Modélisation des applications distribuées à architecture dynamique : Conception et Validation

Mohamed Hadj Kacem

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE ET DE L'UNIVERSITÉ DE SFAX

Délivré par *l'Université Toulouse III - Paul Sabatier et la Faculté des Sciences Économiques et de Gestion - Sfax*

Discipline : Informatique

Présentée et soutenue par

Mohamed HADJ KACEM

Le Jeudi 13 Novembre 2008

Modélisation des applications distribuées à architecture dynamique : Conception et Validation

---

JURY

Président		
Me.	Brigitte PRADIN-CHEZALVIEL	Professeur à l'Université Paul Sabatier, Toulouse III
Rapporteurs		
M.	Rafik BOUAZIZ	Professeur à la FSEG-Sfax, Tunisie
M.	Flavio OQUENDO	Professeur à l'Université de Bretagne-Sud
Examinateur		
Me.	Michelle SIBILLA	Professeur à l'Université Paul Sabatier, Toulouse III
Directeurs de thèse		
M.	Khalil DRIRA	Chargé de Recherche au LAAS-CNRS
M.	Mohamed JMAIEL	Professeur à l'ENIS, Sfax, Tunisie
Invité		
M.	Ahmed HADJ KACEM	Professeur à la FSEG-Sfax, Tunisie

Laboratoire d'Architecture et d'Analyse des Systèmes
**
LAAS-CNRS

Unité de Recherche en développement et contrôle d'applications distribuées
***
ReDCAD

Résumé

Nos travaux de recherche consistent à apporter des solutions de modélisation conformément à l'approche MDA. Nos recherches consistent à fournir des solutions permettant de guider et d'assister les activités de modélisation des architectures logicielles. Il s'agit principalement de proposer une démarche de conception orientée modèle permettant de décrire l'architecture logicielle en tenant compte de trois aspects : le style architectural, les opérations de reconfiguration et le protocole de reconfiguration.
Nous proposons des notations visuelles permettant de décrire de façon compatible avec UML 2.0 l'architecture logicielle. La technique de description que nous adoptons est orientée règles, basée sur les théories de transformation de graphe, permettant, ainsi, de décrire la dynamique structurelle.
Nous proposons une extension d'UML 2.0 par un nouveau profil formé de trois méta-modèles. Nous proposons aussi une approche de validation basée sur des règles intra-modèle et des règles inter-modèles. Nous adoptons également une approche de vérification. Cette approche permet, dans une première étape, une transformation automatique du style architectural et de chaque opération de reconfiguration vers le langage *Z*. Elle permet dans une deuxième étape, de vérifier la consistance du style architectural et la conformité de l'évolution d'une architecture vis-à-vis de son style architectural. Nous utilisons le système de preuve Z/EVES. Finalement, nous proposons une démarche de modélisation des architectures logicielles dynamiques, appelée X, permettant de décrire les différentes étapes pour modéliser l'architecture logicielle. La démarche proposée est inspirée de la méthode MDA et 2TUP.
Le profil et la démarche X que nous avons proposés ont fait l'objet d'une implémentation et d'une intégration, sous forme de plug-in java, dans l'atelier d'aide à la conception FUJABA. Le plug-in implémenté est disponible sur l'URL : http ://www.laas.fr/∼khalil/TOOLS/X.zip.

Mots-clés : Adaptabilité architecturale, Architecture Dynamique, Profil UML, Validation, Vérification formelle, Démarche de modélisation

Abstract

Title : Modeling of distributed applications in dynamic architecture : Design and Validation

Abstract : The Adaptability of networked service-oriented systems is necessary to guarantee the continuity of services under changing infrastructure constraints and evolving user requirements. The architectural and the behavioural dynamic reconfigurations constitute the two acting categories allowing adaptability of such software systems to be managed.

We propose to support the architectural reconfiguration-based adaptation and we propose a UML-based approach for describing, validating and checking dynamic software architectures. We elaborate a new UML profile associated with three meta-models that we define in order to describe (i) different architectural styles, (ii) their dynamic instances together with the associated reconfiguration operations, and (iii) their reconfiguration protocol. Our approach supports describing the architectural constraints that should be preserved during architecture evolving. The proposed approach supports automatic validation through a set of rules we define and implement for ensuring the correctness of the models w.r.t. the meta-models. These models are automatically transformed into *Z* specifications. This allows designers to prove, by elaborating *Z* theorems, the consistency of the defined architectural styles and the conformity of the associated instances and their transformation rules. We define a *Design Process*, called X, allowing guiding and assisting the architects to model the dynamic software architectures. The X Design Process is partially based on the MDA approach and 2TUP Process. Our design process is based on PIM and PSM parts. The proposed design process uses an iterative and incremental development approach and it is architecture-centric.

A software environment supporting the different features of this approach has been developed and integrated as a plug-in in the open-source FUJABA tool. The plug-in is available at URL : http ://www.laas.fr/∼khalil/TOOLS/X.zip.

Keywords : Architectural adaptation, Dynamic architecture, UML Profile, Validation, Formal verification, Design process

A mes parents pour leur patience et leur amour.
A mon épouse Imen.
A toute ma famille.
A vous tous !

Remerciements

Mes remerciements vont avant tout aux membres de jury qui ont accepté d'évaluer ce travail de thèse.

Je remercie vivement et particulièrement : M. Rafik BOUAZIZ, professeur à la FSEG Sfax-Tunisie et M. Flavio OQUENDO, professeur à l'Université de Bretagne Sud-France, qui ont accepté de juger ce travail et d'en être les rapporteurs.

Je remercie également Me. Brigitte PRADIN-CHEZALVIEL, professeur à l'Université Paul Sabatier de Toulouse-France et Me. Michelle SIBILLA, professeur à l'Université Paul Sabatier de Toulouse-France pour l'honneur qu'elles m'ont fait de participer à ce jury de thèse en tant qu'examinateurs. Je remercie particulièrement Me. Brigitte PRADIN-CHEZALVIEL pour l'honneur qu'elle m'a fait en présidant mon jury de thèse.

Je tiens à exprimer ma profonde reconnaissance à M. Khalil DRIRA, directeur de ma thèse coté Toulouse-France, chargé de recherche au LAAS-CNRS pour ses qualités tant scientifiques qu'humaines. La rigueur scientifique, la très grande compétence, ses encouragements et sa gentillesse, ont été déterminants pour les résultats obtenus.

De même, je remercie M. Mohamed JMAIEL, directeur de ma thèse coté Sfax-Tunisie, professeur à l'École Nationale d'Ingénieurs de Sfax et directeur de l'unité de recherche en Développement et Contrôle des Applications Distribuées (ReDCAD) pour la confiance qu'il m'a témoignée tout au long de ce travail. Ce travail a grandement profité de son encadrement, de ses précieux conseils et de ses qualités humaines.

Je suis très reconnaissant à M. Ahmed HADJ KACEM, professeur à la Faculté des Sciences Économiques et de Gestion de Sfax-Tunisie pour sa contribution dans le co-encadrement de ce travail. Ses conseils d'ordre méthodologique, technique et rédactionnel ont grandement contribué à la qualité de ce mémoire. Je le remercie également pour l'honneur qu'il m'a fait en participant au jury.

Je tiens à remercier les étudiants de mastère qui, au cours de leurs projets, se sont intéressés à ma problématique de thèse et m'ont apporté du sang neuf et des idées nouvelles. Je cite notamment Achraf BOUKHRIS et Mohamed Nadhmi MILADI.

Mes remerciements vont également vers les membres de l'unité de recherche LARIS, Sfax-Tunisie, dont j'ai fait partie ainsi que les membres de l'unité de recherche ReDCAD,

Sfax-Tunisie, dont je suis actuellement membre.

Je remercie également les membres du groupe OLC du LAAS-CNRS, Toulouse-France, dont je suis également membre, pour l'accueil chaleureux, pour les discussions scientifiques et l'amitié cultivée au sein de ce groupe.

Je remercie, mes collègues à l'Institut Supérieur d'Informatique et de Mathématiques ISIM Monastir-Tunisie, mes collèges à l'Institut Supérieur d'Informatique et de Multimedia ISIM Sfax-Tunisie et mes collègues à la Faculté des Sciences Économiques et de Gestion FSEG Sfax-Tunisie pour les encouragements et l'aide qui m'ont permis de finir cette thèse et surtout ceux qui m'ont fait profiter de leur grande expérience et avec qui j'ai eu de nombreuses discussions enrichissantes.

Mes plus vifs remerciements s'adressent, plus particulièrement, à Walid HADJ KACEM, Riadh BEN HALIMA, Slim KALLEL et Imen LOULOU pour les discussions enrichissantes que nous avons eues et les informations que nous avons échangées.

Un immense Merci à mes parents pour leur soutien sans faille, à toute ma famille, à tous mes amis proches ou éloignés, pour leurs encouragements et leurs marques d'affection. Bien que je ne les liste pas ici, je suis certain que chacun s'y reconnaîtra.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Les applications émergentes récentes sont de plus en plus distribuées, ouvertes et à architecture complexe. De telles applications sont composées d'un nombre, généralement important, d'entités logicielles dispersées sur le réseau coopérant pour fournir à leurs utilisateurs les services qu'ils requièrent. Cette complexité est liée, par exemple, à l'extension des réseaux de communication, à l'hétérogénéité matérielle et logicielle, au fort degré d'interaction et aux exigences d'évolutions nécessaires et permanentes. Le but étant d'offrir des services de plus en plus performants, robustes et sûrs, et qui s'adaptent aux différents besoins des clients. L'adaptation ou la reconfiguration logicielle constitue l'un des enjeux et défis techniques les plus importants. En effet, l'adaptabilité d'une application est devenue impérative dans un contexte de systèmes fortement distribués et contenant un grand nombre de composants. Dans ce cas, une reconfiguration ad-hoc n'est pas envisageable.

L'adaptabilité dans les applications logicielles peut être classée en deux catégories. La première concerne l'adaptation comportementale. Elle traite la modification du comportement de l'application et de ses composants et impliquerait, par exemple, la redéfinition d'une méthode dans l'interface d'un composant. La deuxième catégorie, dans laquelle nous pouvons classer nos travaux, concerne l'adaptation structurelle et implique une reconfiguration au niveau architectural. Ce type de reconfiguration porte sur l'organisation de l'architecture et consiste, par exemple, à remplacer un composant défaillant par un autre composant qui possède les mêmes fonctionnalités ou rediriger un lien (une connexion) d'un composant défaillant vers un composant susceptible d'offrir de meilleures fonctionnalités.

Afin de faire face à cette complexité, l'OMG (Object Management Group) a proposé l'approche dirigée par les modèles (MDA : Model Driven Architecture). Dans cette approche, les modèles constituent la couche nécessaire pour fournir le niveau d'abstraction demandé. L'approche MDA préconise l'utilisation d'UML (Unified Modeling Language) pour l'élaboration des modèles. En effet, UML, et grâce à sa version 2.0, est devenu un standard pour la conception et la modélisation des architectures des applications distribuées à base de composants.

Cependant, même avec cette intention d'être général, UML ne peut pas couvrir tous les contextes et offre ainsi un mécanisme d'extensibilité basé sur les profils. Un profil UML permet la personnalisation et l'ajustement d'UML pour prendre en charge des domaines spécifiques qui ne peuvent pas être représentés avec UML dans son état original.

Nos travaux de recherche entrent dans ce contexte et traitent les applications distribuées à base de composants. Dans nos travaux de thèse, nous nous concentrons sur la modélisation de la dynamique de l'architecture logicielle au niveau conceptuel (design time). Nous traitons essentiellement le problème de l'évolution des exigences représentant un changement dans l'activité soutenue par le système modélisé. Nous associons des opérations de reconfiguration architecturales à ces situations pour adapter le système à ces changements. Nos travaux consistent à apporter des solutions de modélisation conformes à l'approche MDA. Ainsi, nous proposons des solutions de modélisation permettant de guider et d'assister les activités de conception des applications distribuées à architecture dynamique. Nous proposons une démarche de conception orientée-modèle permettant de modéliser, de valider, de vérifier et d'implémenter une architecture logicielle dynamique.

Le profil UML, que nous définissons dans ce manuscrit, couvre plusieurs maillons de la chaîne de description architecturale. Il offre trois méta-modèles permettant de décrire le style architectural, les opérations de reconfiguration et le protocole de reconfiguration. Le premier méta-modèle décrit le style architectural. Il décrit les types de composants constituants l'architecture, les connexions qui les associent ainsi que les contraintes et les propriétés architecturales. Le deuxième est basé sur des techniques orientées-règles pour la reconfiguration architecturale dynamique. Ainsi chaque étape élémentaire de l'évolution de l'architecture est décrite par une combinaison d'opérations de reconfiguration permettant sa transformation d'un état vers un autre. Chacune de ces opérations spécifie les actions de transformation élémentaires relatives, par exemple, aux composants logiciels qu'il faut introduire ou supprimer, ou aux connexions (entre composants) à introduire ou à supprimer. Le troisième décrit le protocole de reconfiguration. Il spécifie l'organisation et l'enchaînement entre les différentes opérations de reconfiguration décrites au niveau dynamique.

