qu'une interface client spécifie les fonctionnalités qu'un composant requiert pour son fonctionnement.

Le connecteur Fractal

Comme Darwin, Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie comme un lien orienté entre une interface client et une interface serveur. Ce lien permet aux composants d'interagir. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

La membrane possède à la fois des interfaces externes, qui sont accessibles de l'extérieur du composant, et des interfaces internes, accessibles seulement par son contenu. Dans le modèle Fractal, une interface interne ne peut exister que symétriquement à une interface externe. Ce mécanisme d'interface interne sert principalement à

permettre les traversées de membranes des composites en conservant la sémantique de la liaison. Ainsi la liaison permet de définir à la fois les interactions entre deux composants mais permet aussi de définir les liens de délégation entre les interfaces d'un composite et les composants de son contenu.

La configuration Fractal

Fractal est un modèle hiérarchique, un composant peut posséder au sein de son contenu d'autres composants. Il est identifié dès lors comme un composite. Le contenu d'un composite définit une configuration pour la mise en œuvre des services définis au niveau de ses interfaces métiers. Enfin, et c'est une spécificité du modèle Fractal, un composant peut appartenir au contenu de deux composites qui ne sont pas imbriqués l'un dans l'autre. Il est alors dit *partagé*.

Différentes mises en œuvre du modèle de composant

Fractal dispose de différentes mises en œuvre du modèle de composant pour différents langages de programmation : Julia [Bruneton *et al.*, 2004a], l'implantation de référence en Java, Think [Fassino *et al.*, 2002] une implémentation en C du modèle pour le développement d'applications embarquées et de noyaux extensibles de systèmes d'exploitation, FracNet [Escoffier and Donsez, 2005] une implémentation pour le *framework* .Net., ou encore AO-Kell [Seinturier, 2005], PLASMA [Layaïda and Hagimont, 2005] et FracTalk ⁶ des mises en œuvre du modèle à l'aide respectivement des langages AspectJ, C++, et SmallTalk.

L'ADL Fractal

En amont du modèle de composant Fractal, il est possible de décrire l'architecture d'une application à base de composants à l'aide de l'ADL Fractal. Basé sur XML, ce langage définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes de composants, d'interfaces, de liaisons et d'attributs.

L'ADL Fractal, dans la lignée d'ADL comme xADL [Dashofy *et al.*, 2001], est un ADL extensible. Il permet d'intégrer facilement de nouveaux concepts au niveau de la description de l'architecture en modifiant la grammaire du langage. Cette grammaire est en effet construite à partir d'un ensemble de modules prenant en charge chacun une caractéristique du modèle de composant comme les interfaces, les liaisons ou les attributs. Il est possible de définir son propre module pour ces propres caractéristiques ou simplement redéfinir des caractéristiques existantes pour définir la syntaxe de son langage de description d'architecture.

3.2.2.2 Le déploiement

Comme ArchJava présenté dans la sous-section précédente, le déploiement n'est pas pris en compte dans la spécification Fractal. Dans la mise en œuvre par défaut du modèle Fractal, les composants Fractal appartiennent à la même JVM. Un sous projet de Fractal, Fractal-RMI permet l'utilisation de RMI (Remote Methode Invocation) pour les interactions entre composants distants. Cependant, le placement de composants reste à spécifier manuellement.

Une proposition récente, FROGi [Donsez *et al.*, 2004] vise à réunir le modèle de composant Fractal et la plateforme de services OSGi [Alliance, 2003]. L'approche FROGi permet de bénéficier des atouts du modèle de composant Fractal pour la conception hiérarchique du système à base de composants tout en permettant de bénéficier des avantages des plates-formes orientées services OSGi pour les aspects distribution, déploiement, et désactivation de composants, trois fonctionnalités non abordées pour le moment par le modèle de composant Fractal.

⁶<http://csl.enscm-douai.fr/FracTalk>

3.2.2.3 Définition de la dynamique

Dans Fractal, l'architecture logicielle correspond à un graphe d'instances de composant interconnectées. Au cours du cycle de vie cette architecture évolue : création/destruction de liaisons ou d'instances. En Fractal, la description de l'architecture au travers de l'ADL est statique.

Néanmoins, le modèle de composant Fractal offre par l'intermédiaire de son API des mécanismes pour faire évoluer dynamiquement l'architecture d'une application. Il est ainsi possible de reconfigurer dynamiquement une architecture et créer à l'exécution de nouveaux composants.

Reconfiguration dynamique

Fractal offre une API pour naviguer et introspecter la structure des composants : découvrir des interfaces (*component*), le contenu des composites (*content-controller* et *super-controller*) et naviguer le long des liaisons entre interfaces (*binding-controller*).

En outre, il est possible de modifier dynamiquement la structure d'une application Fractal : modifier le contenu des composites (*content-controller*), créer ou détruire des liaisons (*binding-controller*).

Instanciation

Une deuxième caractéristique intéressante liée à la gestion de la dynamique concerne la création de nouvelles instances de composant. Fractal offre des modèles pour créer dynamiquement de nouveaux composants : l'instanciation simple à l'aide d'une *Factory* ou les *Templates*.

La création à l'aide d'une *Factory* crée pour un composant primitif la membrane du composant à partir de la définition des interfaces que l'on trouve dans la définition du type de composant. Il place la membrane autour de l'entité réalisant les opérations de l'interface. La création d'un composite à l'aide d'une *Factory* crée, là aussi, la membrane du composite à partir d'une définition des interfaces du type de ce composite comme un composant normal. Cependant, aucun contenu n'y est ajouté. C'est la principale différence avec la notion de *Template* qui instancie un composite et une configuration pour ce composite. La notion de *Template* enrichit quelque peu la notion de type de composant que l'on trouve dans Fractal en offrant la possibilité de définir un type de composite par l'intermédiaire des interfaces qu'il fournit et qu'il requiert, mais aussi par l'intermédiaire de la configuration mise en place en son sein pour réaliser les opérations liées à ces interfaces.

Tout comme Java, Fractal ne définit pas d'opération explicite de destruction de composants. La seule façon de retirer un composant d'une application est de supprimer toutes ses relations (liaisons, composition) de tous les composants de l'application et de l'arrêter. Rien dans Fractal ne garantit qu'il sera alors détruit, seulement qu'il n'aura plus d'impact sur l'application.

3.2.2.4 Description de comportement et contrats d'interface

Fractal est un modèle de composant extensible. Il est possible d'étendre le modèle pour y ajouter des éléments structurels ou comportementaux. Ainsi, si Fractal ne propose pas au sein du modèle de composant de mécanismes pour spécifier le comportement de chaque composant de manière abstraite, différents travaux en cours viennent étendre ce modèle afin d'y ajouter les différents niveaux de contrats d'interface. Ainsi, ConFract [Collet *et al.*, 2005] travaille sur la définition d'assertions au niveau des interfaces de composants Fractal. Barros *et al* [Barros *et al.*, 2005] proposent un modèle de description de comportement pour les composants hiérarchique en se basant sur Fractal.

3.2.2.5 Exemple

La figure 3.5 décrit l'architecture d'un serveur Web à l'aide de Fractal. On constate la possibilité au niveau de l'architecture de définir des interfaces multiples. La sémantique de ces interfaces multiples n'est pas définie au niveau de l'ADL mais sera mis en œuvre au niveau du contenu du composant.

La phase de développement du contenu du composant est limitée à la mise en œuvre d'une classe Java qui doit implanter les interfaces serveur et posséder des références sur ces interfaces clientes comme le présente la figure 3.6. Cette mise en œuvre très simple de la partie métier des composants Fractal est très intéressante, car en s'intégrant très bien au niveau du langage cible, le développeur conserve ses habitudes de développement et bénéficie des avantages fournis par les nombreux environnements de développement intégrés comme Eclipse ou NetBeans pour Java.

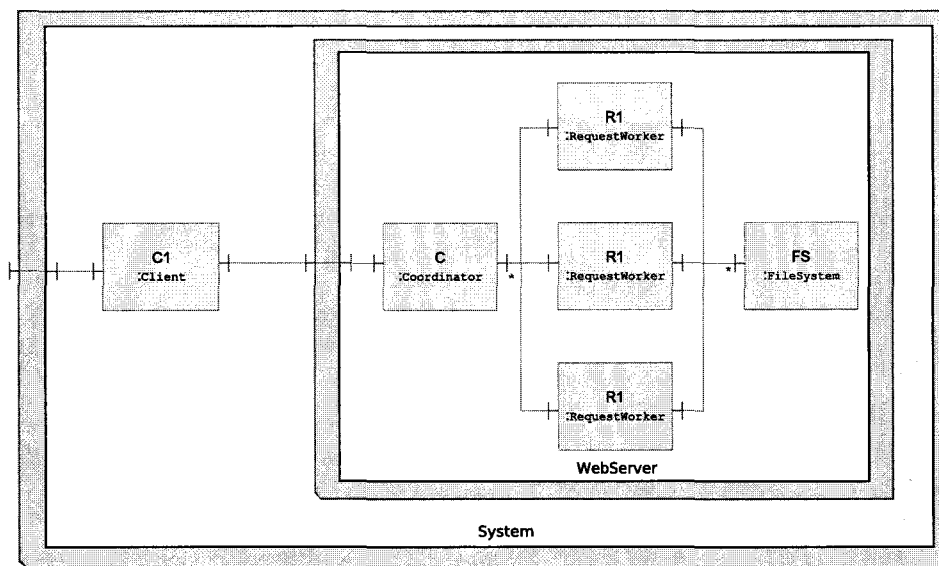


FIG. 3.5 – Mise en œuvre d'un serveur Web avec Fractal

```

1 public class Coordinator implements ServiceHttp, BindingController {
2     //////////////////////////////////////
3     ////Référence sur l'interface requise (interface collection) //
4     //////////////////////////////////////
5     protected HashMap<String,handleRequest> h = new HashMap<String,handleRequest> ();
6
7     //////////////////////////////////////
8     ////Interface ServiceHttp correspond à l'interface fournit de ce composant //
9     //////////////////////////////////////
10    public Document get ( String URL ){ ... }
11    public Document post ( String URL, Data ){ ... }
12
13    //////////////////////////////////////
14    ////Interface BindingController permet de gérer dynamiquement les interactions de ce composant avec les composants RequestWorker //
15    //////////////////////////////////////
16    public String [] listFc () { ... }
17    public Object lookupFc ( String clientName ) throws NoSuchInterfaceException { ... }
18    public void bindFc ( String clientName, Object serverName ) throws NoSuchInterfaceException, IllegalBindingException, IllegalLifecycleException { ... }
19    public void unbindFc ( String clientName ) throws NoSuchInterfaceException, IllegalBindingException, IllegalLifecycleException { ... }
20 }

```

FIG. 3.6 – Structure d'une classe prenant en charge le métier d'un composant Fractal, le contenu du composant

3.2.2.6 Évaluation de Fractal

Fractal est un modèle simple et léger. Sa prise en main est aisée pour les programmeurs et les architectes issus de l'objet. En outre, comme ArchJava, les concepts architecturaux sont encore présents au niveau de l'implantation, ce qui offre une traçabilité très intéressante pour la compréhension de l'architecture. Son ADL offre une représentation de l'architecture indépendamment de tout langage de programmation. D'un point de vue programmatique, son intégration à Java est très propre et ne contient aucune extension syntaxique, ce qui permet de bénéficier de l'ensemble des outils modernes de développement d'applications. La structuration entre le code métier et le code lié à l'architecture est meilleure que ce que l'on peut trouver dans ArchJava. Finalement, son architecture extensible permet d'étendre facilement Fractal pour la définition de nouveaux concepts architecturaux.

La principale lacune de Fractal est paradoxalement sa légèreté. Sa proximité avec le modèle objet et le manque d'information liée au comportement de l'application font de Fractal ADL (non étendu) un langage pauvre et difficile à analyser. L'analyse de la composition au niveau d'une liaison se limite à un travail de compatibilité des signatures des interfaces comme dans un modèle objet. Cependant, sa facilité d'extension offre la possibilité de surmonter cette lacune.

En outre, son modèle d'interface proche de Java, avec une distinction entre les interfaces fournies qui ne possèdent que des opérations fournies et les interfaces requises qui ne possèdent que des opérations requises (à la différence de la définition de port dans ArchJava, Darwin, ou UML 2.0, voir Section 3.2.1, 3.1.2, 3.3.2) rend le modèle Fractal peu adapté à la modélisation de service. Un composant *client* souhaitant interagir avec une application de facturation aura en effet les méthodes requises suivantes « Créer-CommandeClient », « CréerLigne-CommandeClient », « Valider-CommandeClient », *etc.* et la méthode fournie suivante « Demander-NuméroClient ». Pour une telle interaction Fractal impose la définition de quatre interfaces (2 requises et 2 fournies) et de deux liaisons. La notion de port dans les ADLs est un agrégateur de méthodes en vue modéliser l'interaction entre deux entités distantes. Cette absence de notion de port engendre la définition de deux interfaces, une serveur et une client et deux liaisons pour la modélisation d'une même interaction. Si du point de vue de la mise en œuvre, ceci n'est pas très contraignant, d'un point de vue conceptuel c'est regrettable.

Enfin, la description de l'ADL reste pour le moment statique. Cette lacune n'est pas gênante au niveau de la mise en œuvre, car Fractal dispose des mécanismes de gestion de la dynamique au niveau du modèle de composant. Cependant, conceptuellement, comme la description d'architecture constitue un des documents pivots pour spécifier la structure d'une application, il est nécessaire que cette dynamique puisse être représentée ou du moins contrainte.

3.2.3 SOFA

SOFA (*SOFTware Appliance*) [Kalibera and Tuma, 2002] est un modèle de composant développé par l'université Charles (Prague). Son objectif est la construction d'application à partir d'une composition de composants.

3.2.3.1 Présentation du modèle de composant

SOFA propose les éléments caractéristiques des langages de description d'architecture, à savoir le composant, le connecteur, et la configuration. SOFA est sans aucun doute l'approche la plus à cheval entre un langage de description d'architecture ou un modèle de composant académique. De par son équipe d'origine ayant travaillé sur les intergiels et sur CORBA, nous avons choisi quelque peu arbitrairement de classer SOFA dans les modèles de construction d'application à base de composants.

Le composant SOFA

Dans SOFA, les composants sont primitifs ou composites. Un composite est défini comme un assemblage de sous-composants alors qu'un primitif ne peut contenir de sous-composants. Un composite ne contient pas de fonctionnalités propres, toutes les fonctionnalités sont définies au sein des primitifs.

Les composants de SOFA sont définis par leur *frame* et leur *architecture*. La notion de *frame* est à rapprocher à la notion de type de composant présentée dans la section 2.1.1.2. C'est une approche boîte noire du composant. Une *frame* définit les interfaces requises et fournies par le composant. Elle déclare aussi de manière optionnelle certaines propriétés permettant de paramétrer le composant. SOFA est un modèle hiérarchique. Ces objectifs principaux sont l'analyse de la composition de composants. L'*architecture* est une approche boîte grise du composant. Elle définit soit un assemblage de sous-composants permettant de réaliser une *frame* soit elle indique que le composant est un primitif. Dès lors, aucun sous-composant ne concourt à la réalisation des services de la *frame*.

Le connecteur SOFA

Les connecteurs dans SOFA sont des éléments de premier ordre au même titre que les composants. Ils n'ont pas fondamentalement de différence avec les composants. Ils sont définis à l'aide de *connector-frame* pour la vision boîte noire et *connector-architecture* pour la vision boîte grise. L'objectif des connecteurs de SOFA est de permettre de capturer uniquement la logique métier au sein des composants pendant que les connecteurs prennent en charge la sémantique de l'interaction et les problèmes de déploiement.

Les *connector-frames* sont définis à partir d'un ensemble de rôles. Un rôle est une interface générique pouvant être liée à une interface de composant. Le *connector-architecture* décrit la partie interne du connecteur. Comme pour le composant, cette partie peut être déclarée primitive directement implantée ou elle peut être déclarée composée, elle contient alors un ensemble de composants et de connecteurs.

SOFA fournit trois types de connecteur prédéfinis : *CSProcCall* avec une sémantique d'appel de procédure, *EventDelivery* avec une sémantique d'envoi de message et *DataStream* pour une sémantique d'échange de flux. Il fournit aussi un type de connecteur utilisateur qui permet de définir sa propre sémantique. Cette dernière sera alors réalisée à l'aide d'un assemblage de composants et de connecteurs. De manière implicite, si aucun connecteur n'est spécifié, un connecteur possédant une sémantique d'appel de procédure est introduit.

La configuration SOFA

Il y a quatre moyens de venir s'accrocher à une interface de composant. La première appelée le *binding* permet de créer une liaison entre une interface requise et une interface fournie de deux composants de même niveau. La deuxième appelée *delegating* lie une interface fournie du composant englobant à une interface fournie d'un de ses sous-composants. La troisième nommée *subsuming* permet de lier un port requis d'un sous-composant à un port requis du composant englobant. Enfin, la dernière nommée *exempting* permet de déclarer qu'une interface d'un composant n'est pas liée. Les trois premiers types d'accroche sont réalisés par des connecteurs.

DCUP : une plate-forme d'exécution pour SOFA

SOFA fournit une mise en œuvre de référence de son modèle : DCUP [Plasil *et al.*, 1997]. Son objectif principal est la transparence de l'adaptation dynamique. DCUP permet le remplacement de composants à chaud au cours de l'exécution. Ses objectifs principaux sont l'adaptation dynamique, la prise en compte de l'état d'exécution pendant l'adaptation, et le support de téléchargement automatique et dynamique des composants. DCUP étend le modèle de composant SOFA pour permettre la mise à jour de composants à l'exécution.

3.2.3.2 Le déploiement

Dans SOFA, le déploiement est configuré par un descripteur de déploiement. Pour chaque instance de composant définie dans l'architecture, le descripteur de déploiement précise l'hôte où le composant va fonctionner. La tâche de déploiement est centralisée et suppose que chaque hôte est un serveur qui a la capacité d'instancier à distance des composants.

Génération de connecteurs en utilisant CORBA

Dans SOFA, les connecteurs sont utilisés pour connecter des composants s'exécutant sur des hôtes différents en fournissant des mécanismes d'appel à distance sur les composants. D'un point de vue conceptuel, SOFA ne définit aucune contrainte sur l'intergiciel utilisé. Au niveau de la mise en œuvre, SOFA travaille au dessus de CORBA. Ainsi, les talons sont générés automatiquement à partir d'une description IDL obtenue à partir de la description CDL (*Component Definition Language*) de SOFA. Cette génération est exécutée au moment du déploiement quand il est possible de connaître quels sont les composants qui doivent interagir par l'intermédiaire de tels connecteurs.

Gestion des versions

SOFA introduit dans son modèle de composant la notion de nœud. Un nœud SOFA (*SOFA node*) est composé de cinq parties. Le cœur de ce nœud est le *Template repository* (TR). Cet entrepôt contient les mises en œuvre des composants (les binaires) et leurs descriptions abstraites (CDL description). Parmi les autres éléments d'un nœud SOFA, on compte une partie fabrication, deux parties en charge du déploiement distribué des composants et une partie en charge de l'exécution des composants.

Un nœud SOFA n'est pas nécessairement attaché à un unique hôte, il peut être réparti sur plusieurs machines.

Un nœud SOFA supporte au niveau de l'entrepôt un mécanisme de gestion de version qui permet d'y ranger plusieurs versions d'un même composant. Ce système de version est très intéressant au niveau du suivi de l'évolution d'une architecture logicielle.

3.2.3.3 Définition de la dynamique

Comme pour Fractal, la description de l'architecture dans SOFA est statique. Une architecture au sens SOFA contient une liste des instances de composant ainsi que leurs liaisons. Si ce schéma sert au moment du déploiement pour l'instanciation et la configuration des composants, ce schéma peut rapidement devenir caduc suite à l'évolution interne de l'application. En outre, suite à une évolution liée à l'action de l'architecte, seul un mécanisme de gestion de version permet de garder une trace de l'évolution de l'architecture.

3.2.3.4 L'expression du comportement

Les interactions peuvent être capturées formellement au sein de SOFA. Chaque appel de méthode ou retour forme un événement. Cet événement est une unité atomique pour la description du comportement, il ne peut pas être divisé en sous-événements. Ainsi pour une méthode *m*, quatre types de message sont définis :

- *!m^* définissant un appel de méthode côté client,
- *?m^* définissant la réception d'un appel de méthode côté serveur,
- *!m\$* définissant l'émission du message de retour du serveur vers le client,
- *?m\$* définissant la réception du retour par le client.

Deux opérateurs sont introduits pour décrire la séquence (;) et l'alternative (+). Ce langage fournit une syntaxe concise pour la définition de LTS (*Labelled Transition Systems*) qui modélise le comportement d'un composant.

Trois types de protocole sont définis. Le premier exprime l'échange de messages au niveau d'une interface, le deuxième exprime l'échange de messages pour un *frame*, le troisième définit l'échange de messages pour une *architecture*. Les deux premières expressions de comportement sont écrites par l'architecte. La dernière est déduite à partir des sous-composants de l'architecture.

À partir des descriptions de comportement des éléments d'une architecture SOFA, il est possible de vérifier la compatibilité sémantique d'un code source, c'est-à-dire, vérifier si celui-ci respecte la description de comportement du *frame*. De même, il est possible de vérifier si une *architecture* respecte la description de comportement de son *frame*.

3.2.3.5 Exemple

La figure 3.7 présente l'exemple du serveur Web décrit à l'aide du langage CDL de SOFA. Le langage CDL présente de nombreuses similitudes avec le langage IDL2. Il se distingue par la possibilité de décrire des types de composant (*frame*) et des mises en œuvre de composant (*architecture*).

3.2.3.6 Évaluation de SOFA

SOFA est un modèle de composant issu d'une équipe qui a beaucoup travaillé dans le domaine des objets répartis. Bénéficiant des différents travaux dans le domaine de l'architecture logicielle, il propose un modèle classique avec une bonne séparation entre la notion de type de composant (appelé *frame*) et la notion d'implantation (appelé *architecture*). Les rôles introduits utilisés dans les connecteurs permettent de gérer la répartition au sein des composants.

Le principal attrait de SOFA repose sur une description du comportement des composants permettant une analyse de l'assemblage et une analyse de la conformité d'une implantation (*architecture*) par rapport à son type (*frame*). Il propose dès lors un contrat de niveau 3 selon la taxonomie de Beugnard *et al* [Beugnard *et al.*, 1999] pour la description des interfaces de composant. Cependant, comme Darwin, le contrat de niveau 2 est inexistant dans SOFA, il n'y a pas de possibilité de préciser ce que s'engage à respecter le composant.

SOFA comme Fractal propose au niveau du modèle de composant une notion d'interface requise ou fournie plutôt qu'une notion de port. Dès lors, comme pour Fractal, si cette distinction ne pose pas de problème au niveau de la mise en œuvre, elle limite la pertinence du modèle de composant au moment de la conception en ne permettant pas d'identifier ensemble les opérations fournies et requises participant à la réalisation d'un service avec un tiers. Ce problème est toutefois nuancé dans SOFA car les mécanismes IDL de CORBA de passage de valeur en *IN*, *OUT* et *INOUT* permet de contourner en partie le problème.

Enfin, SOFA comme Fractal propose une approche statique de la description d'une architecture logicielle. Il ne propose pas un modèle d'invariants du système permettant de définir un cadre pour la dynamique de l'architecture. Ainsi, une architecture représente un assemblage de composants à un instant *t*. Dès lors, même si la mise en œuvre du modèle permet le remplacement à l'exécution de composants, la prise en compte de la dynamique au niveau du modèle est assez restreinte.

3.2.4 Bilan sur les modèles de composant académiques

L'étude des différents modèles à composants récents montre l'intérêt de conserver la notion de composant logiciel et d'architecture logicielle dans le développement des applications. Les composants logiciels apportent une unité de structuration du code de l'application, une unité de réutilisation et de déploiement ainsi qu'une unité

```

1 module SOFA {
2   module example {
3     module webserver {
4       interface ClientCoordinatorInterface {
5         //définition des méthodes en idl2
6       };
7       interface CoordinatorRequestWorkerInterface {
8         //définition des méthodes en idl2
9       };
10      interface RequestWorkerFileSystemInterface {
11        //définition des méthodes en idl2
12      };
13
14      frame Coordinator {
15        requires
16          CoordinatorRequestWorkerInterface Crw;
17        provides
18          ClientCoordinatorInterface Cc;
19      };
20      frame RequestWorker{
21        provides
22          CoordinatorRequestWorkerInterface Rwc;
23        requires
24          RequestWorkerFileSystemInterface Rwf;
25      };
26      frame FileSystem{
27        provides
28          RequestWorkerFileSystemInterface FsRw;
29      };
30      frame WebServer{
31        provides
32          ClientCoordinatorInterface Cc;
33      };
34    };
35  };
36 };
37 architecture CUNI ::SOFA::example::webserver ::Coordinator implements SOFA::example::webserver ::Coordinator primitive;
38 architecture CUNI ::SOFA::example::webserver ::RequestWorker implements ::SOFA::example::webserver ::RequestWorker primitive;
39 architecture CUNI ::SOFA::example::webserver ::FileSystem implements ::SOFA::example::webserver ::FileSystem primitive;
40
41 architecture CUNI ::SOFA::example::webserver ::WebServer implements ::SOFA::example::webserver ::WebServer {
42   inst ::SOFA::example::webserver ::Coordinator c;
43   inst ::SOFA::example::webserver ::RequestWorker r1;
44   inst ::SOFA::example::webserver ::RequestWorker r2;
45   inst ::SOFA::example::webserver ::RequestWorker r3;
46   inst ::SOFA::example::webserver ::FileSystem fs;
47
48   bind c:Log to logger:Log using CSProcCall;
49   bind r1:Rwf to fs:Fsfw using CSProcCall;
50   bind r2:Rwf to fs:Fsfw using CSProcCall;
51   bind r3:Rwf to fs:Fsfw using CSProcCall;
52   delegating Cc to c:Cc;
53   ....
54 };

```

FIG. 3.7 – Mise en œuvre d'un serveur Web avec SOFA

granulaire offrant une vue plus ou moins précise des éléments constitutifs de l'application. La prise en compte des connecteurs permet de séparer les aspects liés à la mise en œuvre de la communication des aspects métiers de l'application. Les capacités d'un modèle de composant à pouvoir reconfigurer dynamiquement une application, c'est-à-dire modifier les éléments constitutifs d'une application et leur interaction est un pas en avant vers la construction d'applications plus flexibles.

Si le choix de retrouver les concepts architecturaux au niveau de la mise en œuvre d'une application semble une proposition intéressante pour conserver une bonne traçabilité de la conception à la mise en œuvre, il reste tout de même d'un point de vue programmatique nécessaire de séparer les parties du code concernant la gestion de l'architecture des parties métier de l'application.

Finalement, on constate que chacune de ces approches a redéfini son propre langage de description d'architecture bien que l'on trouve facilement de nombreuses similitudes entre ces langages. Comme ces langages ont souvent une place centrale dans ces modèles de composant académiques, il serait intéressant de disposer d'un

modèle de description d'architecture abstrait commun afin de favoriser l'interopérabilité entre ces approches de construction d'applications à base de composants.

3.3 Les standards

Les ADLs ont pendant longtemps souffert d'être une approche très académique, peu utilisée par le milieu industriel du génie logiciel. Ce manque d'utilisation s'explique par l'absence pendant longtemps de langage standardisé. Récemment, la prise en compte du concept de composant dans le langage UML (*Unified Modelling Language*) de l'OMG (*Object Management Group*) pour la conception d'application par objets et l'avènement du langage AADL défini par la SAE (*Society of Automotive Engineers*), organisme de standardisation mondiale dans le domaine du transport ont montré le besoin pour les industriels de disposer d'un langage de description d'architecture standard.

3.3.1 AADL

MetaH [Vestal, 1998] est un ADL dédié aux systèmes avioniques qui a été développé par Honeywell depuis 1991 pour le DARPA et l'US Army. En 2001, MetaH a été pris comme base pour un effort de standardisation visant à définir AADL (Architecture Analysis & Design Language), un langage de description d'architectures avioniques, sous l'autorité du SAE (*Society of Automotive Engineers*), organisme de standardisation mondiale dans le domaine du transport.

L'étude d'AADL est intéressante pour au moins deux raisons. Ce langage issu d'un organisme de normalisation rencontre un très bon accueil auprès des industriels du domaine. En outre, AADL est un des premiers langages de description d'architecture à proposer de définir des informations sur le niveau quatre de la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999].

AADL vise à répondre aux besoins spéciaux des systèmes embarqués temps réel tels que les systèmes avioniques. En particulier, il peut décrire les mécanismes standards de flots de contrôles et de données utilisés dans de tels systèmes, et des aspects non fonctionnels importants tels que les exigences temporelles, les fautes et erreurs, le partitionnement temporel et spatial et les propriétés de sûreté et de certification.

3.3.1.1 Présentation du modèle

AADL est à l'origine défini par différents partenaires du domaine de l'avionique, mais il s'adresse dans les faits à tous les systèmes embarqués temps réel, et pas uniquement aux systèmes avioniques.

AADL est d'abord un langage textuel. Une annexe au standard, sur le point d'être votée, définit une notation graphique associée. Une autre annexe définira un profil UML pour AADL.

Les composants AADL

AADL considère deux niveaux de description du composant. Le premier, le type, correspond à l'interface fonctionnelle du composant, ce qui est visible des autres composants. Le second, l'implémentation, décrit le contenu du composant. Le modèle est hiérarchique, le contenu d'un composant consiste à définir ses sous-composants, les connexions en son sein et les délégations.

En plus d'être typé, chaque composant appartient à une catégorie prédéfinie. Ces catégories représentent la nature des éléments manipulés. On retrouve dix grandes catégories dans AADL. Ainsi un composant peut être une mémoire (memory), un périphérique (device), un processeur (processor), un bus, une donnée (data), un sous-programme (subprogram), un processus léger (thread), un groupe de processus léger (thread group), un processus (process), un système (system).

Les composants communiquent les uns avec les autres par l'intermédiaire de ports. Ces ports sont des points d'entrée et/ou de sortie d'un composant. Il y a trois catégories de port :

- *data* : transportent des données ;

- *event* : semblables aux signaux ;
- *data event* : des signaux qui transportent des données.

3.3.1.2 Les connecteurs AADL

Les interactions entre les composants sont matérialisées par la définition de connexions entre leurs ports. Une connexion permet de relier deux ports. Les ports peuvent être déclarés en entrée (*in*), sortie (*out*), ou entrée-sortie (*in out*). Une vérification est faite qu'il y a bien conformité de type et de sens entre les ports connectés. Les connecteurs n'ont pas de comportement associé, mais il est possible de leur ajouter des propriétés sur le délai de transmission des données (voir le paragraphe *propriétés*).

La configuration AADL

De par la nature même des composants, le modèle d'AADL est hiérarchique. Par exemple, un processus peut contenir des processus légers. En outre, la notion de système qui peut lui-même être composé de sous-systèmes permet de décomposer le système sur une infinité de niveaux.

Le schéma 3.8 montre un système contenant deux processus, eux-mêmes contenant chacun un processus léger. Une succession de ports et de connexions établit une liaison entre les deux processus légers.

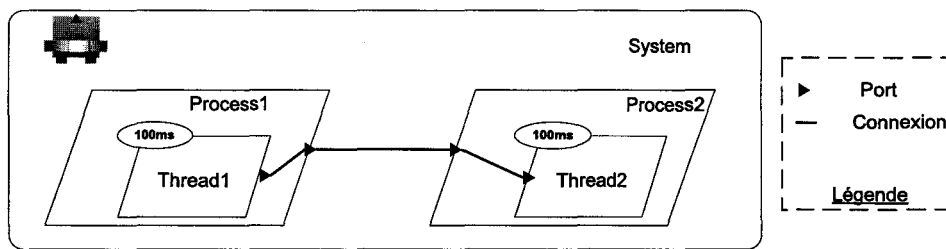


FIG. 3.8 – Exemple simple AADL

Les propriétés

À chaque élément de la description architecturale, on peut associer des propriétés et leur attribuer des valeurs. Celles-ci permettent de caractériser le composant. Certaines propriétés sont prédéfinies, c'est-à-dire qu'elles sont identifiées par un nom, un type et la liste des catégories de composant sur lesquelles elles s'appliquent. Par exemple, les tâches (*processus léger*) disposent de propriétés temps-réel telles que la période, l'échéance ou la durée d'exécution. De nouvelles propriétés ainsi que de nouveaux types de propriétés peuvent également être définis par l'utilisateur et associés à des catégories de composant. Ce mécanisme de propriétés permet de prendre en compte toute notion propre aux besoins de l'utilisateur.

Les annexes

Un autre mécanisme d'extension, appelé l'*annexe*, existe dans AADL. Elle permet d'utiliser des déclarations écrites dans d'autres sous-langages pour ajouter de l'information sur un élément de l'architecture. Complémentaire des propriétés, une annexe sert à ajouter des précisions facultatives, alors qu'une propriété est très liée à la description architecturale.

3.3.1.3 Le déploiement

La prise en compte des éléments matériels dans la description d'une architecture produite avec AADL engendre automatiquement la prise en compte du déploiement de l'application sur une infrastructure matérielle. Cette prise en compte de l'infrastructure matérielle est dirigée par le besoin de connaître un maximum d'information sur cette infrastructure afin de vérifier, à partir de la description d'architecture, certaines propriétés des applications temps réel. La granularité du déploiement dans AADL ne consiste donc pas à placer les composants logiciels sur une infrastructure physique constituée d'hôtes, elle met en relation les éléments liés à l'infrastructure et les éléments liés au logiciel, l'ensemble de ces composants communiquant au travers de ports.

3.3.1.4 Définition de la dynamique

AADL introduit au cœur du langage un mécanisme de gestion de la reconfiguration : les modes. Les modes représentent des configurations alternatives du système, seulement un mode étant actif à la fois. Spécifier un ensemble de modes permet de décrire les différentes configurations admissibles d'un système. Le passage d'un mode à un autre est spécifié par un diagramme de transition et déclenché par un évènement. Ce mécanisme de mode permet de capturer une partie de la dynamique de l'architecture en spécifiant les évolutions possibles de l'architecture.

Ce mécanisme a cependant l'inconvénient d'être une approche déclarative où toutes les configurations possibles doivent être décrites. Cela peut, dans le cas d'architectures possédant un degré de variabilité important, se révéler difficile.

3.3.1.5 Spécification du comportement

La spécification du comportement dans AADL n'est pas encore complètement définie. Si AADL fournit la possibilité de décrire un diagramme de flot permettant de décrire avec une abstraction importante le transfert des données au sein des composants AADLs, il n'y a pas encore de modèle permettant de décrire précisément les interactions entre composants. Une annexe au standard est cependant planifiée pour proposer un langage standard permettant de décrire ces interactions.

3.3.1.6 Exemple

Nous reprenons dans la figure 3.9 l'exemple du serveur Web décrit à l'aide de la syntaxe graphique de AADL. Cette architecture contient cinq processus communiquant par l'intermédiaire de leurs ports. Cette description de l'architecture n'est pas descendue au niveau de granularité du processus léger (*thread*) pour une question de lisibilité, mais l'architecture de chaque processus pourrait être décrite à l'aide de *thread*. Nous fournissons certaines informations relatives au déploiement à l'aide de la description des bus, des processeurs et de la mémoire. Ainsi au sein du serveur Web, le client sera déployé sur un système contenant un processeur et de la mémoire. Ce système pourra alors communiquer avec un deuxième système par le biais d'Internet. Ce deuxième système contient quatre processus en charge du traitement de requêtes. Il pourra accéder à un troisième système par l'intermédiaire d'un réseau local (*LAN*) avec un troisième système, le système de fichiers.

3.3.1.7 Évaluation d'AADL

La première force d'AADL est de fédérer un ensemble d'industriels de l'industrie du transport autour de ce standard.

Le deuxième atout d'AADL réside dans sa prise en compte de plusieurs sémantiques au niveau du composant, entre autres, la description des éléments matériels qui sont peu définis dans les modèles académiques. Enfin, le

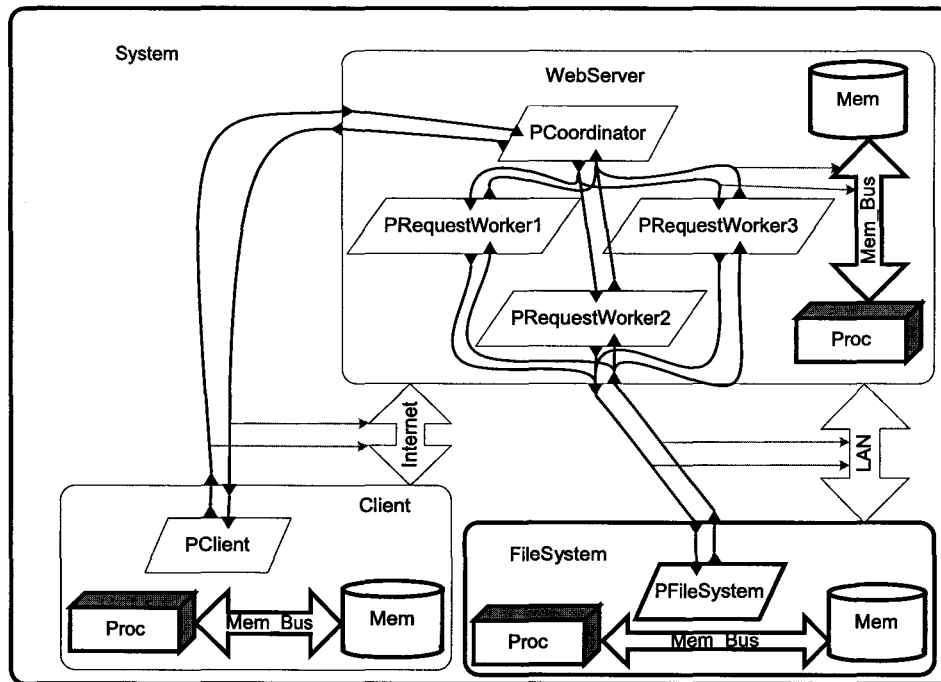


FIG. 3.9 – Architecture d'un serveur Web décrit à l'aide de AADL

troisième atout concerne les concepts de propriétés et d'annexes qui permettent d'ajouter de l'information non fonctionnelle à la description de l'interface du composant tel que des informations liées à la qualité de service fournie ou requise par un composant.

Cependant, il manque à AADL une abstraction du comportement des composants qui permettrait de spécifier la relation entre les sorties d'un composant et ses entrées. Il manque aussi pour le moment une sémantique claire de la composition. De ce fait, il existe des outils d'analyse qui exploitent les propriétés, les annexes et la description de l'assemblage pour garantir des propriétés de qualité de service, mais il n'y a pas de sémantique définie au niveau de la norme permettant de détecter des problèmes de composition liés au comportement des composants.

Enfin, l'exemple de la figure 3.9 montre que AADL est un langage de description d'architecture déjà très orienté mise en œuvre. Les concepts de base du langage ont une forte implication quant aux choix liés à la réalisation des modules logiciels. Son utilisation comme langage de conception utilisable tout au long du cycle de vie du logiciel reste encore à démontrer.

3.3.2 UML 2.0

Après le succès de la première version d'UML [OMG, 2003d], la deuxième version notée UML 2.0 [OMG, 2003c] introduit la notion de diagramme d'architecture, aussi appelée diagramme de structure composite. UML propose une notation unique pour couvrir toutes les phases du développement logiciel. Avec l'ajout de ce diagramme, UML veut combler une des lacunes de la première version en permettant la spécification de l'architecture d'une application.

3.3.2.1 Présentation du modèle

Inspiré des ADLs issus de la recherche académique, UML 2.0 propose les trois concepts fondamentaux des langages de description d'architecture à savoir le composant, le connecteur et la configuration.

Le composant UML 2.0

La notion de composant en UML 2.0 représente un ensemble de constructions qui peuvent être utilisées pour spécifier un système logiciel de taille et de complexité arbitraires. Un composant UML 2.0 est une entité modulaire et réutilisable fournissant et requérant des interfaces⁷ qui peuvent être potentiellement exposées par l'intermédiaire de ports. Un composant est vu comme une boîte noire. La partie interne d'un composant n'est accessible qu'au travers de ses interfaces. Le composant est une entité d'encapsulation. Ses dépendances sont explicitement définies. Il peut donc être facilement réutilisé au sein de plusieurs développements.

Une interface définit un type. Elle contient un ensemble de méthodes et/ou de contraintes. Une interface peut être attachée à un port fourni ou requis ou attachée directement à un composant.

Un port est une entité qui émerge d'un composant. Le port se comporte comme un point d'interaction du composant avec l'environnement typé par plusieurs interfaces. Un port peut être requis, fourni ou complexe. Un port requis ne possède que des interfaces requises et signifie que l'instance du composant doit être connectée à un port de l'environnement fournissant les méthodes requises. Un port fourni ne contient que des interfaces fournies. Un port complexe peut contenir des interfaces requises ou fournies. Ce type de port est très utile pour décrire l'interaction d'un composant avec son environnement dans le cas de service complexe. Un port peut se voir attacher un comportement pour décrire les interactions qui doivent se produire sur le port. Les ports peuvent être connectés si et seulement si leurs spécifications de structure et de comportement sont *compatibles*.

UML 2.0 définit deux types de composant : le composant basique (*Basic Component*) et le composant empaqueté (*Packaging Component*).

Au niveau des caractéristiques, un composant basique étend la notion de *classe*. C'est une entité instanciable qui interagit avec son environnement par l'intermédiaire de ports ou d'interfaces. La description du comportement de ce composant est fondée sur des machines à états.

La notion de composant empaqueté étend la notion de composant basique. Si le composant basique définit la manière dont le composant est vu à l'extérieur du composite, le composant empaqueté est une entité composée d'autres éléments liés de façon cohérente tout au long du processus de développement du logiciel.

Le connecteur UML 2.0

Comme pour les ADLs, les interactions entre les composants sont décrites par des connecteurs. Un connecteur est une entité qui relie des ports ou des interfaces de composant. Dans un diagramme d'architecture, une application ou un composant est constitué de l'assemblage d'autres composants par l'intermédiaire de ports interconnectés. Il existe plusieurs types de connecteur :

- le connecteur d'assemblage qui permet d'assembler deux instances de composant en connectant un port fourni d'un composant au port requis de l'autre composant,
- le connecteur de délégation qui permet de connecter un port externe au port d'un sous-composant interne. L'invocation d'un service sur le port externe sera transmise au port interne auquel il est connecté.

Plusieurs connecteurs peuvent être attachés au même port ou à la même interface.

⁷au sens UML

La configuration UML 2.0

Il y a deux façons de voir le composant en UML : la vision boîte noire présentée précédemment et une vision boîte blanche. Dans la vision boîte noire, le composant n'est défini qu'en fonction des éléments lui permettant d'interagir avec son environnement : ports, interfaces. Au contraire, la vision boîte blanche permet de modéliser les éléments de mise en œuvre des méthodes fournies par le composant.

L'introduction du concept de *partie* (*Part*) dans UML 2.0 rend possible la description de la partie interne d'une classe. Une partie est donc une propriété de la classe, c'est-à-dire elle a le même cycle de vie que l'objet de la classe à laquelle elle appartient. Les parties sont un ensemble d'instances d'autres classes. Comme le composant en UML 2.0 étend la notion de classe, sa structure interne est elle aussi décrite à l'aide de parties qui peuvent être respectivement des instances de classe ou des instances de composant. Si ces parties sont des instances de composant, il est possible de définir l'assemblage entre les ports de ces instances par l'intermédiaire de connecteurs (voir paragraphe précédent).

Une partie représente une ou plusieurs instances d'une classe ou d'un composant grâce à des contraintes de multiplicité. Sans contrainte de multiplicité, l'objet est construit dès que la partie est construite. La multiplicité permet d'indiquer le nombre minimum et maximum d'instances au sein de la part mais aussi le nombre d'instances créées automatiquement lors de la création de la classe ou du composant contenant le part.

UML 2.0 permet donc, comme dans la plupart des langages de description d'architecture, de définir son architecture de manière hiérarchique. En effet, la notion de part permet de définir la réalisation d'un type de composant à partir d'un assemblage de composants.

Un modèle partiellement défini

Dans un souci de consensus global, UML ne définit pas précisément la sémantique de certaines constructions. Sans lister l'ensemble de ces imprécisions, prenons l'exemple de la construction qui permet d'attacher plusieurs connecteurs à un même port ou une même interface. La norme ne précise pas la signification de cette construction au niveau du comportement. Pour une interface ou un port requis connecté à plusieurs autres ports, l'invocation de service est-elle envoyée à tous les ports connectés (*multicast*) ? à un seul port choisi de façon aléatoire ? Pour la norme UML 2.0, cette construction, si elle est autorisée, est un point de variation sémantique. De même, la définition de la compatibilité entre ports ou interfaces connectés est utilisée sans être correctement définie. UML 2.0 définit « *What makes interfaces compatible is a semantic variation point. At a minimum, the provided interfaces must support a superset of the operations and signals specified in the required interfaces.* »⁸. Ce point de variation sémantique inhibe toute possibilité d'analyse de l'assemblage en travaillant uniquement à partir de la norme UML 2.0. Enfin et surtout, la réalisation d'un composant, c'est-à-dire la description de la façon dont un composant est conçu pour pouvoir mettre en œuvre les services qu'il expose, est très peu précise. Un composant peut ainsi contenir un assemblage d'autres composants ou il peut être construit à partir d'objets. La différence entre types de composant ou instances dans les diagrammes représentant le contenu d'un composant n'est pas très claire. Il est possible de mettre au sein d'un composant un diagramme représentant des types de composant ou un diagramme d'instances de composant interconnectées.

Ces imprécisions de la norme UML 2.0 en font un langage semi-formel qui peut difficilement être utilisé tel quel pour décrire l'architecture d'une application. Il est alors nécessaire de profiter des mécanismes d'extension d'UML pour préciser la sémantique du langage utilisé.

3.3.2.2 Le déploiement

Comme UML 1.5, la version 2.0 prend en compte le déploiement de l'application sur une infrastructure physique. Le diagramme de déploiement montre ainsi la disposition physique des matériels qui composent le système

⁸page 180 de la norme

et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de *nœuds*. Les nœuds sont connectés entre eux, à l'aide d'un *support de communication*. La nature de ces supports et leurs caractéristiques peuvent être précisées. Les diagrammes de déploiement définissent des instances de nœuds (un matériel précis), ou des classes de nœuds. UML fournit à l'aide de ce diagramme, les constructions de base qui permettent de spécifier le déploiement d'une application, c'est-à-dire la répartition des entités logicielles de l'application sur une infrastructure physique. Cependant, là aussi, UML 2.0 n'associe ni processus, ni information minimale que l'on doit trouver pour caractériser un nœud.

3.3.2.3 Définition de la dynamique

UML 2.0 propose différentes approches qui permettent de caractériser la dynamique de l'application. Entre autres, le diagramme de séquences permet de représenter la création et la destruction d'instances d'objets au cours du cycle de vie de l'application. Ces constructions permettent de modéliser la reconfiguration d'une architecture en autorisant la création ou la destruction d'instances de composant ou de connecteur. Cependant, aucune opération spécifique n'existe pour cet aspect lié à la reconfiguration. La gestion du redéploiement ou la migration d'un composant au sein d'un autre composite ne sont pas des opérations simples à modéliser sur un diagramme de séquences.

La notion de parties dont on fixe la multiplicité au niveau du diagramme d'architecture est intéressante au niveau de la description de la dynamique. En effet, cela permet sur un diagramme d'architecture de fixer un cadre à l'évolution du contenu d'un composant en termes d'instances contenues. La multiplicité est donc importante à préciser d'abord dans un but descriptif, mais également pour avoir une analyse réaliste du modèle lorsque le modèle UML 2.0 sert de base à la vérification.

3.3.2.4 Spécification du comportement

Différents diagrammes permettent de modéliser le comportement de l'application. Le diagramme de séquences est une représentation intuitive pour définir les interactions entre deux entités d'une application. Il permet à l'architecte de créer au fur et à mesure sa solution. Les diagrammes de séquences servent également à la problématique de test. Les traces d'exécution d'un test peuvent en effet être représentées sous cette forme et servir de comparaison avec les diagrammes de séquences réalisés lors des phases d'analyse et de conception. Le diagramme de séquences est très riche en termes de nombre d'opérateurs. On trouve ainsi la possibilité de modéliser l'envoi ou la réception de message asynchrone ou synchrone. UML 2.0 introduit les notions de fragments combinés (appelés aussi *combined fragment* ou *inline frame*). Un fragment combiné représente des articulations d'interactions. Il est défini par un opérateur et des opérandes. L'opérateur conditionne la signification du fragment combiné. Il existe dix opérateurs définis dans la notation UML 2.0 : L'alternative (*alt*), l'option (*opt*), l'interruption (*break*), le parallélisme (*parallel*), la séquence (*weak or strict sequencing*), l'interdiction (*neg*), la section critique (*Critical*), la séquence optionnelle (*Ignore*), l'assertion (*assert*) et la boucle (*loop*).

L'ensemble de ces constructions permet d'avoir un bon pouvoir d'expression au niveau du langage de description des interactions entre deux composants.

Dans certains processus, les diagrammes de séquences sont remplacés par des MSCs (*Messages Sequence Chart*) [ITU, 1999b, Cobben *et al.*, 1998]. Avec un pouvoir d'expression proches, les MSCs offre l'avantage de disposer d'outils de validation plus complets et plus éprouvés que ceux que l'on trouve autour des diagrammes de séquences.

3.3.2.5 Exemple

Nous illustrons la description structurelle de l'architecture du serveur Web présentée pour les autres approches. On retrouve sur ce schéma dans le formalisme défini par UML 2.0 le fait que le serveur Web est composé de cinq

composants C , $R1$, $R2$, $R3$ et FS .

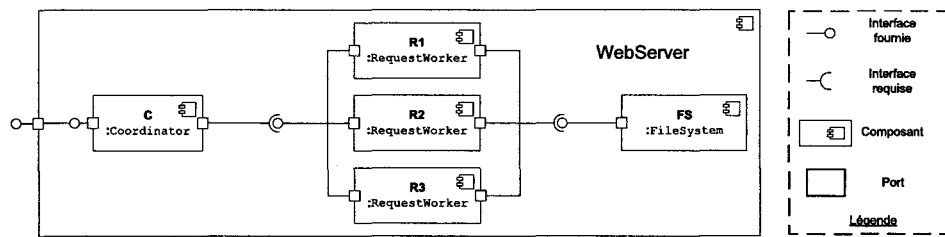


FIG. 3.10 – Architecture d'un serveur Web décrit à l'aide d'UML 2.0

3.3.2.6 Évaluation d'UML 2.0

À la lecture de cette succincte présentation du diagramme d'architecture dans UML 2.0, on découvre une certaine convergence entre les ADLs et le modèle de composant d'UML. Ainsi, si la plupart des constructions de base pour la description d'une architecture logicielle existent maintenant dans UML 2.0, les relations entre les concepts restent encore mal définies. Le modèle UML 2.0 permet la construction de modèle d'architecture dont on ne connaît pas précisément la signification de certaines constructions, ou pire, un modèle cohérent par rapport au méta-modèle mais incohérent du point de vue du système réel. Par exemple, le comportement d'un composant peut être modélisé par une machine à états, un port peut se voir attacher un comportement, lorsqu'un composant veut décrire les interactions qui doivent se produire sur le port. Enfin, un composite peut se voir attacher une collaboration. Ces trois informations peuvent dans une certaine mesure être redondantes. Rien n'est pour le moment spécifié pour assurer la cohérence de ces informations. Il est donc pour le moment possible de définir des informations contradictoires au sein de ces différents diagrammes sans qu'aucune contrainte au niveau du méta-modèle ne permette de les détecter.

L'énoncé de ces contradictions justifie encore la nécessité de redéfinir ou préciser sa propre sémantique au travers de *profils* UML ou de langages spécifiques à un domaine (*Domain Specific Language DSL*)[van Deursen *et al.*, 2000][Consel and Marlet, 1998]. Dans ce sens, les différents travaux autour des ADLs peuvent être vus comme la création de DSLs centrés sur la description d'architectures logicielles. Chacun de ces DSLs pourrait être traduit vers un profil UML. Le choix d'un profil UML ou d'un nouveau langage est à notre avis plus idéologique que scientifique.

3.3.3 Bilan sur les normes associées à la description d'une architecture logicielle

Les travaux autour d'UML et de AADL montrent l'intérêt porté par les industriels à l'émergence de standards pour la description d'architecture logicielle. Issus d'organismes de normalisation différents, UML 2.0 et AADL ne sont pas des normes concurrentes comme l'ont pu l'être H323 ou SIP dans le domaine de la téléphonie sur IP. En effet, AADL vise un domaine d'activité précis : les transports, où les contraintes liées au temps réel sont importantes. Au contraire, UML 2.0, qui vise le domaine plus général de la modélisation de tout type de logiciel, ajoute la prise en compte des composants et de la description de l'assemblage comme une information importante à associer à la modélisation d'une application. La preuve de cette complémentarité réside dans le profil UML proposé pour modéliser une application dans le langage AADL au travers des constructions UML.

3.4 Bilan sur les approches à base de composants

Cette étude des différents ADLs, modèles académiques de construction d'applications à base de composants et modèles de composant abstrait issus des organismes de normalisation, permet de constater la grande diversité dans les caractéristiques et les objectifs de ces approches. En effet, même si la partie structurelle du composant semble maintenant relativement stabilisée avec l'utilisation de la notion de port, d'interface, d'opération ou de méthode, la sémantique du composant varie encore beaucoup selon les modèles. Ainsi, un composant peut être assimilé à un processus, une zone de mémoire partagée, ou même un élément matériel. De ces différences d'objectifs et de caractéristiques découlent de nombreux critères d'évaluation possibles d'un modèle de construction d'une application à base de composants. Nous en retenons cinq :

- Une description de la dynamique du système. Une architecture logicielle étant amenée à évoluer tout au long de son cycle de vie, il est primordial de prendre en compte sa dynamique. Deux approches sont possibles pour décrire cette dynamique : la première spécifie explicitement un schéma de création et de destruction d'instances. Cette approche a pour avantage de permettre la simulation de l'application dans tous les états de son cycle de vie. La seconde, plus générale, permet de spécifier un ensemble d'invariants pour le système. Cet ensemble permet de limiter les possibilités d'évolution du système.
- Une bonne abstraction par rapport aux plates-formes d'exécution. En effet, une description d'architecture sert de pivot aux différents acteurs d'un projet informatique. Le langage doit donc être suffisamment abstrait pour ne pas être noyé dans des considérations techniques. En outre, cette description étant généralement définie avant même le choix de la plate-forme cible, il est alors intéressant de pouvoir utiliser une partie de cette description pour générer du code vers la plate-forme cible.
- Une décomposition du système à l'aide de plusieurs dimensions. En effet, la notion de composant permet de structurer le système de manière fonctionnelle selon les services fournis et requis. D'autres niveaux de structuration doivent permettre de conserver la modularité de l'application.
- La description de l'assemblage et des interfaces des composants doit être la plus précise possible. Dans ce sens, un langage de description d'architecture idéal doit fournir suffisamment d'informations au niveau de l'interface du composant pour permettre une réutilisation sûre des composants logiciels.
- Enfin, l'architecte doit pouvoir construire son architecture de manière incrémentale en intégrant au fur et à mesure les préoccupations de son application.

Cette section dresse un bilan récapitulatif des différentes caractéristiques des différents ADLs et modèles de composant abordés dans ce chapitre en se focalisant sur les trois caractéristiques précitées.

3.4.1 Anticipation des changements : prise en compte de la dynamique du système

Globalement, la dynamique de l'architecture n'est prise en compte que par peu d'approches. Par exemple, Wright ne s'y intéresse pas. D'autres modèles de composant comme SOFA ou Fractal fournissent des capacités de reconfiguration dynamique assez avancées. Cependant, leur description de l'architecture propose une vision du système à un instant t . Ainsi, même si leur modèle et leur mise en œuvre proposent des capacités de reconfiguration au cours de l'exécution du système, ces évolutions ne sont ni spécifiées ni encadrées. AADL propose par l'intermédiaire des modes de prévoir l'ensemble des configurations possibles pour le système. Ce mécanisme peut se révéler rapidement inefficace si l'architecture du système peut évoluer énormément. ArchJava est un modèle quelque peu à part. N'étant pas au départ un langage de spécification, mais une extension de langage de programmation pour la prise en compte des concepts architecturaux au niveau de l'implantation, la dynamique du système est présente dans un programme ArchJava. Cependant, cette dynamique étant noyée dans le reste du programme, il n'y a pas pour le moment d'outil permettant l'analyse du système. Darwin au contraire propose une description de la dynamique du système. Dans un objectif d'analyse, Darwin propose d'explicitement les mécanismes d'instanciation des composants du système. UML 2.0 est sur ce point celui qui offre le plus de possibilités de description même si aucun outil d'analyse ne vient pour le moment garantir la cohérence du modèle. En effet, UML 2.0 propose par l'intermédiaire de son diagramme de séquences de décrire explicitement la création d'instances de composant. Il permet en outre de décrire un ensemble d'invariants du système au niveau de la description de l'assemblage. En

effet, un composite est composé de *parties* qui peuvent être soit un ensemble d’instances de composant connectées au travers de leur port, soit un ensemble de types de composant liés par des liens d’assemblage. Les types de composant et les ports possèdent alors une cardinalité qui précise le nombre d’instances de composant qui peuvent appartenir à l’assemblage de composants. Ce deuxième modèle est très intéressant car il est le seul à donner un cadre à la dynamique du système en définissant les types de composant pouvant appartenir à un composite, leurs relations et leur cardinalité. Malheureusement, en UML 2.0, ce diagramme est pour le moment uniquement recommandé. La norme autorise la définition d’un assemblage de composants dont le type d’assemblage n’est pas défini au préalable.

	Reconfiguration dynamique	Cadre pour la dynamique	Description abstraite de la dynamique
Wright	Non	Aucun	Aucune
Fractal/SOFA	Oui	Aucun	Aucune
AADL	Oui	Oui	Mécanisme de mode
ArchJava	Non	Aucun	Au niveau du code de l’application
Darwin	Non	Politiques d’instanciation	Description explicite des mécanismes d’instanciation du système
UML 2.0	Non	Oui notion de type d’assemblage	Oui description des mécanismes d’instanciation via le diagramme de séquences

TAB. 3.1 – Prise en compte de la dynamique

Le tableau 3.2 résume comment il est possible à l’aide des approches étudiées dans ce chapitre de spécifier la dynamique de l’architecture.

3.4.2 Abstraction : indépendance vis-à-vis de la plate-forme d’exécution

La plupart des ADLs proposent une abstraction vis-à-vis de la plate-forme d’exécution. Cependant, ils ne proposent pas tous des mécanismes permettant de profiter de la description d’architecture du système dans la production de l’application. Ainsi Wright dont l’objectif était la production d’une spécification du système en vue de son analyse ne propose pas de mécanismes de projection de code. En outre, son modèle est assez éloigné des langages de programmation. La traçabilité des éléments d’architecture entre la spécification et la projection vers une plate-forme d’exécution est peu intuitive. Darwin et SOFA proposent une unique plate-forme d’exécution pour leur modèle de composant. Cependant leur modèle de composant est suffisamment abstrait pour imaginer une projection vers d’autres plates-formes à composants. Là encore, ArchJava est un cas à part. Comme il ne sépare pas la description de l’architecture de l’implantation des services fournis et requis par ses composants, la génération de code vers une autre plate-forme à composant semble difficile. Néanmoins, comme son compilateur garantit que le programmeur n’enfreint pas les contraintes architecturales, ArchJava est au contraire une plate-forme cible tout à fait intéressante. Fractal est un bon modèle d’abstraction vis-à-vis de la plate-forme d’exécution. Défini lui-même à partir d’un modèle de composant abstrait, plusieurs implantations de Fractal existent et permettent de choisir sa plate-forme cible en fonction du contexte du projet. Ainsi, une implantation en C (think [Fassino *et al.*, 2002]) permet de construire des applications à composants pour de l’embarqué alors que plusieurs implantations en Java (AOKell [Seinturier, 2005], Julia [Bruneton *et al.*, 2004a]) offrent davantage de fonctionnalités et permettent la construction de système d’information d’entreprise. AADL et UML 2.0 sont des modèles abstraits, ils ne possèdent pas de plate-forme propre, mais proposent des mécanismes de génération de code vers des plates-formes à composants existantes.

Le tableau 3.2 résume, pour chacune des approches étudiées dans ce chapitre, l’existence d’un modèle abstrait de description d’architecture logicielle et l’existence d’une plate-forme d’exécution ou de mécanismes de génération de code.

	Langage de description d'architecture	Génération de code / plate-forme d'exécution
Wright	abstrait	non/non
Darwin	oui	non/oui (Regis)
SOFA	oui	non/oui (DCUP)
ArchJava	non	oui (java)/non
Fractal	oui	non/ oui (Julia, Think, etc.)
AADL	oui	oui (ADA, C) / non
UML 2.0	oui	oui / non

TAB. 3.2 – Indépendance vis-à-vis de la plate-forme d'exécution

3.4.3 Modularité : structuration multi-dimensionnelle de l'application

Si les notions de composant, connecteur et configuration offrent un premier niveau de modularité dans la description d'une application, on constate que ces mécanismes ne permettent pas à eux seuls l'obtention d'un bon découpage modulaire. Certaines préoccupations transverses à l'application engendrent de très nombreuses dépendances entre composants. C'est le cas dans une architecture logicielle d'un composant en charge de préoccupations dites techniques comme la trace ou la persistance qui sont liées à de nombreux composants métier.

Si ces dépendances multiples sont, d'un point de vue conceptuel, inévitables (les composants ayant de réelles dépendances entre eux), il faut pouvoir définir un nouvel élément de structuration afin de ne pas dupliquer une information dans de nombreuses parties de la description d'architecture logicielle. Cette caractéristique, qui est essentielle pour permettre la spécification d'architecture logicielle fournissant un bon découpage modulaire, est encore très peu prise en compte dans les approches de l'architecture logicielle présentées dans ce chapitre. Fractal propose au travers des interfaces de contrôle un moyen de regrouper certaines préoccupations techniques du composant au sein d'une même entité dupliquée sur l'ensemble des composants concernés. Cependant, ces interfaces de contrôle étant pour le moment des interfaces fournies, elles ne permettent pas d'injecter une dépendance vers un composant technique.

3.4.4 Réutilisation : définition étendue des interfaces de composant

Comme une partie des ADLs travaille sur l'analyse d'une description d'architecture, beaucoup de langages ont fourni une définition étendue des interfaces de composant afin de disposer d'un maximum d'information sur le composant lors de l'analyse de l'assemblage. Ainsi, Wright, Darwin et SOFA proposent, au niveau des interfaces des composants et des rôles des connecteurs, une description des messages entrants et sortants de ces composants ou de ces connecteurs. Fractal ne dispose au niveau de son modèle qu'un contrat d'interface de premier niveau à l'aide d'une description de services offerts et requis par le composant. Une extension, ConFract, définit un langage de définition de contraintes. D'autres travaux sont aussi en cours pour permettre une définition des messages entrants et sortants d'un composant dans la lignée de SOFA ou Darwin. AADL pour sa part ne propose pas d'abstraction du comportement du composant au niveau du modèle. Il dispose néanmoins de mécanismes d'extension, les annexes et les propriétés qui permettent d'ajouter de l'information pour caractériser le composant. Grâce à ce mécanisme d'annotation, les différents niveaux de contrats peuvent être décrits en AADL moyennant la définition de langages dédiés. Cependant pour le moment ces mécanismes sont principalement utilisés pour des propriétés de qualité de service. Ainsi, certaines propriétés temporelles doivent être décrites au niveau des interfaces de composant. Ces propriétés permettent, après analyse de l'assemblage, de statuer sur le niveau de qualité de service de l'application. UML 2.0 propose, quant à lui, de nombreux mécanismes au niveau du langage pour enrichir la définition de l'interface d'un composant. Si OCL est maintenant convenablement adopté et intégré dans le modèle de composant, la définition d'information relative au comportement à tous les niveaux du composant (composant, port, interface) et le manque de cohérence entre ces informations gêne pour le moment l'analyse d'un assemblage de composants UML 2.0. Finalement, le cas du langage ArchJava est de nouveau à part. Relativement peu utilisée pour l'analyse, la définition des interfaces de composant se limite à une définition syntaxique des services qu'il fournit et qu'il requiert.

Le tableau 3.3 résume le pouvoir d'expression au niveau des interfaces pour chacune des approches étudiées dans ce chapitre.

	basique	comportementale	synchronisation	qualité de service
Wright	oui	non	oui (CSP)	non
Darwin	oui	non	oui (FSP)	non
SOFA	oui	non	oui	non
Fractal	oui	oui (ConFract)	non	non
AADL	oui	non	pas encore	oui (Propriétés et annexes)
UML 2.0	oui	oui (OCL)	oui (diagramme de séquences)	non
ArchJava	oui	non	non	non

TAB. 3.3 – Pouvoir d'expression disponible au niveau des interfaces selon la taxonomie de Beugnard *et al*

3.4.5 Construction incrémentale de la description d'architecture logicielle

La construction incrémentale d'une description d'architecture logicielle vise à offrir à l'architecte la possibilité de définir son architecture en plusieurs étapes. L'intérêt d'un tel mécanisme est double. Il permet de maîtriser la complexité liée à la spécification d'une architecture en permettant de n'adresser qu'une problématique à la fois. En outre, la description de l'architecture logicielle doit être un document central compréhensible par l'ensemble des acteurs d'un projet informatique afin que chacun puisse identifier facilement son rôle au sein de ce projet. Dès lors, une construction incrémentale permet d'obtenir différents niveaux de granularité autour d'une architecture logicielle et permet au client d'avoir une vue de très haut niveau de son système et à l'expert en sécurité de voir en détail les dépendances entre les composants métiers et les composants techniques.

Cette construction incrémentale n'est proposée pour le moment par aucune des approches étudiées dans ce chapitre. L'architecte voulant spécifier son architecture en plusieurs étapes sera obligé de modifier manuellement sa spécification pour détruire des liaisons, ajouter des interfaces, du comportement, *etc.* Cette opération manuelle si elle peut très vite se révéler fastidieuse est en outre inefficace, car l'ensemble des vérifications effectuées à une étape t de la conception sont perdues et doivent être à nouveau effectuées à l'étape $t + 1$.