La modélisation de l'architecture logicielle génère des modèles en se basant sur le profil UML proposé. Ces modèles doivent être valides par rapport à leur méta-modèle. L'approche de validation que nous proposons est basée sur des règles intra-modèle et des règles inter-modèles. Les règles intra-modèle permettent de contrôler la cohérence d'un modèle par rapport à son méta-modèle et les règles inter-modèles permettent de contrôler le passage d'un modèle à un autre. Cette validation assure la validité des modèles et leur conformité par rapport à leur méta-modèle.

Afin d'attribuer une sémantique formelle, nous proposons également une approche de vérification. Cette approche offre, dans une première étape, une transformation automatique du style architectural et de chaque opération de reconfiguration vers le langage *Z*. Elle adopte, dans une deuxième étape, un processus de vérification. Ce processus développé au sein de notre unité de recherche ReDCAD, permet de vérifier la consistance du style architectural et la conformité de l'évolution d'une architecture vis-à-vis de son style architectural. Nous utilisons le système de preuve Z/EVES. Finalement, nous proposons une démarche intitulée X, inspirée de l'approche MDA et 2TUP (2 Tracks Unified Process) du processus UP (Unified Process), permettant de guider et d'assister, étape par étape, les architectes afin de modéliser l'architecture d'une application dynamique tout en suivant le profil proposé.

Dans des travaux antérieurs, nous avons prouvé notre approche par différents cas d'étude. Nous l'avons appliquée à des applications coopératives telles que l'édition coopérative et la revue coopérative. Nous l'avons aussi appliquée à un cas d'étude relatif aux opérations d'intervention d'urgence. Dans ce manuscrit, nous traitons de manière détaillée un cas d'étude relatif à la gestion des patients dans une clinique (Patient Monitoring System).

Le profil que nous avons proposé, l'approche de validation et de vérification ainsi que la démarche X proposées ont fait l'objet d'une implémentation et d'une intégration, sous forme de plug-in java, dans l'atelier d'aide à la conception FUJABA. Le plug-in implémenté est disponible sur l'URL : www.laas.fr/∼khalil/TOOLS/X.zip

Ce rapport est organisé comme suit :

Le chapitre 1 présente une étude sur les architectures logicielles. Il introduit l'approche MDA et ses différentes recommandations. Une section est réservée à l'introduction et à l'étude de quelques styles architecturaux. Une dernière section est consacrée à l'étude des architectures dynamiques et ses différents types.

Le chapitre 2 présente une étude de synthèse sur les techniques de description des architectures logicielles. Une étude sur les langages de description des architectures (ADLs : Architecture Description Language) est présentée. Une section particulière est consacrée à UML en tant que langage de description des architectures. Les techniques formelles sont aussi étudiés pour voir leur possibilité de décrire l'architecture logicielle et plus particulièrement la dynamique architecturale. Une dernière section est consacrée à la présentation des méthodes agiles.

Le chapitre 3 propose un profil UML pour décrire la dynamique des architectures logicielles. Le profil est formé par trois méta-modèles. Le premier étend le diagramme de composants et permet de décrire les éléments de base de l'architecture et les connexions qui les associent (méta-modèle style architectural). Le deuxième étend le premier et permet de décrire les opérations de reconfiguration de l'architecture en termes d'ajout et/ou de suppression de composants et/ou de connexions (méta-modèle opération de reconfiguration). Les opérations de reconfiguration permettant de décrire la dynamique de l'architecture sont définies en se basant sur la technique de réécriture de graphes. Le troisième étend le diagramme d'activité et permet de décrire le protocole de reconfiguration entre les différentes opérations de reconfiguration (méta-modèle protocole de reconfiguration).

Le chapitre 4 propose une approche de validation. Cette validation offre deux scénarios. Le premier, appelé intra-validation, est basé sur des règles de validation intra-modèle permettant de vérifier la cohérence et la conformité entre le modèle et son méta-modèle. Ces règles sont utilisées pour faciliter l'identification des éventuelles incohérences et pour détecter et corriger des erreurs de spécification. Le second, appelé inter-validation, est basé sur des règles de validation inter-modèles permettant d'assister et de guider le passage d'un modèle à un autre. Ces règles permettent de réduire les erreurs et d'assurer la cohérence entre les modèles. La validation que nous proposons est basée sur la technologie XML (Extensible Markup Language).

Le chapitre 5 est composé de deux parties. Nous proposons, dans la première, une

approche assurant une transformation automatique du style architectural et de chaque opération de reconfiguration vers le langage *Z*. Dans la deuxième partie, nous adoptons une approche de vérification. Cette approche est développée, dans notre unité de recherche ReDCAD, dans le cadre des travaux de la thèse de Me. Imen LOULOU [84]. Nous vérifions la consistance du style architectural et la conformité de l'évolution d'une architecture vis-à-vis de son style architectural. Nous utilisons le système de preuve Z/EVES.

Le chapitre 6 décrit une démarche X, inspirée de l'approche MDA et 2TUP du processus UP. La démarche que nous proposons guide et assiste, étape par étape, les architectes afin de modéliser l'architecture d'une application dynamique tout en suivant le profil proposé. La démarche que nous proposons est itérative, centrée sur l'architecture et incrémentale.

Ce manuscrit se termine par un ensemble de conclusions générales et de perspectives des travaux en cours et à venir.

1

Architectures logicielles

Dans le monde des sciences exactes, les définitions sont dotées d'une précision irréprochable. Par contre, dans le monde du génie logiciel d'une manière générale, les définitions et les concepts ne sont pas aussi précis. S. THOMASON [25] a affirmé : "*vous pouvez trouver autant de définitions d'un concept qu'il y a de lecteurs d'un article*".

L'architecture logicielle est une discipline récente du génie logiciel focalisant sur la structure, le comportement et les propriétés globales d'un système et s'adresse plus particulièrement à la conception de systèmes de grande taille. L'objectif de la formalisation des architectures est de promouvoir la réutilisation et la compréhension. Pour aller plus loin et capturer l'expertise de conception pour un domaine particulier, la notion de style architectural est mise en avant. Un style cadre la conception architecturale à travers un ensemble de contraintes et de propriétés. Ainsi, il est possible d'être averti lorsque l'architecture ne respecte pas les propriétés exigées.

Dans ce premier chapitre, nous dressons une étude sur l'approche MDA et ses différentes recommandations. Nous présentons les différents concepts relatifs au domaine de l'architecture logicielle et nous étudions quelques styles architecturaux. Nous présentons également une étude sur les architectures dynamiques et ses différents types.

1.1 MDA : Model Driven Architecture

En novembre 2000, l'OMG (Object Management Group), consortium de plus de 1000 entreprises, initie l'approche MDA (Model Driven Architecture) [107]. Cette approche a pour but d'apporter une nouvelle vision unifiée de concevoir des applications en séparant

la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique. Cette séparation autorise alors la capitalisation du savoir logiciel et du savoir-faire de l'entreprise. A ce niveau, l'approche objet et l'approche composant n'ont pas pu tenir leurs promesses. Il devenait de plus en plus difficile de développer des logiciels basés sur ces technologies. Le recours à des procédés supplémentaires, comme le patron de conception ou la programmation orientée aspect, était alors nécessaire. L'approche MDA entre dans ce contexte et offre la possibilité d'arrêter l'empilement des technologies qui nécessite une conservation des compétences particulières pour faire cohabiter des systèmes divers et variés. Ceci est permis grâce au passage d'une approche interprétative à une approche transformationnelle, où l'individu a un rôle simplifié et amoindri grâce à la construction automatisée.

MDA n'est pas une solution qui va résoudre tous les problèmes, mais elle paraît la seule aujourd'hui capable de fournir des réponses satisfaisantes à toutes ces nouvelles exigences [82]. Dans l'approche MDA, tout est considéré comme modèle, aussi bien les schémas que le code source ou binaire. Le modèle indépendant de plateforme (PIM) et le modèle dépendant de plateforme (PSM) constituent les principaux types de modèles dans le cadre de MDA. Les PIM sont des modèles qui n'ont pas de dépendance avec les plateformes technologiques (ils sont donc indépendants de CORBA, EJB, Services Web, .NET, etc.). Les PSM sont des modèles dépendants des plateformes technologiques et servent de base à la génération de code exécutable.

De plus, nous pouvons mentionner qu'une approche dirigée par les modèles oblige les architectes et les développeurs à focaliser leurs travaux sur l'architecture et le modèle du système. MDA fournit une approche de l'utilisation de modèles pour diriger la compréhension, la conception, la construction, le déploiement, l'opération, la maintenance et la modification de systèmes [107, 20].

La proposition de l'OMG recommande l'utilisation d'UML (Unified Modeling Language), MOF (Meta Object Facility) et XMI (XML Metadata Interchange). Le MOF est le méta-méta-modèle standard unique. XMI est le format qui va permettre l'échange de modèles et de méta-modèles. Il est basé sur XML (Extensible Markup Language). Enfin, UML est l'un des premiers méta-modèles basé sur le MOF adopté par l'OMG [29].

1.1.1 Architecture à quatre niveaux

La modélisation logicielle a radicalement évolué au cours de ces dernières années. Ces changements majeurs sont principalement supportés par l'OMG. Il y a aujourd'hui un consensus autour d'une architecture à quatre niveaux adopté principalement par l'OMG et l'UML [20].

Comme le montre la figure 1.1, quatre niveaux caractérisent ce standard de méta-

modélisation. Le *niveau* M_0 qui est le niveau des données réelles, composé des informations que l'on souhaite modéliser. Ce niveau est souvent considéré comme étant le monde réel. Lorsqu'on veut décrire les informations contenues dans le niveau M_0, cette activité donne naissance à un modèle appartenant au *niveau* M_1. Un modèle UML appartient au niveau M_1. Le *niveau* M_2 est composé des langages de définition des modèles d'information, appelés aussi méta-modèles. Un méta-modèle définit la structure d'un modèle. Le *niveau* M_3 est le niveau le plus abstrait parmi les quatre niveaux dans cette architecture. Il définit la structure de tous les méta-modèles du niveau M_2 ainsi que lui-même. Il est composé d'une unique entité qui s'appelle le MOF.

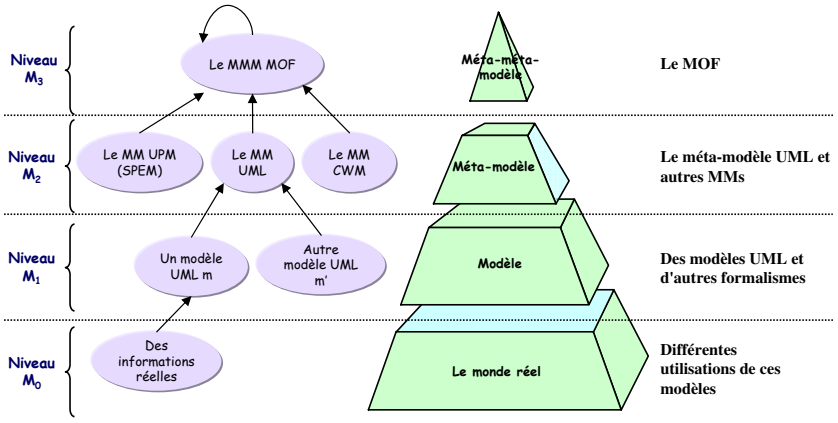

FIG. 1.1 – Architecture à quatre niveaux

Dans l'architecture à quatre niveaux, la validation d'un modèle se fait avec le modèle du niveau suivant. Ce concept de méta-modèle nous semble fondamental. Il aide non seulement à réduire les ambiguïtés et les incohérences de la notation, mais il constitue aussi un précieux atout pour les concepteurs, dans le cadre d'une automatisation du processus de développement logiciel. L'inconvénient majeur de MOF par rapport à nos objectifs c'est qu'il n'offre pas des techniques de validation et des règles de passage d'un niveau à un autre.

Dans notre approche, on s'intéresse uniquement aux trois premiers niveaux. M_0 pour décrire l'existant. M_1 pour spécifier l'existant avec UML et M_2 pour spécifier le méta-modèle de M_1.

1.1.2 MOF : Meta Object Facility

MOF pour (Meta-Object Facility) [106] est un ensemble d'interfaces permettant de définir la syntaxe et la sémantique d'un langage de modélisation. Il a été créé par l'OMG afin de définir des méta-modèles et leurs modèles correspondants [10].