3.4.6 Proposition

Nous proposons au cours de cette thèse de définir un canevas de spécification d'architectures travaillant sur la maîtrise de la complexité des architectures logicielles et l'amélioration de la productivité associée à ce travail. Pour maîtriser la complexité des architectures logicielles, notre approche est guidée par les grands principes du génie logiciel énoncés dans le chapitre précédent. Notre canevas doit ainsi permettre de relever le premier, le deuxième, le troisième et le cinquième défi précédemment cités (voir Section 2.4), c'est-à-dire offrir une bonne abstraction par rapport au système spécifié, proposer différents niveaux de modularisation dans la description d'une architecture logicielle, fournir des mécanismes de composition sûrs afin de permettre la construction incrémentale d'un système et enfin prendre en compte le caractère variable et évolutif d'une architecture logicielle à base de composants. En outre, afin de diminuer les coûts et les temps de construction d'une nouvelle architecture logicielle, notre canevas doit favoriser la réutilisation des briques modulaires d'une architecture logicielle. Pour ce faire, le canevas doit pouvoir garantir que l'utilisation de ces briques de base, dans un contexte différent de celui dans lequel elles ont été conçues, ne casse pas la cohérence d'une architecture logicielle. Cet objectif représente notre quatrième défi.

Deuxième partie

Construire une architecture logicielle à base de composants

Objectifs et présentation générale de SafArchie

Un formalisme d'architecture logicielle idéal doit satisfaire quatre critères essentiels [Garlan and Shaw, 1993] :

- **être expressif** : le formalisme doit permettre de modéliser simplement, sans contorsions inutiles, les caractéristiques essentielles d'une architecture logicielle. Il donne ainsi une description claire et compréhensible de la structure d'une application,
- **être vérifiable** : certaines descriptions d'architecture logicielle (notamment dans l'avionique) sont suffisamment critiques pour que leur correction doivent être garantie par des techniques de vérification ou de preuve. Or celles-ci ne peuvent être appliquées que sur des modélisations faites dans des formalismes dont la sémantique est, d'une part, rigoureusement définie, et d'autre part possède de bonnes propriétés de composition et d'abstraction,
- **être exécutable** : le formalisme doit avoir un caractère exécutable afin que les modélisations d'architectures logicielles ne servent pas seulement de documentation, mais puissent être traitées par des outils de prototypage rapide (pour produire du code) et de génération automatique de tests,
- **être évolutif** : le formalisme doit capturer les invariants du système, mais aussi la dynamique de l'architecture.

Une des difficultés dans les approches à composants provient principalement d'un problème de vocabulaire engendré par une utilisation assez confuse du mot *composant*. En effet, la définition du concept de composant varie grandement par rapport au contexte d'utilisation et à la communauté scientifique qui l'utilise. Il est alors souvent difficile de savoir quelles sont les caractéristiques des composants dont on parle et quelle est leur nature.

La plupart des modèles d'architecture logicielle travaillent sur des instances de composant pouvant être identifiées, déployées, connectées, et administrées. Cependant, dans SafArchie, afin de définir un cadre à l'évolution d'une architecture, il nous semble intéressant de travailler également en amont sur un *modèle de type* d'architecture en identifiant les relations admissibles entre composants. Ce premier travail de spécification permet de mettre en évidence un certain nombre de contraintes que notre architecture logicielle devra vérifier. Nous mettons l'accent particulièrement sur la conformité de ce modèle par rapport à son méta-modèle et la cohérence de cette spécification. Nous présentons dans le chapitre 4 ce modèle de type d'architecture ainsi que la phase d'analyse associée.

Dans un deuxième temps, nous spécifions l'architecture logicielle en termes d'instances de composant. Nous appelons ce modèle l'*architecture logique*. Ce modèle représente l'architecture logicielle de notre application telle qu'elle sera déployée et exécutée. C'est une vision statique à un instant t qui doit vérifier l'ensemble des contraintes de notre modèle de type. Le chapitre 5 présente le méta-modèle associé au modèle logique d'architecture et le modèle de déploiement d'une architecture logicielle spécifiée avec SafArchie sur une infrastructure matérielle.

Le modèle d'architecture logique doit être modifié tout au long du cycle de vie de l'application pour rester représentatif de son architecture même après une évolution de celui-ci. Il existe deux grandes catégories pour l'évolution de l'architecture. Les *évolutions internes* à l'application (création ou destruction d'instances, création ou destruction de liaisons), et les évolutions externes à l'application qui permettent d'ajouter de l'information sur une préoccupation de l'application (déploiement, sécurité) au niveau de la spécification du schéma d'instance. SafArchie se concentre sur les évolutions internes. La prise en compte des évolutions externes est détaillée dans le chapitre 6 de ce document.

Une vue globale de SafArchie est présentée dans la figure 3.11. Ses principaux éléments sont le modèle de type d'architecture et le modèle logique d'architecture.

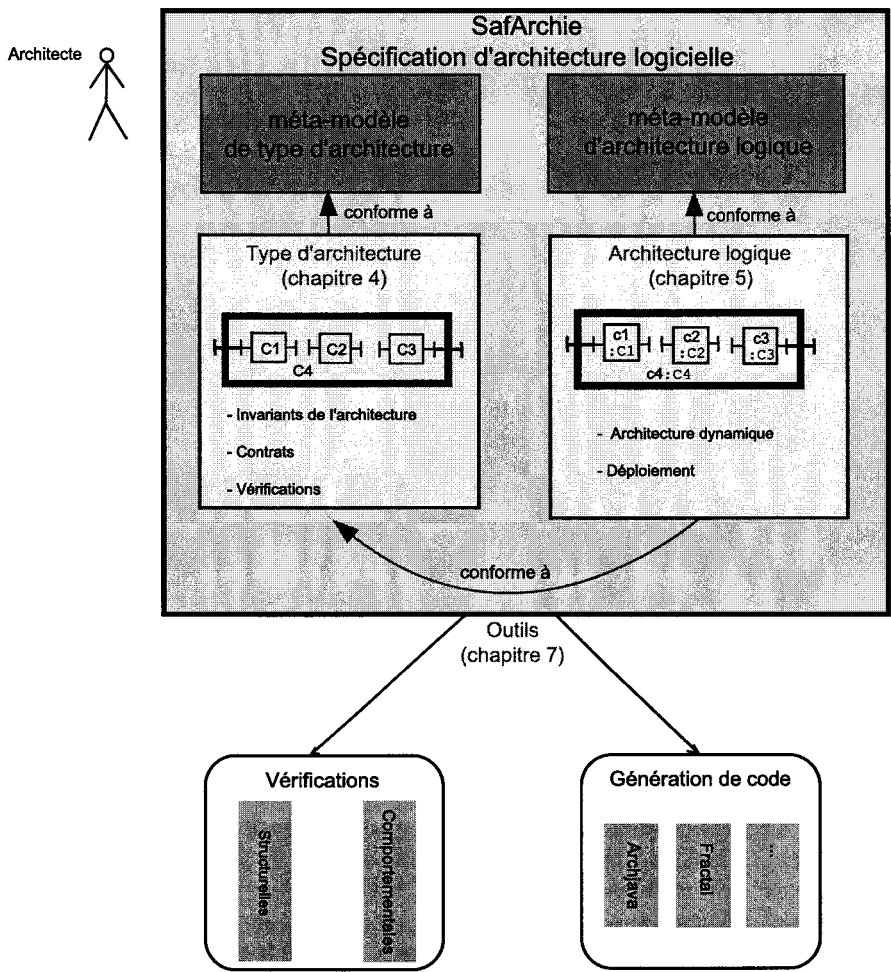


FIG. 3.11 – Présentation générale de SafArchie

Chapitre 4

Spécifier un type d'architecture logicielle cohérent

PRÉAMBULE

Dans ce chapitre, nous présentons en détail le modèle de type d'architecture de SafArchie. Nous présentons dans une première section les bases de ce modèle. Nous détaillons ensuite les extensions apportées à ce modèle structurel, en terme de contrat. Finalement, la dernière section présente les vérifications effectuées sur un type d'architecture logicielle permettant de détecter des problèmes d'incohérence au niveau d'un assemblage de composants.

Sommaire

4.1	SafArchie ADL : vers une notion de type d'architecture logicielle	72
4.1.1	Définition du modèle de type d'architecture	72
4.1.2	Le méta-modèle de type d'architecture	74
4.1.3	Un méta-modèle - différentes représentations	75
4.1.4	Exemple d'architecture : le <i>DataPhoneCenter</i>	75
4.2	Une définition de composant étendue pour un assemblage cohérent	79
4.2.1	Notion de contrat dans SafArchie	79
4.2.2	Contrats structurels d'un composant	79
4.2.3	Contrat comportemental d'un composant	82
4.2.4	Conclusion sur les langages de contrats dans SafArchie	96
4.3	Construire un type d'architecture logicielle cohérent	97
4.3.1	Analyse structurelle de l'assemblage	97
4.3.2	Analyse autour du contrat de comportement	104
4.4	Bilan autour du modèle de type	109

L'objectif du modèle de type d'architecture logicielle est de bâtir un cadre, c'est-à-dire un ensemble de contraintes que devront respecter les futures architectures logicielles, et de mener un premier niveau d'analyse de ces contraintes afin de garantir sa cohérence vis-à-vis de propriétés structurelles et comportementales. Nous présentons dans les deux premières sections de ce chapitre, le modèle de type de SafArchie, le cœur du modèle et les contrats. Le cœur du modèle est bâti à partir de cinq éléments de structure (type de composant, type de composite, type de port, opération et liaison) que nous enrichissons avec différents contrats au niveau de la définition d'un type de composant. Ces contrats présentés dans la section 4.2 précisent la spécification des types de composant. La Section 4.3 présente en détail les analyses effectuées à partir des informations définies dans un type d'architecture afin de garantir sa cohérence.

4.1 SafArchie ADL : vers une notion de type d'architecture logicielle

4.1.1 Définition du modèle de type d'architecture

Dans SafArchie, les cinq éléments qui composent une architecture logicielle sont le composant primitif, le composite, le port, l'opération et la liaison.

4.1.1.1 Le composant

Le *composant* est une entité logicielle définie par l'intermédiaire de son interface avec l'extérieur en vue de sa composition et de sa réutilisation. Il peut posséder des attributs qui caractérisent l'état de ses instances. L'interface du composant met en évidence les opérations que le composant fournit et requiert. Chaque opération est rendue accessible par l'intermédiaire d'un port. Le *port* est l'unité structurelle de composition entre composants. Il est le point d'accès au composant et est potentiellement composé de deux types d'opérations, les opérations fournies et les opérations requises. Un port regroupe une liste d'opérations possédant un lien conceptuel entre elles. Une *opération* définit les structures de données échangées dans une interaction entre deux composants.

Le mode de communication dans SafArchie est le mode synchrone : l'invocation d'une opération requise est bloquante. Ce mode consiste en l'envoi d'une requête suivi de la réception d'une réponse. À partir de ce mode de communication, nous décrivons des modes de communication plus évolués à l'aide de composants dédiés que certains appellent connecteurs [Medvidovic and Taylor, 2000]. SafArchie repose sur un modèle de communication qui permet une délivrance fiable et ordonnée de messages. Cette hypothèse nous permet de garantir que tout message émis d'un composant arrive à son destinataire. De plus, une séquence de messages émis dans un certain ordre arrivera à son destinataire dans le même ordre.

L'objectif du modèle de type est d'imposer un ensemble de contraintes que toute architecture logicielle construite selon ce modèle doit respecter.

Définitions

Une *opération* est définie par sa signature, c'est-à-dire son nom, sa polarité (requis ou fourni), la liste de ses paramètres et leur type, une liste des exceptions typées, et un type de résultat.

Un *type de port* est défini par la liste des opérations qu'il requiert et fournit.

Un *type de composant primitif* est défini par les *attributs* et les types de port qu'il doit posséder.

Un attribut d'un composant est identifié dès la spécification. Il possède un nom et un type, il diminue l'opacité des composants en permettant la connaissance partielle de l'état de celui-ci.

4.1.1.2 L'assemblage

Un assemblage de composants est un ensemble de composants connectés entre eux. La configuration d'une architecture logicielle, troisième concept fondamental des ADLs, définit précisément les composants d'une architecture et leurs interactions au travers des ports des composants. D'une manière générale, une interaction entre deux ports ne peut avoir lieu que si ces deux ports proposent et requièrent des opérations compatibles.

Au niveau du modèle de type, les *liaisons*, appelées *bindings*, matérialisent les interactions possibles entre composants. Chaque liaison doit relier deux types de port compatibles. La notion de types de port « compatibles » est présentée plus en détail dans la section 4.3.

4.1.1.3 Un modèle hiérarchique

SafArchie repose sur un modèle de composant hiérarchique. Un composant peut contenir d'autres composants. Il est alors défini comme étant un *composite*. Un composite a une double spécificité : en tant que composant, il peut lui-même avoir un rôle dans un assemblage entre composants. En tant qu'entité contenant d'autres composants, il connaît la configuration des composants dont il a la charge, c'est-à-dire leurs interactions. Les composants contenus au sein d'un composite sont appelés ses *fils* . Un composant participant à un assemblage appartient nécessairement à un composite qui est appelé le *père* de ce composant.

Au niveau du modèle de type, un type de composite définit à la fois les types de composant qu'il peut contenir, leur cardinalité et leurs liaisons.

Dans le modèle de composant de SafArchie, il existe une différence fondamentale entre les composants primitifs et les composants composites. En effet, outre le fait que les composants primitifs sont des entités opaques (représentées par des boîtes noires), alors que les composites sont facilement introspectables (représentés par des boîtes blanches), les composants primitifs sont une unité de traitement fonctionnelle alors que les composites sont une unité de structuration et de configuration architecturale. C'est pourquoi il existe deux catégories de port, les ports des composants primitifs (*Primitive Port*) et les ports des composants composites (*Delegated Port*). Ces derniers sont des ports délégués. Un composite ne possède pas de ports natifs contenant un traitement associé, il ne fait que déléguer les traitements à un composant par l'intermédiaire des ports délégués. Le port délégué fait donc toujours référence à un port d'un de ses fils à qui il transmet la réalisation de l'interaction. Cette hiérarchie entre les ports ne possède pas nécessairement un seul niveau : un port délégué dp_1 d'un composite A peut ainsi délégué le traitement à un autre port délégué dp_2 d'un composite B contenu dans A . dp_2 déléguera lui aussi le traitement à un troisième port primitif ou délégué.

Cette distinction entre port primitif et port délégué se retrouve au niveau du modèle de type d'architecture logicielle. La notion de type de port primitif (*PrimitivePortType*) est quant à elle toujours associée à un type de composant. Un type de port définit la liste des opérations fournies et requises par le type de port.

La notion de composite permet ainsi d'obtenir une représentation abstraite d'une architecture logicielle de granularité plus importante. Contrairement à certains modèles de composant comme Fractal [Bruneton *et al.*, 2004b], dans SafArchie le composite est une unité d'encapsulation fermée. Un composant ne peut appartenir qu'à un seul composite. Il n'y a pas de partage de composants entre plusieurs composites, ce qui garantit la propriété d'encapsulation.

Définition

Un *type de composite* représente l'ensemble des types de composant qu'il peut contenir, les types de port délégué qu'il exporte et les liaisons entre les types de port contenus en son sein.

Outre les contraintes de typage, nous ajoutons la possibilité de préciser la cardinalité de deux relations entre éléments du modèle de type. Une cardinalité au niveau du type de port permet de préciser le nombre d'instances de ce type de port que pourra posséder un composant. Une cardinalité au niveau du type de composant inclus dans un composite permet de préciser le nombre d'instances de ce type de composant qui pourront être présentes au sein d'un composite. Pour chaque liaison entre deux types de port appartenant à deux types de composant, nous avons donc deux couples : nombre d'instances de composant, nombre d'instances de port que l'on peut comparer.

Définition

Un *type d'architecture logicielle* définit un gabarit⁹ de haut niveau pour une architecture à base de composants. Il est défini à partir de l'ensemble des types de port délégué, types de composant, types de composite et liaisons qu'il pourra contenir.

4.1.2 Le méta-modèle de type d'architecture

La partie structurelle du méta-modèle de type d'architecture de SafArchie est décrite en UML (*Unified Modeling Language*) dans la figure 4.1. Nous y retrouvons dans un formalisme standard les concepts de notre modèle d'architecture et leurs relations décrits dans la section précédente. Le choix d'UML face à MOF (*Meta Object Facilities*), qui est davantage dédié à la définition de méta-modèles, est parfaitement arbitraire. Nous utilisons uniquement les concepts de classe, les associations, les attributs et les types que l'on retrouve au sein d'UML et du MOF.

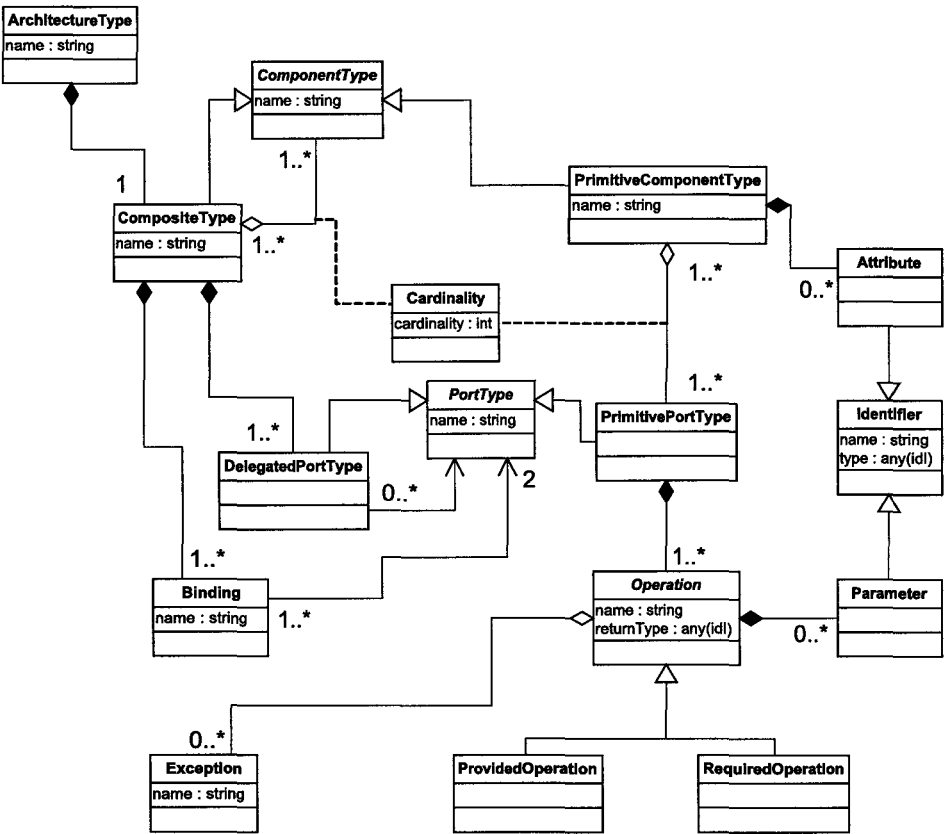


FIG. 4.1 – Méta-modèle de type d'architecture

Un type d'architecture logicielle est composé d'un type de composite. Celui-ci est composé de type(s) de composant primitif (boîtes opaques indivisibles) et/ou de type(s) de composant composite. Les types de composant primitif sont identifiables par leur nom qui doit être unique. Ils sont composés de types de port primitif (*PrimitivePortType*) qui modélisent les types d'interactions d'un composant avec son environnement. Les types de composant primitif possèdent aussi des attributs (*Attribute*). Ces attributs diminuent l'opacité des composants primitifs en permettant la connaissance partielle de l'état de celui-ci. Les types de composite sont identifiés par leur

⁹Par gabarit nous entendons une forme de référence à partir de laquelle sont créés des objets qui présentent des caractéristiques communes.

nom. Ils sont constitués de types de port délégué (*DelegatedPortType*), de types de composant (*ComponentType*) et de liaisons (*Bindings*). Les types de port délégué font référence à un type de port qui réalise le traitement associé à l'interaction.

Un type de port est composé d'opérations. Une opération correspond à une description structurelle d'une partie de l'interaction. La spécification d'une opération correspond à la signature des *méthodes*. La principale différence consiste à identifier et déclarer les opérations requises au même plan que les opérations fournies. Ainsi, les dépendances d'un composant sont exprimées sur un même plan que les fonctionnalités fournies par ce composant. Ce principe est un des fondements de l'approche par composants et sa principale différence vis-à-vis de l'approche par objets.

La déclaration des possibilités d'assemblage se fait au sein des types de composite à l'aide de liaisons. Une liaison identifie deux types de port qui sont alors déclarés liés. Cette liaison identifie une possible connexion au niveau de l'architecture logicielle. Une liaison fournit un unique mode de fonctionnement, l'appel de procédure.

Un type d'architecture est donc un *graphe acyclique* de types de composite, types de composant et types de port. La structure de graphe est plus adaptée qu'une structure arborescente, car un type de port peut être réutilisé au sein de plusieurs types de composant. De même, un type de composant peut être le fils de plusieurs types de composite.

Ce méta-modèle de type d'architecture présente les éléments de base d'un type d'architecture logicielle. Un type d'architecture sera déclaré conforme à ce méta-modèle si il respecte l'ensemble des contraintes liées à ce méta-modèle. Cette vérification structurelle caractérise un type d'architecture logicielle. Des notions de contrat sont ajoutées à ce modèle de type pour étendre sa sémantique et sa précision. Elles sont présentées à la section 4.2.

4.1.3 Un méta-modèle - différentes représentations

Le méta-modèle permet de définir les concepts manipulés au sein d'une description de types d'architecture et leurs relations. SafArchie outille ce méta-modèle à l'aide de différents formalismes. Offrir la possibilité dans SafArchie de manipuler différents formalismes de description est motivé par le souci de proposer à l'architecte de travailler avec un formalisme qui lui est adapté. SafArchie fournit ainsi :

- un langage de description textuelle de types d'architectures,
- une représentation arborescente en XML d'un type d'architecture,
- une représentation graphique.

SafArchie propose le même pouvoir d'expression entre le formalisme textuel et le formalisme XML. Par contre, la représentation graphique, qui a pour objectif de faciliter la compréhension d'une architecture, n'offre qu'un sous-ensemble de ces notions. La grammaire du langage textuel est définie à l'aide d'une EBNF fournie en Annexe A. Le schéma XML décrivant la grammaire du formalisme XML est quant à lui fourni en Annexe B.

La charte graphique pour la représentation des éléments d'un type d'architecture est présentée dans la figure 4.2. Cette charte graphique sera utilisée tout au long de ce document.

4.1.4 Exemple d'architecture : le *DataPhoneCenter*

Pour illustrer le principe du modèle de type, nous présentons l'architecture logicielle d'un système, appelé le « *DataPhoneCenter* », permettant l'accès vocal à des données (Internet, Intranet, mail, système de fichiers). L'utilisateur connecté au serveur peut ainsi surfer en « écoutant » des contenus Internet (messagerie, pages Web) habituellement proposés sur un écran. La synthèse vocale remplace l'écran et la reconnaissance vocale remplace la souris. Nous nous servons de cet exemple tout au long du mémoire pour illustrer nos propos.

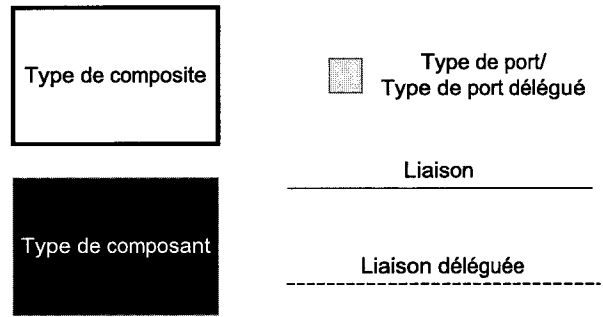


FIG. 4.2 – Charte graphique du modèle de type de SafArchie

4.1.4.1 Contexte d'un tel projet

La genèse d'un tel projet trouve son explication dans les progrès récents que les technologies vocales ont atteints ces dernières années. En effet, après de longues années de balbutiements, ces technologies semblent être devenues assez fiables pour exploitation commerciale. De tels systèmes font cohabiter trois familles de technologies : le langage VoiceXML, la reconnaissance vocale et la synthèse vocale.

Le langage VoiceXML, dérivé de XML, est un langage permettant de construire des scénarios vocaux qui seront joués à l'utilisateur. La reconnaissance vocale permet à l'utilisateur d'interagir avec le serveur grâce à l'utilisation de la parole. Enfin, la technologie TTS (*Text To Speech*) transforme un texte écrit en parole synthétique. La synthèse vocale permet de développer des services ou des applications dotés de la parole tels que : la lecture d'e-mails par téléphone, la messagerie unifiée, les informations en temps réel, les jeux interactifs, les systèmes d'aide à la navigation, etc.

4.1.4.2 L'architecture

L'objectif de spécifier l'architecture d'un tel système est de permettre la création d'une application modulaire facilement déclinable en différents produits à l'aide de configurations différentes. Le type d'architecture explicite alors les dépendances entre briques de bases. En outre, il sert de plan de référence pour comprendre le fonctionnement d'une telle application.

La phase d'analyse des besoins du système *DataPhoneCenter* permet de mettre en évidence ses éléments caractéristiques : le gestionnaire de profil et de facturation, le serveur de ressources VoiceXML, et les différents types de composant applicatif. Le type d'architecture est représenté dans la figure 4.3. Ce type d'architecture est constitué de onze types de composant primitifs et d'un type de composite. Les types de composant constituant le cœur de l'application sont regroupés au sein du type de composite *DataPhoneCenter*. Les six autres types de composant présentent l'intégration d'une telle application dans un environnement existant. Ainsi, le *DataPhoneCenter* peut interagir avec des clients (*Client*), une base de données LDAP (*LDAPDataBase*) pour gérer les ressources liées à l'authentification des clients, des banques pour gérer la facturation des services (*Bank*), des serveurs de courrier électronique (*EmailReader*), des serveurs Web (*WebReader*) et des systèmes de fichiers (*FSAccess*). Pour interagir avec ces différents éléments, le type de composite *DataPhoneCenter* est composé de cinq types de composant. Le gestionnaire de profil (*AA*) reçoit la requête d'initiation de l'utilisateur, à partir de cette requête et des informations liées à l'utilisateur, il commande le serveur de ressources VoiceXML (*MediaCenter*) afin que celui-ci initie la connexion audio avec l'utilisateur et engendre la première requête au composant applicatif demandé. Les composants applicatifs (*EmailReader*, *WebReader* et *FSAccess*) permettent de se connecter à un service de messagerie, à un site d'informations, à un annuaire, etc.

Les liaisons matérialisent les interactions possibles entre deux types de composant. Ces interactions s'effectuent par l'intermédiaire d'appels de procédure effectués au travers des opérations requises d'un port sur les opérations

fournies du port auquel il est lié. La description des opérations requises et fournies pour chaque port de l'architecture du *DataPhoneCenter* est présentée dans la figure 4.4. Ainsi, le type de port *p1* contient une opération fournie *connect* qui lui permet d'initier une connexion et 5 opérations requises par l'intermédiaire desquelles il reçoit et fournit des informations pour gérer la connexion : *getconnectioncapabilities* pour fournir les caractéristiques de la connexion du client, *getpreferredcodec* pour fournir une liste ordonnée de codecs gérés par le client, *disconnect* pour signifier la fin de la connexion, *setCodec* et *setconnectionport* pour fournir respectivement le codec retenu par l'application *DataPhoneCenter* et la référence du composant qui va délivrer le contenu audio.

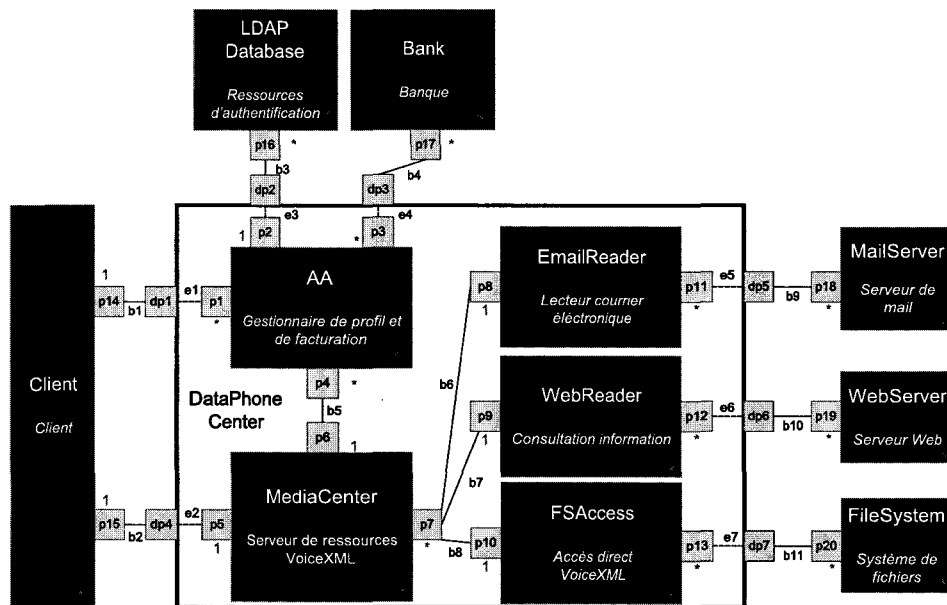


FIG. 4.3 – Type d'architecture pour l'exemple DataPhoneCenter

```

portType p1 {
    provide boolean connect(PersonalInformation pi);
    require string getConnectionCapabilities();
    require string getpreferredcodec();
    require void disconnect();
    require void setcodec(CodecDescription pr);
    require void setconnectionport(PortRef pr);
}

portType p2 {
    require DataSet select(string query);
}

portType p3 {
    require long withdraw(long AccountId, double Amount);
    require long credit(long AccountId, double Amount);
}

portType p4 {
    require void createapp(ApplicationDescription a);
    require void reserve();
    require void free();
    provide void destroy();
}

portType p5 {
    provide void receivedata(AudioStream data);
    require void receivedata(AudioStream data);
}

portType p6 {
    provide void createapp(ApplicationDescription a);
    provide void reserve();
    provide void free();
    require void destroy();
}

portType p7 {
    require string[] getvoicexml(URL url);
    require void prepare();
    require void cancel();
}

portType p8,p9,p10 {
    provide string[] getvoicexmlscene(URL url);
    provide string[] putvoicexmlscene(URL url);
}

portType p11 {
    require boolean authentication(string user, string pass);
    require string[] list();
    require Message getmail(long mailNumber);
    require Message delmail(long mailNumber);
    require void disconnect();
}

portType p12 {
    require string[] get(URL url);
    require string[] put(URL url);
}

portType p13 {
    require long open(string file);
    require string[] read(long fileRef);
    require void close(long fileRef);
}

...

portType p17 {
    provide long withdraw(long AccountId, double Amount);
    provide long deposit(long AccountId, double Amount);
}

...

portType p20 {
    provide long open(string file);
    provide string[] read(long fileRef);
    provide void write(long fileRef, string[] data);
    provide void close(long fileRef);
}

```

FIG. 4.4 – Définition des ports du type d'architecture *DataPhoneCenter*

4.2 Une définition de composant étendue pour un assemblage cohérent

Le premier objectif du modèle SafArchie est de garantir la *cohérence* des futurs assemblages et de détecter des incompatibilités entre composants assemblés grâce à la précision du modèle. Comme notre modèle de type architectural sert de gabarit pour la construction d'architectures logicielles, il est primordial d'assurer un premier niveau de cohérence au niveau du type d'architecture avant d'instancier ce modèle de type. Ainsi, le modèle de type d'architecture devient le garant de la cohérence de l'application.

Cependant, les éléments de base d'un type d'architecture logicielle présentés dans la section 4.1 ne fournissent pas beaucoup d'information sur les types de composant d'une architecture. La notion de compatibilité entre types de port liés se limitent alors à l'étude de propriétés syntaxiques autour des noms des opérations, des types des paramètres, des exceptions et des retours. Cette section présente les informations ajoutées à la définition des types de composant afin de garantir la cohérence d'un type d'architecture face à certaines propriétés structurelles et comportementales. Ces informations sont regroupées sous le vocable « contrat ».

Nous précisons dans la première partie de cette section la notion de contrat dans SafArchie. Puis nous présentons ensuite les contrats permettant d'enrichir la partie structurelle d'un type d'architecture logicielle et les contrats permettant de définir le comportement attendu par les éléments d'un type d'architecture logicielle.

4.2.1 Notion de contrat dans SafArchie

Définition

Un **contrat** est un moyen d'établir d'une façon explicite et non ambiguë les obligations et les droits d'un composant vis-vis de son environnement. Les obligations définissent une ou des contraintes garantissant le résultat du service fourni si et seulement si certains droits sont remplis.

Tous les composants qui veulent utiliser les services offerts sont appelés clients. Le composant qui offre les services est appelé le fournisseur.

Dans SafArchie, les contrats servent à préciser la définition d'un type de composant d'un point de vue syntaxique et sémantique.

Les contrats interviennent comme des garants de la bonne utilisation des composants assemblés. Un composant interagissant avec un autre composant accepte, ou non, le contrat d'utilisation de ce dernier et réciproquement. Les droits d'une partie correspondent directement aux obligations de l'autre partie. Le fournisseur connaît certaines conditions sous lesquelles ses services peuvent être utilisés. Si le client ne respecte pas ses obligations, le fournisseur n'est pas responsable des conséquences. Le fournisseur peut présumer que les conditions sont toujours remplies. Si le client prend soin de remplir les conditions exigées, les obligations du composant, à savoir les contraintes garantissant le résultat du service fourni, seront respectées. Si l'une des deux parties ne respecte pas les conditions spécifiées par le contrat, le contrat est rompu.

4.2.2 Contrats structurels d'un composant

Pour préciser les éléments de structure de la définition du composant, nous ajoutons deux contrats appelés respectivement *contrat de port* et *contrat d'assertions*. Le contrat de port précise les conditions d'utilisation d'un type de port. Le contrat d'assertions précise, quant à lui, les types des paramètres des opérations.

4.2.2.1 Contrat de port

Le **contrat de port** permet de préciser les dépendances entre les opérations d'un type de port d'un composant. Il précise les opérations devant nécessairement être présentes pour le bon déroulement d'une interaction. C'est une expression logique évaluée au moment de la composition. Il n'y a pas de notion de séquence ou d'ordre entre les opérations. En l'absence d'un tel contrat, si deux types de port sont liés, le premier doit fournir l'ensemble des opérations que le deuxième requiert et vice et versa. Le contrat de port permet de relâcher cette contrainte et d'augmenter la généralité des composants.

Langage de contrat de port

Un type de port est identifié par un nom, un ensemble d'opérations qu'il fournit ou qu'il requiert (voir Section 4.1.1). Il appartient nécessairement à un type de composant. Une liaison est déclarée par la mise en relation de types de port identifiés par leur nom. À partir des identifiants des opérations, nous déclarons les dépendances entre les opérations en identifiant l'ensemble des combinaisons des opérations admissibles pour une déclaration de liaison. Ces combinaisons admissibles traduisent l'indépendance de la mise en œuvre d'un service par rapport à une ou plusieurs opérations. Le sous-ensemble des opérations devant nécessairement participer à l'interaction se définit à l'aide de deux opérateurs :

- l'opérateur `+` pour associer différentes opérations devant participer à l'interaction,
- l'opérateur `|` pour autoriser la définition de plusieurs sous-ensembles admissibles.

La grammaire de ce langage est fournie en Annexe C de ce document.

Exemple de contrat de port pour l'application de *DataPhoneCenter*

Dans notre exemple (voir Figures 4.3 et 4.4), l'accès au système de fichiers par l'intermédiaire du port **p20** n'impose pas nécessairement de venir faire un accès en lecture et en écriture. Le port **p20** est alors enrichi à l'aide du contrat de port décrit ci-dessous.

$$(open + read + close) | (open + write + close)$$

Celui-ci précise que la liaison au type de port **p20** autorise deux combinaisons : le client peut ne pas posséder l'opération **write** ou ne pas posséder l'opération **read**. Ce contrat permettra de valider la liaison entre **p13** et **p20**. Sous forme factorisée, ce contrat est équivalent aussi à $open + close + (read | write)$.

4.2.2.2 Contrat d'assertions

Un des objectifs principaux des modèles de composant est d'améliorer la réutilisation de briques logicielles existantes. Cette réutilisation passe par une description précise des composants. Entre autres, il est nécessaire de spécifier avec le plus de précision possible le type des paramètres échangés entre un composant et son environnement. Spécifier ce type consiste à définir un ensemble de valeurs acceptables pour un paramètre. De nombreux travaux autour du concept de *Design by Contract* [Meyer, 1992a] ont montré la difficulté de la définition de types complexes et les problèmes de réutilisation associés. La réutilisation étant un des défis de l'architecture logicielle, SafArch propose d'ajouter un deuxième contrat appelé **contrat d'assertions** qui reprend partiellement les résultats de Meyer. Il consiste à ajouter des contraintes sur les paramètres des opérations, les types de retour des opérations ou les attributs des types de composant.

La notion de *Design by Contract* en programmation par objets permet de décrire l'impact de l'appel d'une méthode sur l'état de l'objet. Ce type de spécification permet de garantir que la mise en œuvre d'une méthode respecte sa spécification (son contrat). Cependant, ces mécanismes explicitant les attentes quant à l'implémentation

d'un composant ne sont pas repris dans SafArchie, car ils ne cadrent pas avec l'objectif de SafArchie, à savoir garantir la cohérence d'un assemblage de composants. Les problématiques liées à la bonne mise en œuvre des spécifications des composants ne sont pas abordées.

Au contraire, l'utilisation de pré et post-conditions dans un contexte de composant fournissant et requérant des opérations est particulièrement adéquate. Elle ouvre la voie à la négociation de contrats au moment de l'assemblage de composants [Chang and Collet, 2005]. En effet, une pré-condition sur les paramètres d'une opération requise définit un contrat que le client s'engage à respecter. Réciproquement, une pré-condition sur les paramètres d'une opération fournie définit un besoin que le fournisseur de service exprime. Réciproquement, une post-condition sur les types de retour d'une opération requise définit une attente du client vis-à-vis du fournisseur de service alors que cette même contrainte placée au niveau de l'opération fournie exprime un contrat que le fournisseur de service s'engage à respecter.

Langage de définition des assertions

Le choix du langage d'expression de contraintes est primordial pour obtenir un pouvoir d'expression suffisant au niveau des assertions. Différents modèles de spécification proposent d'étendre la formalisation de ces contraintes en utilisant le langage naturel dans des commentaires. Le langage naturel offre l'avantage de préciser une spécification en ajoutant une documentation indispensable à un modèle. Cependant, il est bien souvent ambigu et imprécis. Il est alors difficile d'exprimer clairement des contraintes complexes et de lier le texte aux éléments du modèle. Ce problème de langage de contraintes a été soulevé depuis longtemps et pas uniquement dans la spécification d'architectures logicielles. De nombreux langages de spécifications très rigoureux ont d'ailleurs été produits avec une faible utilisation du fait d'un formalisme difficile à appréhender.

Afin de rester convenablement intégré dans une description d'architecture logicielle, le langage de contrat d'assertions doit posséder les caractéristiques suivantes :

- Le langage de définition des contrats d'assertions doit permettre d'augmenter la précision d'un modèle de description de type d'architecture,
- Le langage de définition des contrats d'assertions doit être précis, non ambigu et lisible facilement par l'ensemble des acteurs d'un projet informatique,
- Le langage de définition des contrats d'assertions doit être un langage déclaratif, ses expressions ne peuvent avoir ni effet de bord, ni action sur le comportement du composant,
- Le langage de définition des contrats d'assertions doit être un langage typé de façon à ce que les expressions de ce langage puissent être évaluées afin de déterminer leur correction (*correctness*).

Plutôt que de redéfinir complètement notre propre langage pour répondre à ces exigences, nous avons travaillé à partir d'un langage existant à savoir OCL (*Object Constraint Language*) [OMG, 2003b] que nous avons adapté à notre contexte de description d'architectures.

Les transformations apportées à la grammaire OCL concernent la déclaration de contexte de l'expression OCL afin de prendre en compte la notion de port et la suppression du *constructeur pour post-condition @pre*. Ce dernier permet de récupérer la valeur d'une propriété au début, avant l'évaluation de la post-condition. Son utilisation permet de contraindre l'implémentation d'une méthode dans une approche par objets. La problématique liée à la mise en œuvre correcte du composant n'étant pas prise en compte dans SafArchie, le constructeur pour post-condition n'a pas sa place dans notre langage d'assertions. La définition de contexte est le lien entre une expression OCL et l'entité à laquelle elle s'applique. Dans OCL, cette entité est soit une classe, une interface ou un type de donnée. Dans SafArchie, cette entité est un type de port appartenant à un type de composant pour les pré et les post-conditions ou un attribut appartenant à un type de composant pour les invariants.

Adapté au contexte de SafArchie, les pré-conditions sont des assertions qui doivent être vraies avant l'appel d'une opération. De façon similaire, les post-conditions sont des assertions qui doivent être vraies après l'appel d'une opération. Finalement, les invariants placés généralement sur les attributs des composants doivent être vrais

durant toute la vie de l'instance d'un composant. Les pré-conditions et les post-conditions sont des expressions logiques du premier ordre, des inégalités qui permettent, entre autres, de sous-typer les paramètres et les types de retour des opérations. Les invariants, quant à eux, permettent de sous-typer les attributs d'un composant et donc de garantir un état cohérent de l'ensemble des instances d'un composant.

Exemple de contrats d'assertions pour l'application *DataPhoneCenter*

Par exemple, au niveau du type de port **p3** du type de composant *AA*, la valeur du paramètre de l'opération **withdraw** doit toujours être supérieure à 5 (c'est-à-dire un minimum de cinq euros doit être débité) :

```
context AA : :p3 : :withdraw(amount : long) : long
pre : self.amount > 5
```

Au niveau du composant *Bank*, il ne peut y avoir de débit supérieur à 2000 euros.

```
context Bank : :p17 : :withdraw(amount : long) : long
pre : self.amount < 2000
```

4.2.3 Contrat comportemental d'un composant

Une architecture logicielle à base de composants est un système dit parallèle, c'est-à-dire composé de plusieurs processus qui potentiellement s'exécutent simultanément et s'échangent des informations. Le parallélisme est dit asynchrone car le système ne possède pas une horloge globale qui, à intervalles réguliers, pilote l'exécution des processus. Ainsi, chaque processus est libre d'évoluer à son propre rythme, mais doit se synchroniser avec d'autres processus lorsqu'il veut communiquer ou accéder à des ressources partagées. L'asynchronisme est une hypothèse à la base de l'étude de la cohérence d'un assemblage dans SafArchie.

Comme nous l'avons présenté dans la section 2.2.3 page 22, les systèmes asynchrones n'ont pas de propriétés évidentes de compositionnalité [Garavel, 2003], c'est-à-dire qu'il n'est pas facile de construire un système correct par assemblage de sous-systèmes plus simples et prouvés corrects. Il est donc nécessaire de spécifier la sémantique de la composition de sous-systèmes afin de pouvoir construire le système global et d'analyser sa correction.

Le dernier niveau de contrat, appelé *contrat de comportement*, dans SafArchie permet de décrire le comportement d'un composant primitif en termes de traces des messages qu'il échange avec son environnement. Sans chercher à définir complètement notre propre langage, nous utilisons un formalisme algébrique permettant de décrire le comportement de chaque composant à l'aide d'un système à transitions étiquetées. Ce contrat de comportement décrit les échanges de messages reçus et émis par les instances d'un type de composant en fonction des enchaînements de ses opérations.

4.2.3.1 Les modèles des systèmes à transitions

L'idée d'un système qui reçoit des signaux discrets (les entrées) de l'extérieur et qui produit d'autres signaux (les sorties) agissant sur l'extérieur est très générale. Elle est appliquée par exemple pour modéliser le fonctionnement de circuits électroniques, de systèmes automatiques ou le comportement de systèmes biologiques. Il est donc naturel qu'une théorie mathématique ait été développée pour modéliser de façon abstraite et unifiée, ces systèmes discrets. Les systèmes de transitions finis sont un des formalismes utilisés pour décrire de tels systèmes. Bien que mathématiquement très simple, ce formalisme permet de modéliser, la plupart des aspects des systèmes de processus et joue un rôle important dans la définition de leur sémantique. La plupart des travaux sur la sémantique des processus dits parallèles ou communicants reposent sur la notion d'automate. Un automate fini, constitué d'états et de transitions étiquetées entre ces états, permet de décrire un système dont l'état évolue au cours du temps. À

partir d'une représentation d'un système sous forme d'automate, il est possible d'observer et de vérifier certaines propriétés de ce système, telles que la présence de situation de blocage. Dans ce qui suit, nous présentons le modèle générale des systèmes de transitions étiquetées ainsi que leurs propriétés. Nous décrivons ensuite les automates à entrées sorties qui sont des systèmes de transitions étiquetées qui considèrent deux types de transitions : les entrées et les sorties. Ces automates permettent de spécifier le comportement d'un système qui émet et reçoit des informations de son environnement. Nous nous limitons à des systèmes dont les ensembles d'états et d'interactions sont finis. Cette restriction ne nous pénalise pas, car les systèmes réels peuvent toujours être modélisés par des automates finis.

Systèmes de transitions étiquetées

Le modèle des systèmes de transitions étiquetées est utilisé pour décrire la sémantique d'un certain nombre de langages de spécification, comme CCS [Milner, 1980], CSP [Hoare, 1985], FSP [Magee and Kramer, 1999] ou LOTOS [ISO, 1989], car il fournit une structure mathématique simple appropriée à la modélisation des processus communicants décrits dans ces langages.

Définition

Un système de transitions étiquetées fini, appelé LTS (*Labeled Transition System*) est défini par un quadruplé (S, A, Δ, s_0) où :

- S est un ensemble fini non vide d'états,
- A est un ensemble fini non vide d'actions observables,
- $\Delta \subseteq S \times (A \cup \{\tau\}) \times S$ est l'ensemble des transitions où $\tau \notin A$ est une action interne non observable,
- s_0 est l'état initial du système de transitions étiquetées.

La figure 4.5 illustre la représentation graphique du système de transitions étiquetées E , tel que :
 $E = (\{s_0, s_1, s_2, s_3\}, \{a, b, c\}, \{(s_0, a, s_1), (s_1, b, s_2), (s_1, c, s_3), (s_2, c, s_3), (s_3, d, s_0)\}, s_0)$.

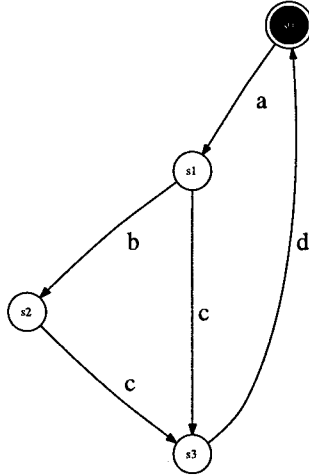


FIG. 4.5 – Représentation graphique d'un système de transitions étiquetées

Dans la suite, pour un LTS $E = (S, A, \Delta, s_0)$ et une trace σ de n actions tel que $\sigma = \mu_1\mu_2...\mu_n \in A^n$, nous utiliserons les notations suivantes :

- (s_1, σ, s_{n+1}) si $(\exists (s_i)_{1 \leq i \leq n+1} \in S^{n+1})(\forall i, 2 \leq i \leq n+1)((s_{i-1}, \mu_{i-1}, s_i) \in \Delta)$,
- (E, σ, s_n) si (s_0, σ, s_n) ,
- (s_1, ε, s_2) si $s_1 = s_2$ ou $(\exists n \geq 1)(s_1, \tau^n, s_2)$,

- $(s_1, \vec{\mu}, s_2)$ si $(\exists (s_3, s_4) \in S)(s_1, \varepsilon, s_3) \wedge (s_3, \mu, s_4) \wedge (s_4, \varepsilon, s_2)$,
- $(s_1, \vec{\sigma}, s_{n+1})$ si $(\exists (s_i)_{1 \leq i \leq n+1} \in S^{n+1})(\forall i, 2 \leq i \leq n+1)(s_{i-1}, \overrightarrow{\mu_{i-1}}, s_i)$,

Mathématiquement un système de transitions étiquetées peut contenir de nombreux états qui ne seront pas accessibles depuis l'état initial s_0 . De même, certaines actions ne pourront jamais être jouées, car elles relient deux états inaccessibles. Seule la *composante connexe* contenant l'état initial est pertinente pour comprendre le fonctionnement d'un système. La composante connexe contenant l'état initial d'un LTS E , notée $CC(E)$, est le système de transitions étiquetées $CC(E) = (S_C, A_C, \Delta_C, s_{0_C})$ tel que $S_C = \{s \in S \mid s \text{ accessible}\}$, $A_C = A$, $\Delta_C = \{(s, u, s') \Delta \mid s \in S_C\}$ et $s_{0_C} = s_0$. Si $E = CC(E)$, on dit que E est initialement connecté.

Les automates à entrées sorties

Une variante des systèmes de transitions étiquetées est obtenue en subdivisant l'ensemble des actions en deux sous-ensembles, les entrées, c'est-à-dire les réceptions, et les sorties, c'est-à-dire les émissions. Ceci permet de distinguer les actions contrôlées par le système, les sorties de celles qui sont contrôlées par son environnement, les entrées. Le modèle obtenu est celui des automates à entrées sorties [Lynch and Tuttle, 1989]. Ce modèle a été utilisé pour modéliser des composantes autonomes dans les systèmes concurrents distribués.

Définition

Un automate à entrées sorties, appelé IOA (Input-Output Automaton) est défini par un élément d'un quintuplé $(S_A, I_A, O_A, \Delta_A, s_{0_A})$ où

- S_A est un ensemble fini non vide d'états,
- I_A est un ensemble fini non vide d'entrées,
- O_A est un ensemble fini non vide de sorties avec $I_A \cap O_A = \emptyset$,
- $\Delta_A \subseteq S_A \times ((I_A \cup O_A) \cup \{\tau\}) \times S_A$ est l'ensemble des transitions où τ est une action interne non observable. Chaque élément de Δ_A est une transition entre un état de départ et un état d'arrivée. Cette transition est soit associée à une action observable (émission ou réception d'une interaction), soit associée à une action interne étiquetée par τ ,
- s_{0_A} est l'état initial de l'automate.

La figure 4.6 illustre la représentation graphique de l'automate à entrées sorties $A = (\{s_0, s_1, s_2, s_3\}, \{x_1, x_2, x_3\}, \{y_1, y_2, y_3\}, \{(s_0, x_1, s_1), (s_0, x_2, s_2), (s_0, x_3, s_3), (s_1, y_1, s_0), (s_2, y_2, s_0), (s_3, y_3, s_0)\}, s_0)$.

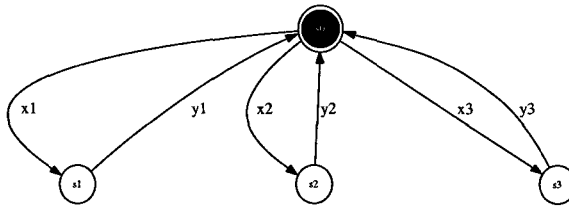


FIG. 4.6 – Représentation graphique d'un automate à entrées sorties

Pour un automate à entrées sorties $A = (S_A, I_A, O_A, \Delta_A, s_{0_A})$, si l'on occulte la distinction entre les entrées et les sorties, on obtient le système à transitions étiquetées correspondant E en regroupant les ensembles I_A et O_A en un seul ensemble d'actions observables. Ce système à transitions étiquetées E est donc défini par $(S_A, I_A \cup O_A, \Delta_A, s_{0_A})$.

Vocabulaire

Une séquence d'actions *observables* σ est dite une trace observable à partir de l'état s s'il existe un état s' tel que $(s, \vec{\sigma}, s')$. L'ensemble des traces observables à partir de l'état s est noté $Tr_E(s)$. L'ensemble des traces observables

à partir de l'état initial s_0 représente l'ensemble des traces observables du système à transitions étiquetées E , on le note Tr_E .

Une trace d'un automate à entrées sorties A à partir d'un état s est la donnée d'une séquence d'actions observables $\sigma \in (I_A \cup O_A)^*$ telle que $(\exists s_n \in S)(s, \vec{\sigma}, s_n)$. L'ensemble des traces de A à partir de s est noté $Tr_A(s)$ et on le note Tr_A dans le cas où $s = s_{0A}$. La notion de trace permet de faire abstraction des actions internes.

Un système à transitions étiquetées est dit *non déterministe* lorsqu'il existe au moins un état s et une séquence d'actions observables σ tels que $(s, \vec{\sigma}, s_1), (s, \vec{\sigma}, s_2)$ et $s_1 \neq s_2$. Dans le cas des automates à entrées sorties, un automate à entrées sorties A est *non déterministe* s'il existe $(s, u, s'), (s, u, s'') \in \Delta^2$ et $s' \neq s''$ pour au moins un état s et une action u ou si l'action interne τ est présente, sinon A est déterministe.

Un état s' est *accessible* à partir d'un état s s'il existe une trace σ telle que (s, σ, s') . Si $s = s_0$ alors s' est dit accessible.

Si pour chaque $s \in S_A$ et tout $x \in I_A$, il existe $s' \in S_A$ telle que $(s, x, s') \in \Delta$, alors l'automate A est complètement spécifié (input-enabled) ; sinon A est partiellement spécifié.

Un état s' d'un automate à entrées sorties A est accessible à partir d'un état s s'il existe $\sigma \in Tr_A(s)$ telle que $(s, \vec{\sigma}, s')$. Si s est l'état initial alors s' est dit accessible.

Dans un automate à entrées sorties A déterministe, un état s et une séquence $\sigma \in Tr_A(s)$ déterminent un unique état final après l'exécution de la trace σ , on le note s_σ . L'ensemble $S(s, \sigma) = \{s' \mid \exists \sigma' \text{ préfixe de } \sigma \text{ tel que } s' = s_{\sigma'}\}$ représente les états accessibles à partir de l'état s par un préfixe de σ .

La composante connexe contenant l'état initial, notée $CC(A)$ est l'automate à entrées sorties $CC(A) = (S_C, I_C, O_C, \Delta_C, s_{0C})$ tel que $S_C = \{s \in S_A \mid s \text{ est accessible}\}$, $I_C = I_A$, $O_C = O_A$, $\Delta_C = \{(s, u, s') \in \Delta_A \mid s \in S_C\}$ et $s_{0C} = s_{0A}$. Si $A = CC(A)$ alors A est initialement connecté.

4.2.3.2 Un langage algébrique pour la description des échanges de messages

Dans SafArchie, le comportement externe des composants est décrit en utilisant des systèmes de transitions étiquetées. Celles-ci sont décrites textuellement comme des processus à états finis à l'aide d'un langage fortement inspiré de FSP (*Finite State Processes*) [Magee, 1999] que nous avons nommé dans SafArchie SFSP (pour *Simple FSP*). La grammaire de ce langage est décrite en Annexe E. La description LTS du système constitue donc la forme graphique du comportement externe du composant et la description SFSP, sa forme algébrique. Ce langage définit le comportement externe d'un composant comme une séquence finie de messages reçus et émis par le composant. Contrairement à FSP, qui travaille à partir de séquences finies d'actions quelconques, SFSP travaille sur une description du comportement externe du composant définie uniquement à partir de quatre types de messages :

- les requêtes émises notées $!m$,
- les réponses à une requête notées $!m\$$,
- les requêtes reçues notées $?m$,
- les réceptions de réponses à une requête émises notées $?m\$$.

Dans SafArchie, le mode de communication est l'appel de procédure modélisé par deux messages. Une opération fournie $op1$ est traduite par un message reçu ($?op1$) qui est suivi de l'émission du retour du message ($!op1\$$). Au contraire, une opération requise $op2$ se traduit par un message émis ($!op2$) suivi de la réception du retour à ce message ($?op2\$$) (Figure 4.1). Le suffixe $\$$ permet de distinguer le message de retour du message d'appel.

La construction d'un langage algébrique de description de comportement vise à fournir un langage simple d'utilisation, intuitif et basé sur un nombre limité d'opérateurs. Dès lors, la description algébrique du comportement d'un type de composant en fonction des messages qu'il reçoit et qu'il émet est concise et précise. Inspiré par FSP, SFSP définit un nombre limité d'opérateurs pour définir le comportement d'un type de composant. Nous utilisons

Définition de l'opération	Traduction en termes de messages	
provide op()	?op, !op\$	message reçu suivi d'un message émis
require op()	!op, ?op\$	message émis suivi d'un message reçu

TAB. 4.1 – Lien entre la spécification du comportement à l'aide de messages et la définition structurelle du composant

principalement la notion de préfixe, de choix indéterministe, d'itération, de composition parallèle, de camoufflage et de réétiquetage.

Le préfixe

Le préfixe permet de décrire une séquence linéaire de messages. Si $!x$ (ou $?x$) est l'émission d'un message (ou la réception) et $!y$ ou $?y$ un autre message alors $(!x \rightarrow !y)$ décrit le comportement d'un composant qui initialement émet le message x et qui ensuite émet le message y . Une forme généralisée de cet opérateur noté $(!x \Rightarrow P)$ avec P un autre contrat de comportement décrit le comportement d'un composant qui initialement émet le message x et qui a, ensuite, le comportement décrit par P .

Le choix indéterministe

L'opérateur de choix permet de proposer une alternative au sein d'une description de comportement.

Si $!m_1$ et $!m_2$ sont deux émissions de messages alors $(!m_1 \rightarrow P \mid !m_2 \rightarrow Q)$ décrit le comportement d'un composant qui initialement émet soit le message m_1 , soit le message m_2 . Lorsque l'un de ces messages est émis, le comportement du processus est celui décrit par le contrat de comportement P si le message m_1 a été envoyé, ou le contrat de comportement Q si le message m_2 a été envoyé. Par défaut, le choix entre deux actions (envoi ou réception de messages) est indéterministe.

L'itération

L'itération permet de spécifier qu'un comportement peut se répéter une ou une infinité de fois.

Si σ est une séquence de messages et P est un processus alors $\sigma^* \rightarrow P$ représente un processus qui peut répéter n fois la trace de messages σ avant de passer à P , n étant compris entre 1 et ∞ .

La composition parallèle

Un opérateur de composition ($\parallel$) permet de modéliser l'assemblage de plusieurs composants.

Si P et Q sont des processus alors $(P \parallel Q)$ représente l'exécution concurrente de P et de Q . L'opérateur $\parallel$ est appelé l'opérateur de composition parallèle.

Définition de la composition

Les systèmes de transitions étiquetées communiquent par un mécanisme de rendez-vous négocié. Les composantes communicantes doivent exécuter une même action simultanément. Ce type de synchronisation est celui utilisé dans CSP [Hoare, 1985]. La communication est dite bloquante dans le sens où si une composante impliquée dans la communication n'est pas dans un état où l'action commune est possible, la réalisation de cette action (l'envoi ou la réception d'un message) sera bloquée. Basé sur ce qui précède, un opérateur de composition parallèle est défini sur les systèmes de transitions étiquetées de la façon suivante :

La composition parallèle de deux systèmes de transitions étiquetées $E_1 = (S_1, A_1, \Delta_1, s_{0_1})$ et $E_2 = (S_2, A_2, \Delta_2, s_{0_2})$, notée $E_1 \parallel E_2$, est définie comme la composante connexe du système de transitions étiquetées $E = (S, A, \Delta, s_0)$ où :

- $S = S_1 \times S_2$,
- $A = A_1 \cup A_2$,
- $s_0 = (s_{0_1}, s_{0_2})$,
- la relation de transition Δ de $E_1 \parallel E_2$ est obtenue au moyen des règles suivantes :
 - $a \in (A_1 \setminus A_2) \cup \{\tau\} \wedge (s_1, a, s'_1) \in \Delta_1 \Rightarrow \forall s_2 \in S_2, ((s_1, s_2), a, (s'_1, s_2)) \in \Delta$
 - $a \in (A_2 \setminus A_1) \cup \{\tau\} \wedge (s_2, a, s'_2) \in \Delta_2 \Rightarrow \forall s_1 \in S_1, ((s_1, s_2), a, (s_1, s'_2)) \in \Delta$
 - $a \in A_1 \cap A_2 \wedge (s_1, a, s'_1) \in \Delta_1 \wedge (s_2, a, s'_2) \in \Delta_2 \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in \Delta$

La composition d'automates à entrées sorties repose sur la synchronisation d'une sortie d'un composant avec l'entrée correspondante dans un composant lié. Par extension par rapport à la relation de composition générale des systèmes de transitions étiquetées, la composition de deux automates à entrées sorties $B_1 = (S_{B_1}, I_{B_1}, O_{B_1}, \Delta_{B_1}, s_{0_1})$ et $B_2 = (S_{B_2}, I_{B_2}, O_{B_2}, \Delta_{B_2}, s_{0_2})$, telle que $O_{B_1} \cap O_{B_2} = \emptyset$, notée $B_1 \parallel B_2$, est définie comme la composante connexe de l'automate à entrées sorties $B = (S_B, I_B, O_B, \Delta_B, s_{0_B})$ où :

- $S_B = S_{B_1} \times S_{B_2}$,
- $I_B = (I_{B_1} \cup I_{B_2}) \setminus (O_{B_1} \cup O_{B_2})$,
- $O_B = O_{B_1} \times O_{B_2}$,
- la relation de transition Δ_B de $B_1 \parallel B_2$ est obtenue au moyen des règles suivantes :
 - $a \in (I_{B_1} \setminus O_{B_2}) \cup \{\tau\} \wedge (s_1, a, s'_1) \in \Delta_{B_1} \Rightarrow \forall s_2 \in S_{B_2}, ((s_1, s_2), a, (s'_1, s_2)) \in \Delta_B$
 - $a \in (I_{B_2} \setminus O_{B_1}) \cup \{\tau\} \wedge (s_2, a, s'_2) \in \Delta_{B_2} \Rightarrow \forall s_1 \in S_{B_1}, ((s_1, s_2), a, (s_1, s'_2)) \in \Delta_B$
 - $a \in (O_{B_1} \setminus I_{B_2}) \cup \{\tau\} \wedge (s_1, a, s'_1) \in \Delta_{B_1} \Rightarrow \forall s_2 \in S_{B_2}, ((s_1, s_2), a, (s'_1, s_2)) \in \Delta_B$
 - $a \in (O_{B_2} \setminus I_{B_1}) \cup \{\tau\} \wedge (s_2, a, s'_2) \in \Delta_{B_2} \Rightarrow \forall s_1 \in S_{B_1}, ((s_1, s_2), a, (s_1, s'_2)) \in \Delta_B$
 - $a \in I_{B_1} \cap O_{B_2} \wedge (s_1, a, s'_1) \in \Delta_{B_1} \wedge (s_2, a, s'_2) \in \Delta_{B_2} \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in \Delta_B$
 - $a \in I_{B_2} \cap O_{B_1} \wedge (s_1, a, s'_1) \in \Delta_{B_1} \wedge (s_2, a, s'_2) \in \Delta_{B_2} \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in \Delta_B$

La composition d'automates à entrées sorties est commutative et associative. L'ordre de composition de plusieurs processus n'a donc pas d'importance.

Le camouflage

SFSP dispose d'un opérateur de camouflage ($\setminus$) et d'un opérateur d'export ($@$) de type port. Il permet de réduire une spécification comportementale vers un sous-alphabet.

Lorsqu'il est appliqué à un processus P , l'opérateur de camouflage $\setminus \{m_1..m_x\}$ cache les messages $m_1..m_x$ de l'alphabet de P et les rend silencieux. Ces messages silencieux sont étiquetés « τ » et ne peuvent être *partagés*, ils ne peuvent servir de point de synchronisation avec d'autres composant de l'assemblage.

Lorsqu'il est appliqué à un processus P , l'opérateur d'export $@ \{m_1..m_x\}$ conserve uniquement les messages $m_1..m_x$ de l'alphabet de P et rend silencieux les autres. Ces messages silencieux sont étiquetés « τ » et ne peuvent être *partagés*.

Ces deux opérateurs sont utilisés lors de la définition d'un composite pour masquer les messages échangés à l'intérieur du composite.

La projection d'un automate vers un sous-alphabet est définie comme suit.

Définition

Pour une séquence $\sigma \in \Sigma^*$ et un sous-ensemble Σ' de Σ , la projection de σ sur Σ' , notée $Pr_{\Sigma'}(\sigma)$, est obtenue en éliminant de σ les symboles n'appartenant pas à Σ' . On note $Pr_A(\sigma)$, la projection de σ sur l'alphabet $(I_A \cup O_A)$ de l'automate à entrées sorties A . Pour un ensemble de traces Y , on note $Pr_{\Sigma'}(Y)$, l'ensemble contenant les projections sur Σ' des éléments de Y . Pour un ensemble X contenant des ensembles de traces, on note $Pr_{\Sigma'}(X)$, l'ensemble $\{Pr_{\Sigma'}(Y) \mid Y \in X\}$.

Ce principe de projection est utilisé pour masquer des états internes du fonctionnement d'un type de composite. Nous gardons en effet uniquement les messages envoyés et reçus par le composite et déclarés dans les types de port délégué.

Le réétiquetage

Le réétiquetage de messages permet de synchroniser des messages possédant des noms différents. Cela correspond à la mise en relation d'opérations ne possédant pas les mêmes noms. Il est appliqué à des processus afin de modifier le nom des étiquettes de messages. La forme générale est la suivante : $/\{\text{nouveau}_1/\text{ancien}_1, \dots, \text{nouveau}_n/\text{ancien}_n\}$.