Il fait partie des standards définis par l'OMG et il peut être vu comme un sous-ensemble d'UML. Le MOF et l'UML constituent le cœur de l'approche MDA car ces deux standards

permettent de créer des modèles technologiquement neutres. Le MOF spécifie la structure et la syntaxe de tous les méta-modèles comme UML, CWM (Common Warehouse Meta-model) et SPEM (Software Process Engineering Meta-model). Ils ont le même formalisme. Le MOF spécifie aussi des mécanismes d'interopérabilité entre ces méta-modèles. Il est donc possible de les comparer et de les relier. Grâce à ces mécanismes d'échanges, le MOF peut faire cohabiter plusieurs méta-modèles différents [10].

MOF 2.0 a été développé en prenant en compte la nouvelle infrastructure d'UML 2.0, permettant ainsi à UML 2.0 d'être utilisé afin de représenter des méta-modèles de MOF 2.0. MOF 2.0 introduit une importante nouveauté, c'est la norme de QVT (Query View Transformation) qui est un langage de requêtes et de transformation de modèles. Le QVT est le résultat d'un effort de normalisation des mécanismes, des techniques et des notations nécessaires à la transformation de modèles sources en modèles cibles. L'objectif du QVT est de définir et d'établir une manière standard pour interroger les modèles du MOF, pour créer des vues et pour définir les transformations de modèles [98].

1.1.3 UML : Unified Modeling Language

Depuis que le groupe OMG a adopté le langage de modélisation unifié en 1997, UML (Unified Modeling Language) s'est rapidement établi comme étant le standard pour la modélisation des systèmes logiciels [48]. UML a été conçu pour unifier différentes notations graphiques de modélisation, être le plus général possible et être extensible afin de prendre en charge des contextes non prévus. La première version d'UML a été publiée en 1997 par l'OMG. Depuis, UML est devenu la référence pour la conception de logiciels. La spécification d'UML définit une notation graphique pour visualiser, spécifier et construire des applications orientées objets [109].

La version 2.0 [126, 116, 19, 48] vise à faire entrer ce langage dans une nouvelle ère de sorte que les modèles UML soient au centre de MDA. Cela revient à rendre les modèles UML stables et productifs et à leur permettre de prendre en compte les plateformes d'exécution. UML 2.0 a introduit de nouvelles constructions qui le rendent plus adapté au développement à base de composants et à la spécification des descriptions architecturales.

La notation UML est décrite sous forme d'un ensemble de diagrammes. La première génération d'UML (UML 1.x), définit neuf diagrammes pour la spécification des applications. Dans UML 2.0, quatre nouveaux diagrammes ont été ajoutés : il s'agit des diagrammes de structure composite (Composite structure diagrams), les diagrammes de paquetages (Packages diagrams), les diagrammes de vue d'ensemble d'interaction (Interaction overview diagrams) et les diagrammes de synchronisation (Timing diagrams) [101, 48]. Les diagrammes UML sont regroupés dans deux classes principales [144] :

– **Les diagrammes statiques** : regroupent les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, et les diagrammes de paquetages.

– **Les diagrammes dynamiques** : regroupent les diagrammes de séquence, les diagrammes de communication (nouvelle appellation des diagrammes de collaboration d'UML 1.x), les diagrammes d'activités, les machines à états, les diagrammes de vue d'ensemble d'interaction, et les diagrammes de synchronisation.

UML est conçu pour prendre en charge une grande variété de contextes. Cependant, même avec cette intention d'être général, UML ne peut pas couvrir tous les contextes et offre ainsi un mécanisme d'extensibilité basé sur les profils. Un profil permet la personnalisation d'UML pour prendre en charge des domaines spécifiques qui ne peuvent pas être représentés avec UML dans son état original. L'approche MDA préconise fortement l'utilisation d'UML pour l'élaboration de PIMs et de la plupart des PSMs. Les spécificités de chaque plateforme peuvent être modélisées avec un profil.

En fait, nous pouvons définir pour chaque système un profil qui regroupe les éléments nécessaires à ses caractéristiques [107].

1.1.4 Profil UML

Un profil UML est une adaptation du langage UML à un domaine particulier [20, 82]. Il est constitué de stéréotypes, de contraintes et de valeurs marquées [102]. Un stéréotype définit une sous-classe d'un ou de plusieurs éléments du méta-modèle UML. Cette sous classe a la même structure que ses éléments de base, mais le stéréotype spécifie des contraintes et des propriétés supplémentaires. Les contraintes peuvent être spécifiées non formellement, mais l'utilisation d'OCL (Object Constraint Language) est préférable pour créer les contraintes de façon formelle et standardisée.

MDA propose l'utilisation de profils UML. L'OMG a standardisé certains profils. Nous citons dans ce qui suit quelques uns :

– Le profil EDOC (Enterprise Distributed Object Computing) vise à faciliter le développement de modèles d'entreprises, de systèmes ou d'organisations en utilisant une approche de collaboration récursive utilisable à différents niveaux de granularité et pour différents degrés de couplages. La gestion de l'ingénierie des processus métier est supportée par l'assemblage de services en visant la stabilité de l'architecture globale dans le cas de modifications locales [67].

– Le profil EAI (Enterprise Application Integration) simplifie l'intégration d'applications en normalisant les échanges et la traduction des méta-données.

– Le profil CORBA (Common Object Request Broker Architecture) permet de représenter à l'aide d'UML le modèle de composants CORBA.

– Le profil modélisation temps réel (Schedulability Performance and Time) pour

modéliser des applications en temps réels.

– Le profil Test permet la spécification de tests pour les aspects structurels (statiques) ainsi que pour les aspects comportementaux (dynamiques) des modèles UML.

– Le profil QoS (Quality of Service) représente la qualité de service et de tolérance aux erreurs.

– Le profil SPEM (Software Process Engineering Metamodel) est défini à la fois comme un profil UML et comme un méta-modèle MOF. SPEM définit les façons d'utiliser UML dans les projets logiciels et permet la création de modèles de processus (pour les PIM exclusivement).

1.1.5 OCL : Object Constraint Language

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes souhaitées. Fort de ce constat, l'OMG a défini formellement le langage textuel de contraintes OCL, qui permet de définir n'importe quelle contrainte sur des modèles UML [20] : le langage OCL (Object Constraint Language) qui est un langage d'expression permettant de décrire des contraintes sur des modèles. Une contrainte est une restriction sur une ou plusieurs valeurs d'un modèle non représentable en UML.

Une contrainte OCL est une expression dont l'évaluation doit retourner vrai ou faux. L'évaluation d'une contrainte permet de savoir si la contrainte est respectée ou non. OCL donne des descriptions précises et non ambiguës du comportement du logiciel en complétant les diagrammes. Il définit des pré-conditions, des post-conditions et des invariants pour une opération. Il permet aussi la définition des expressions de navigation et des expressions booléennes. C'est un langage de haut niveau d'abstraction, parfaitement intégré à UML, qui permet de trouver des erreurs beaucoup plus tôt dans le cycle de vie de l'application. Cette vérification d'erreurs est rendue possible grâce à la simulation du comportement du modèle [108, 12, 110, 6] en utilisant des outils tels que USE (UML-based Specification Environment) [55] et Octopus (OCL Tool for Precise UML Specifications) [44].

1.1.6 XMI : Xml Metadata Interchange

XMI est le langage d'échange entre le monde des modèles et le monde XML (eXtensible Markup Language). C'est le format d'échange standard entre les outils compatibles MDA. XMI décrit comment utiliser les balises XML pour représenter un modèle UML en XML. Cette représentation facilite les échanges de données entre les différents outils ou plateformes de modélisation. En effet, XMI définit des règles permettant de construire des DTD (Document Type Definition) et des schémas XML à partir de méta-modèles, et

inversement. Ainsi, il est possible d'encoder un méta-modèle dans un document XML, mais aussi, à partir d'un document XML il est possible de reconstruire des méta-modèles. XMI a l'avantage de regrouper les méta-données et les instances dans un même document ce qui permet à une application de comprendre les instances grâce à leurs méta-données. Ceci facilite les échanges entre applications et certains outils pourront automatiser ces échanges en utilisant un moteur de transformation du type XSLT (Extensible Stylesheet Language Transformation) [20].

1.2 Architecture logicielle

L'architecture logicielle, comme le montre la figure 1.2, se définit comme une spécification abstraite d'un système en termes de composants logiciels ou modules qui le constituent, des interactions entre ces composants (connecteurs) et d'un ensemble de règles qui gouvernent cette interaction [21, 38, 50, 64, 87]. Les composants encapsulent typiquement l'information ou la fonctionnalité. Tandis que les connecteurs assurent la communication entre les composants. Cette architecture possède, généralement, un ensemble de propriétés d'ordre topologique ou fonctionnelle qu'elle doit respecter tout au long de son évolution.

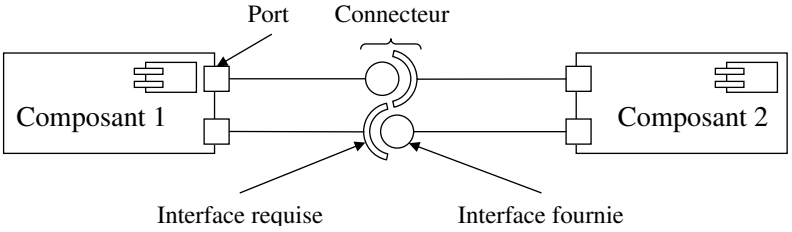

FIG. 1.2 – Les éléments d'une architecture logicielle

1.2.1 Composant

La notion de composant logiciel est introduite 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 [132].

La notion de composant a pris de nombreuses formes dans les différentes approches de l'architecture logicielle [135]. La définition suivante, globalement acceptée, illustre bien ces propriétés : "*Un composant est une unité de calcul ou de stockage. Il peut être primitif ou composé. On parle dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application complète. Deux parties définissent un composant. Une*

première partie, dite externe, comprend la description des interfaces fournies et requises par le composant. Elle définit les interactions du composant avec son environnement. La seconde partie correspond à son contenu et permet la description du fonctionnement interne du composant" [105].

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 et instance de composant.

– Type de composant : un type de composant est la définition abstraite d'une entité logicielle. Nous utilisons le type de composant pour la description du style architectural.
– Instance de composant : une instance de composant est au même titre qu'une instance d'objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implantation de ce type. Nous utilisons l'instance de composant pour la description des opérations de reconfiguration.

1.2.2 Connecteur

Le connecteur (appelé connexion selon notre terminologie) correspond à un élément d'architecture qui modélise les interactions entre deux ou plusieurs composants en définissant les règles qui gouvernent ces interactions [30, 105].

1.2.3 Port

Un composant interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Un port représente un point d'accès à certains services du composant [30]. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports.

1.2.4 Interface

C'est le point de communication qui permet d'interagir avec l'environnement. On la retrouve donc associée aux composants et aux connecteurs. Elle spécifie des ports dans le cas des composants et des rôles dans le cas des connexions. [105].

Pour un composant, il existe deux types d'interfaces :
– Les interfaces fournies décrivent les services proposés par le composant (figure 1.3).

– Les interfaces requises décrivent les services que les autres composants doivent fournir pour le bon fonctionnement du composant dans un environnement particulier (figure 1.3).

Ces interfaces sont éventuellement exprimées par l'intermédiaire des ports (figure 1.3).

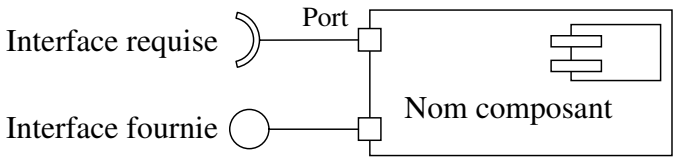

FIG. 1.3 – Modélisation graphique d'un composant

1.2.5 Configuration

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 connexions. On parle aussi de topologie.

Une configuration représente en fait une instance possible d'un style architectural. Plus précisément, une configuration décrit les instances de composants intervenants et les relations qu'elles entretiennent entre elles [88].

La conception d'une architecture logicielle, a une importance capitale pour la réussite d'un projet informatique. Elle est souvent liée au savoir-faire de l'architecte. Une architecture logicielle doit tenir compte des contraintes suivantes :

– La **réutilisabilité** : est la capacité à rendre générique des composants et à concevoir et construire des boites noires susceptibles de fonctionner avec des langages et des environnements variés.

– La **maintenabilité** : est la capacité de modifier et d'adapter une application afin de la maintenir sur une période de vie assez longue. Une architecture bien spécifiée doit être maintenue tout au long de son cycle de vie. La prévision de l'intégration des extensions à l'architecture et la correction des erreurs sont nécessaires dès la phase de conception.

– La **performance** : c'est l'optimisation du temps mis par une application pour répondre à une requête donnée. La performance d'une application dépend de l'architecture logicielle choisie, de son environnement d'exécution, de son implémentation et de la puissance des infra-structures utilisées (e.g. débit du réseau utilisé).