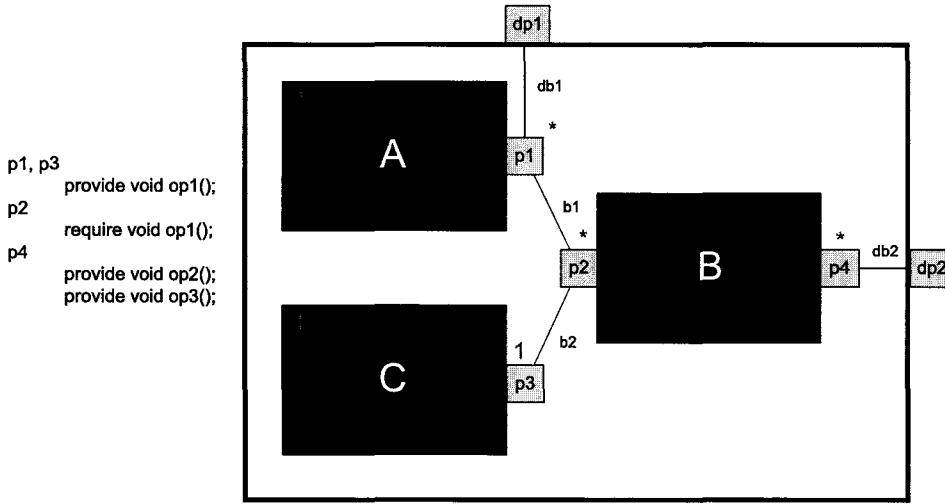
L'opérateur de réétiquetage $/$ prend en entrée un LTS $P = \langle S_1, A_1, \Delta_1, s_{0_1} \rangle$ et une fonction $f : Act \rightarrow Act$ et retourne le LTS modifié suivant $P/f = \langle S_1, f(A_1), \Delta_1, s_{0_1} \rangle$.

Les opérateurs de camouflage, réétiquetage et composition parallèle sont associés à la description d'un type de composite. Ils ne sont pas explicitement spécifiés par l'architecte, mais sont générés à partir de la description structurelle du type d'architecture.

D'autres opérateurs existent en FSP mais ne sont pas autorisés dans les descriptions de contrat de comportement. Par exemple, FSP propose la notion d'actions partagées et d'actions non partagées. Les actions partagées constituent la manière de modéliser l'interaction entre deux processus. Alors que les actions non partagées peuvent être entrelacées de manière arbitraire, une action partagée doit être exécutée au même moment par tous les processus qui participent à cette action. Dans notre modèle, tout envoi et réception de message doivent être partagés. Il n'est pas possible de définir des messages non partagés qui seraient internes à un composant.

À partir du langage algébrique SFSP, il existe une transformation simple d'une spécification comportementale vers un modèle d'automate. Cette transformation permet de définir la sémantique du langage algébrique à l'aide des opérateurs standards sur les LTS.

L'approche proposée dans cette section pour le contrat de comportement d'un type de composant ne peut pas servir tel quel comme description du comportement des instances de composant. En effet, quand l'architecte travaille sur la spécification du type d'architecture, il ne spécifie pas le nombre d'instances de chaque type de port du composant. Dès lors, il n'est pas possible de connaître l'ensemble des messages participants à la description du comportement du composant. Le contrat de comportement exprime donc ces séquences de messages au niveau des types de composant primitif en considérant une instance de port par extrémité de liaison. Cette contrainte permet de simuler une architecture respectant les contraintes fournies par le modèle de type. Le nommage des messages s'effectue de la manière suivante : chaque message est identifié par un nom composé du nom de la liaison, du nom du port et du nom de l'opération auxquelles il est associé. Ainsi une opération nommée op_1 appartenant à un port p_1 lié par une liaison b_1 à un autre port est identifié par deux messages au niveau du contrat de comportement : $b_1.p_1.op_1$ et $b_1.p_1.op_1\$$. Un schéma récapitulant l'identification des messages est présenté Figure 4.7. Le comportement des composants primitifs devra nécessairement être précisé au niveau du modèle logique afin de prendre en compte le nombre d'instances de port (voir Chapitre 5).



$A = ((?db1.p1.op1 \rightarrow !db1.p1.op1\$ \rightarrow A) \mid (?b1.p1.op1 \rightarrow !b1.p1.op1\$ \rightarrow A)).$
 $B = (?db2.p4.op2 \rightarrow !db2.p4.op2\$ \rightarrow ?db2.p4.op3 \rightarrow !db2.p4.op3\$ \rightarrow (!b1.p2.op1 \rightarrow ?b1.p2.op1\$ \rightarrow B)$
 $\quad \mid (!b2.p2.op1 \rightarrow ?b2.p2.op1\$ \rightarrow B)).$
 $C = (?b2.p3.op1 \rightarrow !b2.p3.op1\$ \rightarrow C).$

FIG. 4.7 – Identification des messages dans les contrats de comportement

4.2.3.3 Exemple

Dans l'exemple du *DataPhoneCenter* présenté dans les Figures 4.3 et 4.4, les contrats des cinq types de composant primitif qui composent le *DataPhoneCenter* sont définis ci-dessous.

$AA = (?e1.p1.connect \rightarrow !e1.p1.connect\$ \rightarrow !b5.p4.reserve \rightarrow ?b5.p4.reserve\$ \rightarrow !e3.p2.select \rightarrow ?e3.p2.select\$ \Rightarrow AA1),$
 $AA1 = (!b5.p4.free \rightarrow ?b5.p4.free\$ \rightarrow !e1.p1.disconnect \rightarrow ?e1.p1.disconnect\$ \Rightarrow END \mid !e1.p1.getcapabilities \rightarrow ?e1.p1.getcapabilities\$ \rightarrow !e1.p1.getpreferredcodec \rightarrow ?e1.p1.getpreferredcodec\$ \Rightarrow AA2),$
 $AA2 = (!b5.p4.free \rightarrow ?b5.p4.free\$ \rightarrow !e1.p1.disconnect \rightarrow ?e1.p1.disconnect\$ \Rightarrow END \mid !e1.p1.setcodec \rightarrow ?e1.p1.setcodec\$ \rightarrow !b5.p4.createapp \rightarrow ?b5.p4.createapp\$ \rightarrow !e1.p1.setconnectionport \rightarrow ?e1.p1.setconnectionport\$ \rightarrow ?b5.p4.destroy \rightarrow !b5.p4.destroy\$ \rightarrow !e1.p1.disconnect \rightarrow ?e1.p1.disconnect\$ \rightarrow !e4.p3.withdraw \rightarrow ?e4.p3.withdraw\$ \rightarrow !e4.p3.deposit \rightarrow ?e4.p3.deposit\$ \Rightarrow END);$

Ce premier contrat indique que le type de composant de facturation et d'authentification (*AA*) peut en cas de connexion d'un client (messages $?e1.p1.connect$ et $!e1.p1.connect\$$), activer le type de composant *MediaCenter* ($!b5.p4.reserve$ et $?b5.p4.reserve\$$) en vue d'une nouvelle délivrance de service et récupérer des informations sur le client auprès d'une base de données ($!e3.p2.select$ et $?e3.p2.select\$$). Selon la nature des informations récupérée sur le client auprès de la base de données deux scénarios sont possibles (c'est un choix indéterministe pour le niveau d'abstraction fournie par le langage SFSP). Dans le premier scénario, le type de composant *DataPhoneCenter* rejette la connexion. Dans ce cas, il notifie le type de composant *MediaCenter* afin que celui-ci libère ses ressources $!b5.p4.free$ et $?b5.p4.free\$$) et notifie le client de ce refus à l'aide d'un message de déconnexion $!e1.p1.disconnect$ et $?e1.p1.disconnect\$$). Dans le deuxième scénario, il accepte l'appel du client. Il négocie alors les paramètres audio en fonction des capacités de l'utilisateur connecté ($!e1.p1.getcapabilities$, $?e1.p1.getcapabilities\$$, $!e1.p1.getpreferredcodec$ et $?e1.p1.getpreferredcodec\$$). Cette négociation entraîne de nouveau l'émergence de deux scénarios. Dans un premier cas, si la négociation échoue, le type de composant *DataPhoneCenter* ferme la connexion. Dans ce cas, il notifie le type de composant *MediaCenter* afin que celui-ci libère ses ressources $!b5.p4.free$ et $?b5.p4.free\$$) et notifie le client de ce refus à l'aide d'un message de déconnexion $!e1.p1.disconnect$ et $?e1.p1.disconnect\$$). Dans un deuxième cas, il prévient le client du choix

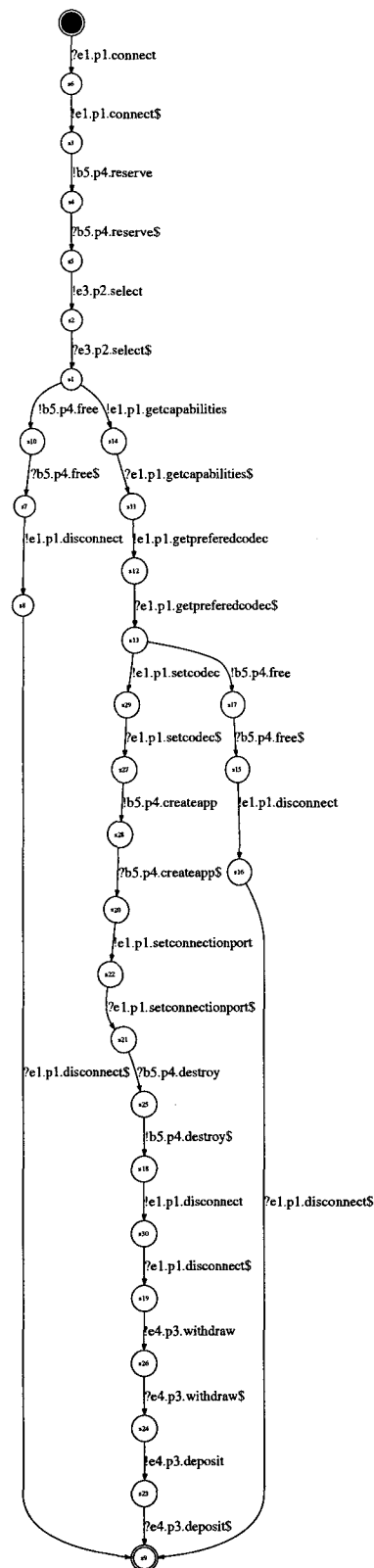


FIG. 4.8 – Automate du comportement du type de composant AA

retenu pour le codec audio (!e1.p1.setcodec et ?e1.p1.setcodec\$) , et crée une nouvelle session au niveau du type de composant *MediaCenter* avec les informations relatives au client (!b5.p4.createapp et ?b5.p4.createapp\$). À la suite de la création de cette session, le type de composant *MediaCenter* indique comment le client peut prendre contact avec lui. Cette information est alors fournie au client (!e1.p1.setconnectionport et ?e1.p1.setconnectionport\$). Le type de composant *AA* se met alors en attente de réception d'une requête de fin de session (b5.p4.destroy et !b5.p4.destroy\$) de la part du type de composant *MediaCenter*. Il déconnecte alors le client (!e1.p1.disconnect et ?e1.p1.disconnect\$) et effectue la facturation liée au service utilisé par le client (!e4.p3.withdraw, ?e4.p3.withdraw\$, !e4.p3.deposit et ?e4.p3.deposit\$). L'automate associé à ce contrat de comportement est décrit dans la figure 4.8 page précédente.

```
MediaCenter = (?b5.p6.reserve →!b5.p6.reserve$ →!b6.p7.prepare →?b6.p7.prepare$ →!b7.p7.prepare →?b7.p7.prepare$ →!b8.p7.prepare →?b8.p7.prepare$ ⇒ MediaCenter1),
MediaCenter1 = (?b5.p6.free →!b5.p6.free$ →!b6.p7.cancel →?b6.p7.cancel$ →!b7.p7.cancel →?b7.p7.cancel$ →!b8.p7.cancel →?b8.p7.cancel$ ⇒ END |?b5.p6.createapp →!b5.p6.createapp$ ⇒ DATAEXCHANGE),
DATAEXCHANGE = (!b6.p7.getvoicexml →?b6.p7.getvoicexml$ ⇒ DATAEXCHANGE1 |!b7.p7.getvoicexml →?b7.p7.getvoicexml$ ⇒ DATAEXCHANGE1 |!b8.p7.getvoicexml →?b8.p7.getvoicexml$ ⇒ DATAEXCHANGE1),
DATAEXCHANGE1 = ((!e2.p5.snddata →?e2.p5.snddata$ →?e2.p5.receiveata →!e2.p5.receiveata$)* ⇒ DATAEXCHANGE |?e2.p5.disconnect →?e2.p5.disconnect$ →!b5.p6.destroy →?b5.p6.destroy$ →!b6.p7.cancel →?b6.p7.cancel$ →!b7.p7.cancel →?b7.p7.cancel$ →!b8.p7.cancel →?b8.p7.cancel$ ⇒ END);
```

Le type de composant *MediaCenter* a un rôle central dans le fonctionnement de l'application. Quand il reçoit une demande de réservation de ressources suivie d'une demande de création d'application (par exemple Lecture d'Email pour M. Dupont), il prend en charge la demande de scenarios VoiceXML et initie le dialogue avec l'utilisateur. En cas de déconnexion de l'utilisateur, il prévient le type de composant de facturation et d'authentification et libère les ressources de chaque composant applicatif. L'automate associé à ce contrat de comportement est décrit dans la figure 4.9 page suivante.

```
EmailReader = (?b6.p8.prepare →!b6.p8.prepare$ ⇒ EmailReader1),
EmailReader1 = (?b6.p8.cancel →!b6.p8.cancel$ ⇒ END |?b6.p8.getvoicexml →!e5.p11.authentication →?e5.p11.authentication$ ⇒ EmailReader2 |?b6.p8.getvoicexml ⇒ EmailReader2),
EmailReader2 = (?e5.p11.list →!e5.p11.list$ →!b6.p8.getvoicexml$ ⇒ EmailReader1 |?e5.p11.getmail →!e5.p11.getmail$ →!b6.p8.getvoicexml$ ⇒ EmailReader1 |?e5.p11.delmail →!e5.p11.delmail$ →!b6.p8.getvoicexml$ ⇒ EmailReader1);
```

Les trois composants applicatifs (*EmailReader*, *WebReader*, et *FSAccess* décrits au niveau du type d'architecture ont un comportement assez similaire. Ils gèrent la traduction entre un protocole d'accès à des données et leur représentation en scénarios VoiceWML. Ainsi à partir d'une requête reçue de la part du type de composant *MediaCenter*, il interroge le service correspondant et fournit le code VoiceXML correspondant à la requête. Les automates associés à ces contrats de comportement sont décrits dans les Figures 4.10 page 93, 4.11 page 93 et 4.12 page 94.

```
WebReader = (?b7.p9.prepare →!b7.p9.prepare$ ⇒ WebReader1),
WebReader = (?b7.p9.cancel →!b7.p9.cancel$ ⇒ END |?b7.p9.getvoicexml ⇒ WebReader2),
WebReader2 = (!e6.p12.get →?e6.p12.get$ →!b7.p9.getvoicexml$ ⇒ WebReader1);
```

```
FSAccess = (?b8.p10.prepare →!b8.p10.prepare$ ⇒ FSAccess1),
FSAccess1 = (?b8.p10.cancel →!b8.p10.cancel$ ⇒ END |?b8.p10.getvoicexml ⇒ FSAccess2),
FSAccess2 = ((!e7.p13.open →?e7.p13.open$ →!e7.p13.read →?e7.p13.read$ →!e7.p13.close →?e7.p13.close$) * ⇒ FSAccess3),
FSAccess3 = (!b8.p10.getvoicexml$ ⇒ FSAccess1);
```

Le contrat de comportement du composite est obtenu par synchronisation du contrat de comportement de ses fils. Cette synchronisation est définie à partir de l'expression présentée ci-dessous. Cette expression est générée

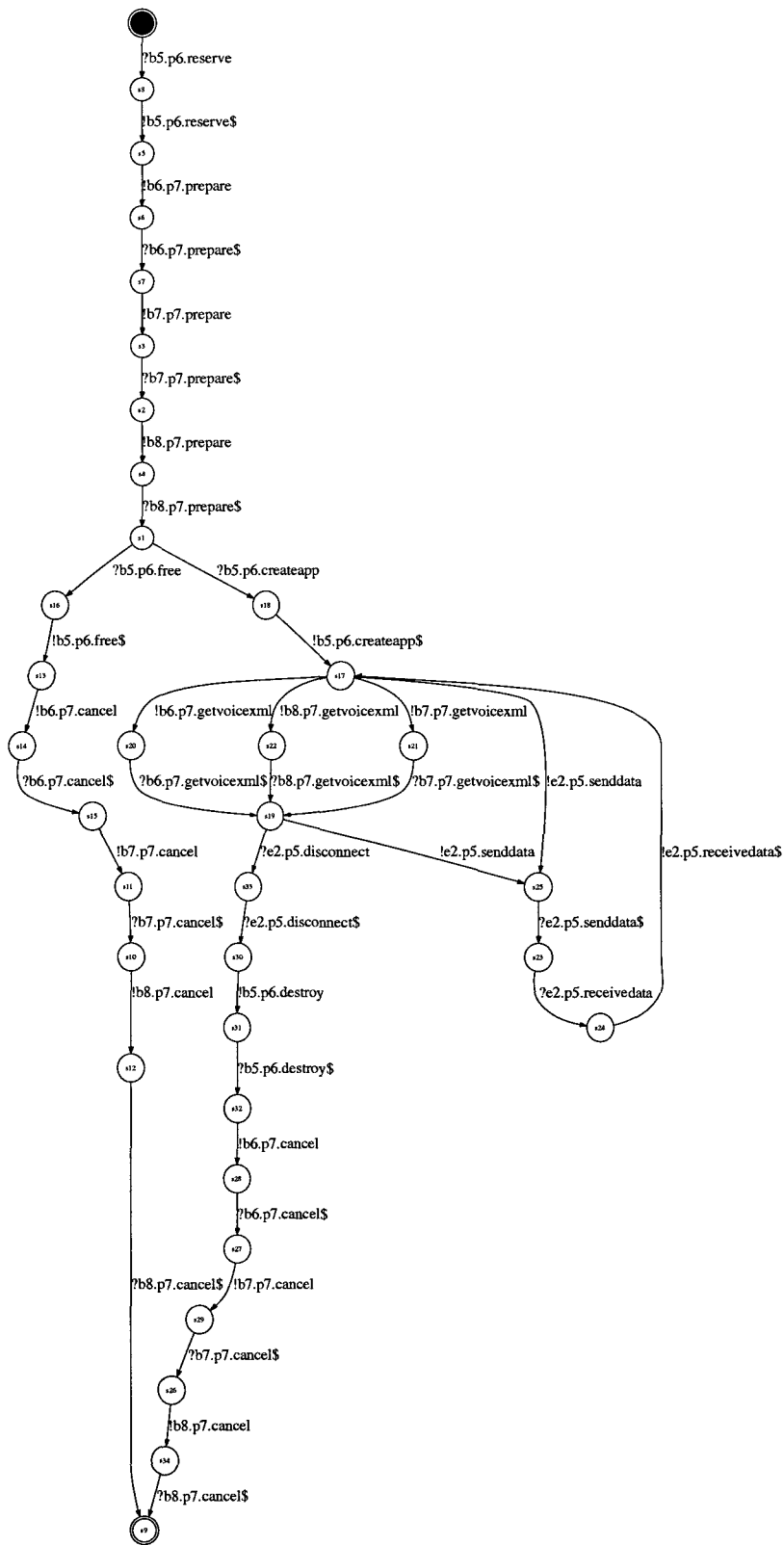


FIG. 4.9 – Automate du comportement du type de composant *MediaCenter*

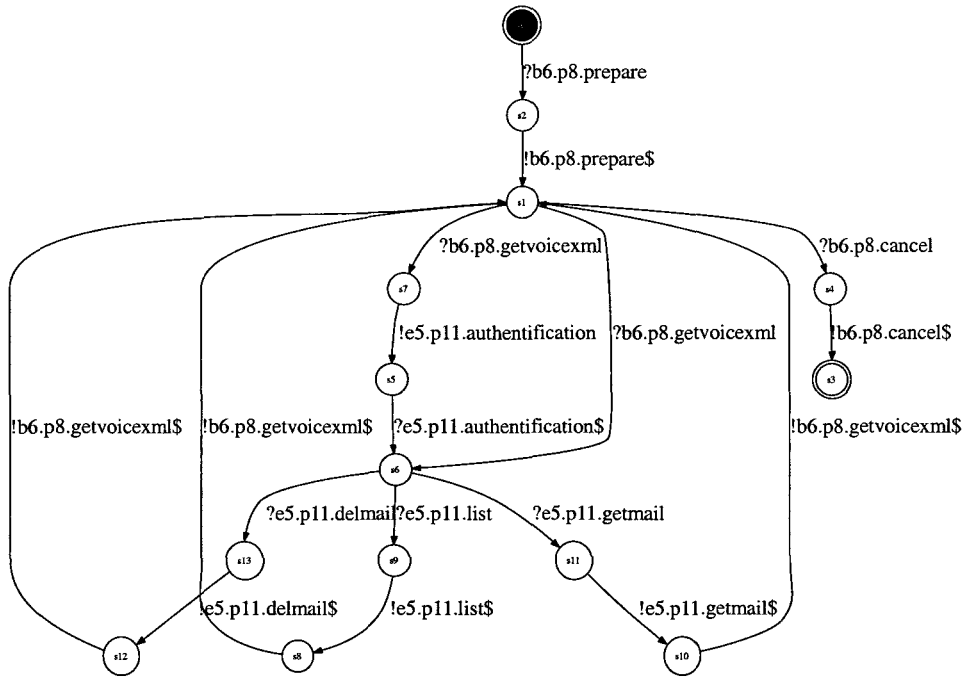


FIG. 4.10 – Automate du comportement du type de composant *EmailReader*

automatiquement à l'aide des informations contenues dans la description de type d'architecture.

$DataPhoneCenter = (AA \parallel MediaCenter \parallel EmailReader \parallel WebReader \parallel FSAccess)$
 $/\{b6.p7/b6.p8, b7.p7/b7.p9, b8.p7/b8.p10, b5.p4/b5.p6\} \setminus \{b5, b6, b7, b8\};$

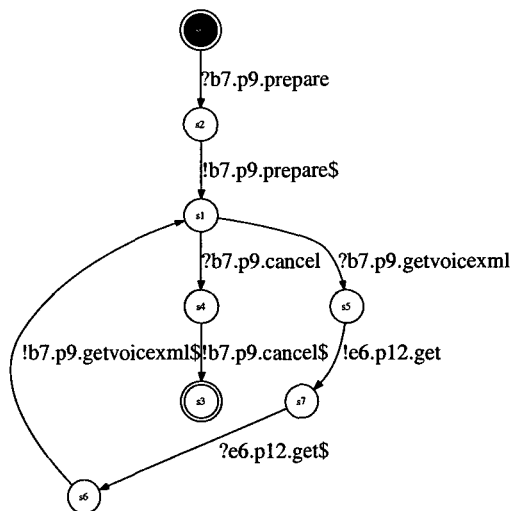


FIG. 4.11 – Automate du comportement du type de composant *WebReader*

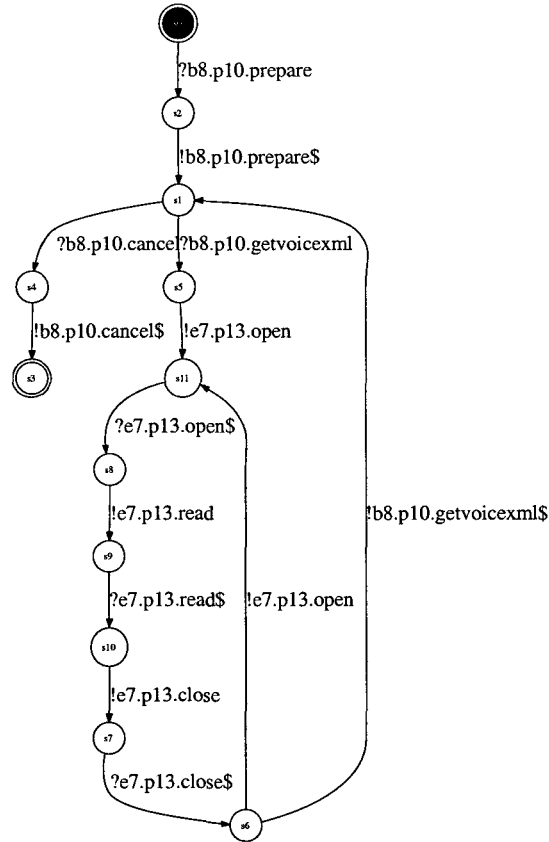


FIG. 4.12 – Automate du comportement du type de composant FSAccess

4.2.3.4 Bilan sur SFSP

Le langage de description de contrats de comportement SFSP n'est volontairement pas très expressif. Il ne prend pas en compte par exemple les valeurs des paramètres des messages échangés ou les conditions associées aux choix internes au composant. Ce manque d'expressivité permet cependant de garder une description de la trace d'exécution du système humainement compréhensible assez proche d'une vision scénario de ce système.

Une autre approche qui peut sembler plus intuitive que d'associer les contrats de comportement au niveau des composants consiste à décrire ces contrats de comportement au niveau des types de port. Cette description aurait l'avantage de faire apparaître le typage comportemental au niveau des types de port permettant ainsi leur réutilisation. Cependant, l'inférence du contrat comportemental du type de composant devient dès lors très difficile à exprimer à partir de ces descriptions de comportement de port. Nous aurions alors une redondance d'informations entre le comportement d'un type de port et le comportement d'un type de composant. C'est pourquoi, nous préférons décrire le comportement global d'un type de composant. L'obtention d'une description de dépendance entre opérations d'un port s'obtient alors par projection du contrat de comportement d'un composant sur l'alphabet défini par les messages reçus et envoyés par le port.

4.2.4 Conclusion sur les langages de contrats dans SafArchie

4.2.4.1 Structuration des contrats

Dans SafArchie, la spécification d'un composant se traduit par un ensemble de contrats qui définissent le plus précisément possible ses exigences et ses aptitudes. Les composants permettent d'encapsuler des parties de système sans demander à leurs utilisateurs de connaître les détails de leur réalisation. Lorsqu'un composant est assemblé avec d'autres composants, leurs contrats sont confrontés pour satisfaire aux différentes hypothèses et contraintes. Cette phase peut être réalisée automatiquement par un outil de vérification ou manuellement par les concepteurs du système en fonction de la richesse du contrat. Dans notre approche, cette phase est réalisée par un analyseur de spécifications dont le fonctionnement est présenté dans la section 4.3.

La plupart des spécifications actuelles de composant (CCM ou Fractal) reposent uniquement sur l'analyse du typage des interfaces offertes et requises (même nom de service, même nombre d'arguments, mêmes types, mêmes exceptions, *etc.*). Certaines extensions de ces modèles permettent de préciser à l'aide de contraintes le typage des paramètres [Collet *et al.*, 2005]. Ces contraintes sont utilisées défensivement à l'exécution pour lever des exceptions en cas de rupture de contrat et non pas préventivement par des outils de vérification ou de compilation sophistiqués. L'approche proposée par SafArchie se veut préventive. Elle permet de travailler sur différents niveaux de contrats. Le principal avantage de ces différents contrats réside dans le fait que l'on peut préciser la sémantique d'utilisation des composants. Les trois types de contrat peuvent être utilisés indépendamment les uns des autres dans la description d'un type d'architecture. Pour comparer les contrats de SafArchie à la classification de Beugnard *et al* [Beugnard *et al.*, 1999], le premier contrat appelé contrat syntaxique est précisé dans SafArchie à l'aide de la déclaration des types de port et des opérations. Le deuxième niveau de contrat est précisé à l'aide du contrat d'assertions et du contrat de port qui explicitent respectivement les conditions d'utilisation et le résultat attendu pour une opération et les conditions d'utilisation d'un type de port. Enfin, le troisième niveau de contrat est précisé par le contrat de comportement dans SafArchie qui permet pour chaque composant de préciser ses interactions avec l'environnement en termes de séquences de messages émis et reçus. Les contrats liés à la qualité de service, qui ne sont pas traités pour l'instant dans SafArchie, pourraient faire l'objet de travaux futurs.

4.2.4.2 Recouvrement entre les contrats

Il existe un recouvrement très faible entre les différents langages de contrats de SafArchie. Cependant, le contrat de comportement et le contrat de port sont en partie redondants. Ils permettent d'exprimer des contraintes antinomiques. En effet, le contrat de port définit les opérations devant nécessairement être présentes afin d'autoriser la création d'une liaison avec un autre port. Le contrat de comportement spécifie comment un comportement interagit avec son environnement. Cette séparation de l'information permet de définir un contrat de port qui autorise des liaisons dont on sait pertinemment à la vue du contrat de comportement qu'elles vont engendrer un interblocage.

Cependant, ces deux contrats sont tout de même nécessaires, car ils ne sont pas de même granularité. En effet, il n'est pas nécessairement aisé à partir du contrat de comportement de voir si une opération donnée peut être omise au niveau d'une liaison entre deux types de port. Dès lors, le contrat de port offre une expression précise de cette information là où le contrat de comportement noie cette information au niveau de l'automate de comportement. En outre, il n'est pas possible de générer automatiquement les contrats de ports à partir des contrats de comportement, car l'existence d'une trace ne possédant pas telle ou telle opération n'impose pas nécessairement qu'il soit possible de concevoir un système où ces informations ne sont pas fournies ou requises par l'environnement. Le contrat de port contient une information sémantique liée au type de composant modélisé qui n'est pas contenue dans le contrat de comportement. Seule l'analyse de la cohérence entre ces deux informations a du sens.

4.3 Construire un type d'architecture logicielle cohérent

L'objectif de SafArchie est de définir un modèle d'architecture en vue de garantir certaines propriétés structurelles et comportementales sur un assemblage de composants. Le modèle de type identifie la liste des types de composant, types de composite et liaisons. Les liaisons identifient les interactions pouvant survenir au cours du cycle de vie de l'application. Notre objectif est de garantir la *cohérence* des futurs assemblages et de détecter des incompatibilités entre types de composant liés, dès la spécification d'un type d'architecture.

Les sections 4.1 et 4.2 ont présenté le modèle de type de SafArchie qui permet la spécification de types d'architectures définis d'un point de vue structurel et comportemental. Ce modèle fournit un ensemble d'éléments qui vont permettre l'analyse de ce type d'architecture logicielle. Au sein de *SafArchie*, un type d'architecture logicielle est caractérisé par deux familles d'informations :

- la définition des composants d'un point de vue structurel et comportemental par l'intermédiaire des contrats,
- la définition des possibilités d'assemblage de ces composants, c'est-à-dire leurs possibles interactions.

À partir de ces informations, l'analyse d'un type architectural consiste à vérifier que ces possibles interactions entre composants respectent l'ensemble des contrats des composants. Ce chapitre définit donc pour les différents contrats du modèle de type de SafArchie la notion de **compatibilité** entre contrats.

Une compatibilité entre deux composants connectés signifie que l'interaction entre ces deux composants est possible et que, par conséquent, les informations nécessaires au respect des contrats structurels et comportementaux d'un composant sont fournies par le composant auquel il est relié. Nous noterons cette relation de compatibilité par le signe utilisé dans les systèmes de typage $\prec$: $A \prec B^*$ la compatibilité entre un type de composant A et un ensemble de types de composant B reliés. Cette relation, en fonction du type de contrat, pourra être plus ou moins contrainte.

Nous détaillons maintenant pour chacun des contrats ce qu'est la relation de compatibilité entre deux types de composant. Nous pourrions ainsi définir un assemblage cohérent à condition qu'il permette le respect des différents contrats d'interfaces des composants formant le composite. En itérant le procédé à l'ensemble des composites d'une architecture, nous pourrions valider un type architectural.

4.3.1 Analyse structurelle de l'assemblage

Dans cette section, nous présentons l'analyse liée aux deux premiers niveaux de contrats (les contrats syntaxiques et sémantiques). Pour chaque analyse de liaison, nous vérifions le respect de l'ensemble des contrats structurels pour l'ensemble des parties de la liaison. Cette analyse est une analyse montante. Nous regardons la compatibilité des contraintes définies sur les opérations puis la compatibilité entre les signatures des opérations puis la conformité sur la signature des types de port. Pour chaque contrat nous définissons trois classes de compatibilité :

- la **compatibilité totale** : dès la conception, nous garantissons le respect des contrats structurels du composant,
- la **compatibilité partielle** : nous informons l'architecte que nous détectons au moment de l'assemblage des risques de non-respect des contrats à l'exécution. Cependant, nous détectons aussi des cas d'utilisation où les contrats sont respectés. Dès lors, seule une analyse à l'exécution permettra de détecter le non-respect des contrats,¹⁰
- la **non compatibilité** : dès la conception, nous détectons que certains éléments d'une liaison ne permettent pas le respect des contrats d'un composant.

SafArchie est un système *ouvert*, dans le sens où nous n'interdisons donc pas la définition de liaisons déclarées partielles, mais nous prévenons l'architecte en lui indiquant les causes de cette compatibilité partielle.

¹⁰La notion de compatibilité partielle est à rapprocher des *warnings* dans un compilateur.

La principale différence entre la compatibilité totale et la compatibilité partielle découle du fait que nous travaillons sur une analyse des types de port indépendante du contexte de fonctionnement. Quand deux types de composant sont déclarés potentiellement compatibles, seul l'ajout de vérifications à l'exécution va permettre de garantir le respect des contrats définis dans le modèle de type.

4.3.1.1 Vérification du contrat d'assertions

Les contraintes ajoutées grâce au contrat d'assertions dans le contexte d'une opération sont définies au niveau des types de composant, mais s'appliquent sur les instances de ces composants. Ainsi, il n'est pas possible de vérifier le respect de ces contrats au niveau du type d'architecture. Par contre, ces assertions définissent des sous-ensembles de paramètres ou d'attributs. Il est possible de vérifier la compatibilité entre ces assertions. En effet, nous pouvons vérifier que les ensembles définis à l'aide des assertions sont comparables.

On considère deux familles d'assertions :

- les *pré* et les *post conditions* renseignent les paramètres d'une opération requise ou fournie et précisent l'état des attributs du composant avant ou après l'interaction. Elles peuvent également définir des dépendances entre des paramètres,
- les *invariants* précisent le type et l'état des attributs d'un composant tout au long du cycle de vie du composant.

Au moment de l'assemblage, les invariants n'apportent pas d'information vérifiable. L'état du composant n'est pas connu. Seules les instances nous permettront de vérifier ces contraintes. Au contraire, les pré et post-conditions définissent à la fois des contraintes sur les attributs et un sous-typage des paramètres des opérations. À partir de cela, les contraintes sur les attributs du composant ne seront pas utilisées au moment de l'assemblage, mais les contraintes sur le sous-typage des paramètres des opérations feront par contre, quant à eux, l'objet d'une validation. Nous pouvons en effet vérifier que les sous-ensembles des données échangées, définis par les pré et les post-conditions d'une opération fournie, sont compatibles avec les sous-ensembles des données échangées d'une opération requise.

En effet, les expressions logiques permettent de définir des sous-ensembles des types de base. Dès lors, nous pouvons comparer les types fournis et requis à l'aide des relations d'ordre ou d'intersection sur l'ensemble des valeurs possibles.

Comparer des ensembles de données consiste à comparer les ensembles de valeurs définies par les déclarations de ces ensembles. Sur un ensemble de types (ensemble de type noté $\mathbb{T}$), nous définissons une relation d'identité ($==$), une relation d'inclusion ($\subseteq$), et une relation d'intersection ($\cap$).

Définition

Ainsi deux ensembles de valeurs $T1$ et $T2$ sont inclus (relation notée $T1 \subseteq T2$) si et seulement si :

$$\forall a \in T1, \exists b \in T2 \mid a = b. \quad (4.1)$$

Définition

Deux ensembles de valeurs $T1$ et $T2$ sont identiques (relation notée $T1 == T2$), si et seulement si :

$$\begin{aligned} T1 &\subseteq T2 \text{ et} \\ T2 &\subseteq T1. \end{aligned} \quad (4.2)$$

Définition

Deux ensembles de valeurs $T1$ et $T2$ possèdent une intersection (relation notée $T1 \cap T2 \neq \emptyset$), si et seulement si :

$$\exists a \in T1 \mid a \in T2 \quad (4.3)$$

À partir de ces règles, l'analyse aboutit à quatre classes de résultat :

- **une compatibilité forte** : les sous-ensembles définis pour les types des paramètres d'une opération requise ou les types de retour d'une opération fournie sont inclus (relation de sous-typage dans les types correspondants des paramètres d'une opération fournie ou le type de retour d'une opération requise),
- **une compatibilité partielle** : il n'existe pas une inclusion entre les types des paramètres d'une opération requise ou les types de retour d'une opération fournie avec les types correspondants des paramètres d'une opération fournie ou le type de retour d'une opération requise. Cependant, il y a toujours une intersection entre les sous-ensembles définis par les contraintes.

Par exemple, soit l'opération fournie suivante :

provide boolean debiter(int value) ; à laquelle on ajoute une assertion sur la valeur du paramètre « value » qui doit toujours être inférieure à 100, ce qui signifie que l'on ne peut débiter plus de cent euros sur le compte. Soit l'opération requise correspondant au terminal de paiement par carte bleue.

require boolean debiter(int value) ; à laquelle on ajoute une assertion pour préciser que la valeur du débit doit toujours être supérieure à 5 euros, le commerçant ne souhaitant pas accepter des montants sur des cartes bleues inférieurs à 5 euros. Dans ce cas, on ne peut garantir la compatibilité forte entre ces deux opérations. On ne peut évaluer, sans une vérification à l'exécution, que l'interaction ne transgressera pas les contrats. Dans ce cas on parle de compatibilité partielle.

- **un deuxième cas de compatibilité partielle** : si une assertion définit un sous-ensemble paramétré par l'état de l'instance d'un composant, l'analyse de cette assertion n'est pas réalisable à partir de notre modèle de type. Nous déclarons alors la liaison de compatibilité partielle et informons l'architecte d'un possible non-respect du contrat à l'exécution,
- **une incompatibilité entre les assertions** : si deux assertions définissent des sous-ensembles ne possédant aucun point d'intersection, la liaison est déclarée non conforme, car, dès l'assemblage, on détecte une faute. Reprenons l'exemple de l'opération débit. Si maintenant l'opération *provide* est contrainte par une assertion lui indiquant que la valeur du paramètre doit toujours être inférieure à 20 alors que l'opération *require* impose une valeur de paramètre supérieure à 30, pour toutes les instances de ce type d'architecture, il y aura un problème lors de l'interaction autour de cette opération.

La vérification des assertions représentant le deuxième niveau de contrat identifie de nombreux assemblages de conformité partielle, car ils décrivent régulièrement des assertions dépendantes du contexte d'exécution et de l'état des composants. Dès lors, les sous-ensembles définis par les assertions ne sont pas évaluable au moment de la conception. Dans ces cas, seule une analyse à l'exécution permet de vérifier le bon fonctionnement d'une interaction. Dans ce cas, la conformité partielle est utilisée pour indiquer qu'il n'est pas possible à partir des seules informations fournies par le modèle de type de garantir le respect du contrat d'assertions.

4.3.1.2 Vérification de l'assemblage au niveau des opérations

Pour un assemblage par les contrats syntaxiques, la relation de compatibilité totale entre signatures d'opération représente le fait qu'il y ait correspondance entre les signatures composées du nom de l'opération, des paramètres, d'un résultat et éventuellement d'exception. Nous établissons alors qu'il y a correspondance entre le quadruplet (nom, paramètre, résultat, exception) d'une opération et le quadruplet (nom, paramètre, résultat, exception) d'une autre opération.

Nous pouvons donner une définition formelle de cette relation de conformité :

Soit $\mathcal{O}$ l'ensemble des opérations,
 soit $\mathcal{N}_o$ l'ensemble des noms d'opérations,
 soit $\mathcal{E}$ l'ensemble des exceptions,



soit $\mathbb{T}_r$ l'ensemble des types de retour possibles,
 soit $\mathbb{T}_p$ l'ensemble des types de paramètres possibles,
 soit $\mathbb{S}$ l'ensemble des polarités des opérations, $\mathbb{S} = \{provide, require\}$.

$\mathbb{O}$ est défini par la relation : $\mathbb{O} = \mathbb{S} * \mathbb{T}_r * \mathbb{N}_o * \mathbb{E}^* * \mathbb{T}_p^*$

Sur $\mathbb{S}$, nous définissons l'opérateur d'inversion $\neg$, tel que $\neg provide = require$ et $\neg \neg s = s$, avec $s \in \mathbb{S}$.
 Sur chaque ensemble nous définissons un opérateur de comparaison noté ($==$) qui retourne vrai si les éléments de l'ensemble sont identiques et faux sinon.

Sur $\mathbb{T}_r$ et $\mathbb{T}_p$, nous définissons une relation d'inclusion (notée $\subseteq$), une relation d'identité (notée $==$) et une relation d'intersection (notée $\cap$). Ces trois relations de typage sont présentées dans le paragraphe précédent (voir 4.3.1.1).

Dès lors nous munissons l'ensemble $\mathbb{O}$ de trois relations : une relation de compatibilité totale entre éléments de $\mathbb{O}$ (notée $:\triangleright$), une relation de compatibilité partielle entre éléments de $\mathbb{O}$ (notée $:\trianglerighteq$), une relation d'incompatibilité (notée $\otimes$).

Soit $(f, g) \in \mathbb{O}^2$

$$f = (s, t_r, n_o, \{e_1, \dots, e_n\}, \{t_{p_1}, \dots, t_{p_m}\}), g = (s', t'_r, n'_o, \{e'_1, \dots, e'_j\}, \{t'_{p_1}, \dots, t'_{p_k}\})$$

Définition

Deux opérations $(f, g) \in \mathbb{O}^2$ sont totalement compatibles (relation notée $f : \triangleright g$), si et seulement si :

$$\begin{aligned} s &= \neg(s') \wedge n_o = n'_o \wedge n = j \wedge m = k \wedge \forall i \in [1..n] \mid e_i = e'_i \\ &\wedge ((s = \{provide\} \Rightarrow t_r \subseteq t'_r, \wedge \forall i \in [1..n], t'_{p_i} \subseteq t_{p_i}) \wedge \\ &\wedge (s' = \{provide\} \Rightarrow t'_r \subseteq t_r \wedge \forall i \in [1..n], t_{p_i} \subseteq t'_{p_i})) \end{aligned} \quad (4.4)$$

C'est-à-dire :

- le nom de l'opération est identique,
- le nombre de paramètres est identique,
- l'ordre des paramètres est identique,
- le type des paramètres est conforme, le type des paramètres d'une opération fournie est un sous-type des paramètres correspondants de l'opération requise,
- les exceptions sont identiques,
- les types des résultats sont conformes, le type des retours d'une opération requise est un sous-type du retour de l'opération requise.

Définition

Deux opérations sont partiellement compatibles (relation notée $f : \trianglerighteq g$), si et seulement si :

$$\begin{aligned} n &= j \wedge m = k \wedge s = \neg(s') \wedge \forall i \in [1..n] \mid e_i = e'_i \\ &\wedge ((s = \{provide\} \Rightarrow t_r \cap t'_r \neq \emptyset \wedge \forall i \in [1..n], t_{p_i} \cap t'_{p_i} \neq \emptyset) \\ &\wedge (s' = \{provide\} \Rightarrow t'_r \cap t_r \neq \emptyset \wedge \forall i \in [1..n], t'_{p_i} \cap t_{p_i} \neq \emptyset)) \end{aligned} \quad (4.5)$$

Nous définissons une relation de compatibilité partielle si le nom de l'opération est différent ou si tous les types échangés possèdent au moins une intersection.

Définition

Enfin, deux opérations sont déclarées incompatibles (relation notée $f \otimes g$), si et seulement si :

$$f \not\supseteq g \quad (4.6)$$

Ce qui correspond à :

$$\begin{aligned} & n \neq j \vee m \neq k \vee s \neq \neg(s') \vee \exists i \in [1..n] \mid e_i \neq e'_i \\ & \vee ((s = \{provide\} \Rightarrow t_r \cap t'_r = \emptyset \vee \exists i \in [1..n], t_{p_i} \cap t'_{p_i} = \emptyset) \\ & \vee (s' = \{provide\} \Rightarrow t'_r \cap t_r \neq \emptyset \vee \exists i \in [1..n], t'_{p_i} \cap t_{p_i} \neq \emptyset)) \end{aligned}$$

Ainsi, toute inversion dans l'ordre des paramètres, toute omission ou ajout de paramètres et d'exceptions engendrera une incompatibilité de la liaison. La définition de l'incompatibilité est équivalente à ce que l'on trouve dans la plupart des modèles à objets [Ducournau *et al.*, 1998, Meyer, 2000].

Exemple de vérification :

provide float add(float x, float y); et **require float add(float x, int y);** sont totalement compatibles, car elles possèdent des signatures compatibles (type compatible, même nom, etc.) et sont de polarité opposée.

Au contraire, **provide float add(float x, float y);** et **require float add(float x, float y, float z);** sont non compatibles, car elles ne possèdent pas le même nombre de paramètres.

4.3.1.3 Vérification de l'assemblage au niveau des types de port

Par défaut, la définition de l'assemblage de deux types de port liés impose à chaque type de port de fournir et de requérir l'ensemble des opérations du type de port auquel il est relié (voir Section 4.2.2.1). Ainsi, pour qu'une liaison entre deux types de port soit déclarée totalement compatible, il faut que l'ensemble des opérations fournies soit requis par l'autre et vice et versa.

Formalisation

Soit $\mathbb{O}$ l'ensemble des opérations, pour chaque élément $o \in \mathbb{O}$, on définit trois relations de compatibilité présentées dans le paragraphe 4.3.1.2 page 99.

$\forall (o_1, o_2) \in \mathbb{O}^2$, on note :

- $o_1 \supset o_2$ deux opérations o_1 et o_2 totalement compatibles,
- $o_1 \supseteq o_2$ deux opérations o_1 et o_2 partiellement compatibles,
- et $o_1 \otimes o_2$ deux opérations o_1 et o_2 incompatibles.

Soit $\mathbb{P}$ l'ensemble des types de port, $\mathbb{P} = \mathcal{P}^{\mathbb{O}}$, $\mathbb{P}$ est l'ensemble des parties de $\mathbb{O}$, c'est-à-dire l'ensemble des ensembles d'opérations. En d'autres termes c'est l'ensemble des types de port que l'on peut construire à partir d'un ensemble $\mathbb{O}$ d'opérations.

Soit $(p_1, p_2) \in \mathbb{P}^2$.

$\forall p = (o_1, \dots, o_n) \in \mathbb{P} \Rightarrow o_i \in \mathbb{O}$.

Par abus de langage, on dit que :

$\forall p \in \mathbb{P}, \Rightarrow p \subset \mathbb{O}$

Nous munissons l'ensemble $\mathbb{P}$ de trois relations : une relation de compatibilité totale notée $\equiv$, une relation de compatibilité partielle notée $\approx$ et une relation d'incompatibilité notée $\not\approx$.

Définition

p_1 et p_2 sont totalement compatibles (noté $\equiv$), si et seulement si :

$$\begin{aligned} \forall o_1 \in p_1 \Rightarrow \exists ! o_2 \in p_2 \mid o_2 : \triangleright o_1 \\ \wedge \forall o_1 \in p_2 \Rightarrow \exists ! o_2 \in p_1 \mid o_2 : \triangleright o_1 \end{aligned} \quad (4.7)$$

Pour chaque opération o_1 de p_1 , il existe une unique opération o_2 de p_2 totalement compatible avec o_1 et pour chaque opération o_1 de p_2 , il existe une unique opération o_2 de p_1 totalement compatible avec o_1 .

Le contrat de port permet d'augmenter la généricité et donc le potentiel de réutilisabilité d'un composant. Dans de nombreux cas, quand nous souhaitons réutiliser un composant, nous le réutilisons partiellement ou différemment de celui qui l'a conçu. Dans le même sens, celui qui le conçoit cherche à offrir un maximum de flexibilité pour l'utilisation de son composant. Ce contrat permet de préciser les opérations minimales qui doivent être fournies et requises pour valider la connexion d'un port. Dès lors, même si l'ensemble des opérations d'un type de port ne trouve pas son vis-à-vis au sein du port lié, si le contrat de port est respecté alors l'assemblage est déclaré partiellement compatible.

Formalisation

Le contrat de port p_1 permet d'identifier le(s) sous-ensemble(s) des opérations de p_1 noté(s) p'_1 devant nécessairement être présents dans l'assemblage :

si $\exists o_1 \in p'_1 \mid \forall o_2 \in p_2, o_2 \otimes o_1$, alors le contrat du port (identifiant l'ensemble p'_1) n'est pas respecté. Ce non-respect du contrat permet de relâcher la contrainte de compatibilité entre deux types de port. En effet par défaut $p'_1 = p_1$, mais le contrat de port permet de définir un sous-ensemble de p_1 , appelé p'_1 tel que si $(p_1) \in P$, le contrat de port p_1 défini $p'_1 \subseteq p_1$.

Ainsi si $\mathbb{O}$ est l'ensemble des opérations.

soit $\mathbb{P}$ l'ensemble des types de port

soit $(p_1, p_2) \in \mathbb{P}^2$

soit $(p'_1, p'_2) \in \mathbb{P}^2$ définis par les contrats de port de p_1 et p_2 tels que $p'_1 \subseteq p_1$ et $p'_2 \subseteq p_2$.

Définition

Les types de port p_1 et p_2 sont déclarés partiellement compatibles (noté $\cong$), si et seulement si :

$$\begin{aligned} \exists o_1 \in p_1 \mid \forall o_2 \in p_2, o_1 \otimes o_2 \\ \vee \exists o_1 \in p_2 \mid \forall o_2 \in p_1, o_2 \otimes o_1 \\ \wedge \forall o_1 \in p'_1 \Rightarrow \exists o_2 \in p_2 \mid o_2 : \triangleright o_1 \\ \wedge \forall o_1 \in p'_2 \Rightarrow \exists o_2 \in p_1 \mid o_2 : \triangleright o_1 \end{aligned} \quad (4.8)$$

Si il existe une opération o_1 du port p_1 tel que pour toutes les opérations o_2 du port p_2 , l'opération o_1 n'est pas totalement compatible avec l'opération o_2 ou si il existe une opération o_1 du port p_2 tel que quelque soit l'opération o_2 du port p_1 , o_1 n'est pas totalement compatible avec o_2 , mais que pour toutes les opérations o_1 identifiées par le contrat de port, il existe une opération en vis-à-vis partiellement compatible avec o_1 , alors p_1 est partiellement compatible avec p_2 .

Définition

Dans le dernier cas, deux types de port p_1 et p_2 de $\mathbb{P}$ sont déclarés incompatibles (noté $\not\cong$), si et seulement si :

$$\begin{aligned} \exists o_1 \in p'_1 \mid \forall o_2 \in p_2, o_1 \otimes o_2 \\ \vee \exists o_1 \in p'_2 \mid \forall o_2 \in p_1, o_1 \otimes o_2 \end{aligned} \quad (4.9)$$

Si une opération identifiée par le contrat de port de p_1 ou p_2 ne possède pas, en vis-à-vis, une opération partiellement conforme, alors p_1 et p_2 sont déclarés incompatibles.

Exemple de vérification :

Si nous étudions la compatibilité entre le port $p13$ et le port $p20$ qui sont liés dans la description de type d'architecture de la figure 4.3, les contrats $p13$ et $p20$ ne sont pas totalement compatibles, car ils ne possèdent pas le même nombre d'opérations. Cependant, si l'on considère les ensembles $p20' = (open, read, close)$ et $p20'' = (open, write, close)$ définis par le contrat du port $p20$ comme les sous-ensembles admissibles pour la liaison, alors on remarque que :

$$\begin{aligned} \forall o_1 \in p'20 \Rightarrow \exists o_2 \in p13 \mid o_2 \supseteq o_1 \\ \wedge \forall o_1 \in p13 \Rightarrow \exists o_2 \in p20 \mid o_2 \supseteq o_1. \end{aligned}$$

Les types des ports $p13$ et $p20$ sont alors déclarés partiellement compatibles.

4.3.1.4 Conclusion sur l'analyse structurelle d'un assemblage

L'analyse structurelle d'un type architecture se fait donc à partir de la description d'un assemblage. Pour un type d'architecture, nous ne parlons plus de compatibilité, mais de cohérence du modèle. En effet, la compatibilité est toujours définie par rapport à un autre élément du modèle de type. La cohérence au contraire est une caractéristique du modèle. La vérification de la cohérence consiste à étudier liaison par liaison pour savoir si les types de port ainsi reliés sont totalement, partiellement, ou non compatibles.

Ainsi, on définit $\mathbb{B}_A$ l'ensemble des liaisons d'un type d'architecture logicielle et $\mathbb{P}_A$ l'ensemble de ses types de port tel que : $\forall b \in \mathbb{B}_A \Rightarrow b = (p_1, p_2), (p_1, p_2) \in \mathbb{P}_A^2$

On définit trois types de propriétés sur A , la **cohérence totale**, la **cohérence partielle** et l'**incohérence**.

Définition

Un type d'architecture A est **totalement cohérent** si et seulement si :

$$\forall b = (p, p') \in \mathbb{B}_A, \mid (p, p') \in \mathbb{P}_A^2, p \equiv p' \quad (4.10)$$

Définition

Un type d'architecture A est **partiellement cohérent** si et seulement si :

$$\begin{aligned} \exists b = (p, p') \in \mathbb{B}_A, (p, p') \in \mathbb{P}_A^2 \mid p \not\equiv p' \\ \wedge \forall b = (p, p') \in \mathbb{B}_A, (p, p') \in \mathbb{P}_A^2 \Rightarrow p \cong p' \end{aligned} \quad (4.11)$$

Définition

Un type d'architecture est **incohérent** si et seulement si :

$$\exists b = (p, p') \in \mathbb{B}_A, (p, p') \in \mathbb{P}_A^2 \mid p \not\cong p' \quad (4.12)$$

Propagation

La propagation des règles de compatibilité présentée dans le Tableau 4.14 permet de déterminer si un type d'architecture logicielle est totalement, partiellement, ou non cohérent.

assertions	opérations	ports	type d'architecture
V	V	V	cohérent
V	V	~	partiellement cohérent
V	~	~	partiellement cohérent
~	~	~	partiellement cohérent
V / ~	V / ~	X	incohérent
V / ~	X	X	incohérent
X	X	X	incohérent

Légende

X = non compatible
~ = partiellement compatible
V = totalement compatible

FIG. 4.14 – Propagation des règles de compatibilité

Cet ensemble de règles de propagation de la conformité permet de valider (ou non) un assemblage. Cependant, cette analyse structurelle n'identifie pas certaines incohérences d'un type d'architecture logicielle telles que celles liées aux incompatibilités entre comportements de types de composant liés au sein de l'assemblage. C'est pourquoi, nous ajoutons en plus des informations structurelles, une description externe du comportement du composant (voir Section 4.2.3). Regardons maintenant quelles sont les propriétés qui peuvent être validées à partir de cette description du comportement externe des composants.

4.3.2 Analyse autour du contrat de comportement

4.3.2.1 Présentation de l'approche

Dans SafArchie, chaque contrat de comportement est représenté sous forme algébrique par une description SFSP. D'un point de vue graphique, le système est représenté par un LTS où les transitions correspondent à des messages entrants ou sortants du composant. Dans SafArchie, les transitions représentent les interactions (entrées ou sorties) du système avec son environnement. On parle alors de système de transitions entrées-sorties ou de IOLTS (*input-output LTS*). Il est alors usuel de parler d'automate à entrées sorties pour désigner un tel système de transitions étiquetées dont l'ensemble des états et celui des transitions sont finis (voir Section 4.2.3.1 page 84). Le comportement d'un composant est alors décrit par un ou plusieurs automates. Les échanges de messages entre composants sont représentés par des synchronisations entre les transitions de ces automates. Le comportement global de chaque composite est modélisé alors par l'automate résultant du produit de synchronisation des automates des composants internes du composite. La composant connexe (ou graphe d'atteignabilité) est l'automate obtenu (à partir d'un automate donné) quand on se restreint aux seuls états atteignables. Seule cette partie de l'automate est vraiment pertinente pour comprendre le comportement global du système.

Une notion de compatibilité comportementale entre services de composants est définie. Elle consiste à vérifier que des automates de comportement des services externes de composants liés au sein d'un composite se composent. L'étude de l'automate résultant du produit de synchronisation permet d'ores et déjà de vérifier la synchronisation des messages émis et reçus par les composants d'un composite. En effet, la composition des automates permet de mettre en lumière si certains enchaînements de messages sont en contradiction et ne peuvent avoir lieu.

Dans notre modèle, chaque message émis ou reçu par un composant est un point de synchronisation ou action partagée entre automates. Toute absence d'un message ou tout ordonnancement incompatible sera alors détecté au moment de l'analyse du comportement.

Comme pour les vérifications effectuées au niveau structurel, SafArchie définit trois niveaux de compatibilité

comportementale entre types de composant liés : la compatibilité totale, la compatibilité partielle et l'incompatibilité. Ces différents niveaux permettent de remonter des informations à l'architecte.

4.3.2.2 Compatibilité totale entre types de composant liés

L'analyse d'un type de composite se fait toujours à partir de l'automate résultant de la composition des différents types de composant qu'il contient.

Prenons un automate A résultant d'une telle composition et comportant un état s . L'état s est dit accessible s'il existe un chemin menant d'un état de départ à s . L'état q appartenant à un automate A est dit co-accessible s'il existe un chemin menant de q à un état final. Un état non co-accessible s'appelle aussi un état-puits.

Définition

La compatibilité totale entre deux automates consiste à vérifier que l'automate résultant de leur composition ne contient pas un état-puits accessible.

Processus de recherche

Pour ce faire, on commence par construire une suite de n ensembles W_n pour trouver tous les états accessibles de l'automate :

- $W_0 = \{s_0\}$
- $W_{n+1} = W_n \cup \{s \in S \mid \exists s_n \in W_n \text{ et } \exists a \in A \text{ tels que } (s_n, a, s) \in \Delta\}$

Autrement dit, on part des états de départ qui sont, par définition, accessibles. Chaque fois que n augmente de 1, on ajoute tous les états liés par une transition aux états qui étaient déjà accessibles. Dès que, pour un n , $W_{n+1} = W_n$, l'ensemble W_n est dit stable, la recherche des états accessibles est terminée.

Ensuite, on cherche les états co-accessibles en appliquant le même principe à partir des éléments finaux F qui sont dans W_n : ($F \cap W_n$). On construit encore une fois une suite de n ensembles Z_n :

- $Z_0 = F \cap W_n$
- $Z_{n+1} = Z_n \cup \{s \in S \mid \exists s_n \in Z_n \text{ et } \exists a \in A \text{ tels que } (s, a, s_n) \in \Delta\}$

L'ensemble des états qui sont accessibles, mais non co-accessibles ($W_n \setminus Z_n$) sont des états-puits de l'automate et permettent de détecter des risques d'interblocage au sein d'un type de composite.

Prenons l'exemple de l'automate présenté dans la figure 4.15 représentant le comportement d'un type de composite. L'analyse de cet automate nous permet de construire l'ensemble $W_0 = \{s_0\}$, l'ensemble $W_n = \{s_0, s_1, s_2, s_3, s_4\}$. $Z_0 = \{s_4\}$, et l'ensemble $Z_n = \{s_0, s_1, s_2, s_4\}$. Comme $W_n \setminus Z_n = \{s_3\}$, il n'y a de compatibilité totale entre les types de composant liés au sein du type de composite dont l'automate est présenté dans la figure 4.15.

4.3.2.3 Compatibilité partielle entre types de composant liés

SafArchie propose une description du comportement de chaque composant avec une abstraction qui introduit un indéterminisme important. Dès lors, de nombreux cas de types d'architecture peuvent conduire à des interblocages. Cependant, ces interblocages n'auront parfois jamais lieu à l'exécution, car ils émanent en effet de branches issues de tests qui ne se révéleront jamais positifs à l'exécution. Dès lors, seule la connaissance des types de composant par l'architecte lui permet de déterminer si l'assemblage contenant de tels interblocages reste valide.

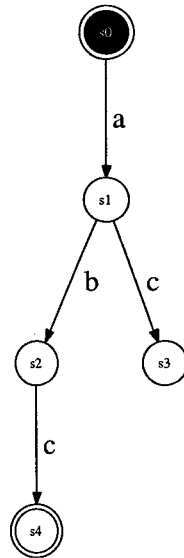


FIG. 4.15 – Exemple d'un automate présentant un interblocage

Définition

Un assemblage est déclaré partiellement compatible au niveau du comportement si et seulement si :

$$W_n \setminus Z_n \neq \emptyset \tag{4.13}$$

et $F \cap W_n \neq \emptyset$

Cette définition correspond à un automate qui contient un ou plusieurs états-puits mais dont au moins un des états finaux est accessible. C'est le cas de l'automate présenté dans la figure 4.15.

4.3.2.4 Incompatibilité entre types de composant liés

Finalement, un assemblage incompatible est un assemblage de types de composant qui engendre nécessairement un interblocage. Aucun état final de l'automate n'est accessible.

Définition

Un assemblage est déclaré incompatible au niveau du comportement si et seulement si :

$$F \cap W_n = \emptyset \tag{4.14}$$

4.3.2.5 Un processus de vérification itératif

La construction de l'automate associé au comportement global du type d'architecture est obtenue de manière itérative dans SafArchie. En effet, seuls les types de composant primitif sont associés à des contrats de comportement et donc des automates. L'automate associé au comportement des types de composite est obtenu par composition des automates de comportement de ses fils. L'automate modélisant le comportement global du système est donc obtenu par compositions successives des spécifications comportementales des types de composant et des types de composite.

La construction de l'automate associé à un composite se fait donc en six étapes :

- identification des acteurs de la composition (les fils du type de composite),
- identification de l'alphabet de synchronisation (union des alphabets des types de port qui sont liés),
- définition du vecteur de synchronisation permettant de synchroniser les messages entrants d'un composant avec les messages sortants. Ce vecteur de synchronisation est paramétré par la relabélisation définie au sein du contrat de comportement du type de composite. Cela permet de synchroniser des messages possédants des noms différents,
- produit de synchronisation des automates à entrées sorties (voir Section 4.2.3.2),
- analyse de cet automate, par recherche d'interblocage et remontée d'information à l'architecte de la trace menant à un interblocage,
- projection vers l'alphabet défini par les types de port délégué (voir Section 4.2.3.2). Cette projection définit la forme factorisée de l'automate à entrées sorties du type de composite.

L'automate du type de composite est alors utilisable dans une composition de granularité plus importante. L'analyse de chaque produit de synchronisation nous garantit l'ordonnement correct des messages au sein de ce type de composite. La sémantique de notre modèle de composant garantit l'encapsulation au sein du composite. Les interactions avec l'extérieur du composite ne seront alors possibles qu'au travers des types de port délégués.

4.3.2.6 Une analyse extensible

Notre contrat de comportement permet, comme nous l'avons vu jusqu'à présent, de décrire le comportement d'un composant du système en décrivant les suites d'échanges de messages pouvant être effectuées par ce composant. Cette forme de description est semblable à la façon de procéder dans un langage de programmation impératif classique : « *le composant fait cela, ensuite il fait cela ou bien cela, etc.* ». On dit donc qu'il s'agit d'une description opérationnelle. Ce qui est nouveau est le fait que les comportements peuvent s'exécuter de façon concurrente, car chaque composant est indépendant et possède un système et un environnement d'exécution propre. Cette relative indépendance entre les comportements des composants de base introduit une difficulté supplémentaire dans l'analyse d'une architecture logicielle.

En effet, une architecture logicielle modélise un système réactif et concurrent qui est généralement composé d'un ensemble de composants qui collaborent entre eux pour réaliser le comportement global du système. La composition des automates permet de vérifier la compatibilité dans l'échange des messages entre composants. Cependant, dans certains cas, bien qu'il puisse être facile de voir que le comportement externe de chacun des composants satisfait une propriété donnée, rien n'assure que cette propriété sera ensuite satisfaite par l'ensemble des processus travaillant de façon concurrente. L'aspect de collaboration introduit donc des difficultés supplémentaires [Garavel, 2003].

Pour les systèmes réactifs et concurrents, un outil formel souvent utilisé pour décrire des propriétés du comportement de tels systèmes est la « logique temporelle ». Contrairement à la logique classique qui ne traite pas explicitement de l'ordonnement des actions, la logique temporelle introduit divers opérateurs, appelés modalités, qui permettent de traiter cet aspect. Par exemple, on peut indiquer qu'une proposition sera toujours vraie, ou encore, qu'elle deviendra vraie dans le futur.

Propriétés d'un système réactif : sûreté et vivacité Il existe deux classes principales de propriétés généralement requises d'un système réactif et concurrent [Pnueli, 1986] [Manna and Pnueli, 1992] :

- propriétés de **sûreté** : une propriété de sûreté (*safety*) vise à indiquer qu'un comportement non désiré est certain de ne pas survenir. En d'autres mots, une propriété de sûreté assure que rien de mauvais ne va arriver (*nothing bad will ever happen*).
- propriétés de **vivacité** : une propriété de vivacité (*liveness*) vise à indiquer qu'un comportement désiré est certain de survenir. En d'autres mots, une propriété de vivacité assure que quelque chose de bon va toujours arriver (*something good will eventually happen*).

Cette classification a un intérêt théorique. Dans [Alpern and Schneider, 1985], Alpern *et al* ont montré que n'importe quelle propriété est la conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Les propriétés de sûreté peuvent, en règle générale, être exprimées à l'aide d'invariants sur l'état du système. Par contre, les propriétés de vivacité impliquent généralement, non pas une quantification universelle sur tous les instants et états possibles du système, mais plutôt une description de contraintes sur des événements futurs : finalement, une certaine propriété deviendra ou pourrait devenir vraie.

SafArchie adresse les problèmes de sûreté au sein d'un assemblage de composants. En effet, l'automate résultant de la composition et l'analyse des traces admissibles permet de comprendre les différents scénarios d'exécution. Cependant, il ne permet pas à l'architecte de vérifier facilement des propriétés globales à un assemblage.

Ces contraintes de sûreté sont exprimées au niveau du composite. Elles sont représentées par un automate contenant la propriété à vérifier. Cet automate est complété à l'aide d'un état *erreur* noté s_{error} . Pour chaque $s_i \in S$, on construit $A'_{s_i} = \{a \in A \mid \exists (s_i, a, s') \in \Delta\}$. Puis pour chaque $a' \in A \setminus A'_{s_i}$, on ajoute (s_i, a', s_{error}) dans Δ . On construit donc un nouvel ensemble $\Delta' = \{(s_i, a', s_{error})\} \cup \Delta$. Pour prouver la propriété de sûreté, on compose ce nouvel automate avec l'automate du type de composite à valider et on vérifie que l'état s_{error} précédemment créé n'est pas accessible.