1.3 Style architectural

Dans n'importe quelle activité relative au domaine du génie logiciel, une question qui revient souvent est : comment bénéficier des expériences antérieures pour produire des systèmes plus performants ? Dans le domaine des architectures logicielles, une des manières, selon [104], est de classer les architectures par catégories et de définir leurs

caractéristiques communes. En effet, un style architectural définit une famille d'architectures logicielles qui sont caractérisées par des propriétés structurelles et sémantiques communes [99].

Les styles d'architecture [54, 47, 79] sont des schémas d'architectures logicielles qui sont caractérisés par :

– Un ensemble de types de composants.

– Un ensemble de connecteurs.

– Une répartition topologique de ces composants indiquant leurs relations.

– Un ensemble de contraintes sémantiques.

Un style n'est pas une architecture mais une aide générique à l'élaboration de structures architecturales [39]. Un style architectural inclut une spécification statique et une spécification dynamique. La partie statique englobe l'ensemble des éléments (composants et connecteurs) et des contraintes sur ces éléments. La partie dynamique décrit l'évolution possible d'une architecture en réaction à des changements prévus ou imprévus de l'environnement. Un style architectural est précisément défini par un ensemble de caractéristiques telles que : le vocabulaire utilisé (les types de composants et de connexions), les contraintes de configuration (contraintes topologiques appelées aussi patrons structurels), les invariants du style, les exemples communs d'utilisation, les avantages et inconvénients d'utilisation de ce style et les spécialisations communes du style. Un style architectural spécifie aussi les types d'analyse que l'on peut faire sur ses instances (systèmes construits conformément à ce style) [51].

Parmi les principaux styles architecturaux, nous citons le style "client-serveur", le style "publier-souscrire" et le style "pipe and filtre".

1.3.1 Style "client-serveur"

Il est sans doute le style le plus connu de tous [18, 137]. Comme le montre la figure 1.4, il se base sur deux types de composants : un composant de type serveur, offrant un ensemble de services, écoute des demandes sur ses services. Un composant de type client, désirant qu'un service soit assuré, envoie une demande (requête) au serveur par l'intermédiaire d'un connecteur. Le serveur rejette ou exécute la demande et envoie une réponse de nouveau au client. La contrainte qui s'impose dans ce type est qu'un composant ne peut être qu'un fournisseur de services ou un demandeur de services. Le style client-serveur consiste alors à structurer un système en terme d'entités serveurs et d'entités clientes qui communiquent par l'intermédiaire d'un protocole de communication à travers un réseau informatique.

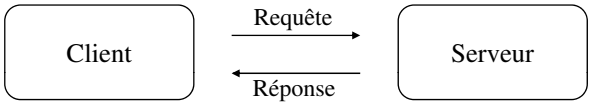

FIG. 1.4 – Le style “client-serveur”

1.3.2 Style “publier-souscrire”

Ce style, comme le montre la figure 1.5, se base sur trois composants. Un composant *producteur* qui va produire des informations. Un composant *consommateur* qui va les consommer et un composant *service d’événement* qui va assurer l’échange d’informations entre les producteurs et les consommateurs. Ces derniers ne communiquent donc pas directement et ne gardent même pas les références des uns et des autres. De plus, les producteurs et les consommateurs n’ont pas besoin de participer activement à l’interaction selon un mode synchrone. Le producteur peut publier des événements pendant que le consommateur est déconnecté, et réciproquement, le consommateur peut être notifié à propos d’un événement pendant que le producteur, source de cet événement, est déconnecté.

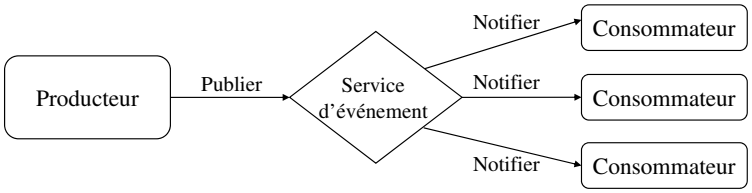

FIG. 1.5 – Le style “publier-souscrire”

1.3.3 Style “pipes and filters”

Dans ce style, comme le montre la figure 1.6, un composant reçoit un ensemble de données en entrée et produit un ensemble de données en sortie. Le composant, appelé *filtre*, lit continuellement les entrées sur lesquelles il exécute un traitement ou une sorte de “filtrage” pour produire les sorties [51]. Les spécifications des filtres peuvent contraindre les entrées et les sorties. Un connecteur, quant à lui, est appelé *pipe* puisqu’il représente une sorte de conduite qui permet de véhiculer les sorties d’un filtre vers les entrées d’un autre. Le style “pipes and filters” exige que les filtres soient des entités indépendantes et qu’ils ne connaissent pas l’identité des autres filtres. De plus, la validité d’un système conforme à ce style ne doit pas dépendre de l’ordre dans lequel les filtres exécutent leur traitement.

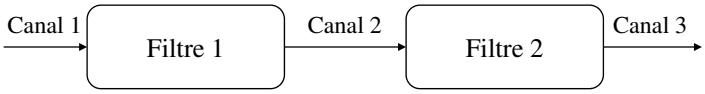

FIG. 1.6 – Le style “pipes and filters”

1.3.4 Avantages des styles architecturaux

L’utilisation des styles architecturaux a un certain nombre d’avantages significatifs. D’abord, elle favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être réappliquées à des nouvelles applications. Elle facilite, pour les autres, la compréhension de l’organisation de l’architecture de l’application si les structures conventionnelles sont bien appliquées et utilisées. Par exemple, concevoir une application selon le style Client-Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants. Finalement, l’utilisation des styles architecturaux peut mener à la réutilisation significative du code : souvent les aspects invariants d’un style architectural se prétendent à des implémentations partagées.

1.4 Architecture dynamique

Les architectures logicielles, comme nous l’avons présenté précédemment, peuvent être décrites comme étant un ensemble de composants et de connecteurs dans un cadre commun. Toutefois, cette approche est une approche statique. En effet, la structure du système, une fois son architecture définie, n’évolue pas dans le temps. Ce type d’architecture commence à perdre de l’ampleur et céder la place au concept d’architecture dynamique suite à de nombreuses raisons détaillées plus tard. Dans la suite, nous présentons une définition et quelques motivations qui sont derrière la naissance de ce concept.

1.4.1 Définition et motivation de la dynamique des architectures

Depuis quelques années, la dynamique des architectures est devenue une activité qui commence à prendre de l’ampleur. Les architectures des systèmes logiciels tendent à devenir de plus en plus dynamiques. En effet, durant son cycle de vie, ce type d’applications peut être amené à évoluer de différentes manières : évolution de l’architecture, ajout de fonctions, remplacements de certains composants par de nouveaux développés selon de nouvelles technologies de programmation ou de communication, etc.

1.4.2 Raison derrière la dynamique des architectures

La dynamique des architectures peut être réalisée pour différentes raisons. Ces raisons peuvent être classées en quatre catégories [70] :

– **Dynamique correctionnelle** : dans certains cas, on remarque que l'application en cours d'exécution ne se comporte pas correctement comme prévu. La solution est d'identifier le composant de l'application qui pose problème et le remplacer par une nouvelle version supposée correcte. Cette nouvelle version fournit la même fonctionnalité que l'ancienne, elle se contente simplement de corriger ses défauts.

– **Dynamique adaptative** : même si l'application s'exécute correctement, parfois l'environnement d'exécution, comme le système d'exploitation, les composants matériels ou d'autres applications ou données dont elle dépend, changent. L'architecture doit donc s'adapter et évoluer soit en ajoutant des nouveaux composants/connexions soit en remplaçant des composants/connexions par d'autres.

– **Dynamique évolutive** : au moment du développement de l'application, certaines fonctionnalités ne sont pas prises en compte. Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités. Cette extension peut être réalisée en ajoutant un ou plusieurs composants/connexions pour assurer les nouvelles fonctionnalités ou même en gardant la même architecture de l'application et étendre simplement les composants existants.

– **Dynamique perfective** : l'objectif de ce type d'adaptation est d'améliorer les performances du système. A titre d'exemple, on se rend compte que l'implémentation d'un composant n'est pas optimisée. On décide alors de remplacer l'implémentation du composant en question. Un autre exemple peut être celui d'un composant qui reçoit beaucoup de requêtes et qui n'arrive pas à les satisfaire. Pour éviter la dégradation des performances de l'application, on diminue la charge de ce composant en installant un autre qui lui partage sa tâche.

Le souci majeur des techniques de conception des systèmes logiciels est donc de pouvoir s'adapter à ces variations et de supporter la dynamicité. Ces variations sont de différents types [112].

1.4.3 Types d'architectures dynamiques

Pour soutenir le développement des architectures dynamiques, plusieurs chercheurs se sont concentrés sur le développement d'approches et de formalismes fournissant des systèmes d'architecture dynamique.

Suite à ces recherches, différents types de dynamiques sont apparus. Parmi ces types nous trouvons :

– **Dynamique d'implémentation** : ce type de dynamique permet de modifier la mise en œuvre d'un composant, c'est-à-dire la manière avec laquelle un composant a été développé (le code du composant) sans changer ni les interfaces ni les connexions. Ce type de changement permet de faire évoluer un composant d'une version à une autre (voir figure 1.7).

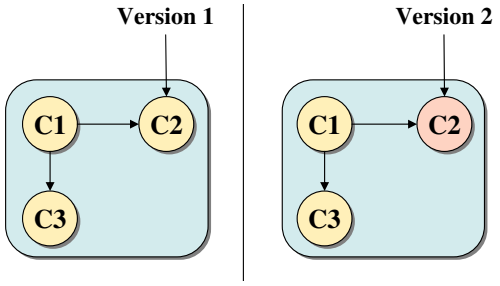

FIG. 1.7 – Exemple de changement d'implémentation

– **Dynamique d'interfaces** : comme nous l'avons détaillé précédemment, un composant fournit ses services à travers des interfaces. La dynamique d'interfaces consiste à modifier les services fournis par un composant au biais de ses interfaces. Ceci, se traduit soit par la modification de l'ensemble des services fournis soit par la modification de la signature d'un service (voir figure 1.8).

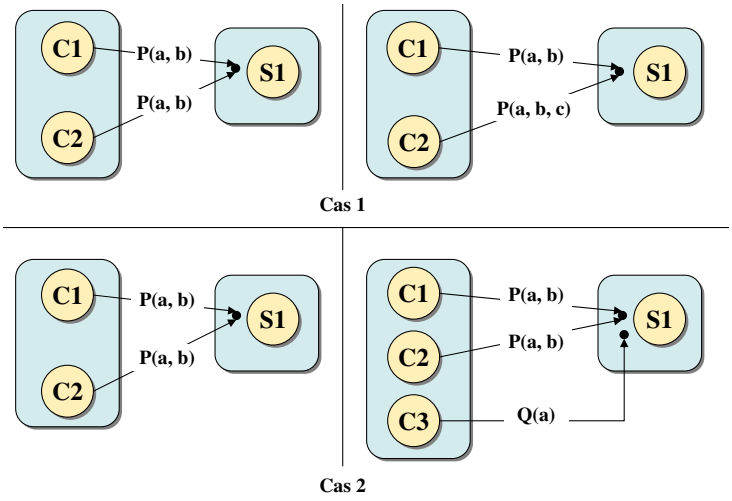

FIG. 1.8 – Exemple de changement d'interface

– **Dynamique de géométrie** : elle est appelée aussi dynamique de localisation ou encore de migration. La dynamique de géométrie modifie la distribution géographique d'une application. En effet, ce type de dynamique altère uniquement l'emplacement des composants. Ainsi, un composant peut changer de localisation en migrant d'un site vers un autre. Comme le montre la figure 1.9, le composant C4 est déplacé du site 2 vers le site 3.

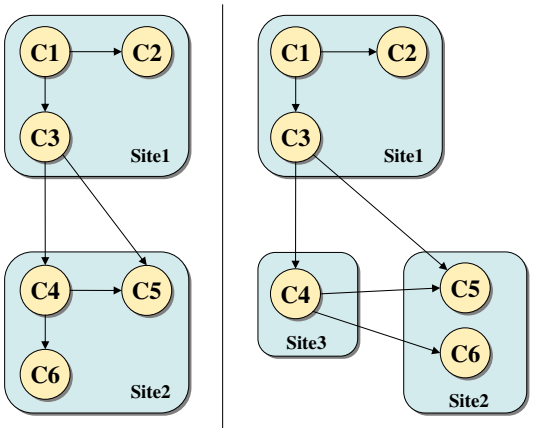

FIG. 1.9 – Exemple de changement géométrique

– **Dynamique de structure** : elle modifie la topologie de l'application. C'est-à-dire dans ce type de dynamique on s'intéresse uniquement au changement de la structure de l'application en termes de composants et de connexions. Cette dynamique se réalise à l'aide de quatre opérations de base [140] : ajout d'un composant, suppression d'un composant, ajout d'une connexion et suppression d'une connexion. Comme le montre la figure 1.10, La connexion T1 est redirigée vers le composant S1.

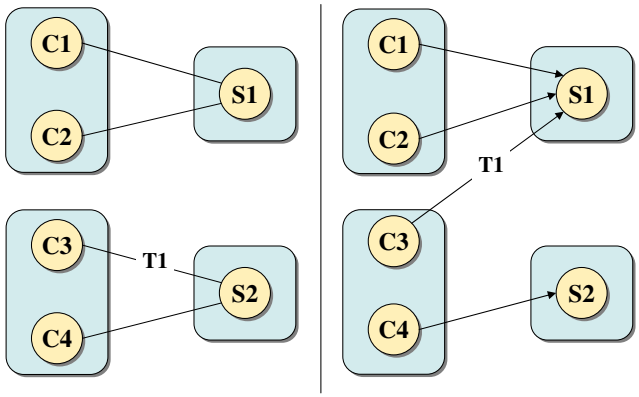

FIG. 1.10 – Exemple de changement de structure

1.5 Conclusion

Dans ce qui précède, nous avons mis notre travail dans son contexte. Notre étude se base essentiellement sur les architectures logicielles. Ces architectures se basent sur la notion de composant. Les architectures à base de composants peuvent changer de structure suite à la variation des exigences des utilisateurs ou à la variation du contexte de l'application.

Dans ce travail, nous nous concentrons sur la modélisation de la dynamique de l'architecture logicielle au niveau conceptuel (design time). Nous remarquons dans ce contexte que l'architecture est à la base de la structure et de l'évolution dynamique des systèmes logiciels. Le développement de ces systèmes exige des approches bien établies garantissant la robustesse de la structure de l'application. En plus, la dynamique de l'architecture doit être contrôlée selon son style architectural afin de préserver des propriétés architecturales du système pendant son évolution et pour ne pas aboutir à des configurations qui risquent de nuire au bon fonctionnement du système.

Avant d'entamer la présentation de notre proposition, nous présentons, dans ce qui suit, un aperçu sur les principaux travaux réalisés dans le domaine de l'architecture logicielle.

2

Description des architectures logicielles : État de l'art

Ce chapitre est organisé en deux parties. Dans une première étape, nous présentons les recherches réalisées pour la description des architectures logicielles. Les contributions existantes liées à ce domaine peuvent être classées principalement, comme le montre la figure 2.1, en quatre axes de recherches. Nous citons les Langages de Description d'Architecture (ADL), le Langage de Modélisation Unifié (UML), les Techniques Formelles (TF) et les Multi-Formalismes (MF). Nous présentons, dans ce qui suit, une étude de chaque approche et nous présentons leurs avantages et leurs limites.

Dans une deuxième étape, nous présentons, comme le montre la figure 2.1, les principaux travaux réalisés dont le but de proposer des méthodes de conception pour les architectures logicielles.

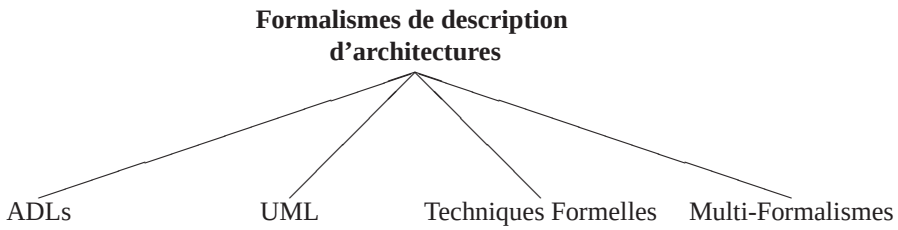

FIG. 2.1 – Formalismes de description d'architectures logicielles

2.1 Langages de description d'architecture (ADL)

Les premiers travaux réalisés pour décrire les architectures logicielles ont été menés dès le début des années 90. Ces recherches ont donné naissance aux langages de description d'architectures (figure 2.2).

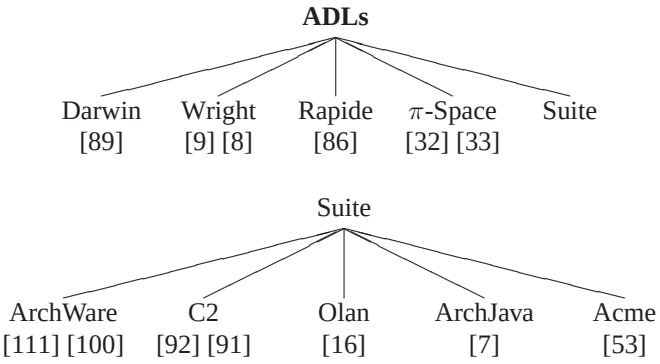

FIG. 2.2 – Les langages de description d'architecture

Il faut préciser qu'il existe plusieurs définitions, mais en général nous adoptons la définition proposée par MEDVIDOVIC et al. [94] "*Les ADLs (Architecture Description Language)* [57, 94, 92, 91, 53] *émergent comme des solutions à base d'une notation textuelle ou graphique, formelle ou semi formelle permettant de spécifier des architectures logicielles*". Ils permettent la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour de la spécification liée à l'architecture. Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans les détails d'implantation. Ils définissent de manière explicite les interactions entre les composants d'un système et fournissent des outils pour aider les concepteurs à décrire l'aspect structurel d'un système.

Les ADLs offrent une grande diversité de notations. Néanmoins, comme le montre la figure 2.3, les concepts de composant, de connecteur et de configuration sont généralement considérés comme essentiels [45]. Les ADLs les plus connus et que nous étudions dans ce rapport sont Darwin [89, 88], Rapid [86], Wright [8], π-Space [32, 33].

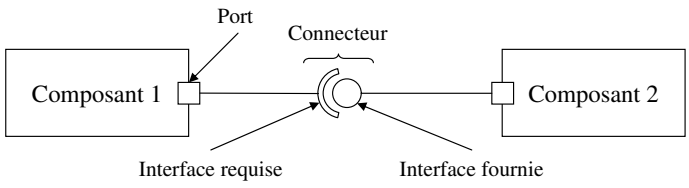

FIG. 2.3 – Les concepts des ADLs

Afin d'illustrer l'aspect dynamique et l'aspect comportemental dans les différents ADLs étudiés, nous présentons l'exemple de l'*édition coopérative des documents* [118, 56]. Comme le montre la figure 2.4, il s'agit d'un système permettant à un ensemble d'utilisateurs (*user*) de coopérer à la rédaction (*write*) et à la lecture (*read*) d'un même document. Le document à éditer est organisé en fragments (*fragment*) numérotées de 0 à N. Un auteur (*writer*) doit avant d'écrire dans une zone, ensemble de fragments consécutifs, la réserver. Une fois que l'auteur a réservé un fragment alors il sera verrouillé jusqu'à la fin de la rédaction. Après la modification, l'auteur peut libérer la zone qu'il a auparavant réservée. Ainsi, chaque fragment du document ne peut se trouver que dans l'un des deux cas de figures suivants : *libre* ou *réservé*.

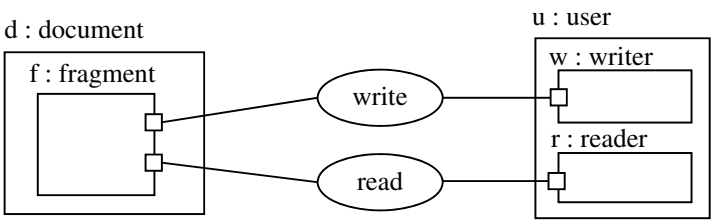

FIG. 2.4 – Exemple de l'édition coopérative des documents

2.1.1 Darwin

Le langage Darwin [89, 88] a été développé par "Distributed Software Engineering Group" de l'Imperial College. Darwin propose un modèle de composants pour la construction d'applications distribuées. Ce langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système [4]. La particularité de Darwin est de permettre la spécification d'une partie de la dynamique d'une application en terme de schéma de création de composants logiciels avant ou pendant son exécution.

Comme la plupart des ADLs, Darwin reprend les trois concepts de base à savoir le composant, le connecteur et la configuration, mais seul le composant est exprimé explicitement.

2.1.1.1 Aspect dynamique et comportemental

Darwin permet de spécifier des composants dynamiques. Il permet de créer des composants en cours d'exécution du système. Le système n'est plus considéré comme un ensemble de composants figés lors de la phase de conception, mais il est possible de spécifier les instants et les emplacements de création dynamique de composants. Darwin fournit deux mécanismes pour décrire la création dynamique : l'instanciation paresseuse et l'instanciation dynamique.

– *L'instanciation paresseuse* permet de déclarer, comme le montre la figure 2.5, des composants qui ne seront pas instanciés immédiatement. Le composant sera instancié au moment où un autre composant souhaite accéder à ce service [13, 4].

```
component Edition{
    inst
        writer : Writer ;
        fragment : dyn Fragment ;
    bind
        writer.reserve – fragment.protect ;
```

FIG. 2.5 – Exemple d'instanciation paresseuse selon Darwin

L'instance `writer` est créée à l'instanciation du composant `Edition`. L'instance `fragment`, quant à elle, est déclarée dynamique avec la clause (`fragment : dyn Fragment`) et ne sera créée que lorsque le `writer` la sollicitera. L'instanciation paresseuse permet donc de décrire la configuration potentielle à l'exécution en déclarant des instances potentiellement nécessaires. Ce mécanisme ne permet d'instancier qu'un seul composant par clause d'interconnexion [13].

– *L'instanciation dynamique* en opposition à l'instanciation paresseuse, de multiples instances peuvent être créées dynamiquement à partir d'une seule clause d'interconnexion. L'instanciation dynamique permet à un composant d'instancier un ou plusieurs autres composants [13, 80].

```
component Edition{
    inst
        writer : Writer ;
    bind
        writer.reserve – dyn Fragment ;
```

FIG. 2.6 – Exemple d'instanciation dynamique selon Darwin

Dans l'exemple de la figure 2.6, l'instance `writer` est créée à l'instanciation du composant composite `Edition`. Le `fragment` n'est pas déclaré, mais un des services du `writer` est relié à la création dynamique d'un `Fragment` (`writer.reserve -- dyn Fragment`). Ainsi, le composant `writer` peut déclencher la création d'un ou de plusieurs `fragments`. Les instances créées ne sont pas référencées et les composants dynamiques ne sont pas accessibles par les autres composants. En d'autres termes, les composants qui sont créés statiquement ne peuvent pas déclencher une communication avec ceux qui sont instanciés dynamiquement, seul l'inverse est possible [80].
Il est à noter que l'instanciation dynamique permet de multiples instanciations au moyen d'une seule clause puisque le lien est spécifié pour le type du composant à instancier (`Fragment`) plutôt que pour une instance de ce type [123].

2.1.1.2 Style architectural

L'inconvénient majeur de Darwin c'est qu'il ne présente pas de mécanisme propre à la formalisation des styles architecturaux [80].

2.1.2 Rapide

Rapide [27, 85, 120] est un langage de description d'architecture dont le but principal est de vérifier, par simulation, la validité d'une architecture logicielle. Il fut proposé à l'origine au projet ARPA (Advanced Research Projects Agency) en 1990. Il est basé sur des événements concurrents. Il est conçu essentiellement pour prototyper des architectures dans des systèmes distribués.

Le langage Rapide ne fournit pas d'outils de génération de code, il permet cependant la description du comportement dynamique et il offre des méthodes de validation (simulation) des architectures qu'il décrit. Il permet aussi de vérifier certaines propriétés comme l'interblocage lors de l'exécution de l'application. Les concepts de base de Rapide sont le composant, l'événement et l'architecture.

2.1.2.1 Aspect dynamique et comportemental

Rapide est capable de modéliser l'architecture des systèmes dynamiques dans lesquels le nombre de composants peut varier quand le système est exécuté. Pour décrire la dynamique, Rapide introduit deux types de variables particulières : les `placeholder` et les `iterator` auxquelles sont associées des règles de création. Le nom d'une variable `placeholder` commence toujours par " `?`". Cette variable désigne un objet qui est susceptible d'être présent. Le nom d'une variable `iterator` commence toujours par " `!`". Cette variable désigne une conjonction d'instances d'un certain type. Les règles de création définissent quand les composants doivent être créés ou détruits lors de l'exécution [80].

```
architecture Edition is
?w : Writer ;
!f : Fragment ;
?service : Service ;
...
connect
?w.request( ?service) ⇒ !f.reply( ?service) ;
end Edition ;
```

FIG. 2.7 – Liaison conditionnelle entre un écrivain et un fragment

Dans l'exemple de l'édition coopérative, comme le montre la figure 2.7, `?w : Writer` est une variable `placeholder` et fait référence à une instance de type `Writer`. `!f : Fragment` est une variable `iterator` et désigne toutes les instances de type `Fragment` présentes dans l'application. Ces déclarations de variables définissent un type d'objet ou un ensemble de types d'objets devant être présents dans l'architecture. Il n'est pas nécessaire de définir les instances de composants présents, car cela peut se faire dynamiquement au fur et à mesure de l'exécution. La partie connexion contient les règles de connexion et les règles de création. Les règles de création définissent les conditions sur les événements qui guident la création de nouveaux composants. Dans l'exemple, la règle de création spécifie l'existence d'un `writer ?w` qui émet avec n'importe quelle donnée de type `Service`, alors que la partie droite de la règle est exécutée. Celle-ci spécifie que tous les `fragment !f` du système reçoivent la même donnée `?service`. Ces règles peuvent être conditionnées par une clause `where`.