4.3.2.7 Exemple

Dans notre exemple du *DataPhoneCenter*, on souhaite globalement vérifier que si le client accède à son mail par exemple, il est facturé. Cette propriété résultant de la synchronisation des différents composants n'est pas forcément facile à mettre en évidence sur le composite global *DataPhoneCenter*. L'expression d'une propriété de sûreté va permettre de garantir une telle propriété.

(1) *property PAIEMENT* = ($b6.p7.getvoicexml \rightarrow e4.p3.withdraw \rightarrow PAIEMENT$);

L'automate résultant de cette propriété est présenté dans la figure 4.16.

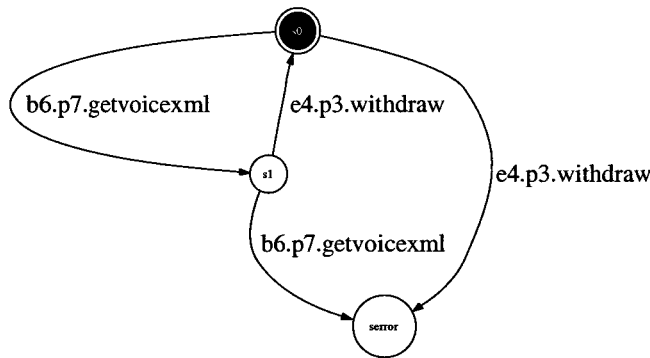


FIG. 4.16 – Automate résultant de la propriété de sûreté *PAIEMENT* (1)

Plus généralement, il est possible d'utiliser des expressions régulières afin de spécifier un ensemble de messages. Ainsi l'expression *property PAIEMENT* = ($*.p7.getvoicexml \rightarrow e4.p3.withdraw \rightarrow PAIEMENT$); permettra de vérifier que pour toute requête demandant l'accès à un contenu VoiceXML du type de composant *MediaCenter* vers un type de composant applicatif ($*.p7.getvoicexml$), le service sera facturé ($e4.p3.withdraw$).

4.3.2.8 Conclusion autour du contrat de comportement

Le langage de comportement associé au modèle de type de *SafArchie* permet de mettre en évidence des problèmes de synchronisation dans l'enchaînement des messages au sein d'un composite. En outre, la vérification de propriété de sûreté permet à l'architecte d'exprimer simplement des propriétés qu'il veut valider sur son assemblage.

L'analyse de comportement permet de mettre en évidence des problèmes de dépendance entre comportements de composants qui seraient impossibles à mettre en évidence à partir uniquement d'une analyse structurelle d'un type d'architecture. Ce problème a été très étudié par les communautés réseau pour prouver la cohérence des protocoles. C'est pourquoi nous proposons de réutiliser les résultats et les outils mis en œuvre au niveau des protocoles dans un contexte d'architecture logicielle. Cependant, comme nous considérons le composant comme une entité de type boîte noire, le comportement exprimé dans les contrats de comportement est non déterministe et indépendant du contexte. De cette opacité liée à notre modèle de composant, il n'est pas possible d'obtenir le comportement exact d'un type d'architecture, une incertitude importante reste liée au contexte d'exécution de l'application.

4.4 Bilan autour du modèle de type

Le modèle de type d'architecture de *SafArchie* se concentre sur les concepts qui nous semblent essentiels dans les modèles à composants au moment de la spécification de type d'architecture : à savoir les éléments de description de l'interface du composant. Le modèle de type propose une description de l'architecture à un niveau d'abstraction important. Le fait de ne pas représenter précisément les instances de composant et leur assemblage au niveau du modèle de type permet de définir un cadre pour l'évolution de l'architecture. La hiérarchie du modèle vise à offrir une abstraction supplémentaire pour exprimer les possibles dépendances entre composants. Un composant n'est pas simplement lié à un autre composant, mais il peut aussi être son fils ou son père. La hiérarchie du modèle se révèle très utile pour deux raisons. Elle offre, pour les différents acteurs participant au développement d'une application, différents niveaux de granularité sur le modèle de l'application. Ceci évite de devoir fixer de manière arbitraire la granularité du composant. En outre, dans une démarche de conception, elle s'intègre bien dans un modèle itératif où l'on construit son système en assemblant des composants de granularité de plus en plus importante.

De par la richesse de la description des types de composant, le modèle de type de *SafArchie* est la base d'outils de vérification utilisés pour détecter certaines incohérences (des composants mal connectés) présentes dans une description d'architecture. Le modèle de type de *SafArchie* précise la sémantique des entités qu'il manipule et il formalise sans équivoque les liaisons entre composants. Enfin, comme le comportement externe des composants est défini, et comme les liaisons font l'objet d'une formalisation non ambiguë, l'analyse de la seule structure d'une application sous sa forme architecturale permet de détecter des risques d'interblocages dus à un mauvais ordonnancement des messages.

Enfin, même si notre modèle assemble des composants de type « *boîte noire* », les trois niveaux de contrat : contrat syntaxique (définition des types de port), contrat sémantique (définition des assertions et définition du contrat de port) et contrat comportemental nous permettent de diminuer progressivement l'opacité des composants que l'on assemble [Büchi and Weck, 1999]. Ces contrats apportent un compromis entre une spécification ouverte et complète des composants, très adaptée pour la vérification, mais inadaptée dans un contexte industriel où l'on cherche à protéger un minimum ses connaissances et une spécification trop pauvre qui ne permet aucune vérification au moment de la conception.

La propriété de substituabilité est aussi consolidée. En effet quand l'ensemble des contrats d'interface est respecté, la substitution d'un type de composant A par un type de composant B ne perturbe pas le fonctionnement de l'application. L'ajout d'information au niveau des interfaces de composant est une approche classique pour permettre une analyse approfondie d'un assemblage [de Alfaro and Henzinger, 2001]. *SafArchie* se distingue

principalement grâce au travail sur la définition du concept de type d'architecture présentée dans cette partie et sur l'analyse associée à ce concept. Cependant, à partir d'un type d'architecture, il n'existe pas de mécanisme simple d'instanciation. En effet, une infinité d'architectures sont conformes à un type. Nous présentons dans la seconde partie l'étape de description d'une architecture logicielle conforme à un type d'architecture logicielle.

Chapitre 5

Construire et déployer des architectures logicielles typées

PRÉAMBULE

Ce chapitre présente deux caractéristiques de la description d'une architecture logicielle dans SafArchie qui viennent compléter la description du modèle de type présentée dans le chapitre précédent. Le premier modèle est le modèle logique. Il permet de décrire une architecture logicielle typée constituée d'un graphe d'instances de composant interconnectées. Le deuxième modèle est le modèle physique. C'est une représentation abstraite de la topologie de l'infrastructure sur laquelle le graphe d'instances de composant est déployé. L'architecture logicielle dans SafArchie correspondra donc au déploiement d'un modèle logique typé par un type d'architecture sur une infrastructure physique.

Sommaire

5.1	Construction d'une architecture logicielle typée	112
5.1.1	Structure d'une architecture logique	112
5.1.2	Une architecture logicielle : instance d'un type d'architecture	114
5.1.3	Un méta-modèle - différentes représentations	114
5.1.4	Modèle logique et contrats d'interfaces de composants	114
5.1.5	Exemple d'architecture logique	115
5.1.6	Bilan sur la notion d'architecture logique	115
5.2	Déploiement d'une architecture logique sur une infrastructure physique	115
5.2.1	Un méta-modèle d'infrastructure physique	116
5.2.2	Exemple de modèle physique d'architecture	117
5.2.3	Placement des composants sur une infrastructure physique	117
5.2.4	Analyse du placement des composants	118
5.2.5	Exemple de description de placement de composants	119
5.2.6	Bilan sur le déploiement dans SafArchie	120
5.3	SafArchie face aux défis de l'architecture logicielle	120

Sur les quatre critères requis pour obtenir un bon langage de description d'architecture à savoir le caractère expressif, vérifiable, exécutable et évolutif (voir Partie II), le modèle de type de SafArchie répond aux trois premiers en offrant une bonne abstraction par rapport aux modèles concrets pris en compte par les plates-formes d'exécution, une description riche des composants permettant l'analyse de l'assemblage, et des mécanismes de génération de code vers différentes plates-formes à composants existantes (voir Chapitre 7 pour ce dernier point). Cependant

concernant la dynamique de l'architecture et son déploiement, aucune information n'est fournie par le modèle de type. C'est pourquoi la description d'une architecture logicielle se fait dans SafArchie avec deux niveaux de description. Le premier est une définition d'un type d'architecture qui caractérise les invariants de l'architecture logicielle. Le second correspond à une définition d'une architecture logique conforme aux invariants définis au sein d'un type d'architecture.

Le lien entre un type d'architecture et une architecture logique n'est pas unitaire. Il est possible de construire une infinité d'architectures logicielles typées à partir d'un type d'architecture donné. D'après notre méta-modèle de type, une architecture logicielle correspond donc à un arbre d'instances de composant connectées au niveau de leurs ports. Mais cette description n'est pas suffisante. Nous considérons dans SafArchie que cette description correspond à un premier niveau d'architecture logicielle que nous nommons : *architecture logique*. La portée d'une architecture logicielle est plus large que ce simple modèle d'instances de composant interconnectées. Une architecture logicielle à base de composants correspond à la représentation à un instant t des instances de ces composants *déployées* sur une *infrastructure physique*. Deux informations sont importantes dans cette définition et vont nous permettre d'enrichir l'architecture logique.

- Une architecture logicielle est une vision statique à un instant t qui doit vérifier l'ensemble des contraintes définies au sein d'un type d'architecture,
- Une architecture logicielle prend en compte la description de l'infrastructure physique sur laquelle est déployé le graphe d'instances de composant interconnectées.

Pour ce faire, nous structurons la démarche de construction d'une architecture logicielle autour de quatre modèles :

- le **type d'architecture** qui définit les types de composant que l'on retrouve au niveau du système et leur possible assemblage.
- l'**architecture logique** qui correspond à un graphe d'instances de composant connectées respectant un type d'architecture précédemment spécifié à l'aide du méta-modèle de type présenté dans la première partie de ce document,
- l'**infrastructure physique** qui spécifie la topologie de l'infrastructure sur laquelle va être déployée notre architecture logique.
- le **déploiement** qui définit la liaison entre le modèle logique et le modèle physique, c'est-à-dire la spécification du placement des composants en fonction des éléments définis au niveau de la topologie de l'infrastructure.

Nous présentons, dans ce chapitre, l'architecture logique et le modèle de définition d'une infrastructure physique.

5.1 Construction d'une architecture logicielle typée

5.1.1 Structure d'une architecture logique

La structure d'une architecture logique est fortement liée au méta-modèle de type d'architecture. Les cinq éléments de base sont le composant primitif, le composite, le port, le port délégué et la connexion. Chacun de ces éléments fait référence à un élément du méta-modèle de type (voir Figure 5.1). Sur cette figure, les éléments grisés sont les éléments de base de la structure du modèle logique. Les éléments représentés en blanc font quant à eux partie du méta-modèle de type d'architecture.

Le composant primitif est typé par un type de composant primitif (*ComponentType*). Cette relation de typage indique que le composant respecte toutes les contraintes définies au niveau du type de composant. Ces contraintes sont de deux ordres. La première indique que le composant possède une liste de ports et de contrats en accord avec la définition du type de composant en termes de signature de ports et de cardinalité. La deuxième précise que si il existe une unique connexion par liaison attachée au type de composant primitif, le composant primitif a le même

contrat de comportement que le type de composant auquel il est associé. Si ce n'est pas le cas, ce contrat devra être précisé afin de prendre en compte les instanciations multiples des types de port.

Le port est typé par un type de port dans le modèle de type (*PortType*). Cette relation de typage indique que le port possède toutes les opérations définies au niveau du type de port. En outre, le port a le même contrat de port défini que celui défini au niveau du type de port.

Une connexion est typée par une liaison déclarée au niveau du modèle de type. La notion de liaison indique une possible interaction entre types de composant. La notion de connexion, quant à elle, définit une interaction entre deux composants. Dès lors, une connexion c entre un port p de type P et un port q de type Q ne pourra avoir lieu que s'il existe une liaison L déclarée entre le type de port P et le type de port Q .

Le composite est typé par un type de composite (*CompositeType*). Cette relation de typage indique que le composite s'engage à respecter trois types de contraintes définis au niveau du type de composite :

- il possède les ports délégués dont le type est celui défini au niveau du type de port admissible pour le type de composite,
- il possède des fils dont le type est celui défini dans le type de composite,
- il possède uniquement des connexions reliant des ports dont les types sont reliés par une liaison (*Bindings*) au niveau du type de composite.

Le port délégué est typé par un type de port délégué. Ce port délégué ne peut alors déléguer son traitement qu'à un port du même type que le type de port pointé par le type de port délégué. En résumé, avec un exemple, si un type de port délégué Q fait référence à un type de port P alors un port délégué q de type Q ne pourra déléguer son traitement qu'à un port p de type P .

Tout élément d'une architecture logicielle est nommé. Ce nom n'a aucun lien a priori avec le nom du type. Il devra être précisé par l'architecte au sein de l'architecture logique.

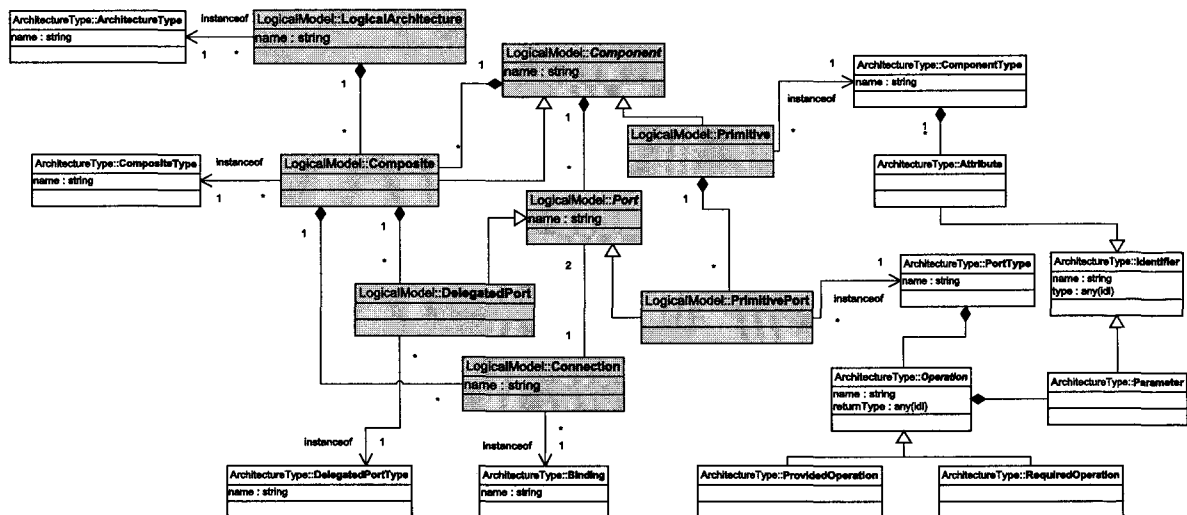


FIG. 5.1 – Modèle logique de SafArchie

5.1.2 Une architecture logicielle : instance d'un type d'architecture

Un modèle logique d'architecture logicielle doit être conforme avec son type. Il doit donc respecter les contraintes fixées par la définition du type d'architecture. Face à la multitude de points de variation potentiels dans le mécanisme d'instanciation entre une architecture et un type d'architecture, nous ne proposons pas de mécanisme automatique pour définir une architecture logicielle à partir d'un type d'architecture. Cependant, la définition d'une architecture logique peut être guidée à partir du moment où l'on a identifié le type d'architecture auquel elle fait référence.

L'analyse de la conformité entre une architecture et son type se fait en quatre étapes :

- en premier lieu, nous vérifions la cohérence du type d'architecture (voir chapitre 4.3),
- dans un deuxième temps, nous vérifions pour chaque composite le respect des contraintes définies dans le type de composite. Ce dernier définit la liste des fils qu'il contient, les cardinalités minimales et maximales admissibles pour chaque type de composant fils, la liste des types de port qu'il peut exporter, et enfin les liaisons qui correspondent aux futures connexions possibles entre instances de composant de l'architecture logicielle. Par défaut, le nombre d'instances de port délégué dans une architecture logique sera rigoureusement le même que le nombre d'instances de port qu'il représente,
- dans un troisième temps, nous vérifions que chaque composant primitif respecte les contraintes définies au niveau du type d'architecture, c'est-à-dire qu'il possède les ports d'un type référencé comme type de port admissible au niveau du type de composant. De plus, le composant primitif doit respecter la cardinalité minimale et maximale de chaque type de port. Cette cardinalité correspond au nombre d'instances minimal et maximal de ce type de port que doit et peut posséder un composant primitif,
- finalement, la dernière étape permet de vérifier des contraintes propres au modèle logique. Le modèle doit être purement hiérarchique. Un composant doit appartenir à un et un seul composite. Toute liaison doit être déclarée entre deux composants différents appartenant au même composite. Tout message du contrat de comportement d'un composant doit faire explicitement référence à une opération appartenant à un port de ce même composant.

Si ces quatre étapes de vérification n'identifient pas d'erreur dans la description de l'architecture, l'architecture logique est conforme au type d'architecture. Le couple *architecture logique* - *type d'architecture* forme alors un premier niveau de description d'architecture logicielle.

5.1.3 Un méta-modèle - différentes représentations

SafArchie propose, comme pour le modèle de type d'architecture, différents formalismes permettant de représenter une description d'architecture logique :

- un langage textuel,
- un langage basé sur XML,
- une description graphique.

Si il existe une bijection possible entre la représentation textuelle et la représentation en XML, la description graphique sert à obtenir une bonne compréhension du schéma d'ensemble. Elle ne permet donc pas de représenter l'ensemble des concepts de l'architecture logicielle.

5.1.4 Modèle logique et contrats d'interfaces de composants

Dans SafArchie, trois contrats permettent de préciser la spécification de types de composant afin d'analyser la cohérence d'un assemblage. Les deux premiers contrats, le contrat de port et le contrat d'assertion permettent de vérifier la compatibilité de deux types de port dont les instances seront connectées. La vérification de la compatibilité n'étant pas dépendante du nombre de ports instanciés de ce type, il n'est pas nécessaire de vérifier de nouveau la compatibilité entre deux ports connectés.

Par contre, le contrat de comportement associé à un type de composant est, quant à lui, fortement dépendant du nombre des messages associés à ce composant. Par conséquent, il est fortement dépendant du nombre de ports instanciés par type de port déclaré au niveau du type de composant. Dans chaque composant primitif, si le nombre de ports instanciés pour chaque type de port est supérieur au nombre de liaisons aboutissant à ce type de port, il est nécessaire de repréciser le contrat de comportement. Dans le cas où il y a égalité entre le nombre de ports instanciés pour chaque type de port et le nombre de liaisons aboutissant à ce type de port, le contrat de comportement du composant peut être obtenu automatiquement à partir du contrat de comportement du type de composant.

Le contrat de comportement d'un composant primitif a la même forme que celui d'un type de composant (voir Section 4.2.3). Seul l'identifiant des messages change. Là où un message est identifié au niveau d'un type de composant par un nom de liaison, un nom de type de port et un nom d'opération, les messages dans un contrat de comportement du modèle logique sont identifiés par un nom de port primitif et un nom d'opération. Comme une unique connexion peut être attachée à un port, il n'est pas nécessaire d'inclure le nom de la connexion au niveau de l'identifiant d'un message.

5.1.5 Exemple d'architecture logique

Pour présenter le modèle d'architecture logique, nous présentons une instanciation du type d'architecture *DataPhoneCenter* en une configuration offrant un pool de connexions de 5 clients simultanés à différents services : mail, accès météo, accès à des scénarios via un protocole d'accès distant à un système de fichier contenant ces scénarios. L'architecture logique possède alors 9 composants primitifs, 57 ports primitifs et 17 ports délégués (voir Figure 5.2). Le composant *aa* a en charge l'authentification et la facturation. Les composants *emailrader1*, *meteoReader* et *smbAccess* font le lien entre des scénarios VoiceXML et des protocoles standards d'accès à certains types de service comme le courrier électronique, la météo sur Internet ou un système de fichier. Enfin, les composants, *mc_x* prennent en charge la traduction entre un contenu « données » décrit sous forme de scénarios VoiceXML et un contenu « audio ».

5.1.6 Bilan sur la notion d'architecture logique

La description de l'architecture logique permet de décrire les instances de composant et leur relation, par exemple la connexion et l'encapsulation. Cette architecture logique est fortement contrainte par la définition du type d'architecture auquel elle fait référence. Ce modèle permet de prendre en compte la dynamique de l'architecture, car elle représente à un instant *t* la structure globale de l'application en termes de composants, de composites et de connexions. À tout instant, ce schéma doit rester conforme à la définition de son type d'architecture.

Parmi les informations importantes à prendre en compte pour comprendre le fonctionnement global de l'application, la répartition et la topologie physique sur laquelle cette application vient s'exécuter sont primordiales. Ces informations permettent de connaître entre autres les capacités de communication entre les composants. Nous proposons dans la prochaine section un premier niveau de topologie physique du système que l'on nomme infrastructure physique sur laquelle l'application est déployée. Cette information est fournie dans SafArchie à partir du modèle physique et du modèle de déploiement.

5.2 Déploiement d'une architecture logique sur une infrastructure physique

La dernière étape de conception dans le modèle d'architecture consiste à spécifier le placement des composants sur une infrastructure physique. La notion d'infrastructure physique permet la prise en compte des informations liées à la répartition. La modélisation de cette infrastructure physique permet la définition du déploiement d'une architecture logique.

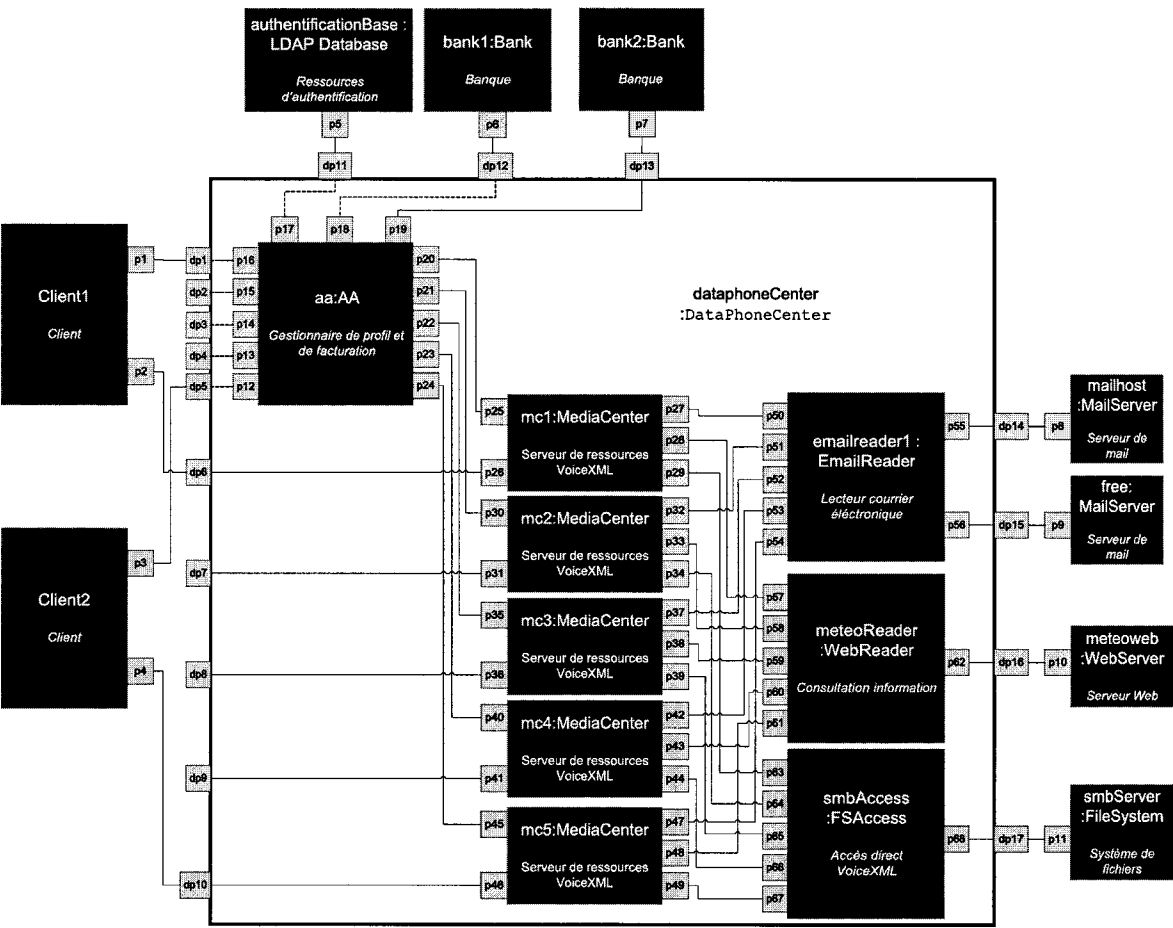


FIG. 5.2 – Modèle logique d’une configuration du DataPhoneCenter

La vision du déploiement dans SafArchie n’est pas associée à une vision « installation de paquets » comme l’on retrouve dans la plupart des distributions Linux [Tharan, 2002] ou dans de nombreuses approches faisant du déploiement [Ketfi, 2004, Dubus, 2005]. Notre modèle de déploiement décrit le résultat pendant l’exécution du graphe d’instances de composant déployées sur une infrastructure physique.

Dans un premier temps, le méta-modèle d’infrastructure physique que nous présentons dans la sous-section suivante cherche à être minimaliste. Nous souhaitons obtenir une vision abstraite de très haut niveau d’un graphe de machines connectées que l’on nomme *sites* dans notre méta-modèle. Dans ce sens, nous ne rentrons pour le moment ni dans des considérations de descriptions précises du contenu et du potentiel de chaque site ni dans une description en détail du fonctionnement des liaisons entre ces sites.

5.2.1 Un méta-modèle d’infrastructure physique

Le méta-modèle d’infrastructure physique est fondé sur cinq concepts (voir Figure 5.3) :

- le concept de **site**. Un site est identifié par son nom. C’est un environnement capable de faire fonctionner différents composants d’une architecture logicielle,
- le concept de **nœud d’interconnexion**. Un nœud est lui aussi identifié par son nom. C’est une entité qui ne peut contenir de composant, mais qui permet de définir des politiques évoluées de communication entre plusieurs sites à l’aide de la définition de *route*,

- le concept d'**interface de communication**. Une interface de communication est attachée à un site ou à un nœud. Elle permet au site ou au nœud de communiquer avec son environnement,
- le concept de **route** permet de définir un chemin de communication entre deux interfaces d'un nœud d'interconnexion. Si aucune route n'est associée à un nœud d'interconnexion, l'ensemble de ses **interfaces de communication** sont connectées. Dès lors, tout message reçu sur une interface de communication est relayé à toutes les autres interfaces de communications du nœud d'interconnexion.
- enfin, le concept de **canaux de communication**. Ces canaux reliant deux interfaces de communication assurent le transport ordonné des messages sans perte d'informations.

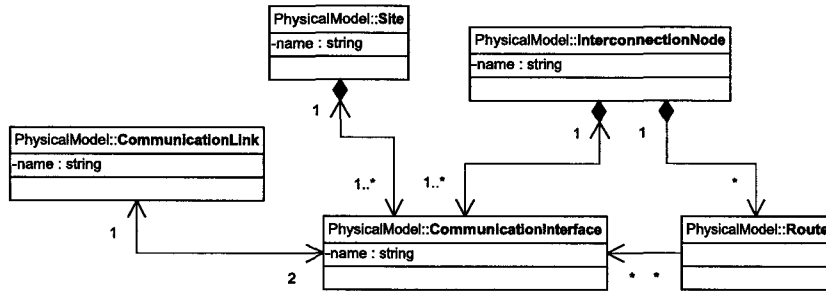


FIG. 5.3 – Modèle physique de SafArchie

5.2.2 Exemple de modèle physique d'architecture

La figure 5.4 présente une description d'infrastructure physique conforme au méta-modèle présenté dans la section 5.2.1. Cette infrastructure physique est composée de sept sites liés par l'intermédiaire de canaux de communication. Ainsi, le site s_1 est directement lié aux sites s_2 , s_3 , s_4 , s_5 et s_6 , eux-mêmes liés au travers d'un nœud d'interconnexion au site s_7 .

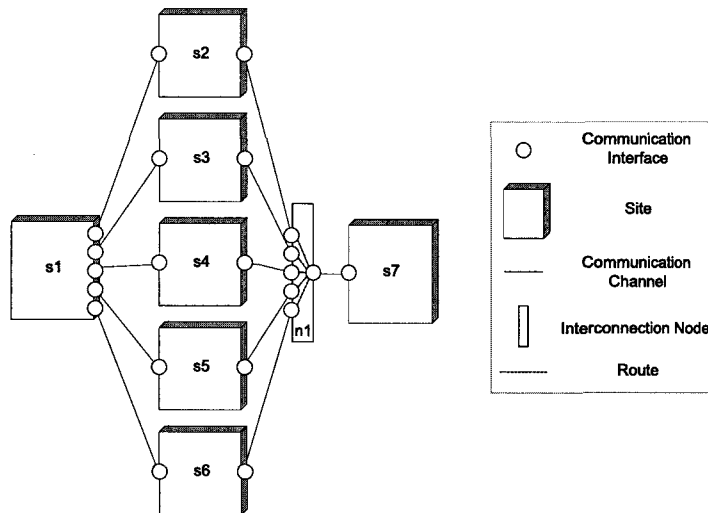


FIG. 5.4 – Exemple d'une infrastructure physique décrite à l'aide de SafArchie

5.2.3 Placement des composants sur une infrastructure physique

Dans SafArchie, la phase de déploiement d'une architecture logique sur une infrastructure physique est pour le moment relativement succincte. Elle consiste à placer chaque instance de composants du modèle logique d'ar-

chitecture logicielle sur un site du modèle physique (voir Figure 5.5). Il n'y a pas pour le moment de contraintes associées au déploiement. Ainsi, on pourra placer plusieurs instances de ces composants sur un site. De même, le déploiement d'un composite consiste à le placer sur un site en particulier. Cependant, ces fils ne doivent pas nécessairement appartenir à ce site. De même, les ports délégués ne doivent pas nécessairement être déployés et associés à ce site. Le fait de déployer un composite sur un site en particulier va permettre de l'administrer au niveau de la gestion de son contenu (gestion des instances de ces fils), de la gestion de l'assemblage, de son cycle de vie ou de son déploiement.

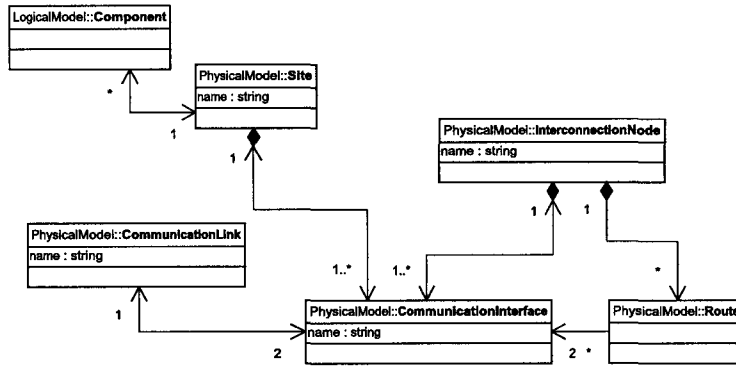


FIG. 5.5 – Modèle de déploiement de SafArchie

5.2.4 Analyse du placement des composants

La principale analyse associée au déploiement consiste à vérifier que l'ensemble des interactions entre les instances de composant logiciel déclarées à l'aide de connexions dans le modèle logique d'architecture, peuvent être prises en charge sur l'infrastructure physique. Pour ce faire, nous définissons les ensembles et les relations suivantes à partir d'une architecture logique et d'un modèle de déploiement.

Soit λ le modèle d'architecture logique et ϕ le modèle d'infrastructure physique,
 soit $\mathbb{P}_\lambda$ l'ensemble des ports $\in \lambda$,
 soit $\mathbb{C}_\lambda$ l'ensemble des composants $\in \lambda$,
 soit $\mathbb{S}_\phi$ l'ensemble des sites $\in \phi$,
 soit $\mathbb{I}_\phi$ l'ensemble des interfaces de communication,
 soit $\mathbb{C}_\phi$ l'ensemble des canaux de communication $\in \phi$,
 soit $\mathbb{N}_\phi$ l'ensemble des nœuds d'interconnexion $\in \phi$,
 soit $\mathbb{R}_\phi$ l'ensemble des routes $\in \phi$.

soit $c : \mathbb{P}_\lambda \mapsto \mathbb{C}_\lambda$ une fonction qui retourne, pour un port donné, le composant auquel il appartient,
 soit $d : \mathbb{C}_\lambda \mapsto \mathbb{S}_\phi$ une fonction qui retourne, pour un composant donné, le site sur lequel il est déployé,
 soit $i : \mathbb{I}_\phi \mapsto \mathbb{S}_\phi$ une fonction qui retourne, pour une interface de communication donnée, le site auquel elle appartient,
 enfin, soit $r : \mathbb{R}_\phi \mapsto \mathbb{N}_\phi$ une fonction qui retourne, pour une route donnée le nœud d'interconnexion auquel elle appartient.

Nous définissons quatre relations pour caractériser la notion de lien de communication *Channel* entre deux interfaces de communications, deux sites, deux composants, ou deux ports (voir équations 5.1, 5.2, 5.3 et 5.4).

Ainsi, il existe une relation *Channel* entre deux nœuds d'interconnexion i_1 et i_2 si et seulement si :

- soit il existe un canal de communication cc_1 entre i_1 et i_2 ,
- soit il existe deux interfaces de communication (i_p, i_q) qui appartiennent à un même nœud d'interconnexion n_1 , il existe une route r_1 au sein de n_1 reliant ces deux interfaces et il existe une relation *Channel* entre les interfaces de communication i_q et i_2 (voir équation 5.1).

$$\begin{aligned}
 & Channel : (i_1, i_2) = True, (i_1, i_2) \in I_\phi^2 \\
 & \Leftrightarrow (\exists cc_1 \in \mathbb{C}c_\phi \mid cc_1 = \{i_1, i_2\}) \\
 & \vee (\exists cc_1 \in \mathbb{C}c_\phi \wedge \exists (i_p, i_q) \in I_\phi^2 \wedge \exists n_1 \in \mathbb{N}i_\phi \wedge \exists r_1 \in \mathbb{R}_\phi \\
 & \mid cc_1 = \{i_1, i_p\} \wedge r_1 = \{i_p, i_q\} \wedge r(r_1) = n_1 \wedge Channel : (i_q, i_2))
 \end{aligned} \tag{5.1}$$

Il existe une relation *Channel* entre deux sites s_1 et s_2 si et seulement si s_1 est égal à s_2 ou si il existe i_1 et i_2 deux interfaces de communication associées respectivement à s_1 et s_2 tel qu'il existe une relation *Channel* entre i_1 et i_2 (voir équation 5.2).

$$\begin{aligned}
 & Channel : (s_1, s_2) = True, (s_1, s_2) \in S_\phi^2 \\
 & \Leftrightarrow (s_1 = s_2) \\
 & \vee (\exists (i_1, i_2) \in I_\phi^2 \mid Channel : (i_1, i_2) \wedge i(i_1) = s_1 \wedge i(i_2) = s_2)
 \end{aligned} \tag{5.2}$$

Il existe une relation *Channel* entre deux composants c_1 et c_2 si et seulement si il existe une relation *Channel* entre les sites s_1 et s_2 sur lesquels sont déployés respectivement c_1 et c_2 (voir équation 5.3).

$$\begin{aligned}
 & Channel : (c_1, c_2) = True, (c_1, c_2) \in \mathbb{C}^2 \wedge (s_1, s_2) \in S_\phi^2 \wedge d(c_1) = s_1 \wedge d(c_2) = s_2 \\
 & \Leftrightarrow Channel : (s_1, s_2) = True
 \end{aligned} \tag{5.3}$$

Enfin, il existe une relation *Channel* entre deux ports p_1 et p_2 si et seulement si il existe une relation *Channel* entre les composants c_1 et c_2 contenant respectivement p_1 et de p_2 (voir équation 5.4).

$$\begin{aligned}
 & Channel : (p_1, p_2) = True, (p_1, p_2) \in \mathbb{P}^2 \wedge (c_1, c_2) \in \mathbb{C}^2 \wedge c(p_1) = c_1 \wedge c(p_2) = c_2 \\
 & \Leftrightarrow Channel : (c_1, c_2) = True
 \end{aligned} \tag{5.4}$$

Définition

Dans SafArchie, un placement d'une architecture logique sur une infrastructure physique est déclaré valide si et seulement si pour toute connexion définie au sein du modèle logique liant les ports p_1 et p_2 , il existe une relation *Channel* entre p_1 et p_2 (voir équation 5.5).

$$\forall cx = (p_1, p_2) \in \lambda \mid (p_1, p_2) \in P^2 \Leftrightarrow Channel(p_1, p_2) = True \tag{5.5}$$

5.2.5 Exemple de description de placement de composants

La figure 5.6 présente le déploiement du composite *dataPhoneCenter* sur une infrastructure physique. Le composite et le composant *aa* sont déployés sur le site s_1 . Chaque instance de *MediaCenter* nécessitant beau-

coup de ressources est déployée sur un site particulier, respectivement s_2 , s_3 , s_4 , s_5 et s_6 . Enfin les composants *emailreader1*, *meteoReader* et *smbAccess* réalisant la création des scénarios XML sont déployés sur le site s_7 .

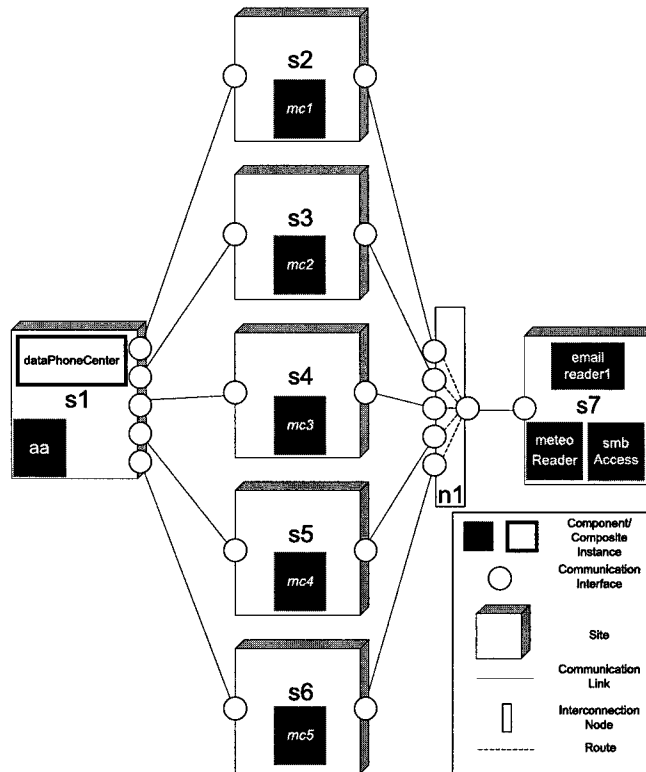


FIG. 5.6 – Déploiement du *dataPhoneCenter* sur une infrastructure physique

5.2.6 Bilan sur le déploiement dans SafArchie

Le modèle de déploiement ne constitue pas une contribution en tant que telle de la thèse. Il fournit néanmoins, les informations relatives au placement des composants sur une infrastructure physique dans la description de l'architecture d'une application. Ces informations nous permettent de vérifier la validité du placement des composants en fonction des capacités de communication nécessaires entre sites et des connexions définies au niveau du modèle logique. Elles nous permettent, en outre, de prendre en compte la facette distribution au moment de la projection vers des plates-formes à composants existantes.

Cependant, cette description du déploiement est pour le moment déclarative ; il est nécessaire d'associer chaque composant d'un modèle logique à un site d'un modèle physique. En outre, le modèle physique est statique. Il n'existe pas, comme au sein du modèle logique et du modèle de type, une description des invariants du modèle physique et de son état à un instant t . Enfin, il n'est pas possible d'associer des informations caractérisant les propriétés des éléments du modèle physique afin de permettre, par exemple, d'ajouter des informations liées à la qualité de service. Ces trois lacunes actuelles liées à la description du déploiement dans SafArchie constituent trois pistes de travail encore à l'étude autour de SafArchie.

5.3 SafArchie face aux défis de l'architecture logicielle

SafArchie propose un cadre pour la construction d'une description d'architecture logicielle. Nous revenons dans cette section sur les défis de l'architecture logicielle énoncés dans la section 2.4 et présentons comment

SafArchie se positionne face à ces différents défis.

L'abstraction

SafArchie fournit un modèle de composant abstrait permettant à l'architecte de manipuler directement les concepts de composant, port, et connexion pour décrire son application ou son système d'information. Nous avons utilisé différents formalismes (textuel, XML ou graphique) manipulant les mêmes concepts pour définir les langages associés à SafArchie. Ces différents langages permettent à l'ensemble des acteurs d'un projet de développement d'avoir une bonne compréhension de la structure de l'application et de partager cette connaissance.

Une description d'architecture SafArchie est structurée autour de trois abstractions : le type d'architecture, l'architecture logique et l'infrastructure physique. Le type d'architecture définit les types de composant admissibles au niveau d'une architecture et leurs interactions. L'architecture logique et l'infrastructure physique sont deux descriptions qui permettent d'obtenir une vision claire et concise de la structure d'une application à l'exécution. L'architecture logique définit le graphe d'instances de composant d'une architecture et leurs interactions. L'infrastructure physique décrit de manière abstraite une infrastructure matérielle. Le déploiement dans SafArchie est le résultat du placement du graphe d'instances de composant sur une infrastructure matérielle.

La réutilisation

SafArchie offre, grâce à une description étendue des interfaces de composant, le moyen d'identifier dès la spécification la cohérence de l'architecture. Les informations fournies à l'aide de ces contrats explicitent des contraintes d'utilisation du composant. La connaissance de ces contraintes facilite la réutilisation des composants au sein de différentes architectures. En effet, il est alors possible de connaître dès la spécification si la réutilisation de ces composants nuit à la cohérence globale de l'architecture. Ainsi, la complémentarité des informations structurelles (spécification des signatures des opérations, contrat de port, contrat d'assertion) et des informations comportementales (contrat de comportement) permet de spécifier les types de composant à partir des trois premiers niveaux de la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999].

L'anticipation du changement

La complémentarité des modèles de type d'architecture et d'architecture logique permet à l'architecte de spécifier deux types d'information sur son application. Le modèle de type définit l'ensemble des invariants de l'architecture. Il identifie les types de composant admissibles au niveau d'une architecture et leurs interactions et offre un moyen de spécifier des architectures au sein desquelles on cadre les évolutions. Le modèle logique donne une vision globale de la configuration d'une application à un instant t . Il identifie les instances de composant réellement présentes au niveau de l'architecture et leurs connexions. Ces deux types d'information permettent de prendre en compte dès la spécification les possibilités d'évolution interne de l'architecture.

Cependant, SafArchie ne fournit pas de mécanisme simple pour prendre en compte les évolutions externes. Ainsi, toute modification d'une description d'architecture logicielle, par exemple pour intégrer une nouvelle préoccupation est effectuée manuellement par l'architecte. En outre, l'analyse de l'impact d'une telle modification ne peut être étudiée localement. Seule une analyse complète de l'architecture logicielle permet de garantir sa cohérence.

La modularité

SafArchie respecte la définition de la modularité grâce sa capacité à structurer un système à l'aide des concepts de composant, de connexion et de composant hiérarchique. Cependant, il propose une seule dimension pour la structuration du système autour des fonctionnalités requises et offertes par un composant. De ce fait, la structuration de l'architecture à l'aide des notions de composant, de connexion et de composant hiérarchique dans SafArchie trouve ses limites pour modulariser certaines préoccupations comme les services techniques.

La construction incrémentale

Enfin, SafArchie ne propose pas non plus de processus de construction incrémentale de descriptions d'architecture logicielle. Entre autres, l'ensemble du travail d'analyse de la cohérence d'une architecture logicielle travaille sur un modèle complètement décrit. L'ajout, a posteriori, de nouvelles informations au sein d'une description d'architecture logicielle est une opération manuelle et nécessite une vérification globale de l'architecture.

Ainsi, pour répondre aux trois derniers défis, l'anticipation du changement, la modularité, et la construction incrémentale qui sont partiellement ou non adressés par SafArchie, nous proposons TranSAT un canevas pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle.

Troisième partie

Enrichir une architecture logicielle à base de composants

Objectifs et présentation générale de TranSAT

SafArchie présentée dans la partie précédente est un langage spécifique de description d'architecture logicielle. Ainsi contrairement à un langage généraliste de modélisation comme UML, l'architecte travaille directement à l'aide des concepts qu'il retrouve au niveau de son métier. L'intérêt d'un tel langage se trouve dans sa concision et sa précision. De plus, la dynamique de l'architecture est prise en compte grâce à la séparation entre la notion de type d'architecture et la notion d'architecture logique. Cependant, il reste difficile pour un architecte de gérer ce que nous appelons les évolutions externes qui correspondent à l'intégration de nouvelles préoccupations au sein d'une architecture logicielle. La prise en compte de ces évolutions externes est motivée par deux caractéristiques principales inhérentes à une description d'architecture : ses évolutions « constantes » pour rester en adéquation avec les besoins des utilisateurs et le besoin d'associer un processus incrémentale afin de maîtriser la complexité de la tâche de construction d'une architecture logicielle .

La séparation en composants et la hiérarchisation offrent un premier niveau pour modulariser une architecture logicielle et offrir un modèle abordable selon une granularité variable pour l'architecte. Il est alors possible, grâce à la hiérarchie, du modèle d'imaginer une construction incrémentale partant de la spécification des composites de granularité la plus importante vers la spécification des composants primitifs. Un tel découpage ne peut à lui seul être à la base d'un bon processus de construction incrémentale d'une architecture. En effet, certaines préoccupations ne peuvent être correctement modularisées à l'aide d'une telle approche et se retrouvent alors noyées au sein d'une description d'architecture logicielle. Ces préoccupations sont alors très difficile à intégrer au cours d'une construction incrémentale de l'architecture. C'est pourquoi nous proposons dans le chapitre suivant TranSAT, un canevas de conception pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle.

TranSAT propose la définition d'un nouveau mécanisme d'assemblage au sein d'une architecture logicielle, le tissage de plan. Il permet de structurer au sein d'un plan les informations relatives à une préoccupation. Il définit alors comment ce plan est intégré à une architecture existante. L'approche par tissage de plans vient compléter les autres mécanismes de modularisation et de composition existants dans SafArchie pour proposer un canevas de conception global de description d'architecture logicielle.

Le chapitre 7 présente la mise en œuvre des concepts mis en avant dans le couple TranSAT/SafArchie au sein d'un atelier de génie logiciel appelé SafArchie Studio. Cet atelier propose une chaîne d'outils pour la construction, la transformation et la vérification d'une architecture logicielle.

Chapitre 6

TranSAT, un canevas de conception pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle

PRÉAMBULE

Ce chapitre présente le canevas de conception TranSAT qui permet d'ajouter une dimension dans la structuration d'une architecture logicielle : la dimension des préoccupations. Cette dimension permet de construire pour chaque préoccupation un plan correspondant à un assemblage de composants dont le but est de réaliser les services liés à cette préoccupation. TranSAT fournit alors une approche par transformation afin de réaliser le tissage de ce plan avec le plan de base.

Sommaire

6.1	Introduction	128
6.1.1	Une approche par transformation	129
6.1.2	Une assistance nécessaire de l'architecte	129
6.1.3	Propriétés attendues du système de transformation d'architecture	129
6.2	Architecture de TranSAT	130
6.2.1	Le plan	131
6.2.2	Le masque de point de jonction	132
6.2.3	Un langage de transformation d'architecture avec une double personnalité	135
6.2.4	Le tisseur	139
6.3	Un processus itératif pour la construction d'architectures logicielles	143
6.3.1	Définition	143
6.3.2	Un processus d'intégration de nouvelles préoccupations	144
6.3.3	Synthèse	148
6.4	Sûreté de fonctionnement	149
6.4.1	Les risques	149
6.4.2	Vérifications statiques sur le patron d'architecture	151
6.4.3	Vérifications dynamiques	157
6.4.4	Conclusion sur l'analyse dynamique	159
6.5	Synthèse	159

6.1 Introduction

De nombreux travaux dans le domaine du génie logiciel ont montré la nécessité d'utiliser des abstractions appropriées pour maîtriser la complexité d'un système informatique [Parnas, 1972, Tarr *et al.*, 1999]. L'introduction des concepts de composant, connexion et composite offre une première dimension de structuration d'une application basée sur les fonctionnalités requises et offertes par un composant. Cependant, cette dimension ne permet pas de séparer clairement les préoccupations fonctionnelles des préoccupations techniques (voir Section 2.4.1). Dès lors, des éléments liés à différentes préoccupations se retrouvent noyés au sein de la spécification d'une architecture. Par exemple, un composant peut posséder de nombreux ports requis et fournis prenant en compte l'ensemble des préoccupations métiers et techniques d'un composant. Ce cas se retrouve généralement au niveau des composants métiers possédant de nombreux ports afin de gérer les interactions avec les autres composants métiers, mais aussi les interactions avec les autres préoccupations techniques comme la persistance, la sécurité ou la transaction. La spécification de l'interface du composant devient alors très difficile à établir en une seule étape.

Disposer d'un deuxième niveau de structuration centré sur les préoccupations se révèle un besoin de plus en plus présent et donc un enjeu crucial pour maîtriser la complexité des systèmes. En effet, dans une démarche de conception guidée par un cahier des charges basé sur un ensemble de cas d'utilisation, il est naturel de se concentrer au préalable sur les préoccupations métiers qui sont clairement explicites au niveau des cas d'utilisation. Dans un second temps, cette spécification est raffinée en intégrant au fur et à mesure les différentes préoccupations techniques. Ces dernières sont elles, le plus souvent, noyées dans les cas d'utilisation. Une approche fournissant une bonne séparation des préoccupations permet de se concentrer sur la complexité des services techniques indépendamment du reste de l'application et de les composer soit à d'autres services techniques, soit à une couche métier. Ainsi, si l'on se place du point de vue général de la construction d'applications, la complexité de ces systèmes est telle qu'il est difficile avec les langages et les outils actuels de construire simplement l'architecture logicielle de ces applications.

Nous proposons avec TranSAT une nouvelle dimension de structuration associée aux préoccupations techniques de l'application dans SafArchie. Cette nouvelle dimension permet de concevoir chaque préoccupation de manière indépendante au sein d'un *plan* correspondant à un assemblage de composants dédié à sa réalisation. Le système final est alors obtenu par tissage de ces différents plans. Comme illustré dans la figure 6.1, le cycle de développement d'une application forme alors un Y.

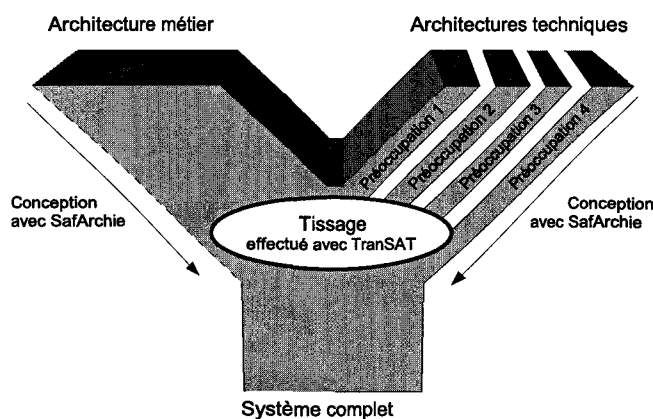


FIG. 6.1 – TranSAT : tissage de plans d'architecture

6.1.1 Une approche par transformation

Nous proposons de décrire les services techniques à l'aide des mêmes entités primaires (composants, connexions et composites) que les services métiers d'une application de manière à pouvoir gérer de manière homogène les applications à tous les niveaux d'abstractions. Le résultat du tissage d'une préoccupation technique au sein d'une architecture logicielle est composé uniquement de ces mêmes entités. Il peut alors être lui-même considéré comme une nouvelle architecture à qui l'on pourra ajouter de nouvelles préoccupations (approche récursive).

Ce choix de l'homogénéité déplace la problématique sur le tissage de deux plans : le plan de base et le plan décrivant la préoccupation à intégrer. Dans TranSAT, nous avons choisi de réaliser ce tissage à l'aide d'une approche par transformation dans laquelle nous spécifions explicitement les modifications à apporter au niveau d'une architecture logicielle initiale (aussi appelée plan de base) afin de permettre l'intégration d'une nouvelle préoccupation. Inspirée par les concepts mis en avant dans la programmation par aspects, l'approche propose une inversion des préoccupations au niveau de la conception des architectures. Nous n'identifions pas les besoins des composants métiers en termes de services techniques, mais ce sont les préoccupations qui viennent transformer l'architecture logicielle. Dans ce sens, le tissage de deux plans est une composition invasive [Abmann, 2003] qui vient transformer la spécification des composants métiers et leur assemblage.

6.1.2 Une assistance nécessaire de l'architecte

Les réflexions autour de TranSAT trouvent leur genèse dans les lacunes actuelles des langages de description d'architectures logicielles. En effet, une problématique importante concerne la description de l'évolution dite externe d'une architecture logicielle et plus particulièrement l'intégration de nouvelles préoccupations. Actuellement, ce travail est, pour l'architecte, souvent manuel rendant l'opération liée à l'intégration d'une nouvelle préoccupation au sein d'une architecture logicielle complexe et risquée, et ceci pour trois raisons :

- sans assistance, cette opération entraîne souvent la mise à jour de nombreuses dépendances. Par exemple, la modification d'une opération sur un port peut entraîner la modification des opérations appartenant à des ports connectés, mais aussi la modification des opérations des types de port identiques,
- l'intégration d'une préoccupation est souvent transverse, ce qui engendre un grand nombre de modifications au sein de l'architecture logicielle originale. Par exemple, pour une préoccupation de sécurité, certaines modifications doivent être apportées à la fois au niveau du client, au niveau du serveur, et au niveau d'un « proxy »,
- enfin et surtout, dans un modèle à composants dans lequel les composants expriment les services fournis et requis, l'intégration d'une nouvelle préoccupation non prévue est problématique. Les éléments de l'architecture ne fournissent pas les ports nécessaires pour l'intégration de cette nouvelle préoccupation.

Dans un souci d'assistance de l'architecte dans cette opération liée à l'intégration de nouvelles préoccupations au sein d'une architecture logicielle, nous proposons un cadre de conception nommé TranSAT associé à SafArchie et permettant l'intégration de ces nouvelles préoccupations.

6.1.3 Propriétés attendues du système de transformation d'architecture

Les propriétés attendues du système sont les suivantes :

- permettre la construction incrémentale d'une architecture logicielle par tissage de plans, chaque plan décrivant l'architecture d'une préoccupation,
- associer à la description de chaque plan, les transformations à effectuer afin de permettre son intégration,
- enfin, décrire ces transformations à partir d'un plan de référence générique afin de créer une entité réutilisable.

La section 6.2 de ce chapitre présente l'architecture de TranSAT centrée autour du concept de *patron d'architecture*. Nous nous focalisons principalement sur deux éléments du patron d'architecture : le *masque de point de jonction* permettant de définir des contraintes sur le lieu d'intégration d'une nouvelle préoccupation et le langage de règles de transformation permettant la spécification de l'intégration de cette nouvelle préoccupation. La troisième section de ce chapitre présente le processus défini autour du canevas de transformation TranSAT identifiant les différents acteurs collaborant autour la définition et de l'intégration d'une nouvelle préoccupation. Enfin, une dernière section présente précisément les vérifications statiques et dynamiques réalisées au moment de la transformation afin de garantir la cohérence de l'architecture résultante de la transformation.

6.2 Architecture de TranSAT

Le canevas de conception TranSAT est construit comme un système de tissage d'aspects au niveau d'une description d'architecture logicielle. Chaque aspect appelé plan dans notre approche prend en charge une préoccupation et vient impacter une architecture logicielle appelée le plan de base afin d'y insérer les informations relatives à une nouvelle préoccupation. Le plan de base n'est pas conscient qu'il va être modifié par une préoccupation : c'est le principe d'*obliviousness* (ignorance) mis en avant par Filman dans [Filman and Friedman, 2000]. Le rôle de l'architecte consiste alors à lier le plan de la préoccupation avec une architecture à modifier en précisant les points de jonction où la préoccupation modifie le plan de base pour pouvoir s'intégrer.

Afin de créer une entité réutilisable au sein de plusieurs architectures, ce canevas de conception est centré autour du concept de *patron d'architecture*. Le patron d'architecture logicielle contient l'ensemble des informations relatives à une préoccupation : le *plan*, le *masque de point de jonction* et les *règles de transformation* :

- le *plan* représente un assemblage de composants mettant en œuvre les services liés à une préoccupation,
- le *masque de point de jonction* décrit un ensemble de contraintes sur le lieu sur lequel le nouveau plan peut être intégré. Le masque de point de jonction a un double rôle dans TranSAT : premièrement, c'est le contrat entre le plan de base qui est modifié et le patron d'architecture. Il précise sous quelles conditions ce plan de base et le plan contenu dans le patron peuvent être tissés. Deuxièmement, ses éléments sont utilisés pour décrire les modifications à apporter pour intégrer une nouvelles architecture,
- enfin, les *règles de transformation* spécifient les modifications à apporter au niveau du plan de base et du plan de ce patron afin de permettre leur tissage créant une architecture logicielle enrichie de la nouvelle préoccupation.

La liaison entre le plan à intégrer et le plan de base est définie en sélectionnant sur le plan de base des points de jonction. De nombreux fragments de ce plan de base respectant le masque de point de jonction est un point de jonction potentiel pour la préoccupation prise en charge par le patron d'architecture. La sélection de ces points de jonction est appelée *la coupe* par analogie avec le vocabulaire que l'on trouve dans le domaine de la programmation par aspects. Dans TranSAT, de par la nature complexe de ces points de jonction qui peuvent être constitués de nombreux éléments de l'architecture logicielle à transformer, nous avons choisi d'appeler ces points de jonction dans la suite de ce document des *lieux d'intégration*. Ce nouveau nom vise à faire la différence entre un point de jonction au sens AspectJ qui représente un point dans le flot d'exécution d'un programme et un lieu d'intégration qui représente un fragment de modèle d'architecture logicielle à partir duquel TranSAT va venir tisser un nouveau plan.

Le résultat d'une transformation est une nouvelle architecture logicielle contenant l'ensemble des éléments du plan de base et du nouveau plan. Cette architecture peut alors servir à nouveau de plan de base pour l'intégration d'une nouvelle architecture. Une nouvelle itération peut alors être effectuée pour intégrer de manière incrémentale d'autres préoccupations.

Une vue générale de l'architecture de TranSAT est présentée dans la figure 6.2. Cette figure illustre le fait qu'un patron d'architecture est composé de trois types d'information, le plan, le masque de point de jonction et les règles de transformation. La coupe sélectionne un ensemble de lieux d'intégration à partir desquels l'architecte souhaite venir tisser le nouveau plan. Le résultat de la transformation peut lui-même être considéré comme un nouveau plan sur lequel l'architecte pourra venir tisser ces nouveaux plans.

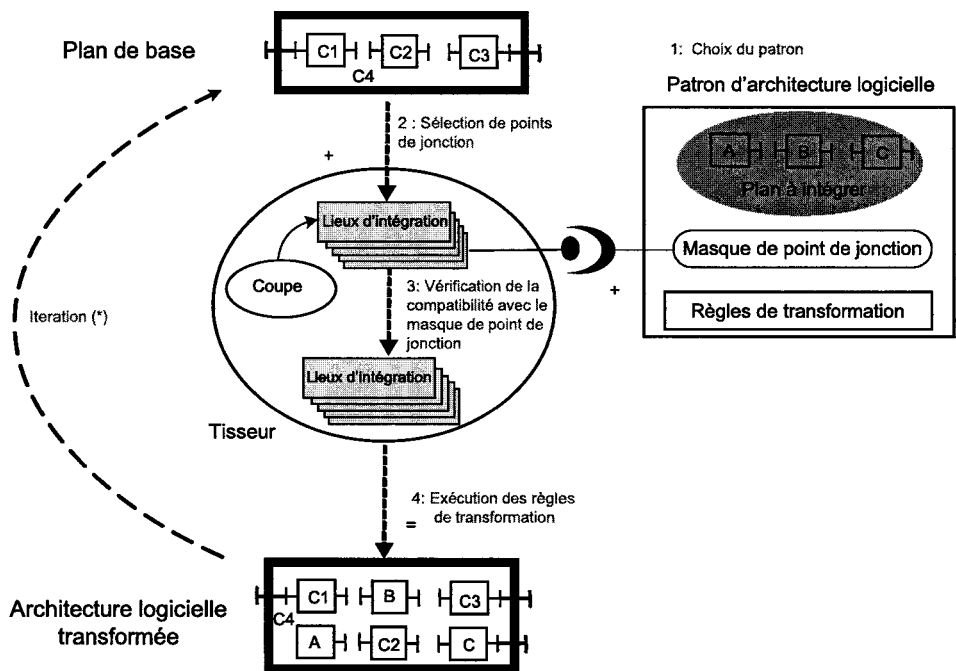


FIG. 6.2 – Vue générale de TranSAT

6.2.1 Le plan

Le premier type d'information que l'on trouve au sein du patron d'architecture est le plan à intégrer. Ce plan est un assemblage de composants construit à l'aide de SafArchie. Cet assemblage a la particularité de ne pas être nécessairement complet. Ainsi, certains ports peuvent être non connectés. En outre, deux composants liés du plan peuvent être intégrés dans des composites différents. Le tisseur préserve la liaison en créant les délégations nécessaires.

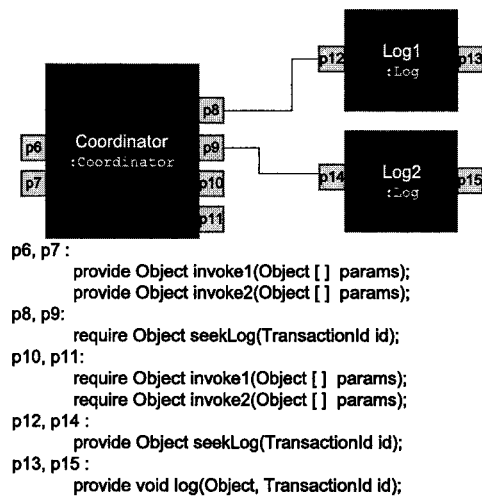


FIG. 6.3 – Plan en charge de la préoccupation d'atomicité

La figure 6.3 présente un plan permettant d'ajouter une propriété d'atomicité lors d'une action mettant en œuvre deux parties. Il est composé de trois composants : un coordinateur *Coordinator* et deux composants de trace *log1*

et *log2*. Les composants de trace possèdent deux ports. Le premier leur permet d'interagir avec le coordinateur, le deuxième leur permet de récupérer des informations sur les actions effectuées par les composants de l'environnement que l'on souhaite rendre atomique. Le coordinateur possède six ports. Deux lui servent à interagir avec les composants de trace faisant partie du plan. Les autres ports lui servent à interagir avec les différents composants du plan de base. La spécification du comportement du coordinateur est présentée dans la figure 6.4 et l'automate résultant de cette spécification est présenté dans la figure 6.5. Ce comportement indique que le coordinateur peut, après avoir propagé l'interaction sur les deux composants, déclencher ou non l'opération inverse sur l'un des deux composants concernés par l'atomicité pour annuler une opération.

```
COORDINATOR = (?p6.invoke1 → ?p7.invoke2 → ?p10.invoke1 → ?p10.invoke1$ → ?p11.invoke2 → ?p11.invoke2$ →
?p6.invoke1$ → ?p7.invoke2$ ⇒ END
| ?p6.invoke1 → ?p7.invoke2 → ?p10.invoke1 → ?p10.invoke1$ → ?p11.invoke2 → ?p11.invoke2$ → ?p6.invoke1$ →
?p7.invoke2$ ⇒ COORDINATEUR1
| ?p6.invoke1 → ?p7.invoke2 → ?p10.invoke1 → ?p10.invoke1$ → ?p11.invoke2 → ?p11.invoke2$ → ?p6.invoke1$ →
?p7.invoke2$ ⇒ COORDINATEUR2

| ?p7.invoke2 → ?p6.invoke1 → ?p11.invoke2 → ?p11.invoke2$ → ?p10.invoke1 → ?p10.invoke1$ → ?p7.invoke2$ →
?p6.invoke1$ ⇒ END
| ?p7.invoke2 → ?p6.invoke1 → ?p11.invoke2 → ?p11.invoke2$ → ?p10.invoke1 → ?p10.invoke1$ → ?p7.invoke2$ →
?p6.invoke1$ ⇒ COORDINATEUR1
| ?p7.invoke2 → ?p6.invoke1 → ?p11.invoke2 → ?p11.invoke2$ → ?p10.invoke1 → ?p10.invoke1$ → ?p7.invoke2$ →
?p6.invoke1$ ⇒ COORDINATEUR2

| ?p6.invoke2 → ?p7.invoke1 → ?p10.invoke2 → ?p10.invoke2$ → ?p11.invoke1 → ?p11.invoke1$ → ?p6.invoke2$ →
?p7.invoke1$ ⇒ END
| ?p6.invoke2 → ?p7.invoke1 → ?p10.invoke2 → ?p10.invoke2$ → ?p11.invoke1 → ?p11.invoke1$ → ?p6.invoke2$ →
?p7.invoke1$ ⇒ COORDINATEUR3
| ?p6.invoke2 → ?p7.invoke1 → ?p10.invoke2 → ?p10.invoke2$ → ?p11.invoke1 → ?p11.invoke1$ → ?p6.invoke2$ →
?p7.invoke1$ ⇒ COORDINATEUR4

| ?p7.invoke1 → ?p6.invoke2 → ?p11.invoke1 → ?p11.invoke1$ → ?p10.invoke2 → ?p10.invoke2$ → ?p7.invoke1$ →
?p6.invoke2$ ⇒ END
| ?p7.invoke1 → ?p6.invoke2 → ?p11.invoke1 → ?p11.invoke1$ → ?p10.invoke2 → ?p10.invoke2$ → ?p7.invoke1$ →
?p6.invoke2$ ⇒ COORDINATEUR3
| ?p7.invoke1 → ?p6.invoke2 → ?p11.invoke1 → ?p11.invoke1$ → ?p10.invoke2 → ?p10.invoke2$ → ?p7.invoke1$ →
?p6.invoke2$ ⇒ COORDINATEUR4);

COORDINATOR1 = (?p8.seekLog → ?p8.seekLog$ → ?p10.invoke2 → ?p10.invoke2$ ⇒ END);
COORDINATOR2 = (?p9.seekLog → ?p9.seekLog$ → ?p11.invoke1 → ?p11.invoke1$ ⇒ END);
COORDINATOR3 = (?p8.seekLog → ?p8.seekLog$ → ?p10.invoke1 → ?p10.invoke1$ ⇒ END);
COORDINATOR4 = (?p9.seekLog → ?p9.seekLog$ → ?p11.invoke2 → ?p11.invoke2$ ⇒ END);
```

FIG. 6.4 – Spécification du comportement du composant Coordinateur

6.2.2 Le masque de point de jonction

Le masque de point de jonction décrit pour un patron d'architecture, les hypothèses faites sur le plan de base pour la spécification de la transformation. Il décrit un ensemble de contraintes sur le contexte architectural dans lequel la préoccupation peut venir s'intégrer. C'est le contrat d'utilisation d'un patron d'architecture. Il définit les attentes du patron d'architecture relatives au(x) lieu(x) d'intégration du plan de base qu'il vient modifier.