Rapide supporte uniquement des manipulations dynamiques où tous les changements d'exécutions doivent être connus a priori. Il permet donc de créer des composants dynamiques et possède des mécanismes permettant leur gestion. Cependant, Rapide ne supporte aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions. En plus, dans le modèle, le concept de connecteur n'apparaît que de manière implicite. Ceci limite la réutilisation [119].

2.1.2.2 Style architectural

Comme Darwin, l'inconvénient majeur de Rapide c'est qu'il ne présente pas de mécanisme propre à la formalisation des styles architecturaux.

2.1.3 Wright

Wright est un langage de description d'architecture créé par ALLEN et GARLAN il fournit des bases formelles pour spécifier les interactions entre les composants (via des connecteurs). Wright est un langage de description orienté plus vers la vérification des protocoles entre les composants, que vers la correction fonctionnelle de l'architecture globale. Il permet de décrire formellement une architecture à l'aide de l'algèbre de processus CSP [9].

Comme les autres ADLs, Wright reprend les trois concepts de l'architecture logicielle à savoir le composant, le connecteur et la configuration.

2.1.3.1 Aspect dynamique et comportemental

La première version de ce langage permettait de décrire uniquement des architectures statiques [9]. En 1998, Wright a été étendu par Dynamic Wright [8] pour supporter les changements dynamiques de la topologie d'une architecture. Cette extension consiste à intro-

duire un programme de configuration (Configurator) décrivant les différentes évolutions possibles de l'architecture.

Pour mettre en évidence les propriétés dynamiques de Dynamic Wright, nous revenons à notre exemple et nous considérons une configuration dans laquelle deux serveurs gèrent l'accès en lecture et en écriture à des documents . Le système est constitué d'un client et d'un serveur (`Primary`) interconnectés via un connecteur. Ce serveur peut tomber en panne à tout moment. L'idée consiste à faire transiter le système d'une configuration à une autre et à mettre à la disposition des utilisateurs un autre serveur (`Secondary`) qui peut remplacer le premier jusqu'à ce que ce dernier soit prêt à fonctionner de nouveau. Il existe donc deux configurations possibles comme le montre la figure 2.8 : le client est soit attaché au serveur principal soit au serveur secondaire [8]. Le serveur primaire signale sa panne à l'aide de l'événement de contrôle `control.down` et son rétablissement avec `control.up`.

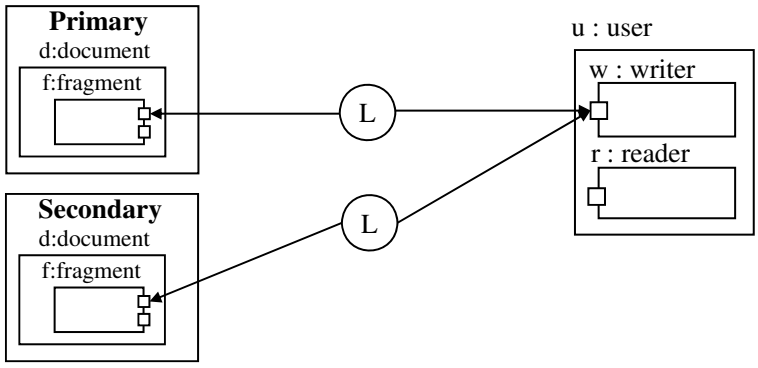

FIG. 2.8 – Configuration selon Wright

Dans l'exemple, comme le montre la figure 2.9, le configurateur attend le déclenchement d'un événement de contrôle. Comme décrit dans la partie `WaitForDown`, deux situations possibles sont spécifiées : le système peut fonctionner correctement et se terminer avec succès ($), ou une erreur peut apparaître avant la terminaison de l'application (réception de l'événement de contrôle `serverDown` de l'instance `Primary`). Si le serveur primaire est détaché du connecteur `detach.Primary.reply.from.L.request` et que le second est attaché à sa place `attach.Secondary.reply.from.L.request`, la nouvelle configuration est alors établie. Au rétablissement du primaire, `WaitForUp` sera activé, modifiant de nouveau la configuration de manière similaire. Les rôles entre les deux serveurs sont alors inversés.

Dynamic Wright propose aussi des actions de reconfiguration "`new`", "`del`", "`attach`", et "`detach`" permettant de décrire les changements de l'architecture en terme de création et de suppression de composants et de connecteurs de manière dynamique. Ces actions sont définies par le programme de configuration.

```
Configurator Edition
    new.U : User
        → new.Primary : FlakyServer
        → new.Secondary : SlowServer
        → new.L : FaultTolerantLink
        → attach.U.request.to.L.reply
        → attach.Primary.reply.to.L.request → WaitForDown
where
    WaitForDown = Primary.control.serverDown
        → detach.Primary.reply.from.L.request
        → attach.Secondary.reply.from.L.request
        → WaitForUP
        $
    WaitForUp = Primary.control.serverUp
        → detach.Secondary.reply.from.L.request
        → attach.Primary.reply.from.L.request
        → WaitForDown
        $
```

FIG. 2.9 – Spécification dynamique selon Wright : tolérance au fautes

2.1.3.2 Style architectural

Wright permet au concepteur de décrire et d'analyser des styles architecturaux. Un style en Wright est composé de deux parties : le vocabulaire et les contraintes sur des configurations. Le *vocabulaire* définit des types de composant et de connecteur. Les *contraintes* définissent les prédicats qui doivent être vérifiés pour n'importe quelle configuration suivant le style. Un style est instancié par une configuration, dont la définition est simplifiée car elle hérite des définitions du style [80].

Wright s'intéresse uniquement aux systèmes ayant un nombre fini de configurations qui sont connues a priori (Wright [8]). Ainsi, il n'est pas possible d'effectuer des opérations de reconfiguration arbitraires durant l'exécution de l'application et en particulier ceux qui sont externes (non prévues d'avance).

2.1.4 π-Space

π-Space [33, 31] est un ADL pour la description des architectures dynamiques à base de composants. Il est formellement fondé sur le π-calcul pour décrire les systèmes dynamiques. Ce langage est étendu par $\sigma\pi$-Space [81] pour la description des styles architecturaux.

Les architectures sont représentées par des configurations de composites, de composants et de connecteurs. Un composite est un élément composé permettant de définir une configuration. Les composants et les connecteurs sont composés d'un ensemble de ports, d'un comportement et d'un ensemble d'opérations (pour les composants) [80].

2.1.4.1 Aspect dynamique et comportemental

π-Space permet de formaliser la dynamique d'une architecture. Lors d'une description, nous pouvons définir quels éléments sont dynamiques. Leurs noms sont alors suffixés du caractère π [80].

Dans cet exemple, comme le montre la figure 2.10, le type client est déclaré comme dynamique. Cela signifie que plusieurs occurrences de type client peuvent être créées dynamiquement. La clause `whenever` permet de spécifier des réactions à des événements. Dans l'exemple, il est spécifié que lorsqu'une nouvelle instance client est créée, celle-ci est attaché au connecteur.

De même, π-Space permet de définir des règles de dynamique permettant de décrire les changements topologiques dus à la création dynamique d'un nouvel élément. La dynamique peut être déclenchée par un composant. Pour cela, des ports spécifiques permettant d'évoquer des reconfigurations `attachementEvolutionPort`, `ComponentEvolutionPort` et `EvolutionPort` sont définis. Ces ports sont sollicités via le mot-clé `evolvable`.

compose Simple
    C π :Client ||
    L :Link ||
    S :Server
where
    attach S@p@reply **to** L@s@sRequest,
    attach S@p@request **to** L@s@sReply,
whenever
    new Cπ $\Rightarrow$ **new** L@cπ,
    new L@cπ $\Rightarrow$ **attach** Cπ@p@request **to** L@cπ@cReply,
    new L@cπ $\Rightarrow$ **attach** Cπ@p@reply **to** L@cπ@cRequest ;

FIG. 2.10 – Spécification dynamique selon π-Space

L'exemple de la figure 2.11 définit le comportement d'un composant. Ce comportement engage un port (pC) qui permet de faire évoluer la topologie. Il permet le déclenchement d'un événement pour une évolution via le port `changementAttachement`.

Le comportement commence par la réception d'un message via le port pC, puis continue

par le déclenchement d'un événement pour une évolution via le port changementAttachement.

```
define behaviour component type ExempleTopologie
      [pC :PortReception [canalReception :[ ] ] ,
      changementAttachement :AttachmentEvolutionPort]
{
ExempleTopologie[changementAttachement] =
      (pC@canalReception () •
      evolvable(changementAttachement) •
      ∼ Boucle[pC] ),
Boucle[pC] = ( ( pC@ canalReception () • ∼ Boucle[pC] ) + $)
}
```

FIG. 2.11 – Changement de la topologie selon π-Space

2.1.4.2 Style architectural

$\sigma\pi$-Space [81] étend π-Space avec des concepts spécifiques aux styles. Dans $\sigma\pi$-Space, un style permet de contraindre un type d'élément. Un style peut être défini pour des composites, des composants, des connecteurs, des ports, des comportements et des opérations. La définition d'un style permet de décrire un ensemble de contraintes qui représentent les caractéristiques d'une famille de types topologiques.

Un style contient une liste d'éléments de construction. Ces éléments possèdent des noms génériques qui sont spécialisés lorsqu'une occurrence de l'élément est utilisée dans une architecture. Ces éléments peuvent être associés à une cardinalité définissant une contrainte sur le nombre d'occurrences dans l'architecture.

Dans $\sigma\pi$-Space, il est possible d'indiquer, dans un style, avec le symbole π qu'un élément peut être dynamique. En plus, il est possible de définir des règles de configuration dans un style de composite, en décrivant les attachements possibles. Le mot `in style` permet de spécifier qu'un type doit satisfaire un style. Le type doit alors respecter les contraintes imposées par le style et il ne peut utiliser dans sa description que les éléments de construction fournis par le style [80].

2.1.5 Conclusion

Les ADLs fournissent une base formelle pour spécifier les architectures logicielles en modélisant les composants, les connecteurs, et les configurations. Ces ADLs permettent de spécifier la vue structurelle d'une architecture logicielle. Selon l'étude que nous avons menée [57], nous avons identifié certaines lacunes. Les points faibles rencontrés dans les ADLs sont de plusieurs ordres. La majorité des ADLs proposent une notation spécifique,

ce qui nécessite un effort de plus pour comprendre le langage. La dynamique structurelle de l'architecture n'est pas bien supportée par les ADLs bien qu'il existe un certain nombre de propositions intéressantes, connues sous le nom de langages de description d'architecture dynamiques (DADLs), mais pas toujours suffisantes (Darwin [90], Dynamic Wright[8] et Rapide [85]). En effet, ces ADLs permettent difficilement de prendre en compte l'extension et l'évolution des besoins. Peu de langages permettent d'introduire des stratégies de reconfiguration et les solutions fournies ne s'appliquent qu'à une description statique de l'application. Les ADLs s'intéressent uniquement aux systèmes ayant un nombre fini de configurations qui sont connues a priori (Wright [8]). Ainsi, il n'est pas possible d'effectuer des opérations de reconfiguration à l'improviste durant l'exécution de l'application et en particulier ceux qui sont externes (non prévues d'avance). En outre, l'aspect comportemental est quasi absent dans la majorité des ADLs à l'exception de quelques uns tel que (Rapide [85]). Cependant, cette description exprime seulement des événements communicants et ne considère pas des opérations de reconfiguration.

La plupart des ADLs ne sont pas satisfaisants pour la description des styles. Ils introduisent eux-mêmes des suppositions spécifiques sur les architectures. Ces suppositions peuvent être incohérentes ou conflictuelles avec celles sous-jacentes au système. Certains ADLs sont spécifiques à un domaine et à un style particulier tel que MetaH [17] qui est un ADL spécifique aux architectures des systèmes multi-processeurs temps réel pour l'aviation. Acme [53] et $\sigma\pi$-Space [81, 31] ne sont pas spécifiques à un domaine, leur style propre est plus générique [80].

Pour l'étude des ADLs, nous nous sommes basés, dans nos recherches, sur la première génération. Cependant, il y a des nouvelles générations des ADLs tels que Prisma [114], Plastik [66] et Pilar [40] qui traitent d'une façon plus appropriée l'aspect dynamique.

2.2 Langage UML

UML [48, 101] est un langage qui offre une notation graphique unifiée connue par la majorité des architectes et apportant une solution pour décrire l'architecture logicielle et ce, tout particulièrement avec l'introduction dans le standard UML 2.0 de la notion de composant comme élément de structuration des modèles.

A l'heure actuelle UML est dans sa version 2.0 et il comporte 13 diagrammes. 6 diagrammes décrivent la structure du système (figure 2.12 : diagramme de classes, diagramme de structure composite, diagramme de composants, diagramme de déploiement, diagramme d'objets et diagramme de paquetage). Les 7 autres diagrammes permettent de décrire le comportement du système (figure 2.13 : les diagrammes d'interactions (diagramme de séquence, diagramme de vue d'ensemble des interactions, diagramme de communication et diagramme de temps), diagramme d'activité, diagramme de machine à états et diagramme de cas d'utilisation).

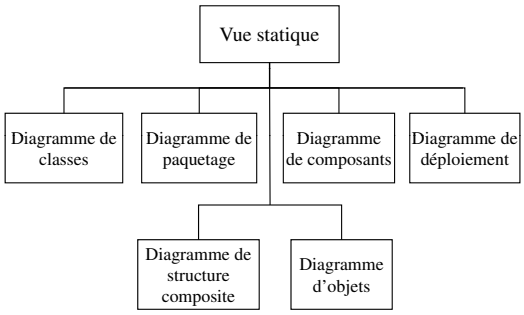

FIG. 2.12 – Diagrammes de structure

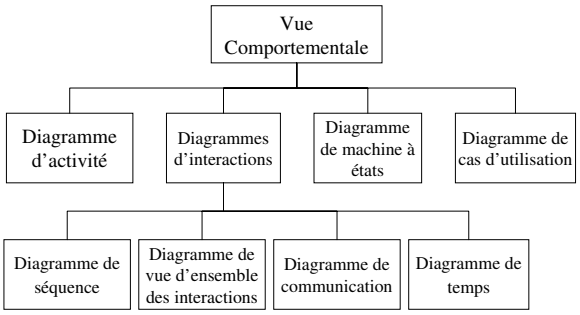

FIG. 2.13 – Diagrammes de comportement

2.2.1 UML et l'architecture logicielle

UML 2.0 apporte des améliorations pour représenter les architectures logicielles. Le diagramme de composants a été révisé et le concept de classificateur structuré (structured classifier) a été intégré. Les classes, les collaborations et les composants sont des classificateurs structurés. UML 2.0 introduit la notion de composant (component) ainsi que la notion de connecteur (connector : assembly connector et delegation connector). L'introduction de ces concepts, ainsi que ceux de port ou encore la distinction entre interfaces offertes et requises fournissent une palette d'éléments de notation intéressante pour la modélisation de l'architecture logicielle.

Un composant peut avoir son propre comportement et détermine un contrat entre ses services fournis et ses services requis, au travers de la définition de ports. Il est ainsi remplaçable par un autre élément dont la définition des ports est similaire [26].

Un connecteur exprime une relation entre les composants. Il s'agit de liens (une instance d'association) qui permet la communication entre deux ou plusieurs instances.

Un port représente les points d'accès par lesquels les messages sont émis et reçus par un classificateur structuré. Les ports ont un type fourni sous la forme d'un ensemble d'interfaces. Ce qui est nouveau en UML 2.0, c'est la possibilité de doter une interface d'attributs et de pouvoir définir des associations entre interfaces. Ainsi par rapport aux interfaces d'UML 1.5, un port peut implémenter (to implement) ou utiliser (to require) ou fournir (to provide) une interface. Quand il s'agit d'une interface fournie, elle caractérise le comportement du composant fourni. Quand il s'agit d'une interface requise, elle caractérise le comportement attendu [26].

UML 2.0 étend les diagrammes de composants d'UML 1.5 d'une simple représentation physique vers la prise en compte des composants logiques. Ainsi, deux vues complémentaires sont possibles pour modéliser les composants :

– Une vue externe (boîte noire). L'attention est portée sur les contrats entre le composant et son environnement.
– Une vue interne (boîte blanche). L'attention est portée sur l'organisation interne du composant en termes de component, sub-component, connector, etc.

Comme le montre la figure 2.14, deux types de connecteurs ont été définis. Les connecteurs d'assemblage (assembly connector) relient une interface requise à une interface fournie. Les connecteurs de délégation (delegation connector) relient un port à l'interface (compatible, et de même type requise/fournie) d'une partie interne. Une flèche indique le sens de la délégation [26].

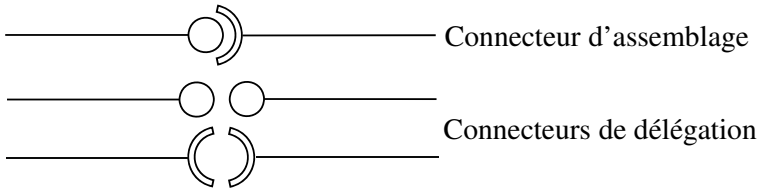

FIG. 2.14 – Types de connecteur

2.2.2 UML en tant que langage pour la description des architectures

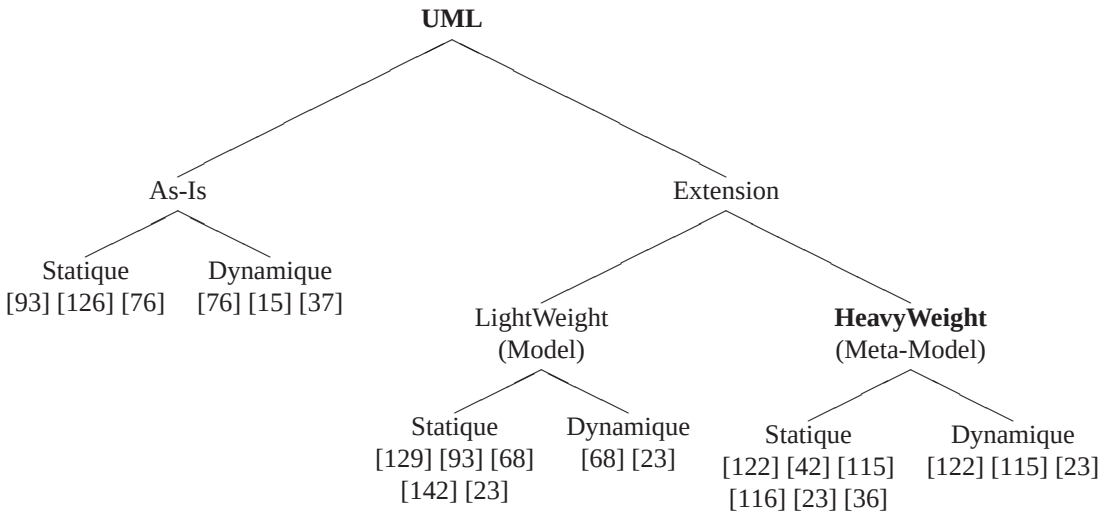

FIG. 2.15 – Le langage UML

La diversité des ADLs proposés ont conduit certains travaux de recherche à considérer

UML en tant que langage de description d'architecture ou de mapper vers la notation UML [126, 139, 116, 52, 50, 69, 63]. Les travaux menés dans ce domaine, comme le montre la figure 2.15, peuvent être divisés principalement en deux approches. La première est appelée "As-Is" et consiste à utiliser la notation existante d'UML sans aucune extension pour représenter l'architecture logicielle. La deuxième, consiste à étendre la syntaxe et la sémantique de la notation UML afin de décrire l'architecture logicielle. Cette extension peut être divisée en "heavyweight" et "lightweight". L'extension "heavyweight" [116, 115, 69] ajoute de nouveaux modèles ou remplace la sémantique existante en modifiant le méta-modèle d'UML. L'extension "lightweight" [129, 93] définit de nouveaux éléments permettant d'étendre la notation UML sans modification du méta-modèle.

Parmi ces travaux, nous citons les recherches de MEDVIDIVOC et al. [93] qui visent une adaptation d'UML. Leurs travaux cherchent à représenter le style d'architecture C2. Dans une première étape, ils proposent l'utilisation d'UML 1.4 tel qu'il est (As-Is) sans considérer ses mécanismes d'extension. Ils utilisent le diagramme de classes pour représenter la structure du système. Ce diagramme illustre les classes issues de l'application, leurs relations d'héritage et leurs associations. Dans une deuxième étape, MEDVIDIVOC et al. cherchent à étendre et à adapter le méta-modèle UML en y ajoutant des stéréotypes nécessaires à la description des architectures tels que les interfaces et les connecteurs. Les traits avec le rond représentent la réalisation des interfaces par les classes, alors que les traits en pointillés avec des flèches orientées vers les ronds représentent des relations de dépendance d'une classe avec l'interface impliquée. Les diagrammes de collaboration sont aussi utilisés pour la représentation des interactions entre les instances de classes [93].

Nous citons aussi les travaux de MARTINEZ et al. qui représentent un exemple des extensions lourdes d'UML 1.4 (heavyweight) [116]. Ils proposent un style architectural appelé C3 basé sur le style C2. Le méta-modèle proposé permet la description syntaxique des architectures logicielle. Les nouveaux constructeurs sont rajoutés au méta-modèle comme sous-classes de la méta-classe ModelElement qui est une sous-classe de Element (la méta-classe racine). Les constructeurs *Constraint*, *Attribute* et *Parameter* définis dans le paquet Core, et les types Boolean et ProcedureExpression définis dans le paquet Data Types sont utilisés. Pour étendre le méta-modèle UML, C3 suit les consignes suivantes :

– Aucun méta-constructeur (méta-classe du méta-modèle UML) n'est supprimé ni modifié dans sa syntaxe ou sémantique.
– Les nouveaux méta-constructeurs doivent avoir un minimum de relations avec les méta-constructeurs existants, c'est-à-dire ils doivent être auto-contenus [115, 96].

2.2.3 Conclusion

Une étude basée sur études de cas [60, 58, 97], nous a permis de constater que UML 1.x, en tant que langage pour les architectures logicielles, ne satisfait pas complètement les besoins structuraux nécessaires pour la description des architectures logicielles. Il ne permet de décrire convenablement ni l'architecture logicielle ni la dynamique structurelle [96]. De ce fait, nous avons fait recours à UML 2.0. Celui-ci offre en effet des améliorations

significatives en ce qui concerne les concepts propres utilisés pour modéliser les architectures logicielles. L'ajout des concepts de port, de connecteur et de diagrammes d'architectures (classes, composants, structure composite) enrichit grandement le pouvoir expressif d'UML 2.0 et le rend plus adapté à supporter la modélisation des architectures logicielles.

L'apport majeur d'UML 2.0 par rapport à UML 1.x est qu'il offre un langage mieux défini et plus ouvert. Une étude, basée sur des études de cas [97] a montré que UML 2.0 permet la modélisation de l'architecture logicielle et de décrire le comportement interne de l'application. Cependant, et malgré les différents diagrammes proposés, il ne permet pas de décrire la dynamique structurelle d'une architecture logicielle en termes d'ajout et/ou de suppression de composants et/ou de connexions.

2.3 Techniques formelles

Le troisième axe de recherche dans le domaine des architectures logicielles consiste à tirer profit des avantages des techniques formelles existantes en les appliquant à la conception architecturale. L'objectif est de pouvoir offrir aux développeurs la possibilité de raisonner, analyser et vérifier l'architecture d'un système logiciel.