Le masque de point de jonction est une notion qui ne se retrouve pas actuellement dans les langages de programmation par aspects comme AspectJ. En effet, dans AspectJ, la coupe et les *advice*s sont définis pour un

entre deux ports quel que soit le nombre de composites traversés par cette connexion. La suppression du caractère hiérarchique du modèle de composant se révèle là aussi intéressante. En effet, le choix de l'encapsulation d'une interaction dans un composite ou non se révèle très souvent un choix de conception spécifique. La prise en compte de ce choix au niveau du masque de point de jonction nuirait beaucoup à la genericité du patron d'architecture en interdisant des points d'accroches sur un plan de base qui pourtant ont du sens en termes d'interaction entre composants.

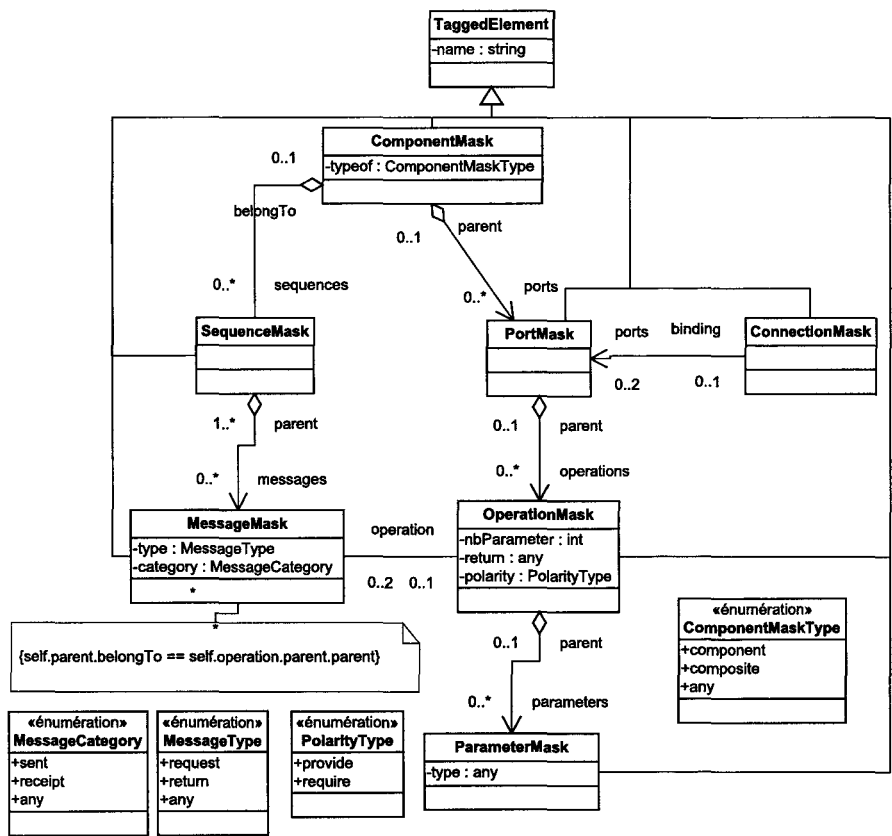


FIG. 6.6 – Le modèle de masque de point de jonction

En programmation par aspects, la notion de point de jonction est liée au comportement du programme : un point de jonction est un point dans le flot d'exécution autour duquel on ajoute du comportement. Ces points de jonction, sont presque uniquement exprimés à partir d'un élément structural du programme. Pourtant, certaines préoccupations dans un système, par exemple transactionnel, s'expriment aussi bien en fonction d'éléments comportementaux que d'éléments structuraux. Du point de vue de l'architecture, comme nous disposons d'informations structurales et comportementales, nous permettons la définition de contraintes sur ces types d'élément. Ainsi, un masque de point de jonction peut définir des contraintes à partir de deux éléments de base de la description du comportement à savoir les enchaînements de messages et les messages. Le *SequenceMask* et le *MessageMask* permettent de définir des séquences linéaires de messages attachées à la définition des masques de composant. Différents attributs permettent de préciser les contraintes définies au niveau d'un *MessageMask*. Il est ainsi possible de définir une contrainte sur le sens d'un message (envoi, réception, indifférent) et sur son type (requête, retour ou indifférent). Comme pour le modèle de composant, un message est forcément associé à une opération. Une contrainte assure que cette association relie un message d'une séquence de messages d'un composant à une opération de ce même composant.

Différents éléments du masque de point de jonction peuvent être indéfinis ou partiellement définis à l'aide du caractère générique *. Ce caractère permet, au niveau de la définition des paramètres, des opérations ou des ports, de substituer le reste de la spécification. Le choix a été fait au niveau du masque de point de jonction d'interdire

toute contrainte portant sur le nom des éléments de l'architecture cible. Ces critères de nommage étant là encore trop spécifiques et liés à des choix de conception de l'architecte.

Finalement, tout élément du masque de point de jonction hérite de l'élément *TaggedElement* et est de fait une *référence* identifiée par son nom. Cette référence sert dans la définition des règles de transformation pour identifier les éléments du masque de point de jonction qui sont utilisés par les règles de transformation.

Exemple de masque de point de jonction

La figure 6.7 présente l'exemple du masque de point de jonction qui permet de définir le contexte d'intégration du plan prenant en charge l'atomicité. Ce masque de point de jonction définit que le lieu d'intégration au niveau du plan de base doit contenir trois composants identifiés respectivement par Cm_1 , Cm_2 et Cm_3 . Ces trois composants doivent être reliés entre eux par l'intermédiaire de deux connexions bm_1 et bm_2 reliant respectivement le port pm_1 de Cm_1 avec le port pm_3 de Cm_2 et le port pm_2 de Cm_1 avec le port pm_4 de Cm_3 . Chaque port doit contenir au moins deux opérations de même polarité, *provide* pour les opérations de pm_3 et pm_4 et *require* pour les opérations de pm_1 et pm_2 . Une dernière contrainte indique que les opérations (opm_1, opm_5), (opm_2, opm_6), (opm_3, opm_7) et (opm_4, opm_8) doivent avoir la même signature mais être de polarité inverse.

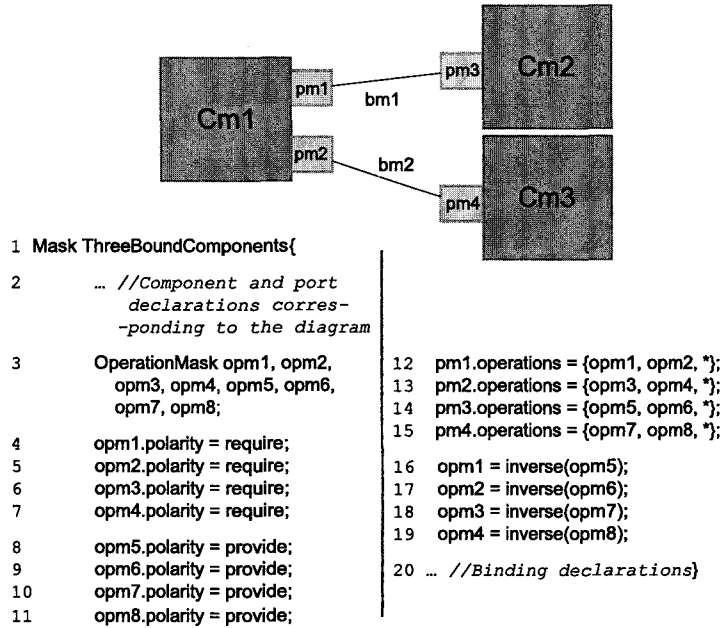


FIG. 6.7 – Exemple de masque de point de jonction pour le plan prenant en charge l'atomicité

6.2.3 Un langage de transformation d'architecture avec une double personnalité

La dernière brique constitutive d'un patron d'architecture logicielle concerne les règles de transformation. Ces règles spécifient à intégrer les opérations à effectuer afin d'intégrer le plan au sein d'une architecture logicielle existante. Elles sont décrites uniquement à partir des éléments du masque de point de jonction et du plan afin de conserver la propriété d'indépendance du patron par rapport au contexte d'intégration. Au niveau des architectures logicielles, la plupart des approches travaillent sur la reconfiguration permettant l'adaptation d'une architecture à son contexte d'exécution ou l'intégration de nouvelles fonctionnalités. Dans ces travaux, des langages dédiés de reconfiguration sont apparus comme Fscript [David, 2005] permettant des adaptations structurelles. Cette focalisation sur les mécanismes de reconfiguration est principalement liée au fait que les plates-formes d'exécution offrent

ce seul degré de liberté pour modifier une architecture. Cependant, les apports récents issus du rapprochement entre composants et aspects offrent au développeur des mécanismes de transformation de ces composants bien plus intrusifs permettant la modification locale de l'interface d'un composant. Ces transformations modifient alors aussi bien la structure du composant que son comportement. TranSAT vise à permettre une abstraction à ces deux types de transformation d'une architecture logicielle : la définition des reconfigurations et la définition de l'introduction.

D'une manière générale, la reconfiguration considère les composants comme une boîte noire indivisible. Les modifications apportées à l'architecture sont donc des modifications du graphe de composants (ajout, suppression de composants ou de connexions entre composants). La reconfiguration ne casse pas l'encapsulation du composant, elle ne touche ni à son interface, ni à son fonctionnement interne.

Au contraire, l'introduction considère qu'il est possible de faire évoluer le composant. L'introduction propose donc des modifications intra-composant. Dans une description d'architecture où le composant est défini par l'intermédiaire de son interface contenant des informations structurelles et comportementales, l'introduction propose des mécanismes pour modifier cette interface en cherchant à maintenir la cohérence entre les informations structurelles et les informations comportementales.

6.2.3.1 Vers un langage dédié à la spécification des transformations d'une architecture logicielle

Des solutions existent pour transformer une architecture logicielle par exemple à l'aide d'APIs de navigation sur le modèle (générées à partir du méta-modèle), en utilisant des feuilles de styles XSLT [W3C, 2005] venant modifier la représentation XML d'une architecture logicielle [Tran *et al.*, 2005] ou en utilisant des langages de transformation de modèles générale comme ATL [Jouault and Kurtev, 2005] ou *Simple TRL* [Sriplakich, 2003]. Ces approches offrent l'avantage pour l'architecte de disposer d'une liberté complète sur les modifications qu'il souhaite apporter, mais possèdent en contrepartie trois inconvénients majeurs :

- la définition d'une transformation complexe en utilisant des feuilles de style XSLT, ou en travaillant à partir d'un langage généraliste et de l'API de navigation au sein du modèle de composant se révèle souvent très verbeuse. Par conséquent, même des opérations conceptuellement simples nécessitent une quantité de code disproportionnée,
- l'utilisation directe des APIs de navigation pour programmer ces transformations, par exemple en Java, ou l'utilisation d'un langage de transformation généraliste comme XSLT rend difficile toute analyse de haut niveau de l'impact de ces transformations sur la cohérence de l'architecture résultante,
- enfin, et cela rejoint en partie le premier point, les concepts mis en avant dans SafArchie comme le composant, le port, les opérations ne sont pas des primitives que l'on trouve dans XSLT ou dans un langage de programmation généraliste. Ce manque d'adéquation entre les concepts présents dans le modèle transformé et les concepts présents au niveau du langage de transformation engendre une surcharge de code pour l'architecte. Le manque de spécialisation du langage de transformation engendre un problème de pouvoir d'expression. L'architecte souhaitant utiliser ce langage se retrouve alors avec des constructions inadéquates à son domaine d'expertise.

Ces trois problèmes - notation verbeuse, vérifications très difficiles, et adéquation insuffisante entre le langage et le domaine visé par ce langage - correspondent précisément aux arguments en faveur de la construction d'un nouveau langage dédié à la description des règles de transformation.

C'est pourquoi nous utilisons pour définir les règles de transformation un langage dédié permettant de spécifier les modifications apportées au niveau du plan de base lors du tissage d'un nouveau plan. Ces transformations sont spécifiées uniquement à partir des éléments du masque de point de jonction et des éléments du nouveau plan. Nous présentons dans cette section le cahier des charges de ce langage, le modèle et la syntaxe concrète que nous avons mis en œuvre afin de permettre la spécification des règles de transformation.

6.2.3.2 Le langage de règles de transformation

Les règles de transformation spécifient comment tisser un nouveau plan et un plan de base en fonction d'un lieu d'intégration respectant les contraintes définies au niveau du masque de point de jonction. Ces règles définissent les transformations à apporter au plan de base et au nouveau plan ainsi que leur ordre d'application.

Les primitives d'introduction

Le tableau 6.1 présente les primitives utilisées afin de changer localement la structure d'un composant. Ces primitives permettent de modifier l'interface d'un composant en lui ajoutant de nouveaux ports et de nouvelles opérations, mais aussi en supprimant des ports ou en déplaçant une opération au sein d'un autre port.

La primitive *Port Pr in Cp* permet de définir un nouveau port *Pr* au niveau d'un composant *Cp*. Ce nouveau port ne possède pas d'opérations.

La primitive *Pr.destroy()* permet de supprimer un port. Seuls les ports définis au niveau du masque de point de jonction peuvent être détruits. Ces ports ne sont pas nécessairement complètement spécifiés. La primitive *destroy* appliquée à un port, ne le détruit que si celui-ci ne contient plus aucune opération. Dans ce cas, toute connexion associée à ce port est détruite.

	Port	Operation
create	Port <i>Pr</i> in <i>Cp</i>	Operation <i>Or</i> = <i>op</i> in <i>Pr</i> Operation <i>Or</i> ₁ = <i>op</i> replaces <i>Or</i> ₂
destroy	<i>Pr.destroy()</i>	N/A
move	N/A	<i>Or.move(Pr)</i>

Cp : ComponentRef; *Pr* : PortRef;
Or : OperationRef; *op* := *Or* | *inverse(Or)* N/A : Not applicable;

TAB. 6.1 – Primitives d'introduction

La primitive d'ajout d'une opération *Operation Or = op in Pr* ajoute dans *Pr* une nouvelle opération *Or* possédant la même signature que *op*. Ce type d'ajout d'opération doit nécessairement engendrer une mise à jour du comportement afin de prendre en compte les nouveaux échanges de messages associés à *Or*. Une règle de transformation modifiant le comportement du composant contenant *Pr* comme celles présentées dans la figure 6.8 doit donc être définie.

L'ajout d'une nouvelle opération au sein d'un composant est liée à la modification du contrat de comportement de ce composant. Cette modification est spécifiée en utilisant une approche à base de reconnaissance de forme sur le contrat de comportement *patron* $\Rightarrow$ *résultat*. Une telle règle permet d'insérer le message associé avec une nouvelle opération, *op*, avant, après ou autour de l'appel ou de la réponse d'une opération *m*. Le patron spécifie la séquence de messages de *m* qui peut être séparée ou non par une séquence de messages notée *x*. Le résultat spécifie comment les messages associés à l'opération *op* sont insérés par rapport aux messages de *m*. La partie gauche de la règle peut aussi faire référence à des *MessageMask* définis au sein du masque de point de jonction. La figure 6.8 montre quelques exemples de telles règles. La ligne 1 par exemple spécifie la réception d'un appel à l'opération *?m* suivi par l'envoi de la réponse à cette requête *!m\$*. Le résultat de l'intégration spécifie que juste après la réception de la requête sur *?m*, le composant envoie une requête à *!op* avant d'effectuer le traitement de *m* (*x* $\rightarrow$ *!m\$*). C'est le cas classique d'une interception avant l'exécution d'une méthode dans une approche objet. Le résultat peut être décrit en utilisant les différents opérateurs présentés dans la section 4.2.3.2. Ainsi la deuxième ligne de la figure 6.8 spécifie un résultat où l'opération *op* n'est pas systématiquement appelée en cas de réception de requête sur l'opération *m* (*!op* $\rightarrow$ *?op\$* $\rightarrow$ *x* | *x*).

Les primitives *Operation Or*₁ = *op* replaces *Or*₂ et *Or.move(Pr)* ajoutent aussi une opération à un port. Dans ces cas-là, cependant, la mise à jour du contrat de comportement est automatique. Dans le cas du

$$\begin{aligned} ?m \rightarrow x \rightarrow !m\$ &\Rightarrow ?m \rightarrow !op \rightarrow ?op\$ \rightarrow x \rightarrow !m\$; \\ ?m \rightarrow x \rightarrow !m\$ &\Rightarrow ?m \rightarrow (!op \rightarrow ?op\$ \rightarrow x|x) \rightarrow !m\$; \end{aligned}$$

 1
2

FIG. 6.8 – Transformation du comportement

replaces, chaque message associé à Or_2 dans le contrat de comportement est remplacé par le message correspondant au niveau de Or_1 . Les opérations doivent avoir la même polarité afin d'assurer que ce remplacement soit valide au niveau de l'automate de comportement.

Dans le cas de la primitive `move`, TranSAT restreint l'utilisation de la primitive à un déplacement d'une opération entre deux ports du même composant. Ainsi, les messages associés à cette opération sont déjà présents au niveau du contrat de comportement. Comme le nom des messages dans le contrat de comportement contient à la fois le nom de l'opération et le nom du port, l'utilisation de la primitive `move` met à jour le contrat de comportement en associant le nouveau nom de port aux messages concernés. Un déplacement d'une opération au sein du masque de point de jonction ne sera possible que si le port de destination et le port de départ appartiennent au même composant. Cette garantie n'est possible que si le *componentMask* à qui appartiennent ces deux ports possède une contrainte précisant qu'il doit nécessairement être un composant primitif.

Les primitives de reconfiguration

Les primitives de reconfiguration prennent en charge, quant à elles, les modifications des interactions entre composants lors de l'intégration d'une nouvelle préoccupation. Ces modifications concernent la création et la destruction de connexions, la création et la destruction de composites et l'ajout et le retrait de composants au sein d'un composite. Le tableau 6.2 présente les différentes primitives de reconfiguration que l'on peut utiliser au niveau des règles de transformation.

	Component	Connexion	Composite
create	N/A	Connexion $Br = \{Pr_1, Pr_2\}$	Composite Cr Composite Cr_1 in Cr_2
destroy	N/A	$Br.destroy()$	$Cr.destroy()$
move	$Cp.move(Cr)$	N/A	$Cr_1.move(Cr_2)$

Cp : ComponentRef ; Cr : CompositeRef ; Pr : PortRef ;
 Br : ConnexionRef ; N/A : Not applicable ;

TAB. 6.2 – Primitives de reconfiguration

La création des connexions est toujours faite de manière relative dans les règles de transformation. La primitive Connexion $Br = \{Pr_1, Pr_2\}$ connecte un port Pr_1 et un port Pr_2 même si ceux-ci n'appartiennent pas à des composants d'un même composite. Le tisseur se charge de créer les ports délégués et les délégations pour que cette interaction puisse avoir lieu.

Exemple des spécifications de règles de transformation pour l'atomicité

Les règles associées à l'introduction de la préoccupation d'atomicité sont présentées dans la figure 6.9.

Pour intégrer l'atomicité, il est nécessaire d'ajouter un port p_1 au composant Cm_1 afin de créer un port qui ne contient que les opérations opm_1 et opm_2 . On réalise la même action pour les opérations opm_3 et opm_4 du port pm_2 que l'on place dans le port p_2 . Ainsi si les ports pm_1 ou pm_2 contiennent plus de deux opérations, les autres opérations restent dans leur port d'origine. Les primitives $pm_1.destroy()$ et $pm_2.destroy()$ spécifient la

destruction des ports pm_1 et pm_2 . Cette suppression n'est autorisée que si les ports sont vides. Dans ce cas les connexions attachées à ces ports sont aussi détruites.

L'action de placer les opérations opm_5 , opm_6 et opm_7 , opm_8 dans de nouveaux ports p_4 et p_6 des composants Cm_2 et Cm_3 est effectuée pour les mêmes raisons que précédemment, c'est-à-dire isoler les opérations capturées par le masque de point de jonction dans des ports séparés afin de pouvoir les lier à d'autres composants. La destruction des ports pm_3 et pm_4 n'a lieu que si les ports ne contiennent plus aucune opération.

Operation $o_{6a} = inverse(opm_1)$ replaces $p_6.invoke1$ et les trois autres primitives similaires modifient l'interface du composant coordinateur pour l'adapter au contexte d'intégration.

Enfin, la dernière primitive d'introduction modifie l'interface des composants Cm_2 et Cm_3 afin de pouvoir interagir avec les composants $log1$ et $log2$. Ainsi, Port p_3 in Cm_2 ajoute un port à Cm_2 . Operation $o_3 = inverse(p_{13}.log)$ in p_3 introduit une opération dont la signature est inverse de la signature de $p_{13}.log$. Afin de garder la cohérence entre les opérations structurelles et les informations comportementales du composant Cm_2 , cette introduction nécessite une mise à jour explicite du comportement du composant Cm_2 . Cette mise à jour est effectuée à l'aide des primitives des lignes 31 à 35.

Les primitives de reconfiguration placent les composants du plan du patron au sein du plan de base. Finalement, la dernière étape consiste à créer les nouvelles liaisons. Si deux ports liés n'appartiennent pas au même composite, comme par exemple la liaison entre les ports *Coordinator.p10* et *Cm2.p4*, les délégations sont créées automatiquement au cours de la transformation. Un exemple d'intégration de ce patron à l'aide de ces règles de transformation est présenté dans la figure 6.11.

6.2.3.3 Synthèse

La capacité du patron à être réutilisé pour différents plans de base est une caractéristique très importante. Regrouper le plan, le masque de point de jonction et les règles de transformation au sein d'une même entité devient intéressant à partir du moment où il est possible de réutiliser ces patrons au sein de différentes architectures logicielles. Il est alors possible de constituer des entrepôts de patrons. La genericité du patron est principalement mise en œuvre grâce au masque de point de jonction qui définit un contrat entre les attentes du patron d'architecture et le lieu d'intégration de l'architecture logicielle.

Le choix au niveau du patron d'architecture, c'est-à-dire du plan, du masque de point de jonction et des règles de transformation de ne pas s'occuper de la hiérarchie du modèle de composant, laissant au tisseur la charge de créer les bons ports délégués et les bonnes délégations permet de ne pas empêcher l'intégration d'un plan d'architecture pour des raisons liées à l'encapsulation des composants au sein des composites.

La spécialisation du langage, basé uniquement sur douze primitives, contraint la manière d'écrire les règles de transformation d'un patron d'architecture. D'un autre côté, la bonne connaissance du fonctionnement de chacune de ces primitives et l'absence de boucle ou de primitives de choix dans le langage permettent une première analyse statique de l'impact de l'intégration de la préoccupation prise en charge par le patron (voir Section 6.4.2).

6.2.4 Le tisseur

Le patron d'architecture constitue l'élément générique de TranSAT. Le tisseur au contraire est dépendant du contexte d'intégration. Il est associé à un couple (patron d'architecture, plan de base). Il permet d'intégrer un nouveau plan d'architecture sur un plan de base. Le tisseur est ainsi le centre du Y présenté dans la figure 6.1.

Contrairement à une description d'architecture logicielle ou à un patron d'architecture, le tisseur ne contient pas d'information. Il est, d'un point de vue abstrait, un opérateur de transformation qui prend en entrée trois types d'information : l'architecture logicielle à transformer, une spécification des différents lieux d'intégration sur cette architecture et le patron d'architecture. À partir de ces informations, il produit une nouvelle description

```

// Cm1 transformation
Port p1 in Cm1;
opm1.move(p1);
opm2.move(p1);
... Similarly for the port p2 and the operation masks opm3 and opm4 of pm2

// Cm2 transformation
Port p4 in Cm2;
opm5.move(p20);
opm6.move(p20);
... Similarly for the port p6 in Cm3 and the operation masks opm7 and opm8 of pm4

// Port destruction
pm1.destroy();
pm3.destroy();
... Similarly for the ports pm2 and pm4

// Coordinator transformation
Operation o6a = inverse(opm1) replaces Coordinator.p6.invoke1;
Operation o6b = inverse(opm2) replaces Coordinator.p6.invoke2;
... Similarly for the operations of port Coordinator.p7

Operation o10a= inverse(opm5) replaces Coordinator.p10.invoke1;
Operation o10b= inverse(opm6) replaces Coordinator.p10.invoke2;
... Similarly for the operations of the port Coordinator.p11

// Introduction of p3 within Cm2
Port p3 in Cm2;
Operation o3 = inverse(log1.p13.log) in p3;
?opm5 → x → !opm5$
    ⇒ ?opm5 → x → !o3 → ?o3$ → !opm5$;
?opm6 → x → !opm6$
    ⇒ ?opm6 → x → !o3 → ?o3$ → !opm6$;

// Cm3 Transformation
... Similarly to Cm2 for the transformation of port p5

// Component introduction
Coordinator.move(Cm1.parent);
log1.move(Cm2.parent);
log2.move(Cm3.parent);

Binding b6 = {Cm1.p1, Coordinator.p6};
Binding b7 = {Cm1.p2, Coordinator.p7};

Binding b10 = {Coordinator.p10, Cm2.p4};
Binding b11 = {Coordinator.p11, Cm3.p6};

Binding b13 = {log1.p13, Cm2.p3};
Binding b15 = {log2.p15, Cm3.p5};

```

FIG. 6.9 – Règles de transformation pour la préoccupation d'atomicité

d'architecture logicielle. D'un point de vue concret, il représente à la fois le vérificateur de modèle et le moteur de transformation. Le vérificateur de modèle est capable de s'assurer que les lieux d'intégration fournis en entrée respectent le masque de points de jonction défini au sein du patron d'architecture. Le moteur de transformation interprète, quant à lui, les règles de transformation.

6.2.4.1 Définir un lieu d'intégration

La définition du lieu d'intégration permet à l'architecte d'indiquer précisément où il souhaite intégrer un nouveau plan. Cette tâche spécialise le masque de point de jonction pour un contexte particulier. Le lieu d'intégration est défini par l'architecte, c'est le principal travail qu'il doit fournir afin de réaliser le tissage entre un nouveau plan à intégrer et une architecture logicielle existante, le tisseur se chargeant de la transformation.

La définition d'un lieu d'intégration est très peu assistée. L'architecte associe à chaque élément du masque de point de jonction un élément dans un plan de base. Ce manque d'assistance se justifie là aussi par le manque

d'information sémantique contenue dans une description d'architecture logicielle en générale. Il semble en effet, difficile à l'aide des informations contenues dans un plan de base de définir que telle ou telle préoccupation doit être intégrée sur tel ou tel lieu d'intégration.

Exemple d'intégration de l'atomicité au niveau de l'architecture du *DataPhoneCenter*

En reprenant une partie du modèle logique décrivant l'interaction entre la banque et le composant d'authentification, nous souhaitons maintenant ajouter un mécanisme d'atomicité pour les transactions entre les deux banques (voir Figure 6.10).

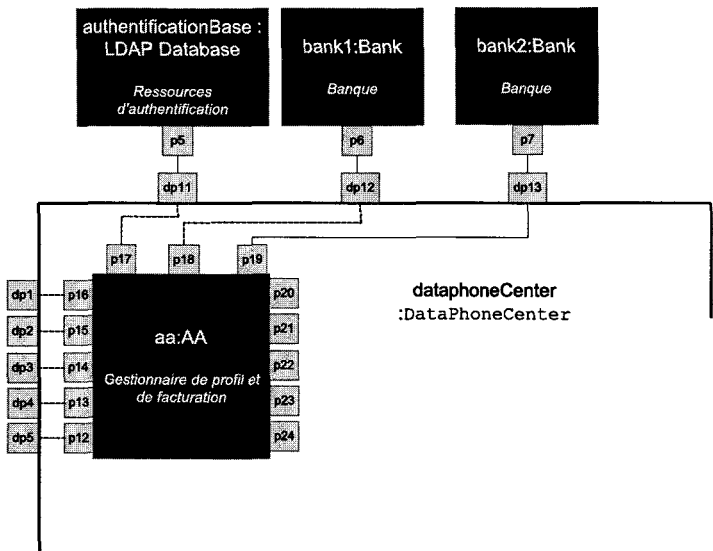


FIG. 6.10 – Plan de base

Le lieu d'intégration consiste à associer Cm_1 avec le composant *aa*, Cm_2 avec le composant *Bank₁*, Cm_3 avec le composant *Bank₂*, pm_1 avec le port *p₁₈*, pm_2 avec le port *p₁₉*, pm_3 avec le port *p₇* et pm_4 avec le port *p₈*. L'ensemble des contraintes du masque de point de jonction sont respectées. Ces ports possèdent en effet comme le montre la déclaration des types de port dans la figure 4.4 page 78 deux opérations *withdraw* et *deposit*. L'application des règles de transformation crée une nouvelle architecture pour le *DataPhoneCenter*. Cette architecture est décrite dans la figure 6.11.

La transformation détruit la liaison entre les composants *aa* et *Bank₁* et entre les composants *aa* et *Bank₂*. Les ports *p₆*, *p₇*, *p₁₈* et *p₁₉*, identifiés par pm_1 , pm_2 , pm_3 et pm_4 , sont détruits au cours de la transformation. En effet, ils ne possèdent avant la transformation que deux opérations qui sont déplacées dans des ports créés au cours du tissage. Les ports délégués *dp₁₈*, *dp₁₉*, *dp₂₀* et *dp₂₁* sont créés automatiquement afin de permettre la connexion entre le composant coordinateur et les composants *log1*, *log2*, *bank1* et *bank2*.

6.2.4.2 Notion de coupe

Comme les préoccupations intégrées à l'aide de patron sont souvent transverses à une architecture, l'architecte doit généralement définir un ensemble des lieux d'intégration sur lesquels il souhaite intégrer le nouveau plan. C'est cet ensemble de lieux que l'on nomme la coupe.

Dans TranSAT, un premier modèle abstrait a été proposé par extension du modèle de masque de point de jonction présenté dans la figure 6.6. L'extension consiste principalement à pouvoir ajouter des contraintes de nommage sur les éléments du masque de point de jonction. Une telle approche permet d'obtenir une spécialisation du masque

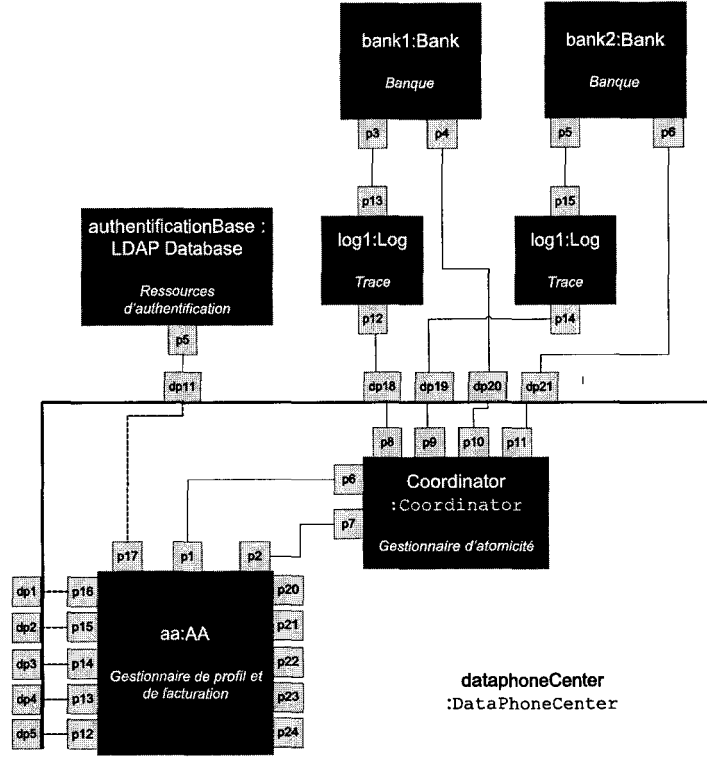


FIG. 6.11 – Intégration de l'atomicité au niveau du *DataPhoneCenter*

de point de jonction au niveau de la définition de la coupe. Le modèle de masque de point de jonction et le modèle de coupe sont alors très proches. La différence entre le masque de point de jonction et la coupe provient principalement du fait que le premier est associé au patron. Il est donc défini indépendamment de tout contexte d'intégration. Son utilisation sur un contexte d'intégration particulier permet de trouver la liste des lieux d'intégration compatibles avec ce patron. Au contraire, la coupe est définie pour un contexte d'intégration particulier. Les fragments d'architecture respectant les contraintes établies au niveau de la coupe sont sélectionnés comme des lieux d'intégration sur lesquels le patron est appliqué. En outre, travailler par spécialisation du masque de point de jonction permet de garantir la cohérence des lieux d'intégration par rapport au masque de point de jonction. Néanmoins, ce modèle reste abstrait. Nous ne lui avons pas attaché de syntaxe concrète, car la richesse d'un tel modèle de coupe engendre une complexité algorithmique importante au niveau de la sélection automatique des lieux d'intégration capturés par cette coupe. En effet pour un masque de point de jonction composé de n_{om} *OperationMask*, n_{pm} *PortMask*, n_{cm} *ComponentMask*, n_{bm} *ConnectionMask* et n_{mm} *MessageMask* définissant les critères d'un lieu d'intégration d'une architecture logicielle composée de n_o *Operations*, n_p *Ports*, n_c *Components*, n_b *Connections* et n_m *Messages*, le nombre de lieux d'intégration potentiels est de :

$$A_{n_o}^{n_{om}} \times A_{n_p}^{n_{pm}} \times A_{n_c}^{n_{cm}} \times A_{n_b}^{n_{bm}} \times A_{n_m}^{n_{mm}} = \frac{n_o!}{(n_o - n_{om})!} \times \frac{n_p!}{(n_p - n_{pm})!} \times \frac{n_c!}{(n_c - n_{cm})!} \times \frac{n_b!}{(n_b - n_{bm})!} \times \frac{n_m!}{(n_m - n_{mm})!}.$$

Ce nombre de lieux potentiels engendre une recherche exhaustive impossible à mettre en œuvre de manière générale pour toute architecture et toute expression de coupe. C'est pourquoi, nous proposons dans une première démarche de laisser cette tâche de définition des lieux d'intégration à la charge de l'architecte, le canevas ne se chargeant que de la propriété liée à la vérification de la compatibilité vis-à-vis du masque de point de jonction. Une des perspectives du travail liée à la définition d'un modèle concret de coupe consistera soit à se baser sur le modèle de coupe abstrait présenté dans cette section et trouver un algorithme de recherche efficace des lieux d'intégration à partir d'une description d'architecture logicielle, soit à définir un modèle abstrait plus simple permettant une recherche exhaustive des lieux d'intégration.

6.2.4.3 Synthèse et limites

Le tisseur peut véritablement être considéré comme un opérateur de transformation au sens mathématique du terme. Il prend un premier paramètre (parti gauche de l'opérateur) qui correspond à l'ensemble des lieux d'intégration définis par l'architecte et possède un deuxième paramètre (la partie droite de l'opérateur) qui correspond au patron à appliquer. Le tisseur vérifie alors la compatibilité entre le parti gauche de l'opérateur et la partie droite, c'est-à-dire la compatibilité entre le(s) lieu(x) d'intégration et les contraintes définies par le masque de point de jonction. Il effectue ensuite la transformation, et finalement vérifie la cohérence de l'architecture résultante du tissage de ces deux plans.

Face à la complexité algorithmique concernant la recherche exhaustive de tous les lieux d'intégration compatibles avec un masque de point de jonction, TranSAT ne fournit pas de mécanismes efficaces permettant de trouver tous ces lieux d'intégration ni de langage de coupe permettant de sélectionner un sous-ensemble de lieux d'intégration compatibles. La définition du lieu d'intégration reste donc pour le moment manuel dans TranSAT. L'architecte associe l'ensemble des éléments du masque de point de jonction avec des éléments du plan de base qu'il souhaite modifier. Ce manque d'assistance à ce moment précis du processus de transformation n'est néanmoins pas dommageable. En effet, dans un processus de construction incrémentale, on peut aisément considérer que l'architecte, quand il choisit le patron, a déjà une certaine idée du ou des lieux d'intégration qu'il souhaite modifier. TranSAT fournit alors un mécanisme d'analyse statique afin de vérifier que les lieux d'intégration choisis respectent l'ensemble des contraintes définies au niveau du masque de point de jonction.

Au cours de la transformation, l'intégration d'un nouveau plan peut se révéler impossible du fait de la présence au sein du plan de base ou du plan à intégrer d'éléments possédant les mêmes noms. En effet, le patron est conçu indépendamment de tout plan de base sur laquelle il est appliqué. En outre, le patron et le plan de base sont parfois construits par des architectes différents. De ce fait, la présence d'un conflit de nommage est possible. Dans SafArchie, les composants sont identifiés par leur nom, les ports par leur nom et le composant auquel ils appartiennent, les opérations par le nom du port auquel elles appartiennent et le nom du composant auquel ce port appartient. En cas de conflit de noms, par exemple au cours de la transformation, l'ajout d'un port à un composant du plan de base possédant déjà un port du même nom, la transformation n'a pas lieu, le conflit de nommage est identifié et l'architecte doit alors proposer une stratégie pour contourner ce problème (changement du nom d'un des deux éléments, changement de patron, de lieu d'intégration, etc.).

6.3 Un processus itératif pour la construction d'architectures logicielles

La première section de ce chapitre définit l'architecture du canevas de conception TranSAT. Elle identifie les principaux éléments intervenant dans la spécification de l'intégration d'une nouvelle préoccupation au sein d'une architecture logicielle. Aucune information quant au mode d'emploi de ces concepts dans une démarche d'intégration de nouvelles préoccupations n'est décrite. Cette section définit précisément le *processus* associé à TranSAT pour permettre à différents acteurs de collaborer dans l'activité liée à la construction incrémentale d'une architecture logicielle.

6.3.1 Définition

L'intégration d'une nouvelle préoccupation au sein d'une architecture logicielle est intrinsèquement un processus. En effet, d'après le sens commun, un processus se définit comme une succession d'étapes élémentaires permettant d'aboutir à un résultat. Il est conçu comme actif et organisé dans le temps¹¹. Dans le domaine du génie logiciel, le processus désigne la méthode de développement d'une application, c'est-à-dire l'ensemble des activités nécessaires au développement d'un logiciel. Il n'existe pas de méthode universelle pour le développement de logiciel. Une telle méthode présupposerait une théorie éprouvée pour le génie logiciel. En effet, le développement

¹¹ Le petit Robert 1994

d'une application dépend du domaine, de la culture de l'organisation et de ses membres. Cette absence de théorie universelle n'engendre pas néanmoins l'impossibilité de définir des processus adaptés à certaines tâches du développement d'une application. L'expérience accumulée sur des projets réalisés, dans des environnements différents permet d'inférer de bonnes pratiques en termes de développement d'application. Enfin, outre les différentes activités qui vont permettre l'intégration d'une nouvelle préoccupation au sein d'une architecture logicielle, nous nous intéressons dans cette section aux différents *acteurs* qui vont contribuer à ce processus et à la définition de leur *rôle*, deux aspects très importants d'un processus de génie logiciel.

Les *acteurs* sont une représentation idéalisée d'une personne, d'un logiciel, d'un processus, d'une organisation qui interagit (depuis l'extérieur) avec le processus.

Le *rôle* d'un acteur définit la manière dont il interagit depuis l'extérieur avec le processus.

6.3.2 Un processus d'intégration de nouvelles préoccupations

6.3.2.1 SPEM

Face à l'impossibilité de définir un processus universel pour le développement d'application dans le domaine du génie logiciel, l'OMG (*Object Management Group*) a travaillé sur un méta-modèle permettant de spécifier de tels processus. SPEM (*Software Process Engineering Metamodel*) [OMG, 2005] est comme son nom l'indique un méta-modèle pour la modélisation des processus de développement logiciel tel que le RUP (*Rational Unified Process*) [Kruchten, 2000].

La description d'un processus en SPEM se structure principalement à partir de trois types d'informations : les acteurs, les produits de travail et la définition de travail. Dans SPEM, la notion d'acteur suit la définition générale définie dans la section 6.3.1. Les produits de travail sont l'ensemble des documents créés au cours du processus. Ces documents peuvent être aussi un cahier des charges, des spécifications, du code, des documents vidéos, *etc.* Enfin la définition de travail identifie l'ensemble des étapes (les documents produits, les acteurs participant) permettant la production de produits de travail. Dans la suite de cette section, nous utiliserons différents modèles SPEM afin de présenter le processus associé à TranSAT.

6.3.2.2 Les acteurs

Dans le cadre de la construction incrémentale d'une architecture logicielle, nous considérons trois acteurs principaux : l'architecte, l'expert de domaine et l'intégrateur de patrons d'architecture.

Le premier rôle de l'*architecte logiciel* consiste à définir à partir d'un cahier des charges le cœur d'une application, c'est-à-dire les entités primaires constituant le logiciel et leur organisation. Ce cœur d'application représente un premier niveau d'architecture logicielle, le plan de base. Dans le cadre de TranSAT, cette architecture est définie par un ensemble de composants, représentant les entités primaires, ainsi que par leurs interconnexions, définissant l'organisation de l'application. Un architecte doit donc définir d'une part, une spécification de haut niveau des composants à l'aide des différents contrats présentés dans le chapitre 4 et un plan global précisant par quelles interfaces ces composants sont mis en relation. Le second rôle de l'architecte consiste à transformer son architecture logicielle par intégrations successives de nouvelles préoccupations. L'architecte en partant des informations contenues au niveau du patron d'architecture définit une coupe identifiant les lieux d'intégration du nouveau plan au niveau du plan de base. Si ces lieux sont conformes au masque de point de jonction, l'architecture est transformée. Le rôle de l'architecte se limite donc au choix du patron et à l'identification des lieux d'intégration.

Le rôle de l'*expert de domaine* consiste à construire des plans d'architecture génériques et réutilisables prenant en charge une préoccupation. Contrairement à l'architecte qui part d'un cahier des charges qu'il cherche à satisfaire, l'expert de domaine, par sa connaissance des standards et des solutions existants au niveau de sa préoccupation spécifie les composants et leurs interactions.

Enfin, le rôle de l'*intégreur* est étroitement lié au rôle de l'expert de domaine. L'intégreur contribue à la création du patron d'architecture en définissant précisément le masque de point de jonction et les règles de transformation. Le travail sur le masque de point de jonction consiste à trouver un bon compromis entre une définition très précise du contexte d'intégration qui permet de disposer de beaucoup d'informations sur les éléments de l'architecture cible au niveau des règles de transformation et une définition très générique du masque qui offre, quant à elle, l'avantage de définir un patron fortement réutilisable. À l'aide des informations contenues dans le masque de point de jonction et dans le plan d'architecture défini par l'expert, l'intégreur spécifie les règles de transformation. Ces règles définissent comment le nouveau plan est tissé au éléments du masque de point de jonction.

6.3.2.3 Les produits de travail

En SPEM, un produit de travail représente comme son nom l'indique tout élément (document, code, binaire, etc.) résultat d'une activité. La description d'un processus en SPEM est toujours vue comme un composant qui prend en entrée des produits de travail et qui transforme cette information en nouveaux produits de travail.

Ainsi dans le cadre de TranSAT, les produits de travail en entrée sont le cahier des charges, la connaissance des standards et des services techniques existants pour les différentes préoccupations et les différents patrons d'architecture existants. Le processus produit une architecture logicielle complète, une architecture logicielle métier, des implantations de composant et de nouveaux patrons d'architecture (voir Figure 6.12).

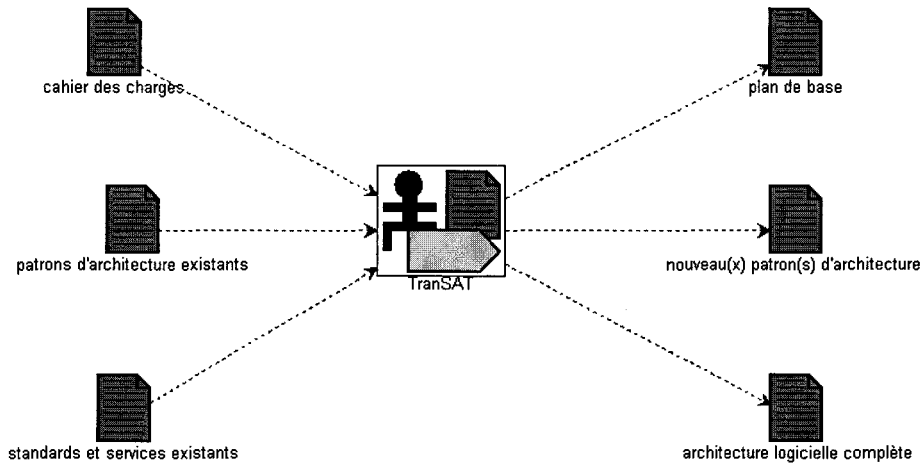


FIG. 6.12 – Vision externe du processus

Les différents produits de travail intermédiaires au niveau du processus sont les spécifications des composants, le plan d'architecture (assemblage de composants), le masque de point de jonction, les règles de transformation, le patron d'architecture et la coupe.

6.3.2.4 Les définitions de travail

Au niveau de granularité le plus élevée, les tâches de chaque acteur sont découpées en *définitions de travail*. Dans le processus de TranSAT, nous identifions quatre définitions de travail à savoir : la définition du plan de base, la définition d'un nouveau plan, la définition d'un patron d'architecture et l'intégration d'une nouvelle préoccupation.

La figure 6.13 présente une vue globale du diagramme de flots correspondant au processus associé à TranSAT. Chaque acteur prend en charge une ou plusieurs définitions de travail.

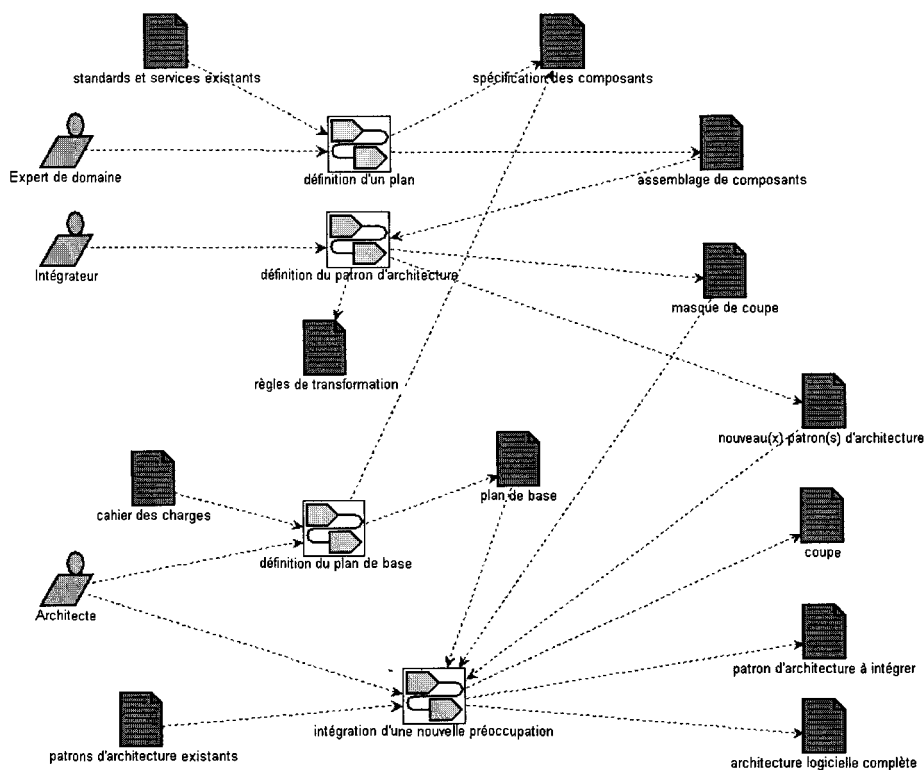


FIG. 6.13 – Vision interne du processus

Une définition de travail est structurée en un ensemble d'activités décrites à l'aide d'un diagramme de flots et d'un diagramme d'activité. Cette structuration permet de préciser en termes d'activités l'enchaînement du travail de chaque acteur.

La définition du plan de base qui prend en entrée le cahier des charges de l'architecte et qui produit une spécification des composants et une description de leur assemblage est constituée de trois activités (voir Figure 6.14). Ces activités sont la définition des interfaces de composant permettant de produire la spécification des composants, la définition de l'assemblage des composants et la validation de l'architecture.

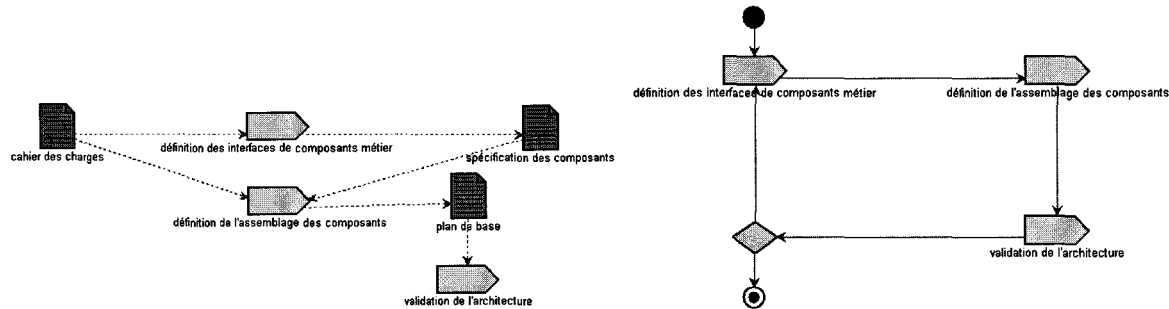


FIG. 6.14 – Diagrammes de flots et d'activités pour la définition d'un plan de base

La définition d'un nouveau plan est prise en charge par un expert du domaine lié à la préoccupation. En entrée, la connaissance des normes existantes et des services disponibles permet à l'expert de spécifier les interfaces des composants et l'assemblage partiel liés à son domaine d'expertise. Le diagramme de flots et le diagramme

d'activités présentés Figure 6.15 sont très proches de ceux liés à la définition d'un plan de base pour l'architecte.

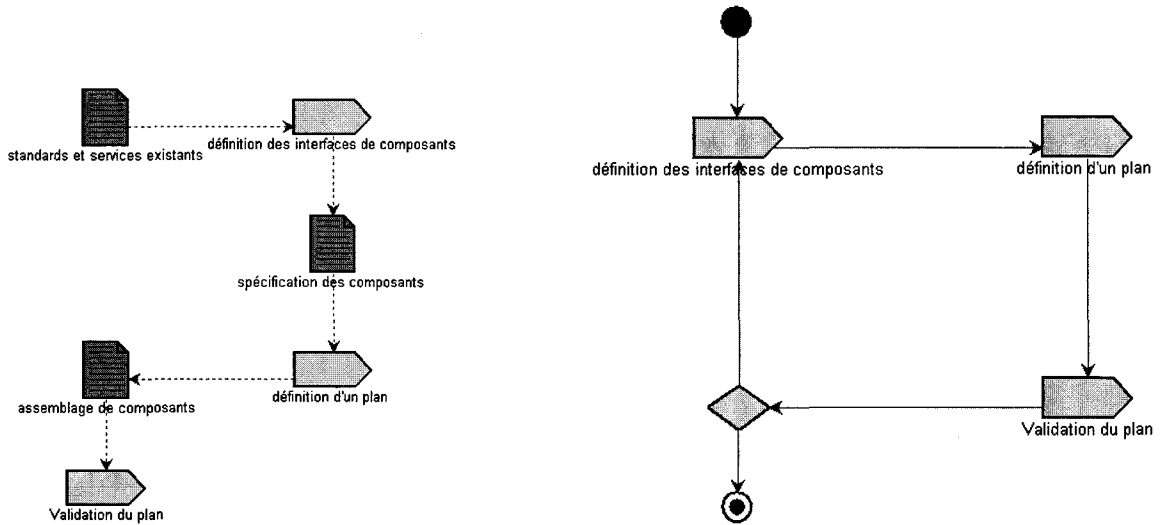


FIG. 6.15 – Diagrammes de flots et d'activités pour la définition d'un nouveau plan

La définition d'un nouveau patron par l'intégrateur se structure autour de cinq activités : la définition du masque de point de jonction, la validation de ce masque, la définition des règles de transformation, la création du patron et la validation du patron d'architecture. La première activité consiste à définir le masque de point de jonction. Comme celui-ci sert de référence pour la définition des règles de transformation, il devra être validé avant que l'intégrateur puisse définir les règles de transformation. La création du patron consiste à regrouper le plan, le masque de point de jonction et les règles de transformation. La validation de ce patron d'architecture conclut la tâche liée à la définition d'un nouveau patron d'architecture (voir Figure 6.16).

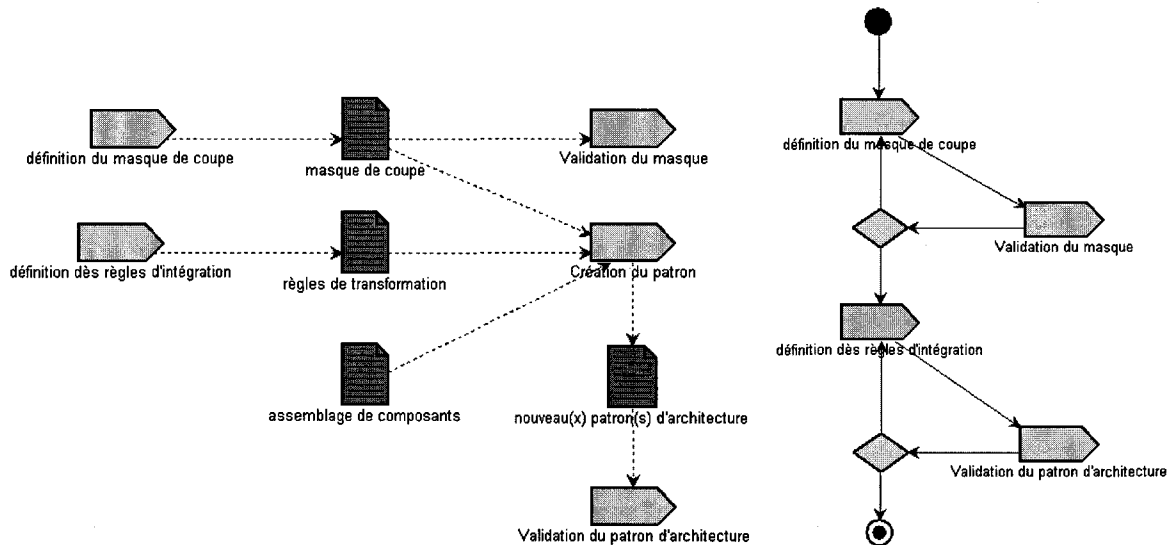


FIG. 6.16 – Diagrammes de flots et d'activités pour la définition d'un nouveau patron d'architecture

L'intégration d'une nouvelle préoccupation au sein d'un plan de base est une tâche prise en charge par l'architecte. Cette tâche se structure en cinq activités : le choix du patron, la définition de la coupe, la validation de

cette coupe, la transformation et la validation de la transformation. Ces activités se déroulent de la manière suivante : l'architecte choisit le patron, définit la coupe, la valide, exécute la transformation et valide le résultat de la transformation (voir Figure 6.17).

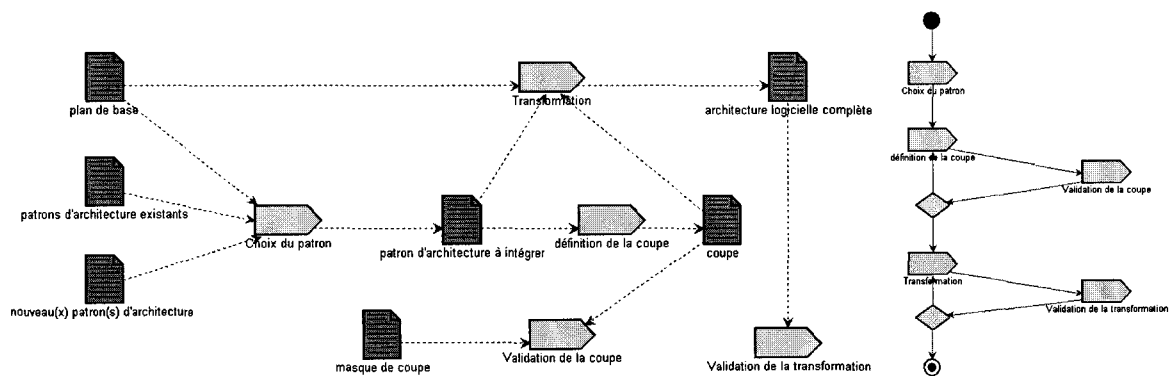


FIG. 6.17 – Diagrammes de flots et d'activités pour la spécification de l'intégration d'une nouvelle préoccupation au sein d'une architecture logicielle métier

6.3.2.5 Responsabilités de chaque acteur

Chaque produit de travail est sous la responsabilité d'un unique acteur qui sera l'interlocuteur privilégié pour les interrogations et/ou les remarques relatives à ce travail (voir Figure 6.18).

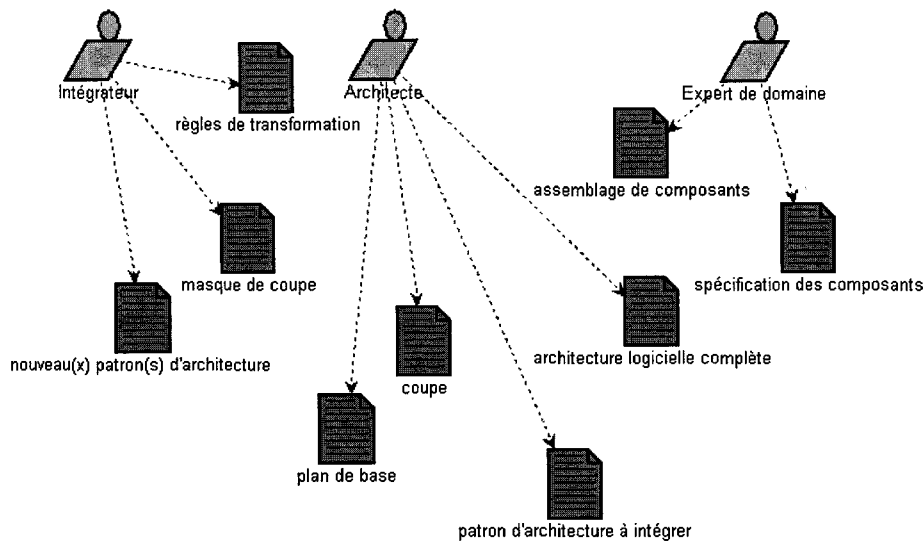


FIG. 6.18 – Diagramme de responsabilités

6.3.3 Synthèse

Les avantages de définir précisément le processus sont les suivants :

- chaque membre de l'équipe de développement comprend quel est son rôle dans le développement de l'application,
- les responsables comprennent mieux l'avancement du travail de chacun grâce à la mise à jour au fur et à mesure de l'architecture logicielle,
- le processus itératif contribue à la maîtrise de la complexité du projet. À chaque étape de transformation, les vérifications permettent d'étudier l'impact de la transformation sur la cohérence globale de l'architecture.

6.4 Sûreté de fonctionnement

L'objectif de TranSAT est d'offrir un canevas de conception pour assister l'architecte lors de l'intégration d'une nouvelle préoccupation. La définition du patron comme entité réutilisable permet à l'architecte de capitaliser le travail lié à l'intégration d'une nouvelle préoccupation. Le processus associé à TranSAT explicite, quant à lui, le travail de l'architecte et permet de découper ce travail entre différents acteurs. Cette section est centrée maintenant sur une caractéristique très importante de TranSAT, à savoir comment nous garantissons la cohérence d'une architecture logicielle transformée. Après un retour sur les risques associés à la transformation d'une architecture logicielle, cette section présente la sémantique de notre langage de règles de transformation qui nous permet d'effectuer une première analyse statique du patron d'architecture. Enfin, la dernière section présente les analyses dynamiques qui permettent de détecter certains types d'erreur qui ne peuvent être repérés à partir des informations contenues au sein du masque de point de jonction.

6.4.1 Les risques

L'intégration de nouvelles préoccupations au sein d'une architecture logicielle engendre des modifications des interfaces des composants, de leur comportement et de leurs interactions. Cette tâche est complexe et risquée. Les nombreuses modifications liées à cette transformation peuvent avoir un impact sur la cohérence de l'architecture résultante de la transformation. Nous identifions trois familles d'incohérences au niveau de l'architecture résultante : les incohérences vis-à-vis des contraintes du modèle de composant, les incompatibilités structurelles entre composants connectés, et les incompatibilités comportementales au sein d'un assemblage de composants.

Problème de conformité par rapport au méta-modèle

Un modèle de composant définit un certain nombre de contraintes sur ces éléments et les relations entre ces éléments. Par exemple, dans SafArchie, un port doit contenir au moins une opération, doit appartenir à un unique composant et doit être connecté à un unique autre port. De même, un composant ne peut-être partagé et appartenir à plusieurs composites (voir Chapitre 5).

En outre, parmi les opérations à effectuer pour intégrer une nouvelle préoccupation au sein d'un plan de base, nous trouvons la création de connexions entre les éléments du nouveau plan et les éléments du plan de base. C'est une erreur de l'architecte que d'essayer de connecter manuellement deux ports dont les composants parents n'appartiennent pas au même composite. La traversée d'une membrane de composite doit nécessairement engendrer la création des ports délégués. Cette tâche, si elle est effectuée manuellement par l'architecte, peut rapidement devenir fastidieuse et propice aux erreurs. De même, l'opération de destruction d'une connexion libère deux ports. Suite à cette déconnexion, ces ports doivent être connectés à d'autres composants ou détruits.

Enfin, la construction de l'automate de comportement associé à chaque composant est propice aux erreurs principalement du fait de la structure complexe de l'automate et de la contrainte liée au maintien de la cohérence entre la partie structurelle et la partie comportementale. En effet, toute opération liée aux ports d'un composant doit apparaître quelque part dans l'automate du comportement du composant. Comme le langage de description d'architecture sépare les descriptions structurelles et comportementales et que le lien entre la partie structurelle

et la partie comportementale est assuré par une contrainte entre le nommage des opérations (partie structurelle) et celui des messages (partie comportementale), il est facile de modifier l'un sans modifier l'autre et inversement. De plus, un automate doit également décrire un comportement significatif ; au minimum, pour chaque opération, l'appel précède un retour. La vérification de ces contraintes exige de connaître tous les chemins de l'automate. Il peut être difficile de le faire manuellement. Finalement, la destruction d'une opération exige de retirer les messages associés partout où ils apparaissent dans l'automate. Ceci peut impliquer de réorganiser l'automate pour éliminer les chemins qui ne sont plus significatifs. Cette tâche peut, elle aussi, se révéler complexe.

Ainsi, l'ensemble des transformations sur l'architecture doit préserver à la fois ce que l'on nomme souvent l'intégrité référentielle, mais aussi l'ensemble des contraintes du modèle de composant SafArchie. Le maintien de cette intégrité et le respect des règles sont difficiles à garantir dans le cas d'une transformation effectuée manuellement par l'architecte.

Compatibilité structurelle entre ports connectés

Chaque nouvelle connexion issue de la transformation met en relation deux ports possédant chacun un contrat de port et une définition des signatures des opérations qu'il requiert ou fournit. Quand l'architecte crée de nouvelles connexions, une erreur consiste à connecter des ports incompatibles. Chaque action sur les connexions doit donc être analysée du point de vue de la conformité par rapport aux contraintes du modèle de composant, mais aussi du point de vue de la compatibilité entre opérations.

Compatibilité comportementale d'un assemblage de composants

Enfin et surtout, l'ajout d'un nouveau composant ou la modification du comportement d'un composant peut engendrer des problèmes d'interblocages au niveau d'un assemblage de composants. Si avec de la patience, il est possible de considérer que l'architecte va pouvoir garantir l'intégrité référentielle de son architecture, la conformité par rapport au modèle de composant et la compatibilité des nouveaux ports liés. Toute modification du contrat de comportement peut impacter la façon dont ce composant se synchronise avec ses voisins. Cette modification peut alors engendrer un interblocage sur une synchronisation d'un assemblage d'un des composites englobants. Ce type d'erreur lié à la forte dépendance entre les comportements des composants est très difficile à détecter.

Problématique

SafArchie dispose d'une suite d'outils qui permet d'ores et déjà de détecter des erreurs liées à la non-conformité par rapport au modèle de composant, des incompatibilités structurelles entre composants liés et des incompatibilités comportementales au niveau d'un assemblage de composants. Cependant, le retour de ces erreurs est inadapté à une construction itérative. Tout d'abord, pour chaque étape de transformation, les outils vont analyser la totalité de l'architecture logicielle dont seulement une partie a été modifiée. En cas de détection d'anomalies, les informations retournées à l'architecte ne tiennent pas compte des règles de transformation. Les rapports d'anomalies sont alors uniquement factuels et ne permettent pas de remonter à l'architecte quelles règles de transformation ont causé cette erreur. Enfin et surtout, les outils d'analyse liés à SafArchie ne peuvent travailler que sur une architecture logicielle complète, c'est-à-dire le résultat de la transformation. Or, de par l'architecture de TranSAT centrée autour d'une entité modulaire et réutilisable, le patron d'architecture, il est nécessaire de pouvoir déplacer un maximum de ces analyses au niveau du patron d'architecture.

L'objectif de TranSAT est de pouvoir fournir un ensemble de vérifications permettant une première analyse du patron indépendamment du contexte d'intégration, une analyse partielle des éléments modifiés par la transformation, et enfin une précision des messages d'erreurs identifiant les règles de transformation ayant entraîné l'apparition d'une erreur au niveau de la description d'architecture. Ainsi, TranSAT vise à effectuer un maximum de vérifications au niveau du patron d'architecture grâce à la connaissance de la sémantique du langage de transformation. En effet, les analyses effectuées statiquement permettent de valider une fois pour toutes certaines propriétés

au niveau du patron d'architecture. Ces propriétés ne devront plus être vérifiées dynamiquement au moment de la transformation. Ces vérifications représentent la partie statique de l'analyse d'une transformation. Pour la partie dynamique dépendante du contexte d'intégration, TranSAT vise à n'effectuer que les vérifications qui sont pertinentes par rapport aux modifications effectuées sur le plan de base. L'analyse des règles de transformation va permettre de déterminer précisément quelles parties de l'architecture doivent être à nouveau vérifiées.

6.4.2 Vérifications statiques sur le patron d'architecture

6.4.2.1 Vérification de la compatibilité structurelle et comportementale entre composants du plans liés

La première chose à vérifier dans l'analyse statique d'un patron d'architecture est la cohérence du plan attaché à ce patron. Comme le plan est construit à l'aide de SafArchie, la cohérence d'un plan à intégrer correspond à vérifier la cohérence de l'assemblage de composants qu'il décrit. Hormis le fait que le plan associé à un patron peut posséder des ports non liés, le reste des vérifications est le même que pour la cohérence d'une architecture définie avec SafArchie. Il faut donc vérifier la conformité du plan par rapport au méta-modèle d'architecture, la compatibilité structurelle des ports connectés, la compatibilité comportementale des composants connectés. Pour ce faire, nous utilisons les mêmes vérifications que celles présentées dans les sections 4.3.1 et 4.3.2.

6.4.2.2 Vérifications statiques du masque de point de jonction

La deuxième source d'information qui est analysée pour garantir la cohérence d'un patron d'architecture concerne le masque de point de jonction. Deux propriétés distinctes sont vérifiées : la conformité du masque de point de jonction du patron par rapport à son méta-modèle et l'adéquation du masque de point de jonction du patron par rapport aux règles de transformation de ce même patron. La conformité du masque de point de jonction consiste à vérifier qu'un masque de point de jonction attaché à un patron est bien une instance du méta-modèle de masque de point de jonction. C'est une analyse de l'arbre syntaxique résultant de l'analyse du langage de transformation. Dans TranSAT, tout élément manipulé par les règles de transformation doit soit être créé par une primitive de transformation, soit faire référence à un élément du plan à intégrer, soit faire référence à un élément du masque de point de jonction. En outre, chaque élément est typé (*Port*, *Connexion*, *Composant*, etc.). L'adéquation du masque de point de jonction du patron par rapport aux règles de transformation vérifie que chaque élément du masque de point de jonction utilisé dans les règles de transformation est du bon type par rapport à la primitive qu'il l'utilise.

6.4.2.3 Vérifications des règles de transformation

Principes élémentaires

- Toute variable doit être définie avant d'être initialisée.
- Les règles de transformation sont prévues pour agir sur un type d'élément de l'architecture en particulier. Les éléments référencés par une règle de transformation doivent être du bon type.
- Chaque objet est référencé par son nom. Tous les noms d'objets sont donc distincts.

Une transformation a la forme S , définie ci-dessous :

$$\begin{aligned}
 S &::= s ; S \mid \varepsilon \\
 s &::= \text{Composite } Cr \\
 &\quad | \text{Composite } Cr_1 \text{ in } Cr_2 \\
 &\quad | \text{Port } Pr \text{ in } Cp \\
 &\quad | \text{Operation } Or = op \text{ in } Pr \\
 &\quad | \text{Operation } Or = op \text{ replaces } Or \\
 &\quad | \text{Binding } Br = \{Pr_1, Pr_2\} \\
 &\quad | Cr.destroy() \\
 &\quad | Pr.destroy() \\
 &\quad | Br.destroy() \\
 &\quad | Cr.move(Cr) \\
 &\quad | Cp.move(Cr) \\
 &\quad | Or.move(Pr) \\
 op &::= Or \mid \text{inverse}(Or)
 \end{aligned}$$

Un composant est identifié par son nom (par exemple C_1).

Un port est identifié par son nom, celui-ci contient le nom du composant possédant le port et le nom du port (par exemple $C_1.p_1$)

Une opération est toujours identifiée par son nom, celui-ci contient le nom du composant possédant l'opération, le nom du port contenant l'opération, et le nom de l'opération (par exemple $C_1.p_1.op_1$), ou une variable.

Au niveau structurel, nous différencions au niveau de l'analyse, la vérification faite sur le contenu, c'est-à-dire relation entre les composites, les composants, les ports et les opérations, de l'analyse faite sur les connexions.

6.4.2.4 Vérification du contenu

Règles

Nous définissons la relation suivante sur les types des objets manipulés par TranSAT :

$$\text{composite}^* > \text{component} > \text{port} > \text{operation}$$

Si $x > y$, alors chaque instance de x doit contenir au moins une instance de y . En outre, chaque instance de y doit être contenue dans exactement une instance de x . Cette relation est une relation d'ordre, elle est donc asymétrique.

Le relation composite^* indique qu'un composite peut contenir un composite et ceci une infinité de fois. L'analyse présentée ci-dessous ne prend pas en compte les connexions entre composants. Celle-ci sera présentée dans la section 6.4.2.5.

Cette analyse définit un environnement ρ constitué d'éléments d'architecture. Chacun de ces éléments est associé à son parent si il en a un et à la liste de ces enfants. ρ est initialisé à partir des informations contenues dans le plan à ajouter et des informations contenues dans le masque de point de jonction. Pour le masque de point de jonction, si $*$ apparaît dans la définition des opérations d'un port, rien n'est ajouté à ρ .

Dans TranSAT, il n'est pas possible de créer ou d'ajouter des éléments afin de les détruire ou de les enlever après. Afin de s'assurer du respect de cette contrainte, l'analyse utilise des listes J et P contenant respectivement les éléments du masque de point de jonction et les éléments du plan. Seuls les éléments contenus dans J et P sont modifiables, ils sont alors retirés de J et de P .

L'analyse de contenu est défini en terme de jugement de la forme $\rho, J, P \vdash s \rightarrow \rho', J', P'$. Chaque élément de l'environnement ρ est associé à un couple (π, ζ) où π représente le parent de cet élément et ζ l'ensemble de ces enfants. $\rho, J, P \vdash s \rightarrow \rho', J', P'$ est un jugement qui définit l'environnement de départ (ρ, J, P) , l'opérateur de transformation (s) et l'environnement d'arrivée (ρ', J', P') . Si des conditions sont définies sur les paramètres de l'opérateur de transformation ou sur les éléments de l'environnement de départ, le jugement a la forme $\frac{\text{condition}}{\text{jugement}}$ comme montré ci-dessous.

Transformation :

$$\rho, J, P \vdash \varepsilon \rightarrow \rho$$

$$\frac{\rho, J, P \vdash s \rightarrow \rho', J', P' \quad \rho', J', P' \vdash S \rightarrow \rho''}{\rho, J, P \vdash s \ S \rightarrow \rho''}$$

Primitives :

Composite Cr : ajoute Cr à l'environnement avec la valeur $(None, [])$

$$\rho, J, P \vdash \text{Composite } Cr \rightarrow \rho[Cr \mapsto (None, \emptyset)], J, P$$

Composite Cr_1 in Cr_2 : Si l'entrée pour Cr_2 dans ρ est (π, ζ) , alors l'entrée pour Cr_2 devient $(\pi, \zeta + [Cr_1])$ et l'entrée pour Cr_1 devient $(Cr_2, [])$.

$$\begin{aligned} &\rho[Cr_2 \mapsto (\pi, \zeta)], J, P \vdash \text{Composite } Cr_1 \text{ in } Cr_2 \\ &\rightarrow \rho[Cr_1 \mapsto (Cr_2, \emptyset), Cr_2 \mapsto (\pi, \zeta \cup \{Cr_1\})], J, P \end{aligned}$$

Port Pr in Cp : Si l'entrée pour Cp dans ρ est (π, ζ) , alors l'entrée pour Cp devient $(\pi, \zeta + [Pr])$ et l'entrée pour Pr devient $(Cp, [])$.

$$\rho[Cp \mapsto (\pi, \zeta)], J, P \vdash \text{Port } Pr \text{ in } Cp \rightarrow \rho[Pr \mapsto (Cp, \emptyset), Cp \mapsto (\pi, \zeta \cup \{Pr\})], J, P$$

Operation $Or = op$ in Pr : Si l'entrée pour Pr dans ρ est (π, ζ) , alors l'entrée pour Pr devient $(\pi, \zeta + [Or])$ et l'entrée pour Or devient $(Pr, [])$. Pr ne peut-être le parent de op , ou de l'inverse de op ($inverse(op)$).

$$\frac{\pi_1 \neq Pr \quad Or_2, inverse(Or_2) \notin \zeta_2}{\rho[Or_2 \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2)], J, P \vdash \text{Operation } Or_1 = Or_2 \text{ in } Pr \rightarrow \rho[Or_1 \mapsto (Pr, \emptyset), Or_2 \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or_1\})], J, P}$$

$$\frac{\pi_1 \neq Pr \quad Or_2, inverse(Or_2) \notin \zeta_2}{\rho[Or_2 \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2)], J, P \vdash \text{Operation } Or_1 = inverse(Or_2) \text{ in } Pr \rightarrow \rho[Or_1 \mapsto (Pr, \emptyset), Or_2 \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or_1\})], J, P}$$

Operation $Or_1 = op$ replaces Or_3 : Or_1 et Or_3 doivent avoir le même type et la même polarité. Or_3 doit être dans le masque de point de jonction ou dans le plan et ne doit pas avoir été déplacé. Or_1 a le parent de Or_3 . Le parent de Or_3 retire Or_3 de ses enfants et ajoute Or_1 comme un nouveau fils. Le parent de Or_3 ne peut pas être aussi le parent de op ou de l'inverse de op .

$$\frac{Or_3 \in J \cup P \quad \pi_1 \neq \pi_2}{\rho[Or_2 \mapsto (\pi_1, \emptyset), Or_3 \mapsto (\pi_2, \emptyset), \pi_2 \mapsto (\pi_3, \zeta_3)], J, P \vdash \text{Operation } Or_1 = Or_2 \text{ replaces } Or_3 \rightarrow \rho[Or_2 \mapsto (\pi_1, \emptyset), Or_1 \mapsto (\pi_2, \emptyset), \pi_2 \mapsto (\pi_3, \zeta_3 - \{Or_3\} \cup \{Or_1\})], J, P}$$

$$\frac{Or_3 \in J \cup P \quad \pi_1 \neq \pi_2}{\rho[Or_2 \mapsto (\pi_1, \emptyset), Or_3 \mapsto (\pi_2, \emptyset), \pi_2 \mapsto (\pi_3, \zeta_3)], J, P \vdash \text{Operation } Or_1 = inverse(Or_2) \text{ replaces } Or_3 \rightarrow \rho[Or_2 \mapsto (\pi_1, \emptyset), Or_1 \mapsto (\pi_2, \emptyset), \pi_2 \mapsto (\pi_3, \zeta_3 - \{Or_3\} \cup \{Or_1\})], J, P}$$

$Cr.destroy()$: Cr ne doit contenir aucun enfant. Il doit faire partie du masque de point de jonction et ne doit pas avoir été déplacé. Son parent retire Cr de la liste de ces enfants.

$$\frac{Cr \in J}{\rho[Cr \mapsto (\pi_1, \emptyset), \pi_1 \mapsto (\pi_2, \zeta_2)], J, P \vdash Cr.destroy() \rightarrow \rho[\pi_1 \mapsto (\pi_2, \zeta_2 - \{Cr\})], J, P}$$

$Pr.destroy()$: Pr ne doit contenir aucun enfant. Il doit faire partie du masque de point de jonction et ne doit pas avoir été déplacé. Son parent retire Pr de la liste de ces enfants.

$$\frac{Pr \in J}{\rho[Pr \mapsto (\pi_1, \emptyset), \pi_1 \mapsto (\pi_2, \zeta_2)], J, P \vdash Pr.destroy() \rightarrow \rho[\pi_1 \mapsto (\pi_2, \zeta_2 - \{Pr\})], J, P}$$

$Cr_1.move(Cr_2)$: Cr_1 ne doit pas avoir été déplacé. Cr_2 ne doit pas être le parent courant de Cr_1 . Cr_1 doit être soit dans le plan soit dans le masque de point de jonction. S'il est dans le masque de point de jonction, la position originale de Cr_2 ne peut-être dans le masque de point de jonction. De même si Cr_1 est dans le plan, dans sa position originale, Cr_2 ne peut-être dans le plan. Il est enfin nécessaire de vérifier qu'un composite n'est pas déplacé dans un de ces fils.

$$\frac{Cr_1 \in J \quad Cr_2 \notin J \quad \pi_1 \neq Cr_2}{\rho[Cr_1 \mapsto (\pi_1, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Cr_1.move(Cr_2) \rightarrow \rho[Cr_1 \mapsto (Cr_2, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2 \cup \{Cr_1\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Cr_1\})], J - \{Cr_1\}, P}$$

$$\frac{Cr_1 \in J \quad Cr_2 \notin J}{\rho[Cr_1 \mapsto (None, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2)], J, P \vdash Cr_1.move(Cr_2) \rightarrow \rho[Cr_1 \mapsto (Cr_2, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2 \cup \{Cr_1\})], J - \{Cr_1\}, P}$$

$$\frac{Cr_1 \in P \quad Cr_2 \notin P \quad \pi_1 \neq Cr_2}{\rho[Cr_1 \mapsto (\pi_1, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Cr_1.move(Cr_2) \rightarrow \rho[Cr_1 \mapsto (Cr_2, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2 \cup \{Cr_1\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Cr_1\})], J, P - \{Cr_1\}}$$

$$\frac{Cr_1 \in P \quad Cr_2 \notin P}{\rho[Cr_1 \mapsto (None, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2)], J, P \vdash Cr_1.move(Cr_2) \rightarrow \rho[Cr_1 \mapsto (Cr_2, \zeta_1), Cr_2 \mapsto (\pi_2, \zeta_2 \cup \{Cr_1\})], J, P - \{Cr_1\}}$$

$Cp.move(Cr)$: Cp ne doit pas avoir été déplacé. Cr ne doit pas être le parent courant de Cp . Cp doit être soit dans le plan soit dans le masque de point de jonction. Si il est dans le masque de point de jonction, la position originale de Cr ne peut-être dans le masque de point de jonction. De même si Cp est dans le plan, la position originale de Cr ne peut-être dans le plan.

$$\frac{Cp \in J \quad Cr \notin J \quad \pi_1 \neq Cr}{\rho[Cp \mapsto (\pi_1, \zeta_1), Cr \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Cp.move(Cr) \rightarrow \rho[Cp \mapsto (Cr, \zeta_1), Cr \mapsto (\pi_2, \zeta_2 \cup \{Cp\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Cp\})], J - \{Cp\}, P}$$

$$\frac{Cp \in J \quad Cr \notin J}{\rho[Cp \mapsto (None, \zeta_1), Cr \mapsto (\pi_2, \zeta_2)], J, P \vdash Cp.move(Cr) \rightarrow \rho[Cp \mapsto (Cr, \zeta_1), Cr \mapsto (\pi_2, \zeta_2 \cup \{Cp\})], J - \{Cp\}, P}$$

$$\frac{Cp \in P \quad Cr \notin P \quad \pi_1 \neq Cr}{\rho[Cp \mapsto (\pi_1, \zeta_1), Cr \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Cp.move(Cr) \rightarrow \rho[Cp \mapsto (Cr, \zeta_1), Cr \mapsto (\pi_2, \zeta_2 \cup \{Cp\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Cp\})], J, P - \{Cp\}}$$

$$\frac{Cp \in P \quad Cr \notin P}{\rho[Cp \mapsto (None, \zeta_1), Cr \mapsto (\pi_2, \zeta_2)], J, P \vdash Cp.move(Cr) \rightarrow \rho[Cp \mapsto (Cr, \zeta_1), Cr \mapsto (\pi_2, \zeta_2 \cup \{Cp\})], J, P - \{Cp\}}$$

Or.move(Pr) : Le parent courant de *Or* doit être le même composite ou composant que le parent de *Pr*. *Or* doit être dans le masque de point de jonction et ne doit pas avoir été déplacé.

$$\frac{Or \in J \quad \pi_2 = \pi_3}{\rho[Or \mapsto (\pi_1, \zeta_1), Pr \mapsto (\pi_2, \zeta_2), \pi_1 \mapsto (\pi_3, \zeta_3)], J, P \vdash Or.move(Pr) \rightarrow \rho[Or \mapsto (Pr, \zeta_1), Pr \mapsto (\pi_2, \zeta_2 \cup \{Or\}), \pi_1 \mapsto (\pi_3, \zeta_3 - \{Or\})], J - \{Or\}, P}$$

Après l'analyse, les conditions suivantes devront être respectées :

- Pour un objet déclaré dans le masque de point de jonction, si l'on considère un élément (π_0, ζ_0) dans l'environnement de départ, il n'y a aucune contrainte sur π . Par contre, ζ ne peut être vide que si ζ_0 est vide lui aussi. L'idée est que si aucun fils ne correspond à l'origine alors nous pouvons supposer qu'il existe tout de même un fils au départ qui n'est pas capturé par le masque, ce fils sera nécessairement présent à la fin de la transformation. Au contraire, si un élément est capturé et que ζ_0 n'est pas vide, cet élément peut-être le seul fils de cet objet (structure arborescente de ρ). Donc si l'ensemble ζ est vide à la fin de la transformation, alors potentiellement la structure arborescente de l'architecture ne sera pas respectée.
- Pour un composite dans le plan ou créé par la transformation possédant une entrée (π, ζ) dans l'environnement final, ζ ne peut pas être vide.
- Pour un composant primitif dans le plan possédant une entrée (π, ζ) dans l'environnement final, ζ ne peut pas être vide, π ne peut-être *None*.
- Pour tout port primitif ayant une entrée (π, ζ) dans l'environnement final et déclaré dans le plan ou créé au sein des règles de transformation, π ne peut pas être *None* par construction et ζ ne peut-être vide.
- Pour tout port délégué ayant une entrée (π, ζ) dans l'environnement final et déclaré dans le plan ou créé au sein des règles de transformation, π ne peut pas être *None* par construction et ζ doit être vide.
- Pour toute opération ayant une entrée (π, ζ) dans l'environnement final et déclaré dans le plan ou créé par transformation, π ne peut être *None* par construction.

Dans les règles présentées ci-dessus, ρ , J et P représentent l'environnement et les listes utilisés tout au long de l'analyse. Cet environnement et ces listes sont mis à jour au cours de l'analyse.

La clause d'arrêt a la forme suivante : $\rho, J, P \vdash x \mapsto (\pi, \zeta)$, où ρ , J and P sont l'environnement et les listes utilisés au cours de l'analyse et $x \mapsto (\pi, \zeta)$ est un élément de l'environnement final ρ . Chaque élément de cet environnement final doit satisfaire les clauses suivantes :

$$\begin{aligned} & \frac{x \in J \quad \rho(x) = \emptyset}{\rho, J, P \vdash x \mapsto (\pi, \emptyset)} \\ & \frac{x \in J \quad \zeta \neq \emptyset}{\rho, J, P \vdash x \mapsto (\pi, \zeta)} \\ & \frac{x \notin J \quad (x \in Cr \vee x \in \zeta) \quad \zeta \neq \emptyset}{\rho, J, P \vdash x \mapsto (\pi, \zeta)} \\ & \frac{x \notin J \quad x \in Pr \quad \pi \neq None \quad \zeta \neq \emptyset}{\rho, J, P \vdash x \mapsto (\pi, \zeta)} \\ & \frac{x \notin J \quad x \in Or \quad \pi \neq None}{\rho, J, P \vdash x \mapsto (\pi, \emptyset)} \end{aligned}$$

6.4.2.5 Les connexions

Nous considérons les connexions séparément, du fait du type et de la particularité de ces structures. Les règles associées à l'analyse des connexions sont les suivantes :

- chaque port dans l'architecture est connecté à un autre port par l'intermédiaire d'une connexion.
 - quand il existe un port libre (au sens non connecté) dans le plan, les règles de transformation le connectent à un autre port ;
 - quand une règle de transformation casse une connexion entre deux ports p_1 et p_2 , une autre règle de transformation devra connecter ces deux ports à d'autres ports ;
 - quand les règles de transformation créent un nouveau port, une autre règle de transformation doit le connecter à un port disponible ;
- les parents de deux ports connectés n'appartiennent pas nécessairement au même composite. Par contre ces parents sont différents. De plus ces parents ne sont pas des descendants directs l'un par rapport à l'autre,
- les opérations définies dans deux ports connectés doivent être compatibles. Les opérations capturées à l'aide du caractère * du masque de point de jonction sont incompatibles avec toute autre opération,
- comme pour les autres éléments, une connexion ne peut-être créée pour être détruite au cours de la même transformation.

L'analyse des connexions démarre de l'environnement PE calculé dans l'analyse précédente. PE contient l'ensemble des entrées de ρ à la fin de l'analyse précédente à l'exception des ports qui sont retirés au cours de la transformation. L'analyse définit aussi une liste de ports non connectés UP. Cette liste est initialisée par l'ensemble des ports du nouveau plan non initialement liés. Aucun port du masque de point de jonction n'est ajouté à UP. Le masque de point de jonction ne contient pas suffisamment d'information pour savoir si un port du masque non lié sera effectivement non lié au niveau du plan de base. Enfin, l'analyse définit une liste BE contenant la liste des connexions et les ports qu'elles relient. L'analyse se préoccupe autant des connexions du plan et du masque de point de jonction que des connexions créées au sein de la transformation.

Les opérations pertinentes sont les suivantes :

Port p p est ajouté à UP.

p.destroy() p doit être un élément de UP. Il est alors supprimé de UP. Il avait été supprimé de ρ lors de l'analyse sur le contenu.

Binding b = {p1,p2} p1 et p2 doivent être deux éléments de UP. Ils sont alors retirés de UP. Si l'on prend $PE(p1) = (\pi_1, \zeta_1)$ et $PE(p2) = (\pi_2, \zeta_2)$. π_1 et π_2 doivent être différents. ζ_1 et ζ_2 doivent être compatibles. Il doit exister une correspondance f de ζ_1 à ζ_2 tel que pour toute opération o, f(o) a une signature compatible et une polarité opposée (voir Section 4.3.1.2). Une entrée $b \rightarrow (p1,p2)$ est alors ajoutée à BE.

b.destroy() b doit être une connexion définie dans le masque de point de jonction. Il doit exister $BE(b) = (p1,p2)$. L'entrée pour b est alors supprimée et p1 et p2 sont ajoutées à UP.

A la fin de l'analyse, UP doit être vide.

6.4.2.6 Conclusion sur l'analyse statique

L'utilisation d'un langage dédié pour la spécification des règles de transformation et l'analyse statique du patron d'architecture permet de garantir la conformité d'une architecture résultante par rapport au modèle de composant. De plus, le masquage de la hiérarchie au niveau de l'expression des règles de transformation annihile le risque de création de connexion traversant la membrane d'un composite, le tisseur ayant la charge de la création des ports délégués adéquats. La cohérence entre la structure et le comportement au sein de l'architecture est garantie à l'aide de deux mécanismes : une mise à jour automatique des noms de messages au sein de l'automate pour les

primitives de déplacement d'une opération et de remplacement des opérations et une analyse des modifications apportées au contrat de comportement pour les opérations liées à l'ajout d'une opération à un port. L'analyse statique des modifications apportées au contenu de l'architecture et l'analyse des modifications apportées aux connexions permettent principalement de garantir que la transformation va préserver la conformité par rapport au modèle de composant.

Cette analyse statique effectuée uniquement à partir des informations contenues dans le patron d'architecture permet de s'affranchir de l'analyse de la conformité du résultat d'une transformation face au modèle de composant. Les principaux risques résident maintenant sur l'apparition d'une incompatibilité structurelle ou comportementale au niveau de l'architecture logicielle.

6.4.3 Vérifications dynamiques

Les vérifications dynamiques visent à garantir, au moment de l'intégration d'une préoccupation au sein d'une architecture logicielle donnée, que les modifications liées à cette intégration n'engendrent pas d'incompatibilités structurelles ou comportementales.

6.4.3.1 Vérification structurelle

La vérification structurelle est simple. Elle se base sur la définition de la compatibilité entre types de port dans SafArch (voir Section 4.3.1). L'analyse des règles de transformation et, plus particulièrement, l'analyse des connexions permet de créer une liste *NP* de connexions. Cette liste regroupe l'ensemble des connexions nouvellement créées et les ports qu'elles relient ainsi que l'ensemble des connexions dont au moins l'un des deux ports a été modifié au cours de la transformation. Pour chacune de ces connexions, nous vérifions la compatibilité structurelle entre les types des ports reliés. Seules ces connexions doivent être vérifiées, les autres connexions ayant en effet été vérifiées au préalable soit au moment de l'analyse du nouveau plan à insérer (voir Section 6.4.2.1), soit au moment de l'analyse du plan de base (voir Section 4.3).

6.4.3.2 Vérification du comportement

L'analyse de l'impact des modifications du comportement est plus difficile à mettre en évidence. En effet, une modification d'un contrat de comportement peut entraîner un interblocage au niveau d'un assemblage d'un composite englobant à un niveau de hiérarchie beaucoup plus élevé. Par exemple, dans la figure 6.19, la transformation d'une architecture logicielle possédant trois composants primitifs (A, B, D) et deux composites (C et E) consiste à ajouter un composant (F) pour interagir avec (B). L'interface de (B) est alors modifiée pour permettre son interaction avec (F). Comme la modification du comportement de B et l'introduction de F modifient le comportement de C, il peut a priori entraîner des erreurs au niveau de l'assemblage de (E) et récursivement si E appartenait lui-même à un autre composite. L'objectif au niveau de l'analyse du comportement de TranSAT consiste à déterminer sous quelles conditions il est possible de ne pas propager l'analyse du comportement à tous les assemblages d'une architecture logicielle.

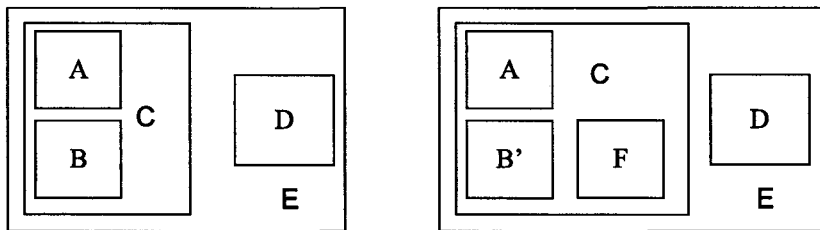


FIG. 6.19 – Exemple d'architecture

Règles

Pour déterminer si le comportement d'un composite et les propriétés de sûreté associées à ce composite ont besoin d'être à nouveau calculés, nous identifions trois cas qui dépendent de l'évolution de l'alphabet de synchronisation, noté A , et de l'évolution des automates des comportements des composants du composite notée $(E_1, \dots, E_n)$:

- cas d'un composite dont l'alphabet de synchronisation a été modifié ($A! = A'$) : toute introduction de messages ou suppression de messages au sein d'un automate participant à un assemblage nécessite de vérifier la synchronisation de ces automates et nécessite la vérification des propriétés de sûreté exprimées au niveau du composite en charge de cet assemblage,
- cas d'un composite dont l'alphabet de synchronisation reste identique et dont aucun des comportements de ces fils n'a été modifié ($A! = A' \wedge \forall i \in \{1..n\}, E_i = E'_i$) :
Dans ce cas évidemment, le comportement du composite reste identique et n'a pas besoin d'être à nouveau calculé,
- cas d'un composite dont l'alphabet de synchronisation reste identique, mais dont l'un des comportements des composants du composite est modifié ($A! = A' \wedge \exists i \in \{1..n\}, E_i! = E'_i$). C'est le cas classique des composites participant à un assemblage, dont le contenu est modifié par la transformation mais pas l'interface. Dans l'exemple de la figure 6.19, le contenu de C est modifié, mais pas son interface. Dans ce cas, la synchronisation sera à nouveau effectuée si et seulement si l'un des comportements modifiés n'est pas « équivalent » à son comportement de départ. Il existe de très nombreuses relations d'équivalence au niveau du comportement de deux processus [van Glabbeek, 2001]. Leur pertinence dépend fortement de l'utilisation faite de ces automates.

Équivalence comportementale

Dans SafArchie, la modélisation de comportement définit un comportement indéterministe. Du fait de cet indéterminisme, deux comportements de composants sont dits équivalents dans SafArchie si ils sont fortement bisimilaires.

Définition : Soient $E_1 = (S_1, L_1, \Delta_1, s_{0_1})$ et $E_2 = (S_2, L_2, \Delta_2, s_{0_2})$ deux systèmes de transitions étiquetées avec $L_1 = L_2$. Une relation $R \subseteq S_1 \times S_2$ est une bisimulation forte si et seulement si :

$$\begin{aligned}
 & (\forall (s_1, s_2) \in R) (\forall a \in L_1 \cup \{\tau\}) \\
 & ((s_1, a, s'_1) \in T_1 \Rightarrow \exists s'_2 \mid (s_2, a, s'_2) \in T_2 \wedge (s'_1, s'_2) \in R) \wedge \\
 & ((s_2, a, s'_2) \in T_2 \Rightarrow \exists s'_1 \mid (s_1, a, s'_1) \in T_1 \wedge (s'_1, s'_2) \in R)
 \end{aligned}$$

Deux états sont dits fortement bisimilaires lorsqu'ils ont la possibilité d'exécuter les mêmes actions, et que chaque état atteint après l'exécution d'une action à partir de l'un des deux états est fortement bisimilaire à un état atteint après l'exécution de la même action à partir de l'autre état. Deux systèmes de transitions étiquetées $E_1 = (S_1, L_1, T_1, s_{0_1})$ et $E_2 = (S_2, L_2, T_2, s_{0_2})$ sont dits fortement bisimilaires, on note $E_1 \sim E_2$, si et seulement si il existe une bisimulation forte $\mathfrak{R}$ telle que $(s_{0_1}, s_{0_2}) \in \mathfrak{R}$.

Les différentes règles associées à l'analyse de l'impact des modifications comportementales sur la compatibilité permettent de limiter la portée des vérifications comportementales à effectuer sur une architecture transformée. Cependant, la contrainte liée à la définition de l'équivalence de deux automates est forte et ne pourra pas toujours être assurée. En outre, l'ajout d'une connexion entre deux composants appartenant à des composites différents modifie les alphabets de synchronisation de nombreux composites. Nous souhaitons, au niveau des perspectives sur l'analyse dynamique de l'impact des modifications sur le comportement, travailler sur une caractérisation plus fine des transformations liées au comportement. Cette caractérisation vise à pouvoir statiquement, dans certains

cas, garantir la compatibilité comportementale uniquement à partir des informations contenues au sein du patron d'architecture.

6.4.4 Conclusion sur l'analyse dynamique

Grâce aux différentes analyses réalisées statiquement sur le patron d'architecture, les vérifications dynamiques se limitent à une étude de la compatibilité structurelle entre ports nouvellement connectés et une étude de l'impact des modifications comportementales sur la synchronisation des automates de comportement des composants de l'architecture. Au niveau des perspectives, l'objectif consiste à continuer à travailler sur l'analyse statique afin de limiter au maximum les analyses à effectuer dynamiquement au moment de la transformation.

6.5 Synthèse

TranSAT permet l'intégration de nouvelles préoccupations au sein d'une description d'architecture logicielle. Il définit une nouvelle entité appelée plan d'architecture. Ce plan correspond à un assemblage de composants en charge d'une préoccupation. TranSAT définit un opérateur de tissage permettant de composer ces plans pour produire l'architecture complète du système. Ce mécanisme de composition est intrusif, il ne respecte pas l'encapsulation des composants et peut modifier à la fois les interactions entre les composants et le comportement des composants. L'action associée au tissage de deux plans est décrite à l'aide d'un langage dédié au sein des règles de transformation. Enfin, TranSAT permet l'intégration d'un même plan au sein de différents plans de base en utilisant les mêmes règles de transformation. Pour cela, il introduit la notion de masque de point de jonction qui spécifie un ensemble de contraintes sur le lieu d'intégration sur lequel le plan peut être tissé. Dès lors, les règles de transformation sont décrites uniquement en fonction des éléments du plan à intégrer et du masque de point de jonction. Les trois informations : plan à intégrer, masque de point de jonction, règles de transformation sont regroupées pour constituer un élément d'architecture réutilisable : le patron d'architecture (Section 6.2). La construction d'une architecture logicielle se fait donc à deux niveaux de granularité. Sur le premier, les composants liés à une préoccupation sont assemblés pour la prendre en charge. Dans un deuxième temps, l'architecture logicielle est construite par applications successives de patrons sur un plan de base.

TranSAT fournit à l'architecte différents langages dédiés pour spécifier le masque de point de jonction et les règles de transformation (Sections 6.2.2 et 6.2.3). La précision du langage de règles de transformation fournit à l'architecte un langage possédant un pouvoir d'expression important travaillant à partir des concepts qu'il souhaite exprimer. Le deuxième avantage lié à l'utilisation d'un langage spécifique au pouvoir d'expression volontairement limité est de permettre d'effectuer des analyses statiques plus poussées des programmes qu'avec un langage de transformation généraliste.

TranSAT définit un processus autour du canevas de conception (Section 6.3). Ce processus permet de distribuer le travail lié à l'utilisation de TranSAT entre différents acteurs. Cette séparation des rôles lors de la construction d'une architecture logicielle est sinon très difficile.

TranSAT propose un ensemble de vérifications statiques et dynamiques pour garantir la cohérence de l'architecture résultante de la transformation (Section 6.4). Ainsi, les vérifications statiques menées sur le patron permettent de garantir un certain nombre de propriétés uniquement à partir des informations contenues dans le patron (Section 6.4.2) : par exemple, il n'y pas de ports qui se retrouvent déconnectés après la transformation, aucun port ne peut se retrouver connecté deux fois, ou il n'y a pas d'oubli de délégation d'interfaces de composants lors de la création d'une nouvelle interaction. Ces vérifications sont effectuées une fois pour toutes et ne doivent pas être effectuées pour chaque nouveau contexte d'intégration. Dans TranSAT, les vérifications dynamiques sont réduites au minimum en tenant compte des modifications apportées au plan de base (Section 6.4.3). Cette limite permet d'éviter de devoir vérifier la totalité d'une architecture logicielle dont seulement une partie a été modifiée. L'ensemble de ces vérifications nous permet en garantir en sortie du processus de transformation une architecture logicielle cohérente sans revérifier la globalité de l'architecture logicielle.

Enfin, TranSAT est un bon complément à SafArchie. Le couple TranSAT/SafArchie fournit en effet un modèle de construction d'architecture logicielle pour maîtriser la complexité de ces architectures et favorisant la réutilisation afin de diminuer les coûts de production d'une architecture logicielle. Ainsi, SafArchie permet d'offrir un premier niveau de **modularité**, une bonne **abstraction** par rapport au système et un mécanisme d'**anticipation des changements** internes à l'architecture. TranSAT, quant à lui, se concentre sur la définition d'un **deuxième niveau de modularité** autour du concept de patron. Il fournit en outre un processus de **construction incrémentale** de description d'architectures par intégrations successives de nouvelles préoccupations. En outre, le couple TranSAT/SafArchie favorise la **réutilisation** grâce aux différentes vérifications qui permettent de garantir la correcte utilisation d'un composant ou d'un patron pour un contexte différent de celui pour lequel ils ont été initialement conçu.

Chapitre 7

Concrétisation de l’approche : un atelier de construction d’architectures logicielles

PRÉAMBULE

Dans ce chapitre, nous présentons l’atelier de génie logiciel appelé SafArchie Studio développé au cours de cette thèse. Cet outil nous sert de plate-forme d’évaluation pour les modèles et les analyses associés à ce travail. Ce chapitre définit et justifie les choix technologiques et architecturaux retenus pour la mise en œuvre de ces multiples modèles, leur analyse et la production de code à partir de ces modèles.

Sommaire

7.1	Introduction et motivations	161
7.2	SafArchie Studio : une architecture modulaire	162
7.2.1	Présentation générale de SafArchie Studio	162
7.2.2	Étendre ArgoUML	165
7.2.3	Vérification de la cohérence	174
7.2.4	Génération de code	177
7.3	TranSAT	180
7.3.1	Trois nouveaux modules pour la définition de nouveaux patrons d’architecture	181
7.3.2	Le tisseur, un moteur de transformation pour intégrer des nouvelles préoccupations dans une description d’architecture logicielle	183
7.4	Bilan et Perspectives sur SafArchie Studio	185
7.4.1	Bilan qualitatif	185
7.4.2	Bilan quantitatif	185
7.4.3	Perspectives	185

7.1 Introduction et motivations

La dernière contribution de cette thèse concerne le développement d’un atelier de génie logiciel SafArchie Studio. L’objectif du projet SafArchie Studio consiste à proposer une chaîne d’outils pour la construction, la transformation et la vérification d’une architecture logicielle. Face à la difficulté d’évaluer quantitativement les propositions liées à cette thèse, SafArchie Studio permet de concrétiser l’ensemble des propositions autour de SafArchie et de TranSAT.

Ce chapitre présente, tout d'abord, l'architecture de SafArchie Studio, motive le choix de mise en œuvre autour d'ArgoUML et décrit les mécanismes et les technologies utilisés pour la création de l'atelier de génie logiciel. L'accent est mis par la suite sur la mise en œuvre de la vérification de la cohérence et la génération de code vers des modèles de composant existants. Finalement, nous présenterons comment TranSAT s'intègre dans SafArchie Studio pour proposer un processus de construction incrémentale de nouvelles descriptions d'architecture logicielle.

7.2 SafArchie Studio : une architecture modulaire

SafArchie Studio est construit sur une architecture modulaire. Ainsi, trois modules étendent l'atelier de génie logiciel ArgoUML pour permettre la définition de nouveaux types d'architecture, de nouvelles architectures logiques et physiques. La section 7.2.1 décrit l'architecture de SafArchie Studio centrée sur ces trois modules. La section 7.2.2 présente les technologies utilisées pour la construction de ces différents modules pour produire un atelier autour des différents méta-modèles définis dans SafArchie. Enfin, la section 7.2.3 détaille la mise en œuvre des différents analyseurs qui nous permettent de garantir la cohérence d'un assemblage de composants. Finalement, la section 7.2.4 présente les mécanismes de génération de code qui permettent de produire les squelettes d'une application Fractal ou ArchJava à partir d'une spécification SafArchie.

7.2.1 Présentation générale de SafArchie Studio

7.2.1.1 Étendre un atelier de génie logiciel existant

Le premier choix que nous avons eu à faire pour la construction d'un atelier de génie logiciel à base de composants a été d'étendre un atelier existant ou d'en construire un nouveau. La construction d'un nouvel atelier permet d'éviter de subir des choix techniques dictés par l'existant. Cependant, l'intégration d'une nouvelle application dans un environnement existant est un cadre classique de réutilisation en génie logiciel. Cela permet un gain de productivité accrue. Le choix de proposer un prototypage de nos travaux autour d'ArgoUML peut donc se résumer en cinq points :

- ArgoUML est un atelier de génie logiciel reconnu. Son but est de guider l'utilisateur dans l'utilisation du logiciel et de la notation UML au travers d'un mécanisme de critiques et d'assistants,
- ArgoUML est basé sur une bibliothèque permettant l'édition de graphiques vectoriels. L'existence d'une telle bibliothèque et son intégration au cœur d'ArgoUML sont deux critères intéressants pour la description des différents modèles d'architecture logicielle. En effet, SafArchie propose une charte graphique pour la description d'une architecture logicielle. Or, bénéficier de cette représentation au niveau de la construction de cette architecture facilite le travail de l'architecte qui bénéficie directement d'une vue abstraite plus facilement lisible qu'une spécification textuelle,
- le troisième point est d'ordre technique. Développé en Java et proposant un mécanisme de système de chargement de modules permettant à quiconque de développer du code pour ArgoUML et de le diffuser séparément, ArgoUML est facile à prendre en main et à étendre. De plus, grâce à la disponibilité de la machine virtuelle Java sur la plupart des systèmes d'exploitation existants, ArgoUML est *de facto* multi plates-formes,
- contrairement à Eclipse [EclipseWWW, 2001 2005] qui propose aussi un mécanisme d'extension, ArgoUML est avant tout centré autour des modèles alors qu'Eclipse est centré autour du code source de l'application. Cette différence de philosophie qui tient à s'estomper avec Eclipse 3.X se traduit au niveau d'ArgoUML par la possibilité pour l'utilisateur d'être guidé par des assistants et des critiques tout au long de la phase de conception,
- enfin, le dernier point est plus philosophique. Placer sous licence BSD, ArgoUML est un projet open source dynamique. Les sources d'ArgoUML étant disponibles sur le Web, il est possible d'analyser le fonctionnement et de participer au développement de ce programme. Notre objectif avec SafArchie Studio est de bénéficier des nombreux avantages d'ArgoUML pour bâtir un atelier de génie logiciel centré sur l'architecture logicielle à base de composants.

7.2.1.2 Architecture de SafArchie Studio

ArgoUML est conçu pour la construction de nouveaux modèles UML. Il base son mécanisme d'extension entre autres sur la possibilité de définir de nouveaux modules. Chaque module est associé à un diagramme qui permet de représenter un modèle lié à un méta-modèle. Dans SafArchie, nous proposons la construction de différents méta-modèles (type d'architecture, architecture logique, architecture physique). Dès lors, pour chacun de ces méta-modèles définis dans SafArchie, nous proposons un module de mise en œuvre. L'architecture de SafArchie Studio est illustrée dans la figure 7.1.

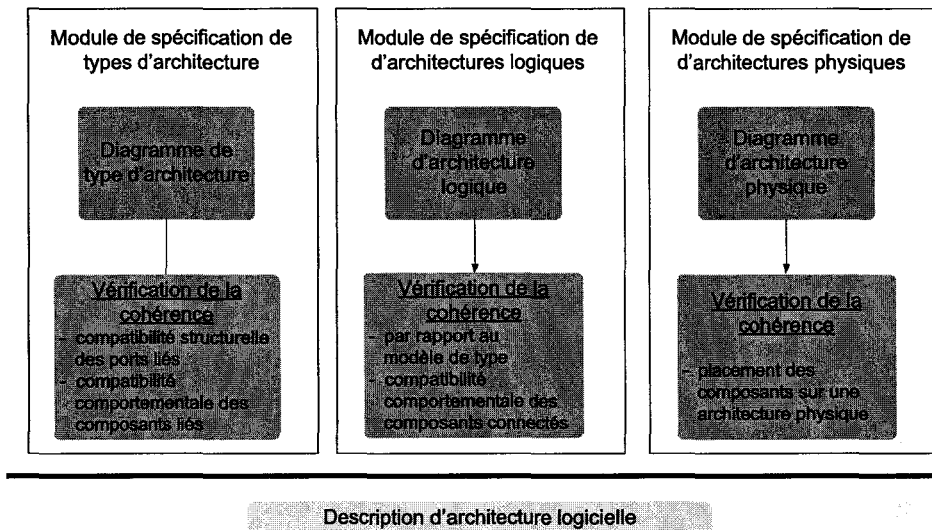


FIG. 7.1 – Trois nouveaux modules pour SafArchie Studio

Pour la partie relative à SafArchie, SafArchie Studio se caractérise par l'ajout à ArgoUML de trois modules. Le premier prend en charge la spécification d'un nouveau type d'architecture, le deuxième d'une architecture logique typée et le troisième d'une architecture physique sur laquelle nous déployons les instances de composant de l'architecture logique. Pour chacun de ces modèles, nous développons un outil d'analyse de la cohérence qui vérifie respectivement :

- pour le modèle de type, la compatibilité structurelle entre types de port liés et la compatibilité comportementale entre types de composant liés (voir Section 4.3.1 et 4.3.2),
- pour le modèle logique, la conformité avec le modèle de type auquel il fait référence et la compatibilité comportementale entre composants connectés (voir Section 5.1),
- pour le modèle physique, l'adéquation entre le modèle physique et les besoins de connectivité entre composants du modèle logique (voir Section 5.2).

7.2.1.3 Éléments caractéristiques de SafArchie Studio

L'espace de travail de SafArchie Studio est calqué sur celui d'ArgoUML. Il est composé de quatre panneaux principaux appelés *vues* (voir Figure 7.2). Ces quatre panneaux proposent quatre points de vue différents sur le modèle en cours d'édition : modèle UML pour ArgoUML ou modèle d'architecture pour SafArchie Studio.

- le premier panneau situé en haut à gauche, nommé *NavigatorPane*, permet d'obtenir une vue arborescente d'un modèle (dans notre cas un type d'architecture ou un modèle d'architecture). L'arbre permet en outre de manipuler le modèle grâce à quatre types d'actions :
 - un menu contextuel,

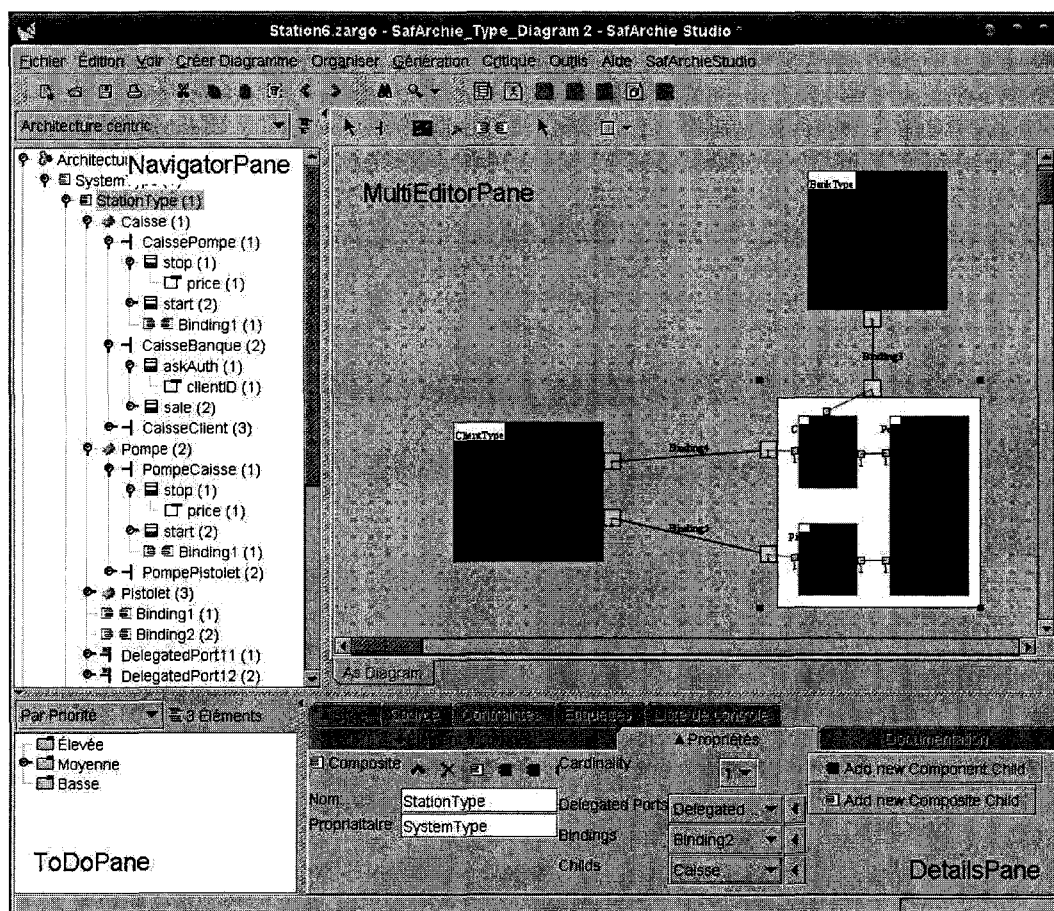


FIG. 7.2 – Copie d'écran de SafArchie Studio sur l'exemple de la station

- une action *glisser/déposer* avec une sémantique de liaison (bouton gauche). Cette action permet de lier une déclaration de type de port ou de composant à un autre composite,
- une action *glisser/déposer* avec une sémantique de déplacement (bouton droit). Cette action permet de déplacer un port ou un composant dans un autre composite,
- une action *glisser/déposer* avec une sémantique de copie (bouton gauche). Cette action permet de copier la définition d'un élément du modèle à un autre nœud du modèle.
- le deuxième panneau, positionné en haut à droite et nommé *MultiEditorPane*, est le tableau de conception proprement dit. Composé d'un ensemble de calques, il permet la construction d'un graphe connecté. Nous associons à chaque élément un menu contextuel pour faciliter la construction de l'architecture,
- le troisième panneau, situé en bas à gauche et nommé *ToDoPane*, permet de faire apparaître un certain nombre de conseils ou de critiques à l'architecte. Les remarques sont classées par ordre de priorité, ce classement permet d'ordonner un ensemble de feuilles de route pour l'architecte,
- enfin, le quatrième panneau, situé en bas à droite et nommé *DetailsPane*, offre une vue d'édition sur les propriétés de l'élément du modèle en cours de sélection.

La cohérence entre ces quatre panneaux est assurée. Toute sélection dans l'arbre, dans le diagramme ou au niveau du panneau d'édition de propriétés permet de choisir les objets équivalents sur les autres panneaux.

Une barre d'outils spécifique au type de diagramme en cours d'édition est disponible au-dessus du panneau nommé *MultiEditorPane*.

La sauvegarde d'un projet dans SafArchie Studio se fait au sein d'un ensemble de fichiers XML définis par différents schémas XML. Ces fichiers sont regroupés dans une archive compressée (.zargo).

7.2.2 Étendre ArgoUML

Pour comprendre plus en détail l'infrastructure logicielle associée à SafArchie Studio, nous présentons dans cette section l'architecture d'ArgoUML centrée sur le motif de conception « Modèle-Vue-Contrôleur » et détaillons comment chaque nouveau module qui constitue SafArchie Studio est produit à partir des différents méta-modèles présentés dans les chapitres 4 et 5.

7.2.2.1 Une architecture centrée autour du patron *Modèle-Vue-Contrôleur*

L'architecture d'ArgoUML est centrée autour du motif de conception « Modèle-Vue-Contrôleur »[MVC, 1979] (voir Figure 7.3). Dès lors, pour chaque nouveau diagramme, une couche « Modèle » contient l'ensemble des informations relatives à ce diagramme : propriétés des éléments, actions possibles sur ces éléments, interactions entre ces éléments. Une couche « Vue » prend en charge la représentation de ce modèle pour l'utilisateur. ArgoUML propose trois vues sur le modèle, une vue arborescente, une vue graphique et une vue d'édition des propriétés. Enfin, une couche « Contrôleur » gère des événements de synchronisation pour mettre à jour la vue ou le modèle. Dans ArgoUML, le modèle ne connaît pas explicitement ses vues et ses contrôleurs. Chaque vue connaît son modèle. Cette liaison directe entre la vue et le modèle se justifie par le fait qu'il y a toujours une relation *un à un* entre un élément de la couche vue et un élément de la couche modèle. Enfin, chaque « Contrôleur » connaît son « Modèle » et sa « Vue ». Le « Contrôleur » a la charge du maintien de la cohérence entre les vues quand le modèle est modifié. Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose.

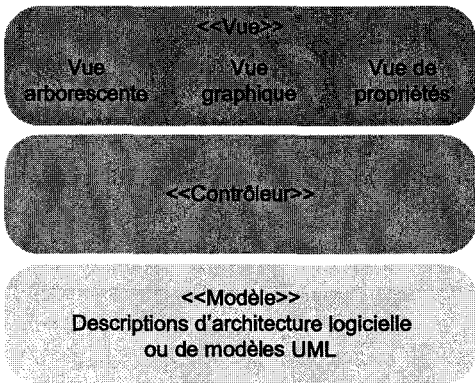


FIG. 7.3 – Modèle MVC de ArgoUML

7.2.2.2 Les mécanismes d'extension d'ArgoUML

ArgoUML propose un mécanisme d'extension basé sur la notion de module. Le fonctionnement des modules est relativement simple. Un fichier de configuration indique de manière absolue à l'atelier le chargement d'une classe au démarrage d'ArgoUML. Cette classe doit alors étendre l'interface *ArgoModule* (voir Figure 7.4). Cette classe est automatiquement instanciée et la méthode *initializeModel()* est appelée par l'atelier pour permettre la configuration du module. Le module est alors chargé. Il est possible de mettre en œuvre une interface qui étend l'interface *ArgoModule* afin d'intégrer au chargement de nouvelles entrées dans les menus de l'atelier ou des nouveaux éléments dans la page de préférences (Voir Figure 7.4).

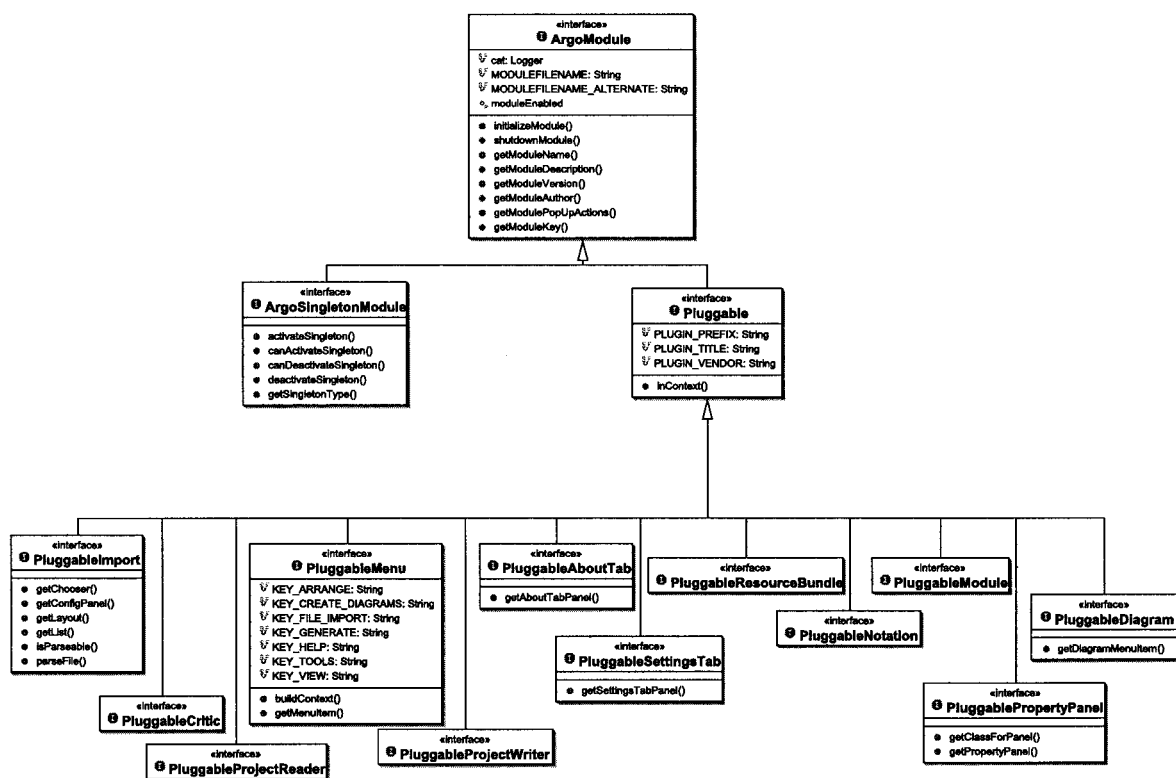


FIG. 7.4 – Interface d'extension d'ArgoUML

Une fois cette classe convenablement mise en œuvre, il est possible grâce à un motif de conception « Singleton » de récupérer une référence sur l'objet *ProjectManager*, qui gère l'atelier, et sur l'objet *Project*, qui contient les données relatives au projet courant, comme par exemple le diagramme en cours d'édition. À partir de là, quand l'utilisateur effectue une demande de création d'un nouveau diagramme, celui-ci est instancié et référencé au niveau du projet courant.

Ce mécanisme d'extension permet d'intégrer facilement de nouveaux modules dans ArgoUML. Néanmoins, pour chaque nouveau module, il est nécessaire de créer une nouvelle couche « Contrôleur », une nouvelle couche « Vue » et une nouvelle couche « Modèle ». L'objectif du développement de SafArchie Studio du fait des nombreux modules à mettre en œuvre consiste à limiter le code à développer pour l'introduction d'un nouveau module en charge d'un nouveau méta-modèle au sein d'ArgoUML.

7.2.2.3 Une nouvelle couche « Contrôleur »

Toute l'architecture d'ArgoUML est centrée autour des types d'élément du modèle manipulés par le diagramme en cours de construction et sur les événements associés à ces types d'élément. Il est nécessaire de déclarer l'ensemble des événements qui peuvent être émis par ce type d'élément afin que la couche contrôle puisse facilement faire le lien entre les événements reçus et le type de l'émetteur. L'ensemble de ces déclarations est effectué à l'initialisation du module sur l'appel de la méthode *initializeModule()* (voir Figure 7.4). Ensuite, l'ensemble de l'atelier fonctionne sur un mécanisme d'événements. La sélection d'un élément ou la modification d'une propriété d'un élément au sein de l'éditeur graphique de l'arbre ou d'un panneau d'édition de propriétés émet un événement de sélection ou de mise à jour qui est notifié aux différentes vues par la couche de contrôle. Les différentes vues sont alors mises à jour afin de garder la cohérence entre le modèle et les vues (voir Figure 7.5).

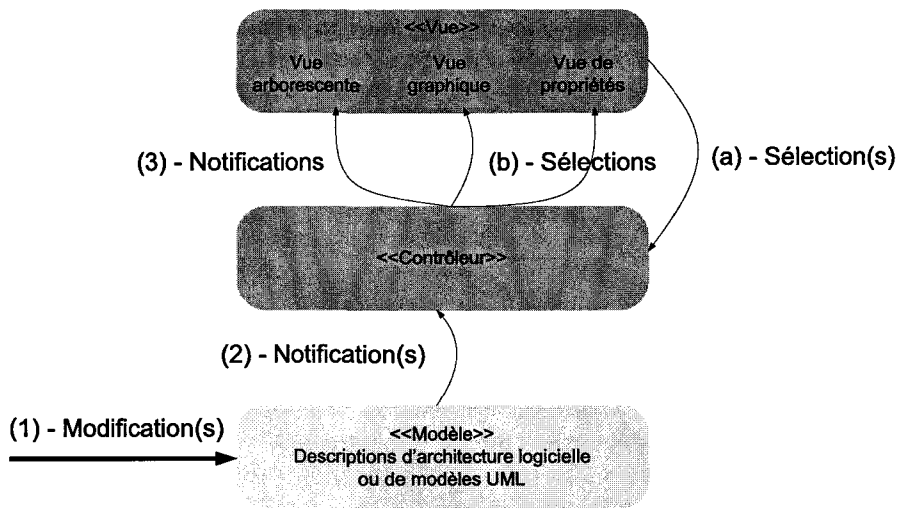


FIG. 7.5 – Un mécanisme d'événements pour la notification de la couche « Contrôleur »

La gestion des événements dans ArgoUML est basée sur un mécanisme d'évènement de la bibliothèque NSUML¹². NSUML a été développée par la société *Novosofts* sous licence GPL. Elle fournit une représentation Java du méta-modèle UML de l'OMG. NSUML fournit surtout un système d'évènements puissant qui permet d'écouter les modifications qui sont appliquées au modèle. Ce système est massivement utilisé par ArgoUML pour propager les informations sur les modifications apportées au modèle aux différents éléments de l'atelier génie logiciel : vue graphique, vue arborescente, et formulaires d'édition de propriétés.

7.2.2.4 Une nouvelle couche « Vue »

ArgoUML offre trois vues différentes du modèle : une vue arborescente, une vue graphique et un formulaire d'édition de propriétés qui permet d'éditer les différentes propriétés d'un élément du modèle. Pour créer SafArchie Studio, il est nécessaire de créer de nouvelles vues pour chacun des types d'élément du modèle.

Créer de nouveaux formulaires d'édition de propriétés et de style

Pour chaque type d'élément d'un méta-modèle que l'on veut voir gérer par ArgoUML (par exemple la notion de type de composant dans SafArchie Studio), il est nécessaire d'associer un panneau d'édition de propriétés qui sera automatiquement affiché lors de la sélection d'un élément de ce type, un panneau de style pour ce type d'élément, ou un panneau de documentation. Ces nouveaux panneaux qui réalisent l'interface *PluggablePropertyPanel* présentée dans la figure 7.4 sont instanciés une fois pour toute à l'initialisation du module sur l'appel de la méthode *initializeModule()*.

Créer une nouvelle vue graphique

Pour la réalisation du panneau *MultiEditor Pane*, ArgoUML utilise la bibliothèque GEF¹³ (*Graphic Editing Framework*). GEF est destinée à représenter des données sous forme d'un dessin vectoriel. Cette bibliothèque fournit outre une API de haut niveau pour la manipulation de dessins vectoriels et la possibilité d'exporter un diagramme en *postscript*, *gif* ou *svg*.

¹²<http://nsuml.sourceforge.net/>

¹³<http://gef.tigris.org/>

GEF fournit au niveau de la couche métier un mécanisme de gestion de graphes basé sur les notions de nœud, de port, et de connexion. Cette prise en compte au niveau de la couche graphique de ces notions se révèle particulièrement intéressante dans la mise en œuvre des diagrammes d'architecture de SafArchie car il existe une relation directe entre certains éléments du modèle métier et des éléments de la couche graphique.

Créer une nouvelle vue arborescente

Il est possible pour créer une nouvelle vue graphique ou de nouveaux formulaires d'édition de propriétés de réutiliser de nombreux objets d'ArgoUML. Cependant, l'arbre fourni par ArgoUML de gestion de projet permettant d'obtenir une vue arborescente sur les modèles en cours de construction est très dépendant du modèle UML de la bibliothèque NSUML. Il est donc difficilement réutilisable. Cependant, ArgoUML laisse la possibilité d'obtenir différentes perspectives au sein de ce panneau de navigation et d'enregistrer de nouvelles perspectives. Dans SafArchie Studio, nous proposons d'ajouter une perspective architecturale. Celle-ci nécessite le développement complet de cette fonctionnalité. Face au manque actuel de cadre dans ArgoUML pour développer un nouvel environnement de navigation arborescente, nous avons choisi au niveau de l'architecture de SafArchie Studio d'intégrer un module du projet de la console d'administration d'OpenCCM développé au sein du projet INRIA Jacquard afin de permettre une configuration fine de la vue arborescente d'ArgoUML. Ce projet, appelé l'*Explorer Framework*, offre l'avantage de pouvoir, à l'aide d'un langage dédié, spécifier finement la représentation arborescente d'un modèle et les actions associées à cette vue arborescente.

L'*Explorer Framework*¹⁴ permet de naviguer facilement sur un ensemble de ressources d'un modèle. Il utilise une représentation arborescente pour la représentation de modèles. La vue graphique de l'arbre permet à l'utilisateur de découvrir dynamiquement la structure d'un modèle et d'interagir avec lui. Cette bibliothèque a été conçue pour être :

- configurable et extensible : l'utilisateur peut ainsi adapter la bibliothèque à son modèle grâce à un fichier de configuration et la définition de quelques classes Java,
- non intrusive : la bibliothèque a été pensée pour n'imposer aucune modification dans la structure et le code du modèle manipulé,
- abstraite : l'utilisateur ne prend pas du tout en charge la gestion du graphisme à part la configuration des icônes des éléments,
- composable : différentes configurations peuvent être associées pour obtenir une vision arborescente regroupant différents modèles.

Architecture de l'explorateur de modèles

La figure 7.6 présente une vue globale sur l'architecture de l'*Explorer Framework*. Pour fonctionner, la bibliothèque a besoin principalement de deux éléments :

- l'implantation de la bibliothèque qui prend en charge les aspects graphiques (Vue arborescente, Glisser & Déposer ou Menus contextuels),
- un ou plusieurs *plug-ins*, dans notre cas un *plug-in* SafArchie Studio qui définit les relations entre les types d'éléments des modèles, les icônes pour chaque type d'élément du modèle et les actions à effectuer en cas de *glisser-déposer* ou de sélection d'actions dans le menu contextuel.

L'intégration de cette bibliothèque dans ArgoUML nous a permis de configurer finement et efficacement les actions au sein de la représentation arborescente d'une architecture logicielle. La principale action liée à cette intégration consiste à réaliser un adaptateur entre le système d'événements de l'*Explorer Framework* et le système d'événement de l'atelier afin qu'une sélection dans l'arbre soit prise en compte comme la sélection d'un élément du modèle et qu'inversement la sélection d'un élément du modèle à l'aide de la vue graphique ou du panneau d'édition de propriétés soit vue comme une sélection d'un élément de l'arbre.

¹⁴http://opencm.objectweb.org/doc/0.9/explorer_plugin.html

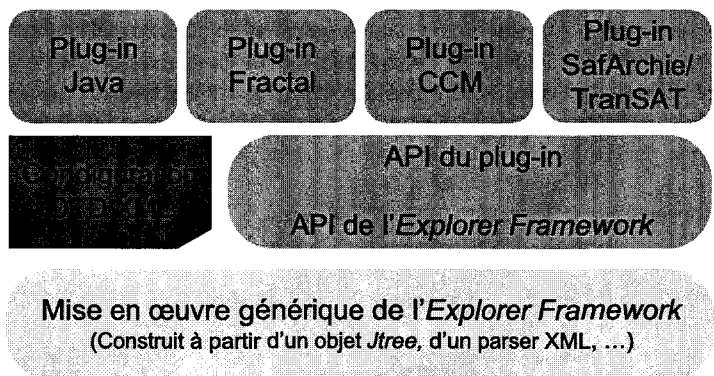


FIG. 7.6 – Architecture de l'Explorer Framework

Réflexions autour de la création d'une nouvelle couche « Vue »

De par son architecture modulaire et du fait de son architecture centrée sur un modèle auquel on associe différents points de vue, ArgoUML se révèle un choix très intéressant. En effet, toute la partie métier de la construction d'une application est alors structurée dans la mise en œuvre du modèle. Le travail de construction de l'atelier se limite alors simplement à un travail d'IHM assisté pour la construction de la vue arborescente par un langage dédié. D'une manière générale, comme il est nécessaire dans ArgoUML de produire un nouveau diagramme pour tout nouveau méta-modèle. Nous souhaitons étendre cette démarche basée sur un langage dédié pour configurer une partie de l'outillage d'un nouveau modèle à la construction de l'ensemble de cet outillage. Ainsi, les panneaux d'édition de propriétés ou la représentation graphique des éléments pourraient être décrits pour pouvoir très facilement créer au-dessus d'ArgoUML un atelier de génie logiciel pour tout type de méta-modèle. Il reste alors la construction du métier de notre application à savoir les APIs de gestion des modèles manipulés par cet atelier.

7.2.2.5 Une nouvelle couche « Modèle »

SafArchie et TranSAT proposent une approche avec de nombreux modèles différents corrélés (type d'architecture, architecture logique, architecture physique, modèle de masque de point de jonction, etc.) et de nombreux formalismes (textuel, XML). Pour chacun de ces modèles, il est nécessaire de permettre son intégration dans ArgoUML, de créer différents analyseurs syntaxiques capables de charger une représentation depuis les différents formalismes proposés, de produire aussi différents traducteurs capables de générer la description de l'architecture logicielle vers le formalisme souhaité par l'architecte. Ces tâches sont par nature répétitives, fastidieuses et propices aux erreurs. Nous avons choisi de centrer notre approche autour de différents mécanismes de génération de code existants afin d'être capable de produire efficacement la couche métier de SafArchie Studio à partir de la représentation de nos méta-modèles sous forme de BNF ou de schéma XML. Nous définissons les objectifs suivants :

- le premier objectif est d'obtenir une uniformisation entre la partie conceptuelle offerte par le méta-modèle, quel que soit le formalisme de représentation utilisé pour ces concepts : langage textuel, fichier XML, graphe d'objets Java,
- le deuxième objectif est de générer des classes qui puissent s'intégrer facilement dans ArgoUML,
- enfin, le dernier objectif est de fournir les mécanismes pour charger directement une représentation textuelle ou XML en un graphe d'objets et sauvegarder un graphe d'objets en une représentation XML en gardant évidemment une bijection entre ces différentes représentations.

Il existe de nombreux travaux qui permettent de réaliser cette correspondance. Nous avons dans SafArchie Studio utilisé massivement le principe d'association de données connues aussi sous le nom de *Data*

Binding et les outils de génération automatique d'analyseurs syntaxiques en particulier un outil appelé *SableCC* [Gagnon and Hendren, 1998] qui permet de générer facilement un analyseur syntaxique, un analyseur lexical et un arbre de syntaxe abstraite en Java à partir de la déclaration de grammaire. Ces technologies nous permettent de générer très rapidement la couche métier intégrable dans *ArgoUML* de chaque méta-modèle présenté dans cette thèse.

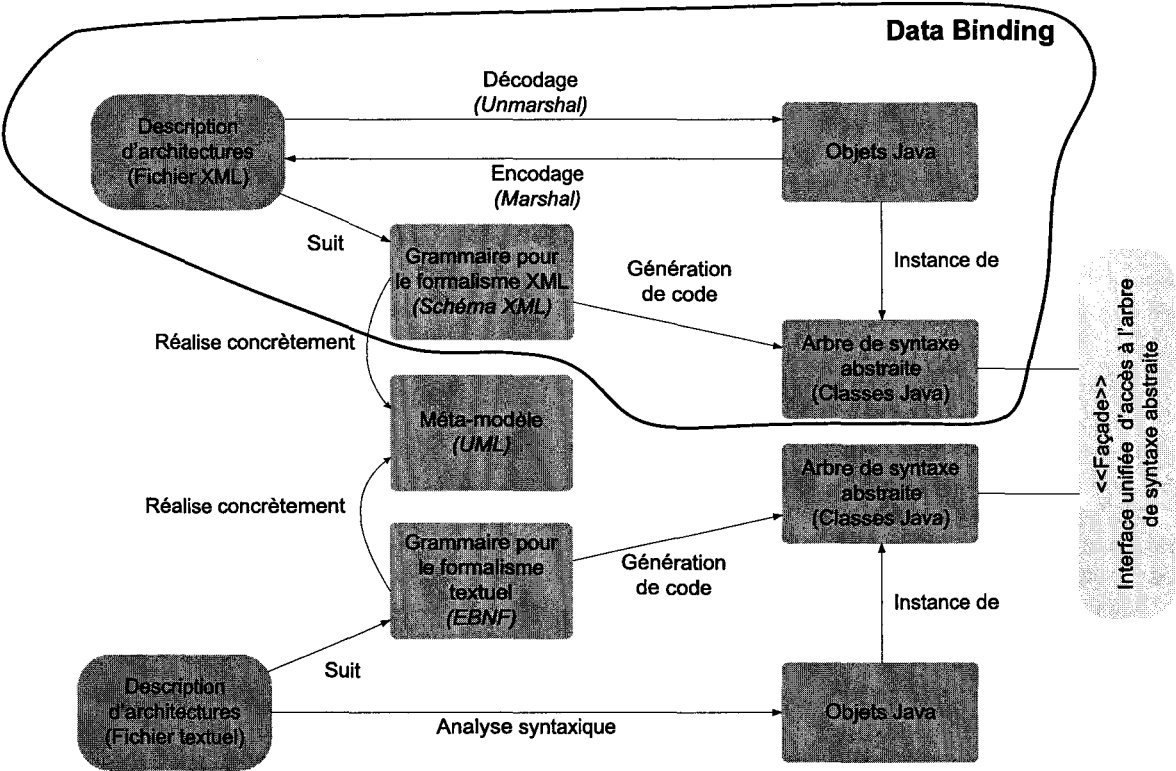


FIG. 7.7 – Mise en œuvre de nouvelles couches métier

JAXB, l'association de données XML/Java

XML est aujourd'hui un format d'échange de données très utilisé. De ce fait, les programmeurs ont réalisé de nombreuses APIs permettant d'accéder aux données XML à travers leurs langages de programmation favoris (par exemple, les APIs *DOM*, *JDOM* et *SAX* en Java). Néanmoins, ces dernières ont des inconvénients. Elles nécessitent l'étude et l'apprentissage d'une nouvelle API. De plus, ces APIs sont trop générales, c'est-à-dire que le programmeur doit adapter le code à son application à chaque fois qu'il veut accéder à des données XML. Finalement, il doit créer lui-même toutes les classes permettant de gérer ces nouvelles données dans son programme. Cette tâche est coûteuse en temps. Pour remédier à ces inconvénients, il existe le *Data Binding* également appelé en français : association de données.

L'association de données est une technologie permettant d'automatiser la transformation d'un modèle de données en un modèle de données objet dans un langage de programmation. Autrement dit, il permet par exemple de convertir les fichiers XML en instances de classes Java. Pour réaliser cela, il y a trois étapes : la génération de classes, l'encodage et le décodage des données.

La partie de la figure 7.7 notée *Data Binding* résume le principe de l'association de données : un document XML suit les règles de grammaire du schéma. Un élément de ce schéma une fois compilé permet de créer une classe

correspondante. Cette dernière permettra de créer une instance d'objet correspondant à l'élément du schéma.

JAXB (*Java Architecture for XML Binding*) est un outil qui analyse un schéma XML et génère à partir de ce dernier un ensemble de classes qui va encapsuler les traitements de manipulation du document. C'est un outil, développé par Sun en Java¹⁵. Le principal avantage est de fournir au développeur un moyen de manipuler un document XML sans connaître XML ou les technologies d'analyse. Toutes les manipulations se font au travers d'objets Java.

L'association de données permet de développer très rapidement des logiciels qui ont besoin d'utiliser des données contenues dans des fichiers XML. Dans notre approche, cette API nous permet de définir toute la logique de chacun des méta-modèles (type d'architecture, architecture logique ou architecture physique) dans un schéma XML et d'obtenir automatiquement une API de gestion de ces modèles. Ainsi, plutôt que de devoir écrire entièrement les différents analyseurs syntaxiques pour nos modèles, nous obtenons directement le moyen de sauvegarder et restaurer une architecture logicielle au sein de notre outil. Le gain en temps de développement est réel, car tout le travail d'analyse conduit à la définition d'un méta-modèle et des différentes grammaires associées à ce méta-modèle. À partir de ces grammaires, tout le travail d'association Java-XML est automatique. Le développeur n'a alors pas besoin de savoir qu'il manipule des données XML.

Intégration de la couche « Modèle » dans ArgoUML

Pour gérer convenablement les éléments de la couche « Contrôleur » et donc s'intégrer correctement au sein d'ArgoUML, chaque élément de modèle a besoin d'hériter de l'objet *MBaseImpl* de la bibliothèque NSUML afin de pouvoir gérer les événements liés à la collaboration entre les différentes vues sur le modèle. En outre, JAXB crée un arbre syntaxique abstrait pour les différents schémas XML modélisant les méta-modèles de l'approche. Les éléments de cet arbre doivent eux aussi être étendus dans l'intégration d'ArgoUML afin de mettre en œuvre certaines méthodes associées à ces objets (la gestion des connexions dans le graphe par exemple). Dès lors pour chaque élément du méta-modèle, une classe de représentation est créée. Cette classe doit étendre à la fois un élément issu de l'arbre syntaxique afin de permettre l'association de données entre la représentation XML et la représentation sous forme de classes mais aussi un élément issu du méta-modèle NSUML afin de permettre l'intégration du nouveau méta-modèle dans ArgoUML. Cependant, comme l'héritage multiple n'existe pas en Java, il n'est pas possible au moment de la mise en œuvre de la classe *ComponentType* de lui faire étendre la classe *ComponentTypeImpl* créée par JAXB et *BaseImpl* du modèle NSUML. En outre, il n'est pas souhaitable d'aller modifier manuellement la classe *BaseImpl* ou les classes générées par JAXB, ce qui rendrait toute modification du méta-modèle extrêmement fastidieuse à propager. JAXB offre un mécanisme d'annotations du schéma XML contenant le méta-modèle à lier avec une représentation sous forme de classes Java qui permet de définir que les classes générées par JAXB héritent elles-mêmes d'une classe particulière (voir Figure 7.8). Cette extension est utilisée dans les différents schémas du modèle pour préciser que ces classes doivent étendre une classe *SafArchieObject* qui permet la prise en charge des événements de modifications du modèle aux différentes vues d'ArgoUML.

```

1 <xsd:annotation>
2 <xsd:appinfo>
3 <jxb:globalBindings generateIsSetMethod="true">
4 <xjc:superClass name="org.argouml.memory.SafArchieObject"/>
5 </jxb:globalBindings>
6 </xsd:appinfo>
7 </xsd:annotation>

```

FIG. 7.8 – Exemple d'annotations pour définir la hiérarchie des objets à l'aide JAXB

En outre, dans le mécanisme d'association de données, le fait de créer des objets qui héritent des objets générés par JAXB permet de passer simplement d'une description d'architecture représentée sous la forme

¹⁵<http://java.sun.com/xml/jaxb/>

d'un graphe d'objet à une représentation sous la forme d'un fichier XML. Cependant, il est nécessaire aussi lors du chargement de ce fichier de créer non pas des instances des classes créées par JAXB, mais des instances des classes étendues permettant l'intégration dans ArgoUML. JAXB propose de nouveau un mécanisme d'annotations du schéma XML qui permet pour chaque élément de ce schéma de préciser quelle classe doit être utilisée lors de la création du graphe d'objet représentant un fichier XML. Ainsi, la figure 7.9 précise que pour les nœuds XML *ComponentTypeModel*, les objets Java à créer doivent être du type *fr.lifl.safarchie.memory.ComponentTypeExtImpl*.

```
1 <xsd:complexType name="ComponentTypeModel">
2   <xsd:annotation>
3     <xsd:appinfo>
4       <jxb:class implClass="fr.lifl.safarchie.memory.ComponentTypeExtImpl"/>
5     </xsd:appinfo>
6   </xsd:annotation>
```

FIG. 7.9 – Exemple d'annotations pour générer une fabrique prenant en compte des extensions des éléments du modèle

L'utilisation conjointe de ces deux types d'annotations du schéma XML correspondant à un nouveau méta-modèle à intégrer permet de considérablement simplifier l'intégration d'un nouveau méta-modèle au sein d'ArgoUML. En effet, cette approche nous permet de générer automatiquement le code lié à la gestion des relations entre entités du méta-modèle, la gestion du code lié à la sauvegarde d'une description d'architecture logicielle vers du code XML et l'analyseur syntaxique permettant d'obtenir un arbre de syntaxe abstraite convenablement intégré dans ArgoUML.

La qualité de code est, en outre, grandement améliorée avec une hiérarchie de classes structurée par rapport aux préoccupations de l'application. Dans notre application, une première couche gère les événements et l'intégration avec ArgoUML. Une deuxième couche prend en charge l'association de données. Finalement, la troisième couche est la partie métier de l'application. Le modèle de classe résultant de cette intégration d'un nouveau méta-modèle dans ArgoUML est alors la suivante pour un élément *ComponentType* du modèle de type (voir Figure 7.10).

SableCC

TranSAT comme SafArchie proposent la mise en place de différents langages textuels pour la spécification d'architectures logicielles. Ainsi, les différents niveaux de contrats sont exprimés à l'aide de nouveaux langages et, pour chaque modèle présenté dans SafArchie (modèle de type, modèle logique, modèle physique), SafArchie propose un formalisme textuel pour décrire une architecture logicielle. Ces langages nécessitent systématiquement la création d'un analyseur permettant de tirer parti de ces éléments de spécification. De nouveau, cette tâche de mise en œuvre devient rapidement dans SafArchie fastidieuse et propice aux erreurs. Pour éviter la construction manuelle d'un nouvel analyseur syntaxique pour chaque langage défini dans SafArchie, nous tirons parti d'un outil appelé SableCC [Gagnon and Hendren, 1998]. SableCC à l'instar de Yacc, CUP, Anagram, Bison ou Beaver permet la construction d'un analyseur en fonction d'une grammaire LALR. SableCC se caractérise par deux choix de conception principaux :

- il utilise le modèle objet pour construire un arbre de syntaxe abstraite typé à partir d'une grammaire définie à l'aide d'une EBNF¹⁶ (*Extended Backus-Naur Form*),
- il est basé une version étendue du motif de conception « Visiteur » qui permet de naviguer facilement dans l'arbre de syntaxe abstraite et permet la mise en œuvre d'actions sur les nœuds de cet arbre.

L'arbre de syntaxe abstraite généré par SableCC est différent de celui généré par JAXB à partir du schéma XML dans le cas de la représentation du même modèle dans différents formalismes. Nous fournissons alors une mise

¹⁶La syntaxe EBNF étend la syntaxe classique BNF de description de grammaire avec les opérateurs *, ? et + qui caractérisent la cardinalité des éléments du langage

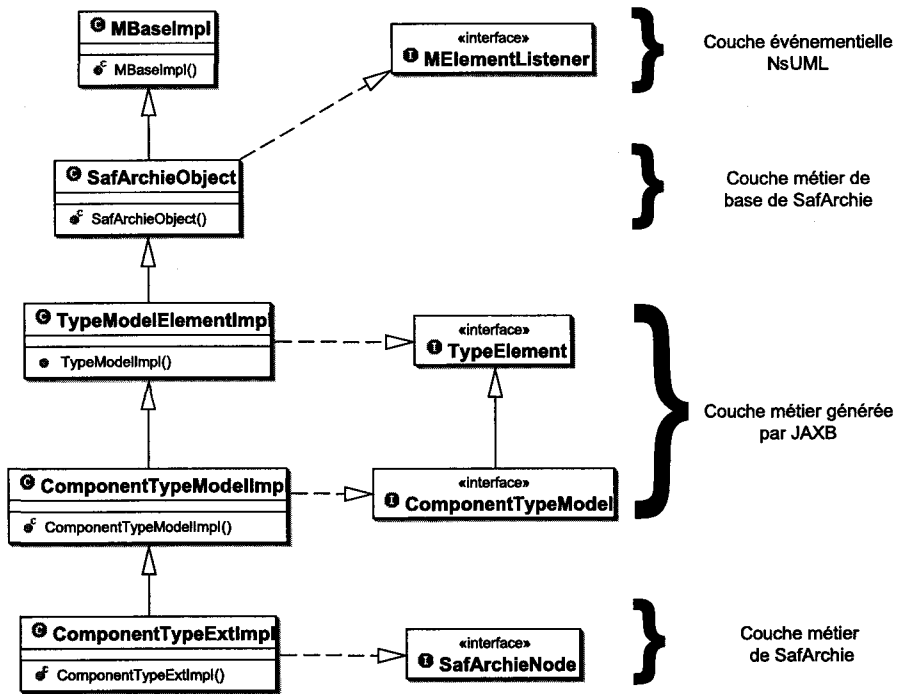


FIG. 7.10 – Hiérarchie de classes pour la couche Métier générée avec JAXB

en œuvre du motif de conception « Façade » pour la navigation au sein du modèle. L'interface de cette façade est issue directement des interfaces de navigations au sein de l'arbre généré par JAXB. Nous fournissons une mise en œuvre de cette façade pour l'arbre de syntaxe abstraite généré par SableCC afin de fournir une interface unifiée de la couche métier, quel que soit le formalisme de description d'architecture utilisé.

7.2.2.6 Discussion autour de la génération automatique de la couche métier

Le choix de représenter les éléments de la couche métier de notre application à partir d'un méta-modèle décrit en UML et de décliner à partir de ce méta-modèle différents formalismes de représentation concrète est attrayant. En effet, le méta-modèle UML permet de représenter de manière abstraite les concepts de notre métier (dans notre cas par exemple les éléments d'un type d'architecture) alors que la grammaire EBNF ou XML schéma précise comment reconnaître et analyser un fichier textuel ou un fichier XML. Le méta-modèle décrit en UML permet ainsi de représenter avec un haut niveau d'abstraction ce que les linguistes appellent l'arbre de syntaxe abstraite. Cependant, il n'y a pas d'association directe entre une représentation UML et une grammaire ou un schéma XML. En effet, UML fournit un certain nombre de constructions qui ne peuvent être convenablement représentées à l'aide d'une EBNF ou d'un schéma XML. Certaines constructions permettent par exemple de définir des langages non réguliers dont la grammaire ne peut être décrite à l'aide d'une EBNF. En outre, la définition de contraintes OCL au niveau du méta-modèle ne peut pas non plus être transcrite dans une grammaire EBNF ou dans un schéma XML.

Pour résoudre ce problème, nous avons utilisé dans la construction de nos différents méta-modèles uniquement certaines constructions du langage UML (composition, héritage de classe abstraite, agrégation) pour définir des arbres de syntaxe abstraite. Les grammaires des langages concrets à savoir le schéma XML ou l'EBNF ont été produites manuellement à partir de la description des méta-modèles en UML.

D'une manière générale, nous avons mis en place une infrastructure dans SafArchie Studio qui permet pour les différents modules et les différents méta-modèles associés de générer une bonne partie du code associé à la couche

métier. L'interface du motif de conception « Façade » qui permet de naviguer au sein du modèle est générée. Les arbres de syntaxe abstraite, les différents analyseurs et générateurs sont eux aussi créés automatiquement à partir des grammaires concrètes de représentation du langage. Ainsi, l'intégration d'une nouvelle couche métier pour la construction d'un nouveau module dans ArgoUML (module de conception de type d'architecture, d'architecture logique, d'architecture physique) se limite à la production à partir du méta-modèle de cette couche métier décrit en UML :

- des grammaires concrètes sous forme d'un schéma XML et d'une EBNF pour le formalisme textuel,
- d'un adaptateur pour l'interface de la « Façade » de la couche objet pour l'arbre de syntaxe abstraite généré à l'aide de SableCC. Cette mise en œuvre de l'interface de la Façade est rendue relativement simple à écrire grâce à la mise en œuvre, au niveau de l'arbre de syntaxe abstraite généré par SableCC, du motif de conception visiteur qui permet d'explorer facilement cet arbre,
- enfin, des classes de gestion des événements liés aux modifications apportées à la couche métier. Ces classes sont intégrées à la couche métier à l'aide d'un mécanisme d'héritage.

7.2.3 Vérification de la cohérence

Le deuxième travail important de mise en œuvre au sein de Safarchie Studio consiste à développer et intégrer différents outils de vérification de la cohérence d'un assemblage de composants. Ces outils visent à réaliser les analyses présentées dans les chapitres 4 et 5. Pour ce faire, pour chaque contrat d'interface, un outil de contrôle de la cohérence a été développé permettant pour les analyses structurelles de garantir la compatibilité de deux ports liés ou de deux opérations de ces ports ou pour les vérifications comportementales de garantir la cohérence d'un assemblage de composants.

7.2.3.1 Analyse du contrat d'assertion

Le contrat d'assertion consiste à définir une contrainte sur les paramètres d'une opération fournie ou requise. Son analyse est composée de deux phases. La première est une analyse syntaxique de l'expression OCL étendue à l'aide d'une vérification sur la correction des éléments comparés dans l'expression. La deuxième phase compare les ensembles définis par deux assertions spécifiées respectivement l'une sur une opération fournie, l'autre sur une opération requise.

Vérification locale de la correction d'une contrainte OCL

Pour mettre en œuvre la vérification locale de la correction d'une contrainte OCL, nous intégrons les travaux autour de la bibliothèque *dresden-ocl* ¹⁷. Cette bibliothèque fournit trois fonctionnalités principales :

- un analyseur syntaxique pour le langage OCL,
- un mécanisme de vérification de type,
- un générateur de code.

Nous utilisons dans SafArch Studio uniquement le mécanisme de vérification de type et l'analyseur syntaxique qui vérifie qu'une contrainte OCL attachée à une opération est convenablement écrite. L'analyseur syntaxique est modifié légèrement pour limiter les expressions OCL qui peuvent être décrites au niveau du contrat d'assertion d'une opération (voir Section 4.2.2.2). L'utilisation de *SableCC* par l'outil *dresden-ocl* nous a permis d'adapter facilement ces modifications.

Le mécanisme de vérification de type consiste à vérifier la correction entre une expression OCL et le modèle avec lequel cette contrainte est associée. Il est important de remarquer que la vérification de type d'une expression

¹⁷<http://dresden-ocl.sourceforge.net/>

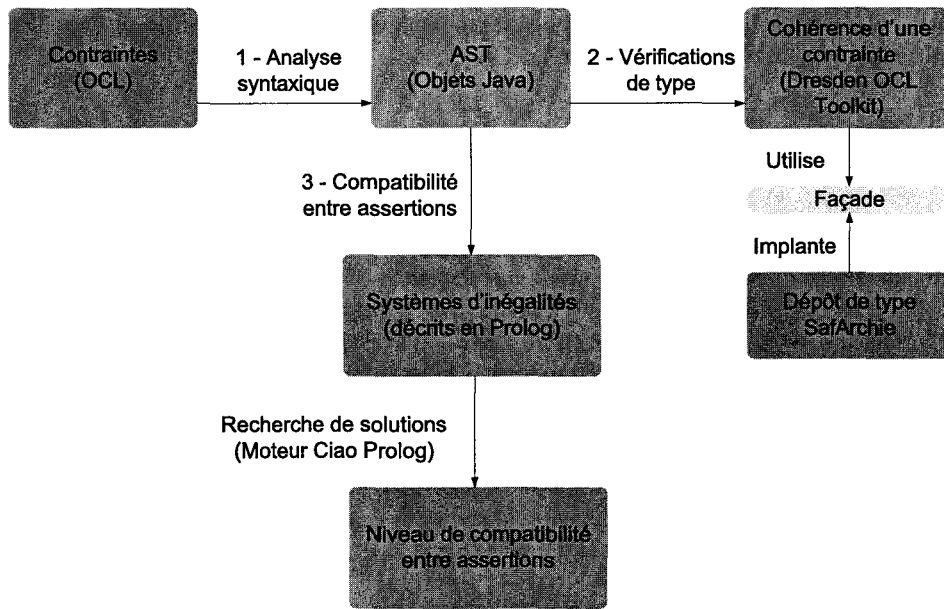


FIG. 7.11 – Vérifications autour du contrat d'assertion

OCL n'est pas possible sans information sur le modèle au sein duquel cette contrainte est définie. Pour récupérer ces informations, tout en limitant un maximum le couplage entre le modèle sur lequel la contrainte est attachée et l'outil OCL, un motif de conception « Façade » est utilisé. L'outil *dresden-ocl* l'utilise pour explorer le modèle et récupérer des informations sur celui-ci. Pour ce faire, dans SafArchie, nous fournissons une mise en œuvre de cette façade pour notre dépôt de type.

Compatibilité entre deux opérations munies d'assertions

Une première étude a été menée pour mettre en œuvre un outil d'analyse statique de compatibilité entre assertions [Tran *et al.*, 2004]. Cet outil ne permet pas pour le moment de raisonner sur tous les types de paramètre d'une opération. La première mise en œuvre travaille uniquement sur les réels, les entiers ou les booléens. Le principe de l'outil consiste à générer un système d'inéquations à partir des assertions définies en OCL. Nous recherchons alors s'il existe une solution à ce système d'inéquations. Si ce système n'a pas de solution, les assertions sont non compatibles. S'il existe une solution, la compatibilité est, au minimum, partielle. Nous construisons alors un deuxième système d'inéquations en prenant l'inverse de l'assertion attachée à l'opération fournie dans le cas où cette assertion concerne un paramètre de cette opération. Sinon, nous prenons l'inverse de l'assertion attachée à l'opération requise si cette assertion concerne le type de retour de cette opération. Si ce système d'inéquation n'a pas de solution, nous sommes dans un cas de compatibilité totale. Si ce système d'inéquation a une solution, il y a intersection entre les types de paramètres restreints à l'aide des assertions. Nous sommes donc dans un cas de compatibilité partielle entre opérations.

Cette résolution d'inéquations est mise en œuvre à l'aide d'un module de résolution de contraintes sur les réels en Prolog. L'expression Prolog à évaluer est générée à partir des contrats d'assertions après analyse syntaxique. Le moteur Prolog utilisé est Ciao Prolog¹⁸. Il permet l'exécution du moteur Prolog depuis un programme Java et permet ainsi son intégration dans ArgoUML.

Cette analyse de la compatibilité entre assertions définies sur des paramètres d'opérations appartenant à des ports liés montre la possibilité de prendre en compte les informations liées aux assertions dans l'analyse statique d'un assemblage de composants. Si l'outillage pour le moment se limite aux assertions sur les réels, les entiers ou

¹⁸http://clip.dia.fi.upm.es/Software/Ciao/ciao_html/ciao_112.html#SEC462

les booléens, nous souhaitons poursuivre la production d'outils sur ce domaine afin d'être capables de prendre en charge un plus grand nombre d'assertions définies en OCL.

7.2.3.2 Analyse du contrat de port

L'analyse du contrat de port permet de détecter lors de la vérification de la compatibilité entre deux types de port s'ils possèdent respectivement les opérations nécessaires au fonctionnement de l'autre port. Pour cela, à chaque opération d'un type de port est associée une valeur booléenne fixée à *vrai* si le type de port lié fournit une opération compatible et fixée à *faux* sinon. Ensuite, chaque contrat de port définit une expression booléenne. Cette expression associée à un type de port est évaluée en fonction du type de port auquel il est lié. L'évaluation des deux expressions (une par type de port) doit être positive pour valider la compatibilité des deux types de ports par rapport au contrat de port. L'absence de contrat de port impose la présence de l'ensemble des opérations compatibles dans le type de port lié.

L'analyseur syntaxique du contrat de port est obtenu directement à partir de la définition du langage grâce à SableCC. La mise en œuvre dans SafArchie Studio consiste à créer l'expression booléenne à partir du contrat de port défini au sein de la description d'architecture (voir Figure 7.12).

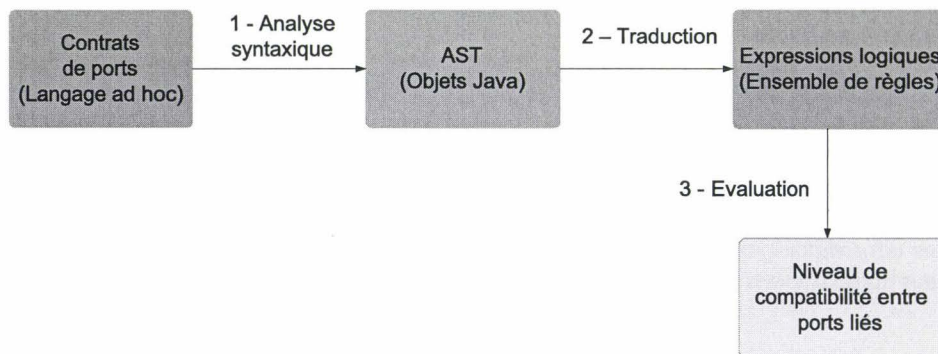


FIG. 7.12 – Vérifications autour du contrat de port

7.2.3.3 Analyse du contrat de comportement

Le langage SFSP, défini dans la section 4.2.3.2, offre une représentation algébrique du comportement externe des types de composants ou des composants. Les analyses associées à la compatibilité d'un assemblage de composants sont, quant à elles, effectuées sur la représentation du comportement à l'aide d'un automate à entrées/sorties. Le premier travail consiste donc à transformer la représentation algébrique du comportement en une représentation sous forme d'automates à entrées/sorties. Pour ce faire, un analyseur syntaxique a été créé pour le langage SFSP à l'aide de SableCC. À partir de cette représentation abstraite de l'arbre, nous traduisons cette représentation algébrique vers une représentation sous forme d'automates.

Le choix a été fait d'utiliser une bibliothèque d'automates JAuto [Bailly, 2005] produites au LIFL. Cette bibliothèque, développée en Java, permet de représenter les automates et offre de nombreux opérateurs de transformation standard sur les automates (projection, produit de synchronisation, etc.). JAuto fournit aussi, grâce à sa bonne conception objet, une bonne généricité par rapport aux éléments manipulés par l'automate. Ainsi, l'automate est capable de manipuler tout type d'objets au niveau des transitions. Les comparaisons se font à l'aide des méthodes *equals()* et *hashCode()* que l'on trouve pour tous les objets Java. En outre, certaines opérations comme le produit de synchronisation sont mises en œuvre à l'aide du motif de conception « Façade ». Cette façade, illustrée dans la figure 7.13, permet de configurer finement la façon dont on synchronise deux automates. La méthode *synchronizable* définit lors d'une synchronisation de deux automates, les états qui peuvent servir de points de synchronisation. Le déroulement du parcours de synchronisation appelle la méthode *synchronize* pour chaque

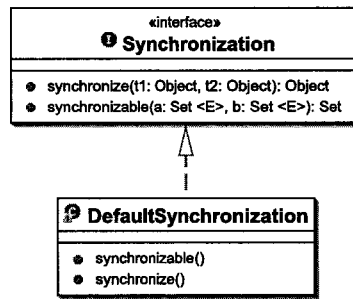


FIG. 7.13 – Façade pour le produit de synchronisation

synchronisation de deux transitions appartenant aux ensembles de transitions a et b . La mise en œuvre de cette méthode permet de décider si l'on souhaite synchroniser les deux transitions t_1 et t_2 . Cette méthode retourne un nouvel objet correspondant à la synchronisation des deux transitions t_1 et t_2 . Cette mise en œuvre est particulièrement intéressante dans notre approche pour la construction du produit de synchronisation. En effet, SafArchie propose la synchronisation de messages émis par un composant et d'un message reçu par un autre. Ces messages ne portent pas nécessairement les mêmes noms. Cette mise en œuvre du produit de synchronisation permet de générer facilement l'automate d'un composite à partir des automates de ses fils sans modification de ceux-ci. Seule la réalisation de l'interface de synchronisation est modifiée afin de prendre en compte les contraintes de nommage au sein d'un composite.

Pour comprendre le comportement d'un composite, il est intéressant d'obtenir une représentation graphique de l'automate qui modélise ce comportement. D'une manière générale, la représentation graphique des graphes est un problème algorithmique difficile. Il est utile de recourir à un programme externe à qui on délègue la génération des graphes. La représentation graphique des différents automates est donc représentée à l'aide de l'outil *Graphviz*¹⁹. Conçu par une équipe des laboratoires de recherche de AT&T (American Telephone & Telegraph), cet outil permet la représentation de graphes très denses comprenant un très grand nombre de nœuds grâce des algorithmes très puissants. Il est très rapide à l'exécution et le rendu est optimisé afin que les liens ne recouvrent pas les nœuds et qu'ils ne se croisent pas. Il fournit surtout un langage dédié minimal pour la représentation de graphe. Dans SafArchie Studio, nous utilisons ce langage pour décrire les automates générés à partir des description SFSP et nous utilisons *Graphviz* pour la génération de la représentation graphique des automates relatifs au comportement de l'application (voir Figure 7.14).

7.2.4 Génération de code

La partie projection d'une description d'architecture Safarchie vers Fractal ou ArchJava se place dans le contexte actuel de la transformation de modèles. Notre modèle de départ est le modèle abstrait de composant SafArchie, le modèle d'arrivée est le modèle concret Fractal ou ArchJava. Pour ce faire, nous avons utilisé cette fonctionnalité de SafArchie Studio comme un cas d'étude pour comprendre les différences d'expression entre une transformation exprimée à l'aide d'un langage déclaratif à savoir XSLT et une transformation exprimée à l'aide d'un langage impératif Java.

Le contexte de notre transformation de modèles est présenté Figure 7.15.

7.2.4.1 Projection vers ArchJava

La projection vers ArchJava [Aldrich *et al.*, 2002] est de loin la plus facile, la proximité des éléments structurels entre les deux modèles permet de projeter directement les concepts composants primitifs, composites et ports vers

¹⁹<http://www.graphviz.org/>

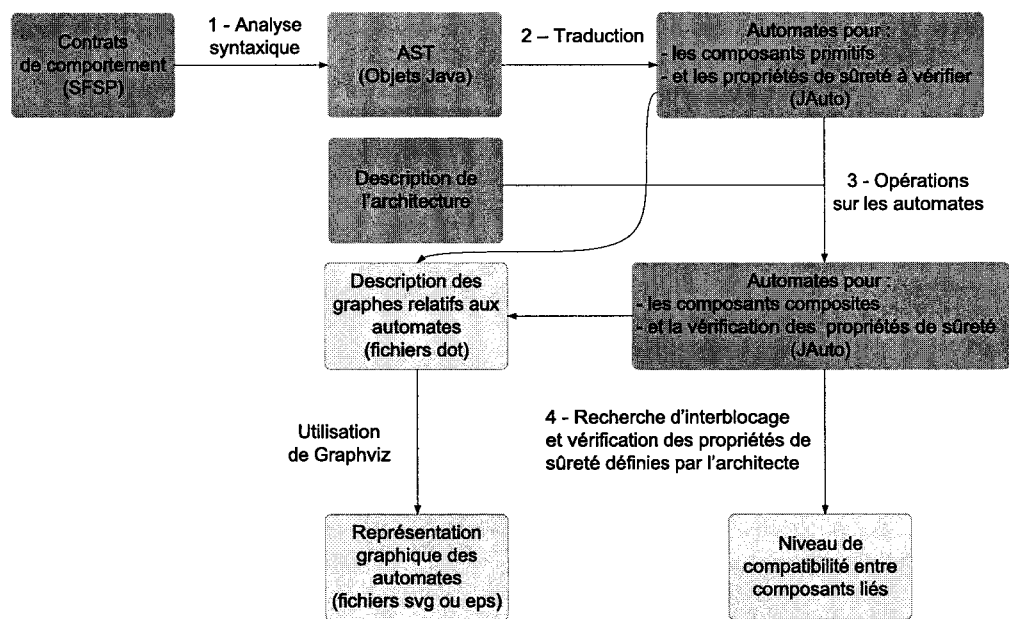


FIG. 7.14 – Vérifications autour du contrat de comportement

ArchJava. En outre, ArchJava distingue pour la notion de connecteur, les notions de classe de connexion qui identifient des ports de types de composant qui peuvent être liés par un connecteur à l'exécution et la notion de connecteur qui lie deux ports de composant. Cette dualité de la notion de connecteur est utile pour prendre en compte au niveau de la mise en œuvre la différence de sémantique dans SafArchie entre la liaison du modèle de type et la connexion du modèle logique.

La génération de code vers ArchJava crée pour chaque composant de l'architecture le code relatif à son interface. Il crée en outre pour les composites les patrons de connexion pour chaque liaison définie dans le modèle

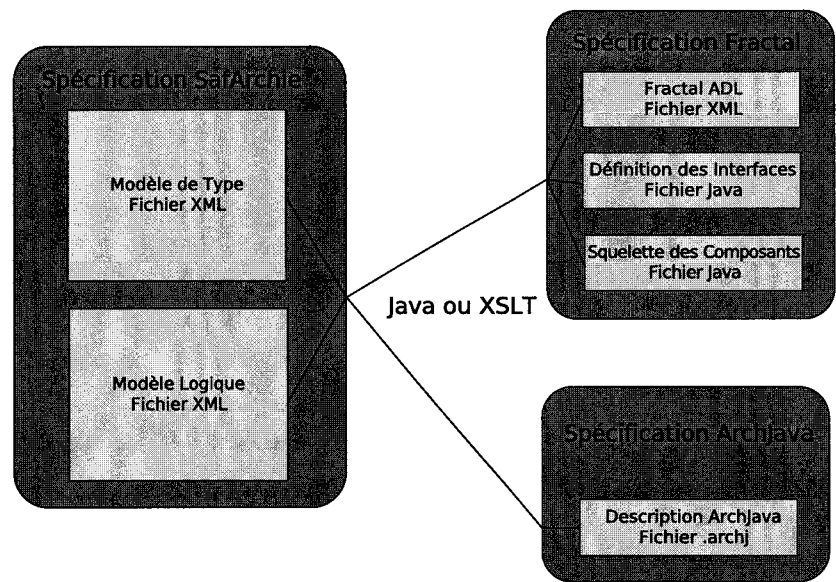


FIG. 7.15 – Vue globale de l'architecture de la projection

de type de SafArchie. Pour chacun de ces composites, un constructeur paramétré par le nom du composite est aussi généré. Celui-ci crée les instances de composant déclaré dans le modèle logique pour ce composite et les connexions entre ces instances à l'aide du mot clé *connect*.

Cette génération de code est très intéressante à la fois pour SafArchie comme pour ArchJava. En effet, SafArchie dispose avec ArchJava d'une plate-forme de mise en œuvre de l'architecture qui apporte des garanties sur le respect des contraintes architecturales fixées par la description d'architecture. La proximité des deux modèles de composant permet à l'architecte de garder une bonne traçabilité entre les éléments spécifiés avec SafArchie et les éléments que l'on retrouve dans la structure du programme ArchJava. Réciproquement ArchJava qui souffre de ne pas avoir une vision abstraite de l'architecture bénéficie avec SafArchie d'une telle abstraction.

7.2.4.2 Projection vers Fractal

La projection vers le modèle de composant Fractal [Bruneton *et al.*, 2004a] du consortium ObjectWeb a permis de mettre en lumière plusieurs difficultés. Il existe, en effet, des différences structurelles importantes entre le modèle Fractal et le modèle SafArchie. Par exemple, l'absence de la notion de port dans Fractal remplacée par la notion d'interface impose la création de deux entités par port SafArchie. Cette séparation des ports pose un problème de nommage pour les éléments de Fractal. Pour le moment, une convention de nommage a été retenue pour la désignation des interfaces Fractal. Le nom des interfaces clientes est préfixé par *client_*, celui des interfaces serveur est préfixé par *server_*.

La génération de code vers Fractal à partir de SafArchie offre la possibilité de générer la description de l'architecture (fichiers XML), les interfaces des composants (interfaces Java) et le squelette des composants (classes Java). L'utilisation de la génération de code permet de se concentrer tout d'abord sur la description de l'architecture à l'aide de SafArchie et puis sur la mise en œuvre de ces composants en Java. Elle permet de garantir la cohérence entre la description de l'architecture Fractal, la description des interfaces et les informations du contrôleur de liaisons. En effet dans Fractal, le nom des interfaces doit être rigoureusement le même entre la description d'architecture et l'utilisation de ces noms dans le contrôleur de liaisons. De même, les signatures des interfaces Java ou des classes d'implémentations doivent-elles aussi être référencées dans le fichier de description d'architecture. Sans support, cette cohérence entre la description d'architecture en XML et la description des interfaces ou des classes de mise en œuvre est difficile à maintenir. Ce manque de support est propice aux erreurs. L'utilisation d'un générateur de code de SafArchie vers Fractal permet de supprimer le travail lié au maintien de la cohérence entre les éléments de spécifications et de se concentrer uniquement sur la mise en œuvre des services offerts par un composant.

7.2.4.3 Bilan

L'étude a montré la faisabilité de cette transformation à l'aide de deux langages (Java ou XSLT). Cependant, nous avons constaté que l'utilisation d'XSLT dans un contexte de transformation de fichiers XML est lourde et difficile à maintenir et à relire. En outre, le problème de la finalité des variables augmente considérablement la longueur du code et oblige à écrire des parties de code redondantes. Par contre, le langage de navigation XPath est lui au contraire intuitif et puissant. Pour XSLT, il manque une syntaxe abstraite de plus haut niveau pour la spécification de transformations.

L'utilisation d'un langage impératif semble plus facile à appréhender et à maintenir, l'utilisation de Java et d'une API comme JDOM de manipulation de document XML permet la prise en main facile des technologies. Il est alors possible de se concentrer sur l'algorithmique du moteur de transformation (principalement sur les problèmes de récursivité dans les parcours d'arbre). Cependant, ce parcours d'arbre nécessite l'utilisation intensive d'itérateurs qui diminue la lisibilité du code et alourdit l'exécution de la transformation.

Ces travaux de transformation de modèles nous ont permis d'apercevoir la complexité de la description d'une transformation de modèles. Comme de nombreux travaux sont en cours pour spécifier un langage dédié à la trans-

formation de modèles, nous avons, au travers de notre application, un cas d'étude intéressant pour évaluer ces différents langages.

7.3 TranSAT

La suite d'outils SafArchie Studio nous permet dans un premier temps de concrétiser l'ensemble des propositions de SafArchie. Il permet de spécifier une architecture logicielle à l'aide de SafArchie, vérifier la cohérence de cette description et générer du code vers les modèles de composant Fractal ou ArchJava. TranSAT, présenté dans le chapitre 6, est un canevas de conception qui vise à améliorer la structuration d'une architecture logicielle grâce à l'ajout d'une entité de modularisation, le patron, et permettre la construction incrémentale d'une architecture logicielle par applications successives de ces patrons. Dans TranSAT, le patron d'architecture est constitué d'un plan à intégrer, d'un masque de point de jonction qui précise le contexte d'intégration de ce plan et d'un ensemble de règles de transformation qui spécifie les modifications à apporter sur une architecture logicielle afin de pouvoir y intégrer ce nouveau plan. Le développement d'outils associés à TranSAT vise à étendre SafArchie Studio afin de permettre la construction d'architecture par assemblage de plans. Les principales caractéristiques de ces extensions sont :

- la spécification de nouveaux patrons, c'est-à-dire :
 - la description de nouveaux plans ;
 - la description de masques de point de jonction ;
 - la spécification des règles de transformation,
- la vérification statique de la cohérence de ces nouveaux patrons, c'est-à-dire :
 - la vérification de la cohérence de nouveaux plans,
 - la vérification de la cohérence de nouveaux masques de point de jonction,
 - l'analyse statique des règles de transformation,
- la recherche des lieux d'intégration compatibles avec un masque de point de jonction,
- la spécification des lieux d'intégrations,
- l'application des règles de transformation sur un lieu d'intégration.

L'architecture de ces outils est construite au-dessus de SafArchie Studio. Trois nouveaux modules permettent respectivement de spécifier un nouveau plan, un masque de point de jonction et les règles de transformation. Pour chacun de ces modules, les choix techniques ont été rigoureusement les mêmes que pour les modules relatifs à SafArchie, c'est-à-dire la génération de la couche « modèle » à partir des différents méta-modèles UML ou grammaires, l'intégration dans l'arbre grâce à un *plug-in* pour l'*Explorer Framework* et le développement des panneaux de propriétés ou de la vue graphique pour chacun de ces méta-modèles. À partir de ces éléments, différents outils permettent de vérifier la cohérence du patron d'architecture (voir Figure 7.16).

Un dernier module a été développé pour mettre en œuvre le tissage d'un plan sur une architecture logicielle (voir Figure 7.17). Ce module prend en charge la définition de lieux d'intégration. Cette tâche peut être réalisée de deux façons par l'architecte. Soit l'architecte spécifie manuellement un ensemble de lieux d'intégration, le module vérifie alors le respect des contraintes du masque de point de jonction. Soit le module assiste l'architecte dans la définition des lieux d'intégration en présentant à l'architecte l'ensemble des lieux compatibles avec un masque de point de jonction. L'architecte sélectionne alors ces lieux d'intégration dans la liste qui lui est présentée.

Finalement, la dernière brique de TranSAT est le tisseur. Il fournit un moteur d'exécution de règles d'intégration capable de transformer l'architecture logicielle et d'y intégrer un nouveau plan en fonction de la définition d'un lieu d'intégration. Le tisseur fournit aussi un outil de vérification permettant de garantir la cohérence de l'architecture résultante de l'intégration d'un nouveau plan. Le choix de structurer cet ensemble d'outils autour de quatre modules différents est aussi renforcé par le processus associé à TranSAT qui identifie différents acteurs. Chaque module fournit alors un atelier dédié aux tâches dont chacun de ces acteurs a la responsabilité.

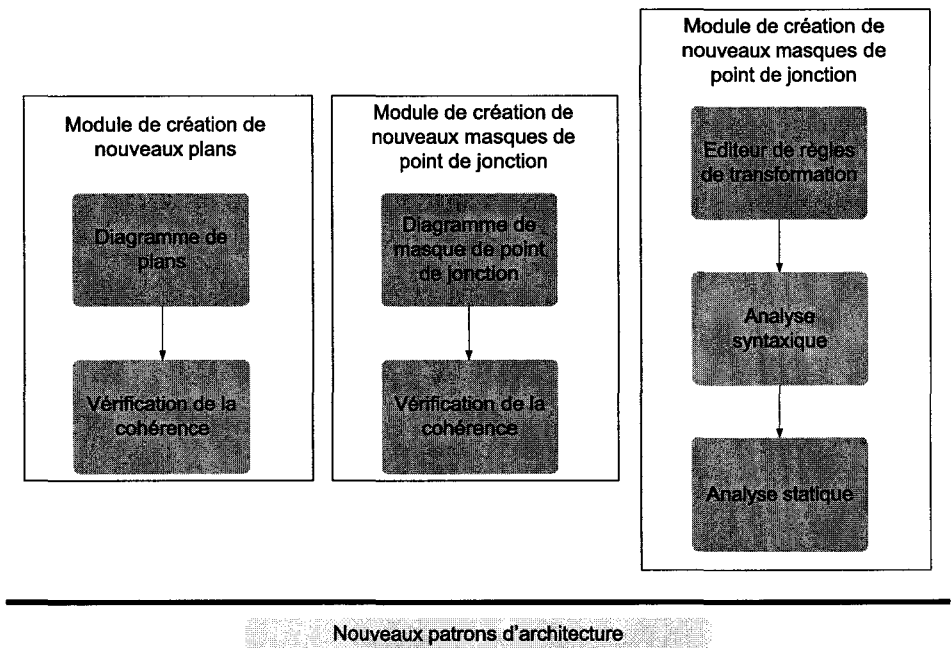


FIG. 7.16 – Trois nouveaux modules pour la production de nouveaux patrons d'architecture

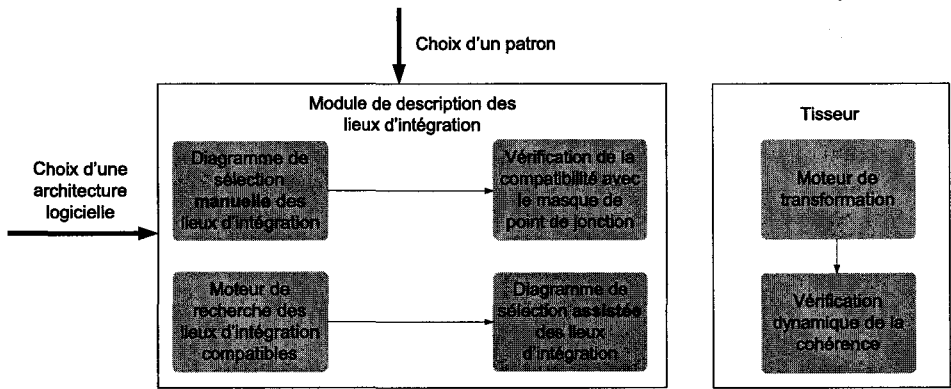


FIG. 7.17 – Un nouveau module pour l'intégration d'un nouveau plan au sein d'une architecture logicielle

Cette section se structure autour de deux axes principaux. Nous présentons, tout d'abord, les trois modules permettant de spécifier de nouveaux patrons d'architecture. Dans un deuxième temps, la mise en œuvre du module en charge du tissage est présentée.

7.3.1 Trois nouveaux modules pour la définition de nouveaux patrons d'architecture

7.3.1.1 Processus de construction de ces modules

Les deux modules permettant la spécification d'un nouveau plan d'architecture et d'un masque de point de jonction sont construits selon le processus d'ajout de modules présenté dans la section 7.2.2. Dans un premier temps, un schéma XML et une EBNF sont spécifiés à partir des deux méta-modèles. Le schéma XML est annoté afin que JAXB puisse générer un arbre de syntaxe abstraite intégré à ArgoUML. La mise en œuvre consiste alors

à :

- étendre la couche « Modèle » pour l'implantation de méthodes spécifiques aux éléments du méta-modèle qui ne peuvent être directement générés à partir du schéma XML,
- initialiser la couche « Contrôleur » pour prendre en compte les événements issus des objets du modèle,
- créer les différentes « Vues » qui permettent de manipuler les éléments du modèle :
 - la vue arborescente est créée en développant une extension à l'explorer pour ce nouveau méta-modèle ;
 - la vue graphique est développée en utilisant la bibliothèque GEF ;
 - la vue propriété est créée en mettant en œuvre différents panneaux d'édition de propriétés (un par type d'objets).

Le dernier module est pour le moment beaucoup plus simple. Il est composé d'un éditeur de texte pour la spécification des règles de transformation et d'un panneau d'édition de propriétés au sein duquel il est possible de préciser le plan et le masque de point de jonction avec lesquels les règles de transformation sont associées.

7.3.1.2 Analyse de la cohérence du patron d'architecture

Pour chacun de ces trois nouveaux modules, différents analyseurs sont développés afin de garantir statiquement certaines propriétés (voir Section 6.4).

Cohérence du plan et du masque de point de jonction associés à un patron d'architecture

La cohérence du plan consiste à garantir que le nouveau plan ne contient pas d'erreurs au niveau des connexions entre composants (voir Section 6.4). Pour ce faire, nous réutilisons, pour vérifier chaque connexion du nouveau plan, les différents outils développés pour SafArchie, permettant de vérifier la cohérence d'un assemblage de composants (voir Section 7.2.3). Pour chaque connexion, un analyseur vérifie la compatibilité structurelle des ports qu'elle relie. Il vérifie aussi la compatibilité comportementale de l'assemblage de composants défini au sein du plan.

Si le plan ou le masque de point de jonction sont spécifiés à l'aide du formalisme textuel ou du formalisme XML en dehors de SafArchie Studio et importés au sein du module de création de nouveau plan, un analyseur syntaxique vérifie la conformité de la description d'architecture avec sa grammaire.

Cohérence des règles de transformation associées à un patron d'architecture

L'analyse des règles de transformation vérifie respectivement la conformité de la spécification avec la grammaire du langage de règles de transformation, le contenu (voir Section 6.4.2.4) et les liaisons (voir Section 6.4.2.5). L'analyse syntaxique est construite à l'aide de SableCC. La vérification statique du contenu et des connexions consiste à construire un environnement à partir des informations contenues dans le plan et dans le masque de point de jonction et à le faire évoluer par rapport à la sémantique de chaque primitive de transformation. Le résultat de l'environnement est ensuite analysé en fonction des clauses de sorties précisées dans la section 6.4.2.4.

L'outil d'analyse est construit à l'aide de Drools²⁰. Drools est un moteur de règles mettant en œuvre l'algorithme de chaînage avant Rete [Forgy, 1982] adapté à un environnement Java. L'avantage de Drools réside dans sa flexibilité, qui permet de l'adapter facilement à la logique de n'importe quel domaine à partir du moment où la logique de ce domaine est décrite à l'aide d'un schéma XML. L'utilisation de Drools pour mettre en œuvre l'analyse statique associée aux règles de transformation consiste à transcrire les règles qui définissent la sémantique du langage (voir Section 6.4.2) en un ensemble de règles qui permettent, soit d'évaluer l'état de l'environnement, soit de

²⁰<http://drools.org/>

simuler l'application d'une primitive de transformation. La première catégorie de règles transcrit les clauses de jugement de chaque primitive pour savoir s'il est possible d'appliquer une primitive de transformation. La deuxième catégorie fait évoluer l'environnement. L'évaluation de l'état final de l'environnement est associée elle aussi à différentes règles qui doivent être vérifiées à la fin de la transformation. La simulation consiste alors uniquement à évaluer ces règles dans un ordre dicté par la spécification de la transformation. Ces règles s'appliquent sur une base de faits définie à partir des informations fournies par le masque de point de jonction et le plan du patron d'architecture auxquels les règles de transformation sont associées.

Vérifier la cohérence d'une transformation à l'aide d'un moteur de règles Drools se révèle une approche intéressante car il est alors facile de faire évoluer la sémantique du langage (ajout de primitive, ajout de contraintes). En effet, ces évolutions s'intègrent alors par la définition de nouvelles règles sans modification des règles existantes.

Ces trois modules fournissent un atelier pour la production de nouveaux patrons d'architecture. Les vérifications syntaxiques permettent de détecter des anomalies dans la construction de ces nouveaux patrons, au niveau du plan du masque de point de jonction ou des règles de transformation.

7.3.2 Le tisseur, un moteur de transformation pour intégrer des nouvelles préoccupations dans une description d'architecture logicielle

Le dernier module associé à TranSAT a en charge la sélection du ou des lieux d'intégration sur lesquels on souhaite intégrer le nouveau plan. L'idée générale de ce module est de pouvoir à l'aide d'un moteur de règles, soit dans le cas d'une spécification du lieu d'intégration manuelle, garantir la compatibilité entre ce lieu et le masque de point de jonction du patron que l'on cherche à intégrer, soit dans le cas d'une spécification du lieu d'intégration assistée, trouver un ensemble de lieux d'intégration compatibles avec le masque de point de jonction.

Une fois la compatibilité entre un lieu d'intégration spécifié par l'architecte et le masque de point de jonction, un tisseur applique les règles de transformation du patron à ce lieu d'intégration. Une dernière étape de vérification garantit la cohérence de l'architecture résultante de la transformation.

7.3.2.1 Recherche des lieux d'intégration compatibles avec un masque de point de jonction

La recherche des lieux d'intégration compatibles avec un masque de point de jonction se fait à l'aide du moteur de règles Drools. Le masque de point de jonction qui spécifie les contraintes qu'un lieu d'intégration doit vérifier sert à générer les règles. La description d'architecture définit les faits, c'est-à-dire les éléments sur lesquels le moteur va chercher à satisfaire ces règles (voir Figure 7.18).

Dans le cas de la vérification de la cohérence des règles de transformation, le fichier de règles Drools est spécifié une fois pour toutes. Au contraire, les règles utilisées pour trouver les lieux d'intégration compatibles avec un masque de points de jonction sont générées à l'aide d'une feuille de style XSLT à partir d'une description du masque de coupe. Cette approche permet à l'architecte de bénéficier, à l'aide du langage de masque de point de jonction, d'un langage de haut niveau pour spécifier les contraintes sur le contexte d'intégration d'une préoccupation. La génération d'un fichier de règles Drools permet de mettre en œuvre, à l'aide d'un moteur de règles éprouvé, la recherche des lieux d'intégration compatibles pour une architecture donnée.

Pour réduire le nombre de lieux d'intégration compatibles dans les cas de larges architectures, l'architecte peut ajouter certaines contraintes supplémentaires. Par exemple, il peut identifier certains éléments du masque de point de jonction pour réduire l'espace de recherche. Cette identification se fait en ajoutant une contrainte sur le nom d'un élément du masque de point de jonction.

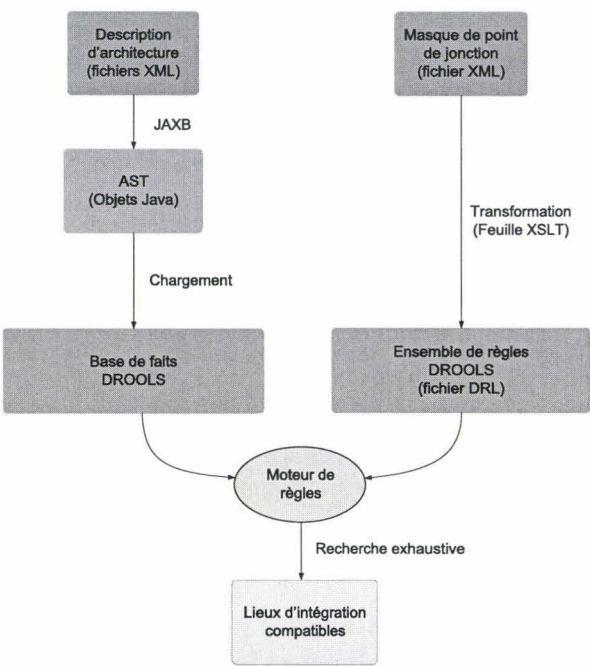


FIG. 7.18 – Mise en œuvre de la recherche exhaustive de lieux d'intégration compatibles

7.3.2.2 Adéquation entre un lieu d'intégration et un masque de point de jonction

La deuxième vérification consiste à, dans le cas d'une spécification manuelle d'un ou de plusieurs lieux d'intégration, vérifier la compatibilité de ces lieux avec le masque de point de jonction. Pour ce faire, les données relatives au lieu d'intégration sont chargées dans la base de faits. Le masque de point de jonction permet la création d'un fichier de règles. La vérification consiste à s'assurer que les règles s'appliquent sur le lieu d'intégration défini par l'architecte (voir Figure 7.19). La stratégie de ne charger que le lieu d'intégration à contrôler dans la base de faits permet de réduire considérablement le coût de la vérification de la compatibilité entre un masque de point de jonction et un lieu d'intégration.

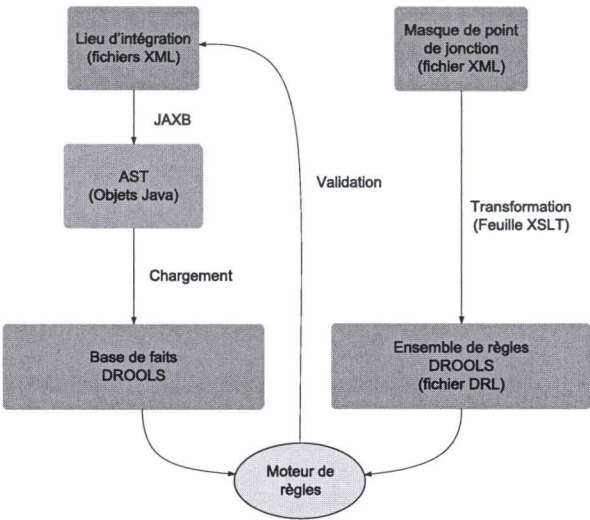


FIG. 7.19 – Validation d'un lieu d'intégration spécifié manuellement

7.3.2.3 Un moteur de transformation pour la description d'architecture logicielle

Le moteur de transformation prend en entrée un lieu d'intégration respectant un masque de point de jonction d'un patron d'architecture. À partir de ce lieu d'intégration, il applique les règles de transformation de ce patron. L'application de ces règles est relativement directe dans le sens où chacune des primitives de transformation du langage est atomique et qu'elle impacte uniquement les éléments du plan ou du lieu d'intégration. Chacune de ces primitives est donc mise en œuvre par une classe qui prend en paramètre du constructeur des références sur les éléments qu'elle manipule. Ainsi, la primitive *Or.move(Pr)* est mise en œuvre par une classe *OperationMouvement* qui prend en paramètre une référence sur une opération et une référence sur un port. L'évaluation des conditions d'utilisation des différentes primitives a été validée de manière statique. La transformation peut donc s'appliquer au paramètre transmis à la primitive.

La deuxième tâche du moteur de transformation consiste à vérifier dynamiquement l'impact de l'intégration d'un plan au sein d'une architecture logicielle (voir Section 6.4.3). Ainsi, pour chaque nouvelle connexion ou pour chaque connexion dont au moins l'un des deux ports a été modifié au cours de la transformation, le tisseur vérifie la compatibilité structurelle entre les types des ports reliés. Nous utilisons pour ce faire les outils d'analyse structurelle développés dans le cadre de SafArchie. Nous vérifions ensuite l'impact de l'intégration d'un nouveau plan sur le comportement de l'application. L'analyse de cet impact n'est effectuée que pour les assemblages de composants pour lesquels, l'introduction du nouveau plan présente un risque, c'est-à-dire ceux dont au moins le comportement d'un des composants a été modifié.

7.4 Bilan et Perspectives sur SafArchie Studio

7.4.1 Bilan qualitatif

Le développement de SafArchie Studio permet de concrétiser l'ensemble des propositions autour de SafArchie et TranSAT et fournit un atelier pour la construction incrémentale de descriptions d'architecture logicielle. Un travail important dans cette mise en œuvre a été effectué pour limiter la quantité de code à écrire. SafArchie Studio bénéficie ainsi de différentes bibliothèques permettant de générer une partie du code de cet atelier. En outre, la réutilisation de bibliothèques comme JAuto ou la boîte à outils Dresden-OCL permet d'intégrer certaines briques logicielles existantes dans le domaine de l'architecture logicielle. SafArchie Studio bénéficie en outre de la bonne structuration d'ArgoUML en module et de son infrastructure basée sur le motif de conception « Modèle-Vue-Contrôleur ». Enfin, l'usage dans le module de spécification de la coupe de TranSAT d'un moteur de règles comme Drools permet d'avoir une structure homogène entre les différents formalismes. L'usage d'XSLT pour générer les fichiers de règles Drools à partir du masque de coupe est un bon exemple de transformation de modèles dans lequel TranSAT offre un langage dédié pour spécifier les masques de point de jonction proches des notions d'architecture générique et Drools fournit un moteur d'inférence à chaînage avant pour la recherche des lieux d'intégration compatibles ou la vérification de la compatibilité d'un lieu d'intégration.

7.4.2 Bilan quantitatif

SafArchie Studio a été construit comme une suite d'outils pour la spécification des architectures logicielles. Le tableau de la page 7.1 présente un bilan quantitatif pour chacune de ces briques de base en rappelant les technologies utilisées et en présentant un tableau synoptique de la production de code associé à cet outil.

7.4.3 Perspectives

Certaines briques de cet atelier de génie logiciel centré autour de l'architecture logicielle sont encore en développement. Par exemple, la mise en œuvre du moteur de transformation est encore à l'étude. De même, un premier

		Briques de base de SafArchie Studio	Technologies utilisées	Entrées	Sorties	Nombre de lignes de code
SafArchie	Éditeur de modèles de type	Élaboration graphique de modèles de type	ArgoUML GEF JAXB JDOM (manipulation de données XML) NSUML OCLToolkit		Modèle de type	20897 dont 11600 lignes générées
	Éditeur de modèles logiques	Élaboration graphique de modèles logiques			Modèle logique	12100 dont 8100 lignes générées
	Explorateur de modèles	Représentation graphique d'un modèle en arbre de navigation	ArgoUML GEF Explorer Framework	Modèle	Arbre de navigation	775 en Java 571 en XSLT
	Vérificateur de contrats	Vérification de la cohérence de l'architecture	Jauto Graphviz OCLToolkit	Type d'architecture Architecture logique	Liste d'erreurs si l'assemblage est incohérent	7876 dont : 6562 générés
	Générateur de code	Modèle SafArchie vers modèle Fractal ou ArchJava	JDOM	Modèle SafArchie logique	Modèle Fractal Modèle ArchJava	881
TranSAT	Analyse statique des règles de transformation	Vérification des règles de transformation par rapport au plan à ajouter et au masque de coupe	Drools JAuto JAXB JDOM SAXON (moteur de transformations XSLT)	Patron d'architecture	Liste d'erreurs si les règles ne sont pas valides	27500 dont : 18500 générées 5600 lignes pour spécifier des règles pour Drools 750 lignes de XML/XSLT 2650 lignes de code Java
	Recherche de lieux d'intégration	Détermination des lieux d'intégration selon le plan de base et le masque de coupe	Drools JAXB JDOM SAXON	Plan de base Masque de coupe Lieu d'intégration partiel ou complet optionnel	Liste de lieux d'intégration	30000 dont : 24000 générées 3500 lignes de XML et XSLT 2500 lignes de code Java

TAB. 7.1 – Bilan quantitatif de SafArchie Studio

prototype a été produit pour la recherche de lieux d'intégration compatibles. Ce prototype est basé sur un moteur d'inférence à chaînage avant : Drools. D'autres travaux démarrent pour construire le même moteur en se basant sur des moteurs d'inférence à chaînage arrière afin de pouvoir ajouter facilement de nouveaux types de contrainte dans le masque de point de jonction comme la contrainte liée à la hiérarchie du modèle de composant afin de pouvoir exprimer des requêtes du type (je recherche un composant A qui contient parmi ses descendants un composant B qui *etc.*).

Il reste, au niveau des perspectives de développement, un travail important à mener sur l'extension du gestionnaire de conception qui avertit l'architecte des erreurs ou des maladroites de construction d'une architecture logicielle. Si, ce travail a été quelque peu mis de côté pour le moment faute de temps, c'est un travail très important pour assister l'architecte dans son travail de conception et transformer SafArchie Studio en un véritable assistant pour l'architecte dans son travail de conception.

D'une manière générale, le développement de SafArchie Studio a fait apparaître le besoin d'un meilleur support des ateliers existants de l'ingénierie dirigée par les modèles. En effet, SafArchie Studio, par le développement de ses sept modules, illustre le besoin pour chaque formalisme dédié de construire un outillage associé à ce formalisme. Cet outillage comprend une couche modèle, mais bien souvent aussi une couche graphique permettant d'appréhender la complexité du modèle à partir de différents points de vue. Si la construction de la couche modèle est maintenant relativement assistée, la construction de la couche graphique reste manuelle, complexe et rapidement fastidieuse. Une des perspectives dans SafArchie Studio consisterait à proposer dans la suite de l'*Explorer Framework*, une vue graphique et une vue d'édition des propriétés facilement configurable à l'aide d'un langage dédié, ceci dans le but de simplifier au maximum la construction d'un module de prise en charge d'un nouveau méta-modèle.

Enfin, une autre perspective au niveau de l'outillage consiste à améliorer l'assistance de l'architecte dans la spécification des règles de transformation d'un nouveau patron. En effet, si l'architecture prend l'habitude de l'outil pour construire son architecture, il est regrettable de revenir à une description textuelle des règles de transformation. Nous souhaiterions étudier la faisabilité d'offrir un module graphique permettant à l'architecte de manuellement modifier un masque de point de jonction pour y intégrer un plan. Les règles de transformation sont alors générées automatiquement à partir des différentes actions de l'architecte. L'objectif est de pouvoir construire une version scénarisée de l'éditeur de règles de transformation dans l'esprit de l'enregistrement de Macro dans la suite Office.

Quatrième partie

Conclusion et perspectives

Chapitre 8

Conclusion et perspectives

8.1 Résumé des contributions

L'objectif de cette thèse est de proposer un canevas de conception qui vise à faciliter la construction d'architectures logicielles et à maîtriser leurs évolutions. Nous avons pour cela proposé une approche autour de SafArchie/-TranSAT dont l'objectif est à la fois de maîtriser la complexité liée à la construction d'une architecture logicielle et de favoriser la réutilisation. Cette approche se fait fort de respecter les grands principes du génie logiciel à savoir la modularité, l'abstraction, l'anticipation du changement, la construction incrémentale et la réutilisation.

La modularité SafArchie propose un premier niveau de structuration d'une application à l'aide des concepts de composant, de connexion et de composant hiérarchique. En outre, pour favoriser la modularisation de certaines préoccupations transverses à l'architecture, TranSAT propose alors une deuxième manière de structurer une application à l'aide de la notion de plan. Chaque plan décrit un assemblage de composant en charge d'une préoccupation. L'architecte bénéficie alors de deux niveaux de structuration pour décrire une architecture logicielle, l'assemblage ou l'imbrication de composants mais aussi la superposition de plans.

L'abstraction Une description d'architecture SafArchie est structurée autour de trois modèles : le type d'architecture, l'architecture logique et l'infrastructure physique. Ces modèles proposent une abstraction sur la structure et le comportement du système qui permet à l'architecte, mais aussi aux différents acteurs d'un projet informatique, de comprendre l'organisation du système et leur rôle dans la construction de ce système.

L'anticipation des changements Nous considérons dans cette thèse deux types de changement pour une architecture logicielle : les évolutions internes et les évolutions externes. Les évolutions internes caractérisent les changements de l'architecture d'une application qui surviennent dynamiquement au cours de son exécution. Au contraire, les évolutions externes représentent les modifications apportées par l'architecte pour conserver une bonne adéquation entre le système et les exigences des utilisateurs.

Pour maîtriser les évolutions internes, SafArchie repose sur les deux niveaux de description de l'architecture qui sont le type d'architecture et l'architecture logique. Un type d'architecture définit les invariants du système. Il représente un cadre structurel pour l'architecture. L'architecture logique est variable et capture l'état du système à un instant t .

Les évolutions externes sont abordées dans TranSAT comme l'intégration de nouvelles préoccupations au sein d'une description d'architecture logicielle. Ainsi, une évolution externe est caractérisée par l'intégration d'un nouveau plan au niveau d'un ou plusieurs lieux d'intégration d'une architecture de base. Le mécanisme de composition des plans d'une architecture est appelé le tissage. Sa sémantique est définie de manière explicite en spécifiant les modifications apportées à l'architecture de base. De plus, pour maîtriser les évolutions externes, TranSAT fournit un ensemble de vérifications qui sont effectuées à chaque intégration d'un nouveau plan pour garantir le résultat de cette transformation.

La construction incrémentale TranSAT permet de définir un processus incrémental de construction d'architectures logicielles où chaque nouvelle étape correspond à l'intégration d'une nouvelle préoccupation au sein d'un plan de base. Ce processus permet de structurer la conception d'une architecture dans le temps, en partant d'une architecture métier vers l'intégration de préoccupations techniques. Il permet aussi de distribuer le travail associé à la construction d'une architecture entre différents acteurs.

La réutilisation Enfin, SafArchie et TranSAT visent à permettre d'augmenter la productivité au sein de ce canevas de conception en favorisant la réutilisation des composants et des patrons. Une des principales difficultés liées à la réutilisation consiste à pouvoir garantir l'utilisation correcte du module logiciel dans un contexte différent de celui pour lequel il a été conçu. Ce contexte peut être l'assemblage d'un composant avec d'autres composants pour SafArchie ou le tissage d'un plan avec un plan de base pour TranSAT.

Ainsi, SafArchie propose une définition étendue des interfaces de composants à l'aide des contrats de port, des contrats d'assertion et des contrats de comportement. Dans SafArchie, ces informations permettent de caractériser la cohérence d'un assemblage de composants. Nous identifions dès lors trois familles de cohérence : la cohérence totale qui indique à l'architecte qu'aucun risque n'est détecté au niveau de son assemblage de composants, la cohérence partielle qui met en évidence des risques de problème au niveau d'une architecture logicielle et enfin l'incohérence qui détecte au sein d'un assemblage un problème empêchant le fonctionnement correct d'une architecture.

De même, le patron d'architecture dans TranSAT est une entité qui permet de modulariser la notion de préoccupation. Il est conçu pour être utilisable pour différentes architectures. Ainsi, le patron est décrit indépendamment du contexte dans lequel il est utilisé. Le masque de point de jonction définit un ensemble de contraintes sur le lieu d'intégration sur lequel peut être attaché ce nouveau plan. Il est lui aussi défini indépendamment d'un contexte d'intégration particulier. De plus, un travail d'analyse des éléments d'un patron est mené afin de garantir statiquement certaines propriétés sur la cohérence de l'architecture résultante du tissage d'un patron sur un plan de base. Ces analyses favorisent la réutilisation du patron. Ils permettent de limiter les vérifications à effectuer dynamiquement pour garantir la cohérence d'une architecture logicielle résultante de l'application d'un patron d'architecture sur un plan de base.

Finalement, nous avons montré comment ce canevas de conception d'architecture logicielle pouvait être intégré dans un atelier de génie logiciel centré sur la notion de composant afin de favoriser la tâche de construction d'une architecture logicielle. Cet atelier de génie logiciel fournit le moyen de concevoir une architecture logicielle par assemblage de composants et tissage de plans, d'analyser ces assemblages et de produire de nouveaux patrons d'architecture statiquement validés. En outre, placé dans un contexte global d'ingénierie dirigée par les modèles, cet atelier permet de générer le code associé à une description d'architecture logicielle pour différentes plateformes à composants existantes.

8.2 Perspectives

Le travail de cette thèse s'est voulu au sein du projet Jacquard un travail prospectif afin de montrer pourquoi la description d'une architecture logicielle reste une tâche difficile et comment un canevas de conception inspiré des grands principes du génie logiciel peut faciliter cette construction. Dès lors, bien que les principes sous-jacents à la construction du canevas SafArchie/TranSAT nous semblent « sains » de nombreuses améliorations sont possibles. Ces améliorations sont de deux types, celles qui représentent des compléments aux parties déjà réalisées et celles qui correspondent à des centres d'intérêt nouveaux ayant pour cadre la description d'architectures logicielles.

Vers l'intégration de nouvelles préoccupations au niveau du type d'architecture à l'aide de TranSAT TranSAT travaille pour le moment uniquement du point de vue du modèle logique d'architecture. Le tissage de plans au niveau d'une architecture logicielle modifie explicitement le graphe d'instance de composants. La modification du modèle de type est implicite et un nouveau type de composant est créé pour chaque composant dont l'interface est modifiée. Cette stratégie permet d'utiliser TranSAT pour de nombreuses préoccupations qui viennent modifier localement une instance de composant. Cependant, nous souhaitons enrichir TranSAT afin de bénéficier de la

dualité (modèle de type, modèle logique) dans la description de nouveaux patrons d'architecture. Nous souhaitons donc étendre TranSAT, le langage de masque de point de jonction et le langage de règles de transformation afin de pouvoir définir des patrons qui modifient explicitement soit le modèle logique soit le modèle de type ou les deux.

Preuve de correction du code généré Une description d'architecture logicielle vérifie tout un ensemble de propriétés de cohérence au niveau de l'assemblage de composants. Ces propriétés sont vérifiables au niveau de la spécification, mais aucune garantie n'est proposée au niveau de l'implantation à l'aide d'une plate-forme à composants. Une de nos prochaines directions de recherche consiste à travailler sur la traçabilité des propriétés architecturales au niveau des plates-formes à composants afin de garantir que l'application reste conforme à la description de son architecture. La vérification de ces propriétés architecturales peut être de deux ordres :

- vérifier que la structure de l'application respecte la structure et les interactions précisées dans la description d'architecture logicielle,
- propager, dans la phase de génération de code, les contraintes de séquençement des messages précisés dans le contrat de comportement.

Dynamique du modèle de déploiement Le modèle SafArchie propose un premier niveau de description de l'architecture sur lequel le système est déployé. Cette description est pour le moment statique. Dans le contexte d'une étude que nous menons au niveau de l'informatique ubiquitaire, cette approche statique du modèle de déploiement semble assez naïve. Nous souhaitons enrichir ce modèle afin d'obtenir deux niveaux de description pour le modèle de déploiement : un cadre définissant un ensemble d'invariants, une description plus volatile des entités physiques présentes à un instant t . La description du placement des composants logiciels sur les entités physiques doit donc être adaptée pour supporter la dynamique du modèle de déploiement.

Réflexion autour de l'ajout d'informations liées à la qualité de service, au niveau des interfaces de composants Le modèle SafArchie propose une analyse de l'assemblage par rapport aux trois premiers niveaux de la taxonomie proposée par Beugnard *et al* [Beugnard *et al.*, 1999]. Il propose ainsi une définition des prototypes des opérations des composants, la possibilité d'ajouter un ensemble d'assertions au niveau de ces opérations, une expression des dépendances entre composants et la spécification du comportement des composants à l'aide d'automates à états finis. Une des extensions possibles de ce modèle consiste à étudier les possibilités d'expression de la qualité de service au niveau de l'interface des composants. Actuellement, seul AADL propose un premier niveau de description de la qualité de service avec la possibilité d'annoter à l'aide de propriétés et d'extension les éléments du modèle de composant. Nous souhaitons dans SafArchie proposer un tel mécanisme d'annotations et quelques exemples de son utilisation afin d'être capable de proposer des outils pour valider une architecture par rapport à certaines propriétés liées à la qualité de service.

Langage de coupe et recherche exhaustive de points de jonction compatibles avec un masque de point de jonction Une autre perspective de TranSAT consiste à continuer à travailler sur l'utilisation de moteurs de règles pour la recherche des lieux d'intégration compatibles avec un masque de point de jonction. Ce mécanisme de recherche doit permettre en outre de proposer un langage de coupe adapté au modèle d'architecture. Nous souhaitons étudier, dans ce sens, comment les travaux sur la logique de description ou sur la logique des prédicats peuvent nous servir pour effectuer les requêtes sur un modèle d'architecture. L'objectif est alors d'associer, avec les modèles de description de masque de point de jonction et de coupe, des langages de contraintes dédiés faciles à appréhender pour l'architecte. La mise en œuvre des recherches définies par ces langages s'effectuera dans les couches basses à l'aide d'un moteur de règles pour modéliser les relations entre les éléments d'une architecture logicielle et traduire ces contraintes. Le choix entre l'utilisation de la logique de description ou de la logique des prédicats sera alors fortement guidé par le pouvoir d'expression que l'on souhaite obtenir au niveau du langage de contraintes de haut niveau. Un des objectifs de ce travail vise à estimer la taille de l'architecture qui pourra être analysée en fonction de la taille et des caractéristiques des requêtes que l'on peut exprimer sur cette architecture.

Bibliographie

- [Aagedal, 2001] J. Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department for Informatics, University of Oslo, juin 2001. 23
- [Abmann, 2003] U. Abmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN : 3540443851. 129
- [Aldrich *et al.*, 2002] J. Aldrich, C. Chambers, and D. Notkin. ArchJava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197, New York, mai 19–25 2002. ACM Press. 2, 31, 39, 177
- [Allen *et al.*, 1998] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, mars 1998. An expanded version of a the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997. 32
- [Allen, 1997] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, janvier 1997. Issued as CMU Technical Report CMU-CS-97-144. 2, 31
- [Alliance, 2003] OSGi Alliance. Osgi service platform, release 3, mars 2003. 45
- [Alpern and Schneider, 1985] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4) :181–185, 1985. 108
- [AspectJ Team, 2002 2003] AspectJ Team. The AspectJ programming guide. Disponible sur <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>, 2002-2003. 15
- [Bailly, 2005] A. Bailly. *Test et Validation de Composants Logiciels*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, décembre 2005. 176
- [Barreto, 2003] L. Porto Barreto. *Conception aisée et robuste d'ordonnanceurs de processus au moyen d'un langage dédié*. Thèse de doctorat, Université de Rennes 1, France, juin 2003. 15
- [Barros *et al.*, 2005] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In *Proceedings of SPIN'05*. Springer Verlag, 2005. To appear. 46
- [Batory and O'Malley, 1992] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4) :355–398, 1992. 14
- [Beugnard *et al.*, 1999] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, juillet 1999. 4, 22, 30, 34, 37, 43, 51, 54, 96, 121, 193
- [Biggerstaff and Perlis, 1989] T. J. Biggerstaff and A. J. Perlis, editors. *Software reusability : vol. 2, applications and experience*. ACM Press, New York, NY, USA, 1989. ISBN : 0-201-50018-3. 12
- [Brown and Wallnan, 1996] A. W. Brown and K. C. Wallnan. Engineering of component-based systems. In *ICECCS '96 : Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, page 414, Washington, DC, USA, 1996. IEEE Computer Society. ISBN : 0-8186-7614-0. 14

- [Bruneton *et al.*, 2004a] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004. ISBN : 3-540-21998-6. 45, 63, 179
- [Bruneton *et al.*, 2004b] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, version 2.0-3, février 2004. Online documentation <http://fractal.objectweb.org/specification/>. 2, 31, 43, 73
- [Büchi and Weck, 1999] M. Büchi and W. Weck. The greybox approach : When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, août 1999. ISBN : 952-12-0508-3. 109
- [Budinsky *et al.*, 2003] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional ; 1st edition, 2003. ISBN : 0131425420. 16
- [Caldiera and Basili, 1991] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2) :61–70, 1991. 12
- [Chang and Collet, 2005] H. Chang and P. Collet. Vers la négociation de contrats dans les composants logiciels hiérarchiques. In *Langages, Modèles, Objets (LMO'05)*, volume 11 of *RSTI L'objet*, pages 239–252, Berne, Suisse, mars 2005. Hermès Sciences. 81
- [Cobben *et al.*, 1998] J.M.H. Cobben, A. Engels, S. Mauw, and M.A. Reniers. *Formal Semantics of Message Sequence Charts - Annex B to recommendation Z.120*. ITU-T, avril 1998. 60
- [Collet *et al.*, 2005] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A contracting system for hierarchical components. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2005. ISBN : 3-540-25877-9. 22, 46, 96
- [Consel and Marlet, 1998] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *LNCS*, pages 170–194, Pisa, Italy, septembre 1998. 15, 61
- [Coulson *et al.*, 2002] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2) :109–126, 2002. 39
- [Dashofy *et al.*, 2001] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, Washington, DC, USA, 2001. IEEE Computer Society. ISBN : 0-7695-1360-3. 45
- [David, 2005] P-C. David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. Phd thesis, Université de Nantes / École des Mines de Nantes, juillet 2005. 135
- [de Alfaro and Henzinger, 2001] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120. ACM Press, 2001. ISBN : 1-58113-390-1. 109
- [DeMichiel, 2003] L. G. DeMichiel. *Entreprise JavaBeans Specification, v2.1*. Sun Microsystems, novembre 2003. 43
- [Dijkstra, 1972] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10) :859–866, 1972. 1
- [Dissaux, 2004] P. Dissaux. Using the AADL for mission critical software development. In *Proceedings of ERTS conference*, janvier 2004. 2
- [Donsez *et al.*, 2004] Didier Donsez, Humberto Cervantes, and Mikael Desertot. Frogi : Déploiement de composants fractal sur OSGi. In *1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels (DECOR 04)*, pages 147–158, 2004. 45
- [Dowling and Cahill, 2001] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag. ISBN : 3-540-42618-3. 39

-
- [Dubus, 2005] J. Dubus. *Modélisation et Génération Automatique d'Infrastructures de Déploiement d'Applications Réparties*. Mémoire de Master Recherche, Université des Sciences et Technologies de Lille, France, juillet 2005. 116
- [Ducournau et al., 1998] R. Ducournau, J. Euzenat, G. Masini, and A. Napilo, editors. *Langages et modèles à objets : état et perspectives de la recherche*. Number 19 in Didactique. INRIA, Rocquencourt (FR), 1998. ISBN : 2-7261-1131-9. 1, 101
- [EclipseWWW, 2001 2005] Eclipse Project website, 2001 - 2005. [Online] <http://www.eclipse.org/>. 162
- [Escoffier and Donsez, 2005] C. Escoffier and D. Donsez. Fractnet : A fractal implementation for .net. Session poster de la 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005), septembre 2005. Disponible sur : <http://www.lifl.fr/jfdlpa05/>. 45
- [Estublier et al., 2005a] J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005. 16
- [Estublier et al., 2005b] J. Estublier, G. Vega, and A. D. Ionita. Composing domain specific languages for wide-scope software engineering applications. In *Proceedings of the ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems (MoDELS'05 - formerly UML conference series)*, Montego Bay, Jamaica, octobre 2005. To appear. 17
- [Fassino et al., 2002] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK : A software framework for component-based operating system kernels. In *USENIX'02*, pages 73–86, Monterey, CA, USA, 2002. USENIX. 45, 63
- [Filman and Friedman, 2000] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, octobre 2000. 130
- [Floyd, 1967] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science : Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society. 21
- [Forgy, 1982] Charles Forgy. Rete : A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1) :17–37, 1982. 182
- [Frølund and Koistinen, 1998] S. Frølund and J. Koistinen. Quality of service specification in distributed object systems design. In *COOTS*, pages 1–18. USENIX, 1998. 23
- [Gagnon and Hendren, 1998] E. M. Gagnon and L. J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98 : Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 140–154, Washington, DC, USA, 1998. IEEE Computer Society. ISBN : 0-8186-8482-8. 170, 172
- [Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995. 17, 18
- [Gansner and North, 2000] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11) :1203–1233, 2000. 15
- [Garavel, 2003] H. Garavel. Défense et illustration des algèbres de processus. In *Actes de l'Ecole d'été Temps Réel ETR 2003*, Toulouse, France, septembre 2003. In Zoubir Mammer. 22, 82, 107
- [Garlan and Shaw, 1993] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993. 1, 14, 69
- [Garlan et al., 1997] D. Garlan, R. Monroe, and D. Wile. ACME : An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, novembre 1997. 2, 43
- [Ghezzi et al., 1991] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN : 0-13-820432. 13
- [Giannakopoulou, 1999] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, janvier 1999. 31, 36
- [Greenfield et al., 2004] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley ; 1st edition, 2004. ISBN : 0471202843. 16

- [Harel, 1987] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, juin 1987. 23
- [Heineman and Councill, 2001] G. Heineman and W. Councill, editors. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Westley, 2001. ISBN : 0-201-70485-4. 1, 13
- [Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, octobre 1969. 21, 33
- [Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 23, 31, 83, 86
- [Holub, 2003] A. Holub. Why extends is evil. In *JavaWorld*, août 2003. Disponible sur : <http://www.javaworld.com>. 1
- [Hoschka, 1998] P. Hoschka. An introduction to the synchronized multimedia integration language. *IEEE Multi-Media*, 5(4) :84–88, 1998. 23
- [ISO, 1989] ISO. *IS8807 : Information Processing Systems - Open System Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behavior*. ISO, février 1989. 23, 83
- [ITU, 1999a] ITU. Specification and Description Language (SDL). ITU-T Recommendation n° Z.100, novembre 1999. 23
- [ITU, 1999b] ITU-T. *Message Sequence Chart (MSC) - Recommendation Z.120*, novembre 1999. 60
- [Jouault and Kurtev, 2005] F. Jouault and I. Kurtev. Transforming models with atl. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005. <http://so-sym.dcs.kcl.ac.uk/events/mtipl/>. 136
- [Jézéquel and Meyer, 1997] J.-M. Jézéquel and B. Meyer. Design by contract : The lessons of ariane. *Computer*, 30(1) :129–130, janvier 1997. 2, 20
- [Kalibera and Tuma, 2002] T. Kalibera and P. Tuma. Distributed component system based on architecture description : The sofa experience. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 981–994, London, UK, octobre 2002. Springer-Verlag. ISBN : 3-540-00106-9. 31, 48
- [Ketfi, 2004] A. Ketfi. *Une Approche Générique pour la Reconfiguration Dynamique des Applications à base de Composants Logiciels*. Thèse de doctorat, Université Joseph Fourier de Grenoble, France, décembre 2004. 116
- [Kiczales et al., 1997] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. 3
- [Kiczales, 1996] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es) :154, 1996. 15
- [Kruchten, 2000] P. Kruchten. *The Rational Unified Process : An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN : 0201707101. 19, 144
- [Lanergan and Grasso, 1989] R. G. Lanergan and C. A. Grasso. Software engineering with reusable designs and code. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, pages 187–195. acm press, New York, 1989. 12
- [Layaïda and Hagimont, 2005] O. Layaïda and D. Hagimont. Plasma : A component-based framework for building self-adaptive applications. In *Proc. SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications*, San Jose, CA, USA, January 2005. 45
- [Leavens et al., 2000] G. T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML : notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. The Association for Computing Machinery, octobre 2000. 21
- [Ledoux et al., 2001] T. Ledoux, M. Blay, E. Bruneton, D. Caromel T. Coupaye, D. Hagimont, J.-M. Menaud, J. Noyé, and M. Riveill. État de l’art sur l’adaptabilité. Technical Report Livrable 1.1, Projet RNTL Arcad, décembre 2001. 3
- [Legond-Aubry et al., 2003] F. Legond-Aubry, G. Florin, and D. Enselme. Modèle abstrait d’assemblage de composants par contrats. Technical Report Livrable 1.4, Projet RNTL Accord, 2003. 22

-
- [Legond-Aubry, 2005] F. Legond-Aubry. *Un modèle d'assemblage de composants par Contrat et Programmation Orientée Aspect*. Thèse de doctorat, Conservatoire National des Arts et Métiers (CNAM), France, juillet 2005. 22
- [Luckham *et al.*, 1995] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, 1995. 2, 31
- [Lynch and Tuttle, 1989] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3) :219–246, 1989. 23, 84
- [Magee and Kramer, 1999] J. Magee and J. Kramer. *Concurrency. State Models and Java Programs*. wiley, 1999. 23, 83
- [Magee *et al.*, 1995] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989, 1995. 2, 31, 35
- [Magee, 1999] J. Magee. Behavioral analysis of software architectures using Itsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999. ISBN : 1-58113-074-0. 85
- [Manna and Pnueli, 1992] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer-Verlag, 1992. [QA76.6M3564(V1)]. 107
- [Martin, 1996] R. C. Martin. The dependency inversion principle. *C++ Report*, 8, mai 1996. 18
- [Marvie and Pellegrini, 2002] R. Marvie and M. Pellegrini. Modèles de composants, un état de l'art. *Coopération dans les systèmes à objets, Numéro spécial de la revue l'Objet*, 8(3) :61–90, septembre 2002. 2-7462-0520-3. 1
- [McIlroy, 1968] M.D. McIlroy. Mass produced software components. In P. Naur and Randell B., editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee. 1
- [Medvidovic and Taylor, 2000] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, page 23, janvier 2000. 2, 24, 31, 72
- [Mercouroff and Parhar, 1997] N. Mercouroff and A. Parhar. Tina computational modelling concepts and object definition language. In *IS&N '97 : Proceedings of the Fourth International Conference on Intelligence and Services in Networks*, pages 15–24, London, UK, 1997. Springer-Verlag. ISBN : 3-540-63135-6. 23
- [Meyer, 1992a] B. Meyer. Applying “design by contract”. *Computer*, 25(10), octobre 1992. 21, 80
- [Meyer, 1992b] B. Meyer. *Eiffel : the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN : 0-13-247925-7. 21
- [Meyer, 2000] B. Meyer. *Conception et programmation orientées objet*. Eyrolles, juillet 2000. ISBN : 2-212-09111-7. 1, 101
- [Mikhajlov and Sekerinski, 1998] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECCOP '98 : Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998. ISBN : 3-540-64737-6. 1
- [Milner, 1980] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980. 23, 83
- [MVC, 1979] Models - views - controllers, décembre 1979. Technical note. 165
- [OMG, 2002a] Object Management Group OMG. *CORBA Component Model, v3.0, formal/2002-06-65*. Object Management Group OMG, juin 2002. 43
- [OMG, 2002b] Object Management Group OMG. Meta-Object Facility (MOF) Specification, avril 2002. Version 1.4. 16
- [OMG, 2003a] Object Management Group OMG. *Omg/rfp/qvt mof 2.0 query/views/transformations rfp*. Misc, 2003. 16
- [OMG, 2003b] Object Management Group OMG. *UML 1.5 Object Constraint Language Specification*, mars 2003. Version 1.5. 21, 81



- [OMG, 2003c] Object Management Group OMG. *Unified Modeling Language : Superstructure*, août 2003. Version 2.0. 31, 57
- [OMG, 2003d] Object Management Group OMG. *Unified Modeling Language (UML) specification*, mars 2003. Version 1.5. 57
- [OMG, 2005] Object Management Group OMG. *Software Process Engineering Metamodel (SPEM) Specification*, janvier 2005. Version 1.1. 144
- [Parnas, 1972] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12) :1053–1058, 1972. 2, 128
- [Pawlak *et al.*, 2004] R. Pawlak, J.P. Retailié, and L. Seinturier. *La programmation orientée aspect pour J2EE*. Eyrolles, 2004. ISBN : 2-212-11408-7. 14
- [Peterson, 1981] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN : 0136619835. 23
- [Plasil *et al.*, 1997] F. Plasil, D. Balek, and R. Janecek. DCUP : Dynamic Component Updating in Java/CORBA. Technical Report 97/10, Dep. of SW Engineering, Charles University, Prague, 1997. 49
- [Pnueli, 1986] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems : A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, LNCS-224, 1986. 107
- [Roshanak, 2000] E. Dincel Roshanak. ADL independent architectural representation in xml, 2000. 2
- [Ruane, 1984] L. M. Ruane. Abstract data types in assembly language programming. *ACM SIGPLAN Notices*, 19(1) :63–67, janvier 1984. 21
- [SAE, 2004] As-2 Embedded Computing Systems Committee SAE. *Architecture Analysis & Design Language (AADL)*. SAE Standards n° AS5506, novembre 2004. 31
- [Seinturier, 2005] L. Seinturier. *Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties*. Habilitation à diriger des recherches, Université de Paris VI, France, décembre 2005. 45, 63
- [SOA, 2004] Elements of service-oriented analysis and design, juin 2004. Online article. 25
- [Soley and the OMG staff, 2000] R. Soley and the OMG staff. *MDA Model-Driven Architecture*, novembre 2000. Online presentation <http://www.omg.org/mda/presentations.htm>. 16
- [Sriplakich, 2003] P. Sriplakich. *Modèles et environnement pour configurer et déployer des systèmes logiciels*. Rapport de dea, Université Pierre et Marie Curie, septembre 2003. 136
- [Syn, 2002] Synopsys Inc., <http://www.systemc.org>. *SystemC v2.0.1 Users Guide*, juin 2002. 23
- [Szyperski, 1996] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996. 1
- [Szyperski, 1998] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998. 14
- [Tarr *et al.*, 1999] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. N degrees of separation : Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999. 2, 15, 26, 128
- [Tharan, 2002] O. Tharan. *Linux : choisir une distribution*, août 2002. Disponible sur : <http://www.linux-france.org/>. 116
- [Tran *et al.*, 2004] H-M. Tran, P. Bedu, L. Duchien, H.Q. Nguyen, and J. Perrin. Toward structural and behavioral analysis for component models. In *The FSE 2004 Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 138–141, Newport Beach, CA, USA, November 2004. 175
- [Tran *et al.*, 2005] H-M. Tran, L. Duchien, P. Bedu H-Q. Nguyen, and J. Perrin. Figures de transformation pour des architectures logicielles. In *Langages, Modèles, Objets (LMO'05)*, volume 11 of *RSTI L'objet*, pages 63 – 78, Berne, Suisse, mars 2005. Hermès Sciences. 136
- [van Deursen *et al.*, 2000] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, juin 2000. 15, 61

-
- [van Glabbeek, 2001] R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001. Available at <http://boole.stanford.edu/pub/spectrum1.ps.gz>. 158
- [Vestal, 1998] S. Vestal. *MetaH Programmer's Manual Version 1.27*. Honeywell, Inc., Minneapolis, MN, 1998. 54
- [W3C, 2005] W3C. XSL transformations (XSLT) version 2.0, avril 2005. Working Draft 4. 136
- [Weyuker, 1998] E. J. Weyuker. Testing component-based software : A cautionary tale. *IEEE Software*, 15(5) :54–59, 1998. 2
- [Wirth, 1977] Niklaus Wirth. Modula : a language for modular multiprogramming. *Softw., Pract. Exper.*, 7(1) :3–35, 1977. 13
- [Zelesnik, 1996] G. Zelesnik. *The UniCon Language Reference Manual*. School of Computer Science Carnegie Mellon University, Pittsburgh, mai 1996. 2

Annexe A

Grammaire du langage textuel de description de type d'architecture

```
Package fr.lifl.safarchie.architecturetype.parser;
```

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = [[32..127] - [cr + lf]];
eol = cr | lf | cr | lf;
point = '.';
```

Tokens

```
blank = (' ' | tab | eol)+;
new_line = cr | lf | cr lf;
architecture_type = 'architectureType';
composite_type = 'compositeType';
primitive_type = 'primitiveType';
binding = 'binding';
delegatedport_type = 'delegatedportType';
port_type = 'portType';
operation = 'operation';
raises = 'raises';

open = '(';
close = ')';
obracket = '[';
cbracket = ']';
assign = '=';
semi = ';';
comma = ',';
namespace_sep = '::';

void = 'void';
float = 'float';
double = 'double';
long = 'long';
short = 'short';
unsigned = 'unsigned';
char = 'char';
boolean = 'boolean';
octet = 'octet';
any = 'any';
string = 'string';

provide = 'provide';
require = 'require';
identifier = (letter_maj | letter_min) (letter_maj | letter_min | digit)*;
```

Ignored Tokens

```
blank;
```

Productions

```
architecture_type_decl = identifier obracket composite_type_decl cbracket ;
composite_type_decl = composite_type_header obracket composite_type_decl* primitive_type_decl* binding_decl* delegatedport_type_decl* cbracket ;

composite_type_header = composite_type identifier ;

primitive_type_decl = primitive_header obracket primitive_body cbracket ;
primitive_header = primitive_type identifier ;
//primitive_body = attribute_decl* porttype_decl* ;
primitive_body = porttype_decl* ;

binding_decl = binding_header binding_body semi ;
binding_header = binding identifier ;
binding_body = open [left]:porttype_ref comma [right]:porttype_ref close ;
```

```

porttype_ref = scoped_name ;

delegatedport_type_decl = delegatedport_type_header delegatedport_type_body semi;
delegatedport_type_header = delegatedport_type identifier ;
delegatedport_type_body = assign porttype_ref ;

attribute_decl = param_type_spec identifier semi;

porttype_decl = port_header obrace port_body cbrace ;
port_header = port_type identifier ;
port_body = operation_dcl+ ;

operation_dcl = operation_type op_return_type_spec identifier parameter_dcls raises_expr? semi ;

operation_type = {require} require
| {provide} provide;

op_return_type_spec = {returntype} param_type_spec |
| {noreturn} void ;
parameter_dcls = {parameter} open param_dcl other_param* close
| {noparameter} open close ;

other_param = coma param_dcl ;

param_dcl = param_type_spec param_name ;
param_name = identifier ;
raises_expr = raises open scoped_name other_exceptions* close ;
other_exceptions = coma scoped_name ;
param_type_spec = {base_type} base_type_spec
| {string} string_type
| {complextype} scoped_name ;

base_type_spec = {float}floating_pt_type
| {integer}integer_type
| {char} char_type
| {boolean} boolean_type
| {octet} octet_type
| {any} any_type ;

floating_pt_type = {simple} float | {double} double ;
integer_type = {signed} signed_int | {unsigned}unsigned_int ;
signed_int = {long}signed_long_int | {short} signed_short_int ;
signed_long_int = long ;
signed_short_int = short ;
unsigned_int = {ulong} unsigned_long_int | {ushort} unsigned_short_int ;
unsigned_long_int = unsigned long ;
unsigned_short_int = unsigned short ;
char_type = char ;
boolean_type = boolean ;
octet_type = octet ;
any_type = any ;

string_type = string ;

scoped_name = {simplescoped} identifier
| {refscoped} namespace_sep identifier
| {recursivescoped} scoped_name namespace_sep identifier ;

```

Annexe B

Schéma XML du modèle de type d'architecture

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  jxb:version="1.0"
  jxb:extensionBindingPrefixes="xjc">
  <xsd:annotation>
    <xsd:appinfo>
      <jxb:globalBindings generateIsSetMethod="true">
        <xjc:serializable uid="1"/>
        <xjc:superClass name="org.argouml.memory.SafArchicObject"/>
      </jxb:globalBindings>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:element name="ArchitectureType" type="ArchitectureTypeModel"/>
  </xsd:element>
  <xsd:element name="CompositeType" type="CompositeTypeModel"/>
  <xsd:element name="PortType" type="PortTypeModel"/>
  <xsd:element name="Parameters" type="parametersType"/>
  <xsd:complexType name="ComponentTypeModel">
    <xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.ComponentTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="TypeElement">
        <xsd:sequence>
          <xsd:element ref="PortType" maxOccurs="unbounded"/>
          <xsd:element name="Attribute" type="attributType" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="BehavioralContract" type="BehavioralContractModel" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="Height" type="xsd:integer" use="optional"/>
        <xsd:attribute name="Y" type="xsd:integer" use="optional"/>
        <xsd:attribute name="X" type="xsd:integer" use="optional"/>
        <xsd:attribute name="Width" type="xsd:integer" use="optional"/>
        <xsd:attribute name="Cardinality" type="CardinalityType" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="PortTypeModel">
    <xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.PortTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="TypeElement">
        <xsd:sequence>
          <xsd:element ref="Operation" maxOccurs="unbounded"/>
          <xsd:element name="AssemblyContract" type="AssemblyContractModel" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="Cardinality" type="CardinalityType" use="required"/>
        <xsd:attribute name="Height" type="xsd:integer" use="optional"/>
        <xsd:attribute name="Y" type="xsd:integer" use="optional"/>
        <xsd:attribute name="X" type="xsd:integer" use="optional"/>
        <xsd:attribute name="Width" type="xsd:integer" use="optional"/>
        <xsd:attribute name="PortID" type="PortID" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TypeOperation">
    <xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.OperationTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="TypeElement">
        <xsd:sequence>
          <xsd:element ref="Parameters" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="AssertionContract" type="AssertionContractModel" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="Sens" type="xsd:boolean" use="required"/>
        <xsd:attribute name="Return" type="type" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TypeElement">
    <xsd:attribute name="Name" type="xsd:string" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="attributType">
    <xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.AttributTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
```

```

<xsd:complexContent>
  <xsd:extension base="TypeElement">
    <xsd:sequence>
      <xsd:element name="Type" type="type"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="parametersType">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.ParameterExtImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element name="Type" type="type"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="nomAttributType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[a-zA-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="typeAttributType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:simpleType name="attributID">
  <xsd:restriction base="xsd:ID">
    <xsd:pattern value="attribut[a-zA-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="Operation" type="TypeOperation"/>
<xsd:simpleType name="type">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[a-zA-Z_][a-zA-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="CompositeTypeModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.CompositeTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element ref="CompositeType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="ComponentType" type="ComponentTypeModel" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="Bindings" type="BindingsType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="DelegatedPortType" type="DelegatedPortTypeModel" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="Height" type="xsd:integer" use="optional"/>
      <xsd:attribute name="Y" type="xsd:integer" use="optional"/>
      <xsd:attribute name="X" type="xsd:integer" use="optional"/>
      <xsd:attribute name="Width" type="xsd:integer" use="optional"/>
      <xsd:attribute name="Cardinality" type="CardinalityType" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="BindingsType">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.BindingExtImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element name="BindingPort" type="PortRef" minOccurs="2" maxOccurs="2"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="PortRef" final="#all">
  <xsd:restriction base="xsd:IDREF">
    <xsd:pattern value="port[a-zA-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="PortID">
  <xsd:restriction base="xsd:ID">
    <xsd:pattern value="port[a-zA-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="DelegatedPortTypeModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.DelegatedPortTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element name="Portref" type="PortRef"/>
      </xsd:sequence>
      <xsd:attribute name="Height" type="xsd:integer" use="optional"/>
      <xsd:attribute name="Y" type="xsd:integer" use="optional"/>
      <xsd:attribute name="X" type="xsd:integer" use="optional"/>
      <xsd:attribute name="Width" type="xsd:integer" use="optional"/>
      <xsd:attribute name="PortID" type="PortID" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="AssertionContractModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.AssertionContractImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element name="Contract" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="AssemblyContractModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.AssemblyContractImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>

```

```

        <xsd:element name="Contract" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="BehavioralContractModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.BehavioralContractImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element name="Contract" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="CardinalityType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="1"/>
    <xsd:enumeration value="2"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="ArchitectureTypeModel">
<xsd:annotation><xsd:appinfo><jxb:class implClass="org.argouml.memory.ArchitectureTypeExtImpl"/></xsd:appinfo> </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="TypeElement">
      <xsd:sequence>
        <xsd:element ref="CompositeType" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="Author" type="xsd:string" use="optional"/>
      <xsd:attribute name="Version" type="xsd:string" use="optional"/>
      <xsd:attribute name="Description" type="xsd:string" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
</xsd:schema>

```


Annexe C

Grammaire du langage de contrat de port

```
Package safarchie.portcontract.parser;
```

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = {[32..127] - {cr + lf}};
eol = cr lf | cr | lf;
point = '.';
```

Tokens

```
blank = ('_' | tab | eol)+;
new_line = cr | lf | cr lf;

open = '(';
close = ')';
assign = '=';
semi = ';';
or = '|';
and = '+';

label = letter_min (letter_maj | letter_min | digit)* point (letter_maj | letter_min | digit)* ;
identifier = letter_maj (letter_maj | digit)*;

contract = 'portcontract';
```

Ignored Tokens

```
blank;
```

Productions

```
portcontractdecl = exp* semi;

exp = {and} {left}:exp and {right}:operation
| {or} {left}:exp or {right}:operation
| {oper} operation;

operation = {operation} label
| {expression} open exp close ;
```


Annexe D

Grammaire du langage de contrat d'assertion

Package test;

Helpers

```
all = [0..127];
lf = 10;
cr = 13;
backslash = 92;
gu = 39;
uppercase = ['A'.. 'Z'];
lowercase = ['a'.. 'z'];
digit = ['0'.. '9'];
number = digit+;
line_terminator = lf | cr | lf cr;
input_character = [all - {cr + lf}];

simple_escape_sequence = backslash | gu | '"' | '?' | '\\' |
  '\a' | '\b' | '\f' | '\n' | '\r' | '\t' | '\v';
octal_digit = ['0' .. '7'];
octal_escape_sequence = backslash octal_digit? octal_digit?;
hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
hexadecimal_escape_sequence = backslash hexadecimal_digit+;
escape_sequence = simple_escape_sequence |
  octal_escape_sequence |
  hexadecimal_escape_sequence;
s_char = [all - {gu + {backslash + {lf + cr}}}] | escape_sequence;
s_char_sequence = s_char+;
h_set = 'Set';
h_bag = 'Bag';
h_sequence = 'Sequence';
h_collection = 'Collection';
```

Tokens

```
comment = '--' [[all - 10] - 13]* [10 + 13]?;
// comment = '--' input_character* line_terminator?;

dot = '.';
arrow = '->';

not = 'not';

mult = '*';
div = '/';
plus = '+';
minus = '-';

context = 'context';
enum = 'enum';
t_pre = 'pre';
t_post = 'post';
t_inv = 'inv';

equal = '=';
n_equal = '<=>';
lt = '<';
gt = '>';
lteq = '<=';
gteq = '>=';

and = 'and';
or = 'or';
xor = 'xor';
implies = 'implies';

l_par = '(';
r_par = ')';
l_bracket = '[';
```

```

r_bracket = ']' ;
l_brace = '{' ;
r_brace = '}' ;
semicolon = ';' ;

dcolon = '::' ;
colon = ':' ;
comma = ',' ;
channel = '#';
at = '@';
bar = '|';
ddot = '..';
apostroph = gu;

t_let = 'let';
t_in = 'in';

t_if = 'if';
t_then = 'then';
t_else = 'else';
endif = 'endif';

t_set = h_set;
t_bag = h_bag;
t_sequence = h_sequence;
t_collection = h_collection;

bool = 'true' | 'false';
simple_type_name = ( uppercase ( lowercase | digit | uppercase | '_' ) * ) |
  h_set | h_bag | h_sequence | h_collection;
name = lowercase ( lowercase | digit | uppercase | '_' ) * ;

new_line = line_terminator;
int = number;
real = number '.' number;
blank = 9 | '\s' * ;
tab = 9;

string_lit = '\'' s_char_sequence? '\'' ;

Ignored Tokens

comment,
new_line,
blank,
tab;

Productions

constraint =
  context_declaration constraint_body+ ;

// added (SableCC)
constraint_body =
  stereotype name? colon expression ;

context_declaration =
  context context_body ;

context_body =
  {component} [component]:name dcolon port_decl |
  {operation} name classifier_type? :
    port_decl = [port]:name dcolon name l_par formal_parameter_list r_par return_type_declaration? ;
    classifier_type = colon name;

classifier_context =
  classifier_head? type_name ;

// added (SableCC)
classifier_head =
  name colon ;

// added (SableCC)
return_type_declaration =
  colon type_name;

formal_parameter_list =
  formal_parameter formal_parameter_list_tail* ;

// added (SableCC)
formal_parameter_list_tail =
  semicolon formal_parameter;

formal_parameter =
  name colon type_name ;

stereotype =
  {inv} t_inv |
  {pre} t_pre |
  {post} t_post ;

expression =
  let_expression* logical_expression ;

if_expression =
  t_if [if_branch]:expression t_then [then_branch]:expression t_else [else_branch]:expression endif ;

logical_expression =
  relational_expression logical_expression_tail* ;

// added (SableCC)
logical_expression_tail =
  logical_operator relational_expression ;

relational_expression =

```

```

    additive_expression relational_expression_tail? ;

// added (SableCC)
relational_expression_tail =
    relational_operator additive_expression ;

additive_expression =
    multiplicative_expression additive_expression_tail* ;

// added (SableCC)
additive_expression_tail =
    add_operator multiplicative_expression ;

multiplicative_expression =
    unary_expression multiplicative_expression_tail* ;

// added (SableCC)
multiplicative_expression_tail =
    multiply_operator unary_expression ;

unary_expression =
    {unary} unary_operator postfix_expression |
    {postfix} postfix_expression ;

postfix_expression =
    primary_expression postfix_expression_tail* ;

// added (SableCC)
postfix_expression_tail =
    postfix_expression_tail_begin feature_call ;

// added (SableCC)
postfix_expression_tail_begin =
    {dot} dot |
    {arrow} arrow ;

primary_expression =
    {lit_col} literal_collection |
    {literal} literal |
//    {feature} path_name time_expression? qualifiers? feature_call_parameters? |
    {feature} path_name qualifiers? feature_call_parameters? |
    {parentheses} l_par expression r_par |
    {if} if_expression ;

feature_call_parameters =
    {empty} l_par r_par |
    {concrete} l_par expression fcp_helper* r_par |
    ( l_par declarator? actual_parameter_list? r_par ) ;

fcp_helper =
    {comma} comma expression |
    {colon} colon simple_type_specifier |
    {iterate} semicolon name colon simple_type_specifier equal expression |
    {bar} bar expression ;

let_expression =
    t_let name let_expression_type_declaration? equal expression t_in ;

// added (SableCC)
let_expression_type_declaration =
    colon path_type_name ;

// added alternatives real and boolean
literal =
    {string} string_lit |
    {real} real |
    {integer} int |
    {boolean} bool |
    {enum} channel name ;

enumeration_type =
    enum l_brace channel name enumeration_type_tail* r_brace ;

// added (SableCC)
enumeration_type_tail =
    comma channel name ;

simple_type_specifier =
    {path} path_type_name |
    {enum} enumeration_type ;

literal_collection =
    collection_kind l_brace expression_list_or_range? r_brace ;

expression_list_or_range =
    expression expression_list_or_range_tail? ;

// added (SableCC)
expression_list_or_range_tail =
    {list} expression_list_tail+ |
    {range} ddot expression ;

// added (SableCC)
expression_list_tail =
    comma expression ;

feature_call =
    path_name qualifiers? feature_call_parameters? ;

qualifiers =
    l_bracket actual_parameter_list r_bracket ;

// "iterate" alternative added (missing in specification)
declarator =
    {standard} name declarator_tail* declarator_type_declaration? bar |
    {iterate} {iterator}:name {iter_type}:declarator_type_declaration semicolon
    {accumulator}:name {acc_type}:declarator_type_declaration equal expression bar ;

```

```

// added (SableCC)
declarator_tail =
  comma name ;

// added (SableCC)
declarator_type_declaration =
  colon simple_type_specifier ;

path_type_name =
  type_name path_type_name_tail* ;

// added (SableCC)
path_type_name_tail =
  dcolon type_name ;

// added to solve keyword conflict
type_name =
  {non_collection} simple_type_name |
  {collection} collection_type l_par simple_type_name r_par;

collection_type =
  {set} t_set |
  {bag} t_bag |
  {sequence} t_sequence |
  {collection} t_collection;

path_name =
  path_name_begin path_name_tail* ;

// added (SableCC)
path_name_begin =
  {type_name} type_name |
  {name} name ;

// added (SableCC)
path_name_tail =
  dcolon path_name_end ;

// added (SableCC)
path_name_end =
  {type_name} type_name |
  {name} name ;

actual_parameter_list =
  expression actual_parameter_list_tail* ;

// added (SableCC)
actual_parameter_list_tail =
  comma expression ;

logical_operator =
  {and} and |
  {or} or |
  {xor} xor |
  {implies} implies;

collection_kind =
  {set} t_set |
  {bag} t_bag |
  {sequence} t_sequence |
  {collection} t_collection ;

relational_operator =
  {equal} equal |
  {n_equal} n_equal |
  {gt} gt |
  {lt} lt |
  {gteq} gteq |
  {lteq} lteq ;

add_operator =
  {plus} plus |
  {minus} minus;

multiply_operator =
  {mult} mult |
  {div} div;

unary_operator =
  {minus} minus |
  {not} not;

```

Annexe E

Grammaire du langage SFSP permettant la description de comportement

```
Package safarchie.sfsp.parser;
```

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = {[32..127] - [cr + lf]};
eol = cr | lf | cr | lf;
point = '.';
send = '!';
receive = '?';
return = '$';
```

Tokens

```
blank = ('_' | tab | eol)+;
new_line = cr | lf | cr lf;

sequence = '->';

iteration = '^';
union = '|';
star = '*';
open = '(';
close = ')';
shuffle = '#';
mix = '||';
lbrace = '{';
rbrace = '}';
cbrace = '{';
relab = '/';
hide = '@';
assign = '=';
semi = ';';
coma = ',';
label = (send | receive) (letter_maj | letter_min) (letter_maj | letter_min | digit)*
        point (letter_maj | letter_min) (letter_maj | letter_min | digit)*
        point (letter_maj | letter_min) (letter_maj | letter_min | digit)* return? ;

identifieur = letter_maj (letter_maj | digit)*;
```

Ignored Tokens

```
blank;
```

Productions

```
behaviors = behavior+ ;
```

```
behavior =
```

```
{primitive} identifieur assign primitive_behavior_description semi
| {composite} identifieur assign composite_behavior_description semi;
```

```
// (A||B)@{. } /(p1/p2, )
```

```
composite_behavior_description = open processcomposition close relabelisation interfacehiding;
```

```
processcomposition = {composition} [left]:identifieur mix [right]:processcomposition
| {identifieur} identifieur;
```

```
// /(p1/p2, ...)
```

```
relabelisation = relab lbrace relabdef relabext* rbrace;
```

```
relabdef = [left]:label relab [right]:label;
```

```
relabext = coma relabdef;
```

Annexe E. Grammaire du langage SFSP permettant la description de comportement

```
interfacehiding = hide obrace label interfacehidingext cbrace;  
interfacehidingext = coma label;
```

```
primitive_behavior_description = {complex} open [left]:seqexpression sequence [right]:union_behavior_description close  
                                |{simple_seq} open [left]:seqexpression sequence [right]:identifier close  
                                | {simple_union} union_behavior_description;
```

```
union_behavior_description = open finiteseqexpression bexpression+ close;  
bexpression = union finiteseqexpression;  
finiteseqexpression = [left]:seqexpression sequence [right]:identifier;
```

```
seqexpression = {recursionwithcoma} open [left]:label sequence [right]:seqexpression close ?star  
                | {recursion} [left]:label sequence [right]:seqexpression  
                | {unique} label ?star;
```

Annexe F

Grammaire du langage textuel de description de type d'architecture

```
Package fr.lifl.safarchie.architecturelogic.parser;
```

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = [(32..127) - (cr + lf)];
eol = cr | lf | cr | lf;
point = '.';
```

Tokens

```
blank = (' ' | tab | eol)+;
new_line = cr | lf | cr | lf;
architecture_logic = 'architectureType';
composite = 'compositeType';
primitive = 'primitiveType';
connexion = 'connexion';
delegatedport = 'delegatedport';
port = 'port';
instof = '@';
```

```
open = '(';
close = ')';
lbrace = '{';
rbrace = '}';
assign = '=';
semi = ';';
coma = ',';
namespace_sep = '::';
```

```
identifier = (letter_maj | letter_min) (letter_maj | letter_min | digit)* ;
```

Ignored Tokens

```
blank;
```

Productions

```
architecture_logic_decl = identifier instanceof obrace composite_decl cbrace ;
composite_decl = composite_header obrace composite_decl* primitive_decl* connexion_decl* delegatedport_decl+ cbrace ;
```

```
composite_header = composite identifier instanceof;
```

```
primitive_decl = primitive_header obrace primitive_body cbrace ;
primitive_header = primitive identifier instanceof;
primitive_body = port_dcl+ ;
```

```
connexion_decl = connexion_header connexion_body semi ;
connexion_header = connexion identifier instanceof ;
connexion_body = open {left}:port_ref coma {right}:port_ref close ;
port_ref = scoped_name ;
```

```
delegatedport_decl = delegatedport_header delegatedport_body semi;
delegatedport_header = delegatedport identifier instanceof ;
delegatedport_body = assign port_ref ;
```

```
port_dcl = port_header instanceof semi ;
port_header = port identifier ;
```

```
instanceof = instof scoped_name;
scoped_name = (simplescoped) identifier
| (refscoped) namespace_sep identifier
| (recursivescoped) scoped_name namespace_sep identifier ;
```


Annexe G

Schéma XML du modèle d'architecture logique de SafArchie

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="PointCutMask">
    <xs:annotation>
      <xs:documentation>Comment describing your root
        element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ComponentMask" maxOccurs="unbounded"/>
        <xs:element ref="BindingMask" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="ComponentMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ComponentMaskModel"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="ComponentMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="SequenceMask" minOccurs="0"
            maxOccurs="unbounded"/>
          <xs:element ref="PortMask" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="typeof" type="ComponentMaskType" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="PortMask" type="PortMaskModel"/>
  <xs:complexType name="PortMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="OperationMask" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="SequenceMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="SequenceMaskModel"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="SequenceMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="MessageMask" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="MessageMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="TaggedElement">
          <xs:attribute name="type" type="MessageType" use="required"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

        <xs:attribute name="sens" type="MessageSens" use="required"/>
        <xs:attribute name="operationMaskRef" type="RefToOperationMask"/>
    </xs:extension>
</xs:complexType>
</xs:element>
<xs:simpleType name="MessageSens">
    <xs:restriction base="xs:string">
        <xs:enumeration value="sent"/>
        <xs:enumeration value="receipt"/>
        <xs:enumeration value="any"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="MessageType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Request"/>
        <xs:enumeration value="Return"/>
        <xs:enumeration value="Any"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PolarityType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Provide"/>
        <xs:enumeration value="Require"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ComponentMaskType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="component"/>
        <xs:enumeration value="composite"/>
        <xs:enumeration value="any"/>
    </xs:restriction>
</xs:simpleType>
<xs:element name="OperationMask" type="OperationMaskModel"/>
<xs:complexType name="OperationMaskModel">
    <xs:complexContent>
        <xs:extension base="TaggedElement">
            <xs:sequence>
                <xs:element ref="ParameterMask" minOccurs="0"
                    maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="polarity" type="PolarityType" use="required"/>
            <xs:attribute name="return" type="xs:string"/>
            <xs:attribute name="nbParameter" type="xs:short"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="ParameterMask">
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="TaggedElement">
                <xs:attribute name="type" type="xs:string" use="required"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
<xs:complexType name="TaggedElement">
    <xs:attribute name="tag" type="xs:ID" use="required"/>
</xs:complexType>
<xs:element name="BindingMask" type="BindingMaskModel"/>
<xs:complexType name="BindingMaskModel">
    <xs:complexContent>
        <xs:extension base="TaggedElement">
            <xs:sequence>
                <xs:element ref="BoundPort" minOccurs="2" maxOccurs="2"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="BoundPort" type="BoundPortType"/>
<xs:complexType name="BoundPortType">
    <xs:attribute name="portRef" type="PortRef" use="required"/>
</xs:complexType>
<xs:simpleType name="PortRef">
    <xs:restriction base="xs:IDREF"/>
</xs:simpleType>
<xs:simpleType name="RefToOperationMask">
    <xs:restriction base="xs:IDREF"/>
</xs:simpleType>
</xs:schema>

```

Annexe H

Grammaire du langage textuel de description de masque de point de jonction

Package fr.lifl.transat.joinpointmask.parser;

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = [(32..127) - {cr + lf}];
col = cr | lf | cr | lf;
```

Tokens

```
blank = (' ' | tab | eol)+;
new_line = cr | lf | cr lf;
joinpointmask = 'Mask';
componentmask = 'ComponentMask';
connexionmask = 'ConnexionMask';
portmask = 'PortMask';
operationmask = 'OperationMask';
parametermask = 'ParameterMask';
messageseqmask = 'MessageSequenceMask';
messagemask = 'MessageMask';
```

```
primitive = 'primitive';
composite = 'composite';
provide = 'provide';
require = 'require';
request = 'request';
return = 'return';
sent = 'sent';
receipt = 'receipt';
any = 'any';
```

```
typeof = 'typeof';
type = 'type';
category = 'category';
returnparameter = 'return';
polarity = 'polarity';
nbparameter = 'nbparameter';
```

```
messages = 'messages';
ports = 'ports';
operations = 'operations';
operation = 'operation';
parameters = 'parameters';
```

```
open = '(';
close = ')';
obracket = '[';
cbracket = ']';
assign = '=';
semi = ';';
comma = ',';
namespace_sep = '::';
point = '.';
seq = '->';
```

```
identifier = (letter_maj | letter_min) (letter_maj | letter_min | digit)*;
integer = digit+;
```

Ignored Tokens

blank;

Productions

```

joinpointmask_decl = joinpointmask identifieur obrace statement* cbrace ;

statement =
    {entity} entity_decl
  | {property} properties_decl
  | {association} association_decl;

entity_decl = {componentmask} componentmask_decl
  | {portmask} portmask_decl
  | {connexionmask} connexionmask_decl
  | {operationmask} operationmask_decl
  | {parametermask} parametermask_decl
  | {messagemask} messagemask_decl
  | {seqmessagemask} seqmessagemask_decl ;
// {recursive_entity_decl} entity_decl ;

componentmask_decl = componentmask ref other_ref* semi;
portmask_decl = portmask ref other_ref* semi;
connexionmask_decl = connexionmask ref other_ref* semi;
operationmask_decl = operationmask ref other_ref* semi;
parametermask_decl = parametermask ref other_ref* semi;
seqmessagemask_decl = messageseqmask ref other_ref* semi;
messagemask_decl = messagemask ref other_ref* semi;

properties_decl = {componenttypedeclaration} componenttype_decl
  | {messagetype} messagetype_decl
  | {messagecategory} messagecategory_decl
  | {operationmask_polarity} operationmask_polarity_decl
  | {operationmask_return} operationmask_return_decl
  | {operationmask_nbparameter} operationmask_nbparameter_decl
  | {parametermask_type} parametermask_type_decl;

componenttype_decl = ref point typeof assign componenttype semi;
    componenttype = {primitive} primitive |
    {composite} composite |
    {any} any;

messagetype_decl = ref point type assign messagetype semi;
    messagetype = {return} return |
    {request} request |
    {any} any;

messagecategory_decl = ref point category assign messagecategory semi;
    messagecategory = {receipt} receipt |
    {sent} sent |
    {any} any;

operationmask_polarity_decl = ref point polarity assign operationpolarity semi;
    operationpolarity = {provide} provide |
    {require} require;

operationmask_return_decl = ref point returnparameter assign type_decl semi;
    type_decl = identifieur;

operationmask_nbparameter_decl = ref point nbparameter assign integer semi;

parametermask_type_decl = ref point type assign type_decl semi;

association_decl =
    {seqmessage2component} seqmessagetocomponent_decl
  | {port2component} port2component_decl
  | {operation2port} operation2port_decl
  | {message2sequence} message2sequence_decl
  | {connexion} connexion_decl
  | {parameter2operation} parameter2operation_decl
  | {message2operation} message2operation_decl;

seqmessagetocomponent_decl = seqmessagetocomponent_header assign seqmessagetocomponent_body semi;
    seqmessagetocomponent_header = ref point messages;
    seqmessagetocomponent_body = obrace ref other_ref* cbrace;

port2component_decl = port2component_header assign port2component_body semi;
    port2component_header = ref point ports;
    port2component_body = obrace ref other_ref* cbrace;

operation2port_decl = operation2port_header assign operation2port_body semi;
    operation2port_header = ref point operations;
    operation2port_body = obrace ref other_ref* cbrace;

connexion_decl = ref assign connexion_body semi;
    connexion_body = obrace [left]:ref coma [right]:ref cbrace;

message2sequence_decl = ref assign message2sequence_body semi;
    message2sequence_body = obrace ref other_message* cbrace semi;
    other_message = seq ref;

parameter2operation_decl = parameter2operation_header assign parameter2operation_body semi;
    parameter2operation_header = ref point parameters;
    parameter2operation_body = obrace ref other_ref* cbrace;

```

```
message2operation_decl = message2operation_header assign message2operation_body semi;  
    message2operation_header = ref point operation;  
    message2operation_body = ref;
```

```
other_ref = coma ref;
```

```
ref= identifier;
```


Annexe I

Schéma XML du modèle de masque de point de jonction de TranSAT

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="PointCutMask">
    <xs:annotation>
      <xs:documentation>Comment describing your root
        element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ComponentMask" maxOccurs="unbounded"/>
        <xs:element ref="BindingMask" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="ComponentMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ComponentMaskModel"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="ComponentMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="SequenceMask" minOccurs="0"
            maxOccurs="unbounded"/>
          <xs:element ref="PortMask" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="typeof" type="ComponentMaskType" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="PortMask" type="PortMaskModel"/>
  <xs:complexType name="PortMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="OperationMask" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="SequenceMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="SequenceMaskModel"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="SequenceMaskModel">
    <xs:complexContent>
      <xs:extension base="TaggedElement">
        <xs:sequence>
          <xs:element ref="MessageMask" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="MessageMask">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="TaggedElement">
          <xs:attribute name="type" type="MessageType" use="required"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

        <xs:attribute name="sens" type="MessageSens" use="required"/>
        <xs:attribute name="operationMaskRef" type="RefToOperationMask"/>
    </xs:extension>
</xs:complexType>
</xs:element>
<xs:simpleType name="MessageSens">
    <xs:restriction base="xs:string">
        <xs:enumeration value="sent"/>
        <xs:enumeration value="receipt"/>
        <xs:enumeration value="any"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="MessageType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Request"/>
        <xs:enumeration value="Return"/>
        <xs:enumeration value="Any"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PolarityType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Provide"/>
        <xs:enumeration value="Require"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ComponentMaskType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="component"/>
        <xs:enumeration value="composite"/>
        <xs:enumeration value="any"/>
    </xs:restriction>
</xs:simpleType>
<xs:element name="OperationMask" type="OperationMaskModel"/>
<xs:complexType name="OperationMaskModel">
    <xs:complexContent>
        <xs:extension base="TaggedElement">
            <xs:sequence>
                <xs:element ref="ParameterMask" minOccurs="0"
                    maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="polarity" type="PolarityType" use="required"/>
            <xs:attribute name="return" type="xs:string"/>
            <xs:attribute name="nbParameter" type="xs:short"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="ParameterMask">
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="TaggedElement">
                <xs:attribute name="type" type="xs:string" use="required"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
<xs:complexType name="TaggedElement">
    <xs:attribute name="tag" type="xs:ID" use="required"/>
</xs:complexType>
<xs:element name="BindingMask" type="BindingMaskModel"/>
<xs:complexType name="BindingMaskModel">
    <xs:complexContent>
        <xs:extension base="TaggedElement">
            <xs:sequence>
                <xs:element ref="BoundPort" minOccurs="2" maxOccurs="2"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="BoundPort" type="BoundPortType"/>
<xs:complexType name="BoundPortType">
    <xs:attribute name="portRef" type="PortRef" use="required"/>
</xs:complexType>
<xs:simpleType name="PortRef">
    <xs:restriction base="xs:IDREF"/>
</xs:simpleType>
<xs:simpleType name="RefToOperationMask">
    <xs:restriction base="xs:IDREF"/>
</xs:simpleType>
</xs:schema>

```

Annexe J

Grammaire du langage de transformation de TranSAT

```
Package fr.lifl.transat.transformationrules.parser;
```

Helpers

```
letter_maj = ['A'..'Z'];
letter_min = ['a'..'z'];

digit = ['0'..'9'];
tab = 9;
cr = 13;
lf = 10;
not_cr_lf = {[32..127] - [cr + lf]};
eol = cr lf | cr | lf;

point = '.';
send = '!';
receive = '?';
return = '$';
```

Tokens

```
blank = (' ' | tab | eol)+;
new_line = cr | lf | cr lf;

port = 'Port';
operation = 'Operation';
composite = 'Composite';
connexion = 'Connexion';

in = 'in';
replaces = 'replaces';
destroy = 'destroy';
move = 'move';
point = '.';

open = '(';
close = ')';
obracket = '[';
cbracket = ']';
assign = '=';
semi = ';';
comma = ',';
becomes = '=>';
sequence = '→';
iteration = '^';
union = '|';
star = '*';

label = (send | receive) (letter_maj | letter_min) (letter_maj | letter_min | digit)*
        point (letter_maj | letter_min) (letter_maj | letter_min | digit)*
        point (letter_maj | letter_min) (letter_maj | letter_min | digit)* return? ;

identifier = (letter_maj | letter_min) (letter_maj | letter_min | digit)* ;
integer = digit+;
```

Ignored Tokens

```
blank;
```

Productions

```
transformationrules_decl = transformation_primitive_statement* ;
transformation_primitive_statement = transformation_primitive semi;
```

```

transformation_primitive = {port_creation} port_creation_decl
| {operation_creation} operation_creation_decl
| {operation_creation_replaces} operation_creation_replaces_decl
| {move} move_decl
| {connexion_creation} connexion_creation_decl
| {composite_creation} composite_creation_decl
| {composite_creation_in} composite_creation_in_decl
| {destroy} destroy_decl
| {behavior_transformation} behavior_transformation_decl;

port_creation_decl = port {portname}:ref in {componentname}:ref;
destroy_decl = ref point destroy open close;
move_decl = {elementname}:ref point move open {destination}:ref close;
operation_creation_decl = operation {operationname}:ref assign {operationref}:ref in {portname}:ref;
operation_creation_replaces_decl = operation {operationname}:ref assign {operationref}:ref replaces {replacedoperation}:ref;
connexion_creation_decl = connexion {connexionname}:ref assign obrace {port1}:ref coma {port2}:ref cbrace;
composite_creation_decl = composite ref;
composite_creation_in_decl = composite {compositename}:ref in {compositedest}:ref;

behavior_transformation_decl = [old]:primitive_behavior_description becomes [new]:primitive_behavior_description;

primitive_behavior_description = {complex} open [left]:seqexpression sequence [right]:union_behavior_description close
| {simple_seq} open [left]:seqexpression sequence [right]:identifier close
| {simple_union} union_behavior_description;

union_behavior_description = open finiteseqexpression bexpression+ close;
bexpression = union finiteseqexpression;

finiteseqexpression = [left]:seqexpression sequence [right]:identifier;

seqexpression = {recursionwithcoma} open [left]:label sequence [right]:seqexpression close ?star
| {recursion} [left]:label sequence [right]:seqexpression
| {unique} label ?star;

ref= identifier;

```