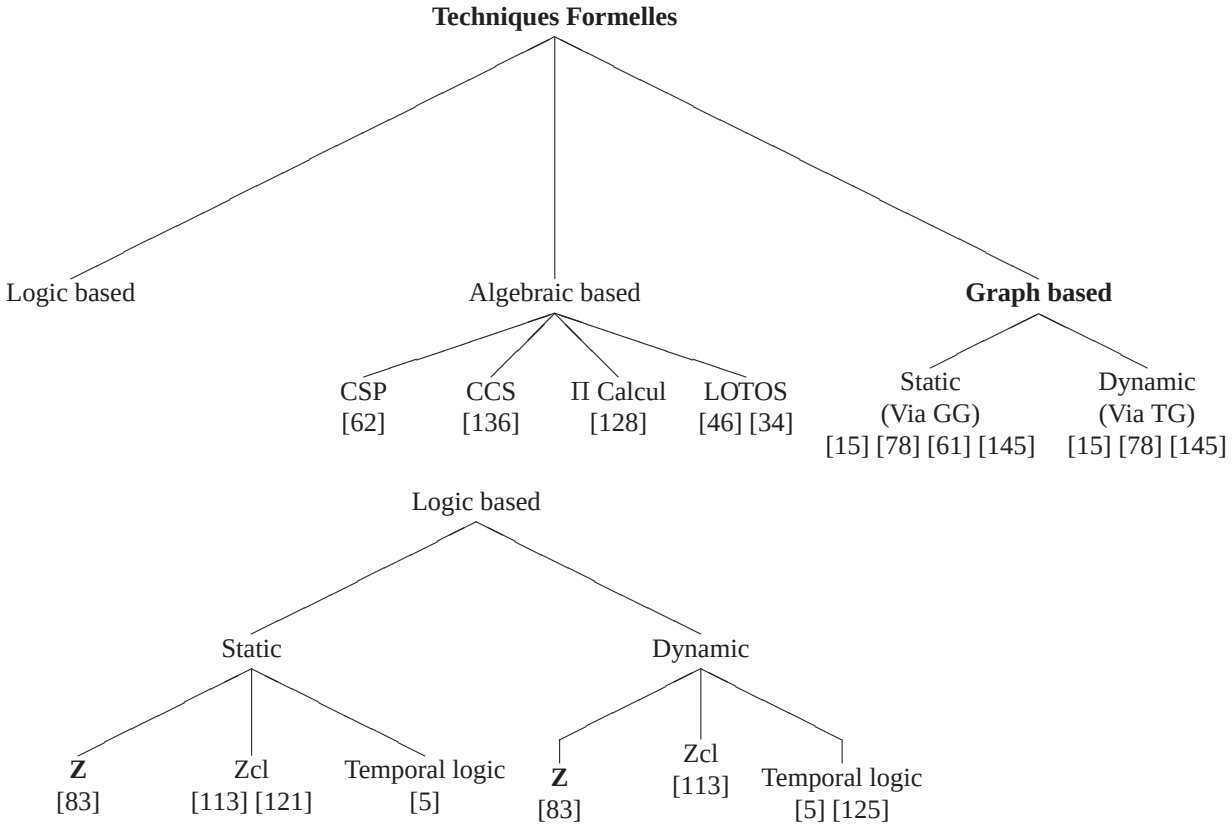

FIG. 2.16 – Les techniques formelles

Les techniques formelles peuvent être classés principalement en trois catégories (figure 2.16) : les techniques basées sur les graphes [78, 61, 145], les techniques basées sur les algèbres de processus [62, 128] et les techniques basées sur la logique [113, 125].

Les travaux formels élaborés dans le domaine de l'architecture logicielle ne possèdent pas les mêmes visions. Néanmoins, ils cherchent tous à pouvoir offrir aux développeurs la possibilité de raisonner et d'analyser une architecture. Cette présentation portera essentiellement sur les techniques basées sur les graphes et plus particulièrement les grammaires de graphe.

2.3.1 Techniques basées sur les graphes

Pour spécifier les architectures logicielles et le style architectural, il est possible d'utiliser les grammaires de graphe.

Les grammaires de graphe, à contexte libre, offrent la possibilité de définir l'architecture logicielle d'un système en terme de graphe. Ainsi, les nœuds du graphe représentent les entités logicielles constituant l'architecture (composant) et les arcs correspondent aux liens de communication entre ces entités (connexion) [78]. L'avantage principal des grammaires de graphe est qu'elles permettent de décrire, en plus de l'aspect statique, l'aspect dynamique d'une architecture logicielle. En effet, pour décrire l'aspect dynamique, il faut utiliser les règles de réécriture de graphe. Un système de réécriture de graphe est spécifié par un ensemble de règles de réécriture où le rôle de chaque règle est de remplacer un sous graphe. Ces règles permettent de décrire clairement les changements de la topologie d'une architecture en termes d'ajout et/ou de suppression de composants et/ou de connexions.

Nous citons principalement les travaux de MÉTAYER qui constituent probablement les premiers travaux de description basées sur les graphes [78]. Le modèle décrit est structuré en deux niveaux. Le premier décrit l'architecture sous la forme d'un graphe et le second décrit les styles architecturaux par une grammaire de graphes. L'évolution de l'architecture est décrite par des règles de transformation de graphes. Le modèle définit une approche formelle basée sur un algorithme de vérification permettant de vérifier a priori et de manière statique la consistance des règles de l'évolution de l'architecture. Cette approche permet de prouver que les contraintes considérées par la description de l'architecture sont préservées par ces règles [96].

Dans HIRSCH et al. [41], les auteurs reprennent le même modèle pour la description de l'architecture et de son évolution dynamique. Ils décomposent la description de l'architecture en trois parties. La première spécifie les règles de la construction de la configuration initiale. La deuxième partie spécifie les règles régissant l'évolution dynamique de l'architecture. La troisième partie spécifie les règles régissant la communication. Ces travaux abordent aussi la problématique de la consistance du point de vue de la communication et des états de composants. Le cycle de vie des composants (comprenant par exemple leur activation et leur désactivation) est simulé en affectant des labels (correspondant à l'état courant des composants) aux nœuds des graphes et des règles de réécriture. La partie décrivant la communication est spécifiée via l'introduction d'événements (en notation

CSP, Hoare) et leur prise en compte en étiquetant les arcs des graphes représentant les connexions entre composants [96].

2.3.2 Conclusion

La plupart des techniques formelles offrent les bases nécessaires pour vérifier et valider la spécification d'une architecture logicielle [8, 14, 145]. Certains travaux sont certes très intéressants [83, 78], dans le sens où ils ont pu mettre en œuvre un système de preuve qui vérifie que toute opération de reconfiguration préserve la consistance d'une architecture vis-à-vis de son style architectural.

Cependant, les grammaires de graphe, comme toute technique formelle, sont basées sur des notations mathématiques qui exigent des connaissances et des expertises assez importantes. Les grammaires de graphe à contexte libre présentent quelques limites. Elles ne permettent pas par exemple de décrire des propriétés quantitatives concernant le nombre maximal d'entités devant exister dans le système. En plus, il n'est pas possible de décrire des propriétés fonctionnelles.

2.4 Multi-Formalismes

Plusieurs travaux de recherche [45, 61, 83, 145] visent à combiner différents formalismes (figure 2.17). L'objectif est de minimiser les inconvénients et d'augmenter le pouvoir expressif en fusionnant les avantages afin de décrire la dynamique de l'architecture logicielle.

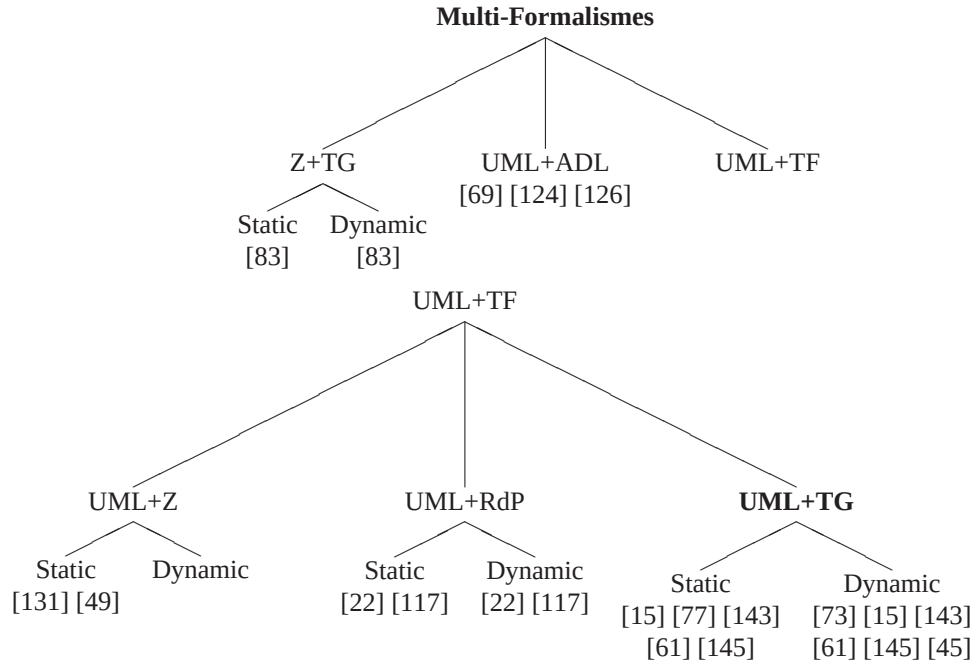

FIG. 2.17 – Les Multi-Formalismes

2.4.1 *Z* et la transformation de graphe

L'approche de LOULOU et al. [83] est basée sur une notation mixte. Cette notation a pour intérêt d'améliorer la lisibilité, de réduire la complexité de la description et de prendre en charge également la dynamique du système. En effet, l'approche utilise la notation *Z* [3] pour décrire l'aspect statique de l'architecture et une notation mixte intégrant les grammaires de graphe et la notation *Z* pour décrire la dynamique structurelle. La notation mixte, comme le montre la figure 2.18, intègre une composante graphique et une composante fonctionnelle. La partie graphique est exprimée par la notation Δ (grammaire de graphe). Quant à la partie fonctionnelle, elle est liée aux pré-conditions imposées aux règles. Ces pré-conditions sont exprimées par la notation *Z*.

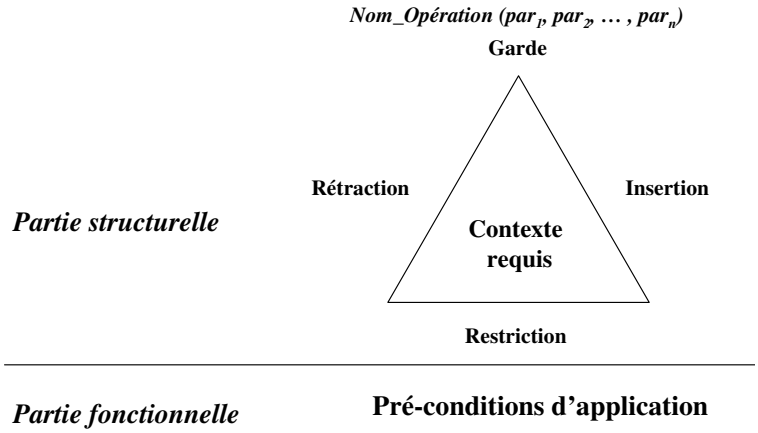

FIG. 2.18 – La notation mixte

Néanmoins, cette approche ne manque pas de limites. En effet, elle ne prend pas en considération l'aspect coordination des différentes opérations de reconfiguration. En plus, la vérification des règles se fait manuellement. Aucun outil n'est proposé. En plus, comme toute technique formelle, elle exige une certaine connaissance mathématique.

2.4.2 UML et les techniques formelles

Nous nous intéressons dans cette section aux travaux qui cherchent à fusionner les règles de transformation de graphe avec la notation UML.

Dans [15, 77], la notation UML et les règles de transformation de graphe sont intégrées pour spécifier l'aspect statique et l'aspect dynamique d'une architecture logicielle. Les diagrammes de classes et les diagrammes d'objets sont utilisés pour décrire l'aspect structurel du système. Les règles de transformation de graphe sont utilisées pour décrire la

dynamique. Cette approche propose aussi la vérification des modèles afin de prouver la conformité d'une spécification par rapport à son style architectural.

L'avantage principal de ces travaux est la combinaison de plusieurs formalismes afin d'augmenter le pouvoir expressif des langages. Cependant, les notations proposées, qui sont basées sur des notations mathématiques difficiles à appréhender, n'offrent pas des outils permettant la validation et la vérification de leurs spécifications. Une telle approche nécessite une connaissance approfondie de la notation mathématique.

2.4.3 Conclusion

Ces travaux permettent la combinaison de plusieurs formalismes afin d'élargir la couverture des différents aspects de l'architecture. Cependant, les notations proposées sont généralement très liées aux transformations de graphe. La plupart de ces travaux n'offrent pas des outils permettant la validation et la vérification de leurs spécifications.

2.5 Synthèse

TAB. 2.1 – Comparaison entre les différentes techniques de description des architectures logicielles.

	ADL	Transformation de graphe	Multi-Formalismes	UML
Avantages	- Premiers travaux qui traitent de l'architecture logicielle. - Langages proposant des solutions cohérentes pour décrire l'aspect structurel.	- Notation très puissante pour décrire la dynamique structurelle. - Notation basée sur une approche mathématique.	- Combine souvent les avantages de deux ou plusieurs notations.	- Notation très populaire. - Standard. - Fournit les bases nécessaires pour décrire l'architecture logicielle.
Limites	- Plusieurs notations et langages. - Absence de standards. - Aspect dynamique n'est pas bien supporté. - Quasi absence d'outillages.	- Absence de standards. - Exige un certain niveau d'aptitude mathématique. - Validation manuelle. - Quasi absence d'outils.	- Les notations proposées sont très liées aux transformations de graphe. - Exige un certain niveau d'aptitude mathématique. - Absence d'outillages. - L'aspect coordination n'est pas supporté.	- La dynamique structurelle n'est pas supportée. - Pas de validation. - Pas d'outils spécifiques.

Selon l'étude menée, nous avons noté (tableau 2.1) que les ADLs souffrent de plusieurs insuffisances pour modéliser l'architecture logicielle. Plusieurs travaux cherchent à surmonter ces problèmes par la définition d'une notation visuelle et unifiée, inspirée de la notation

UML. D'autres travaux cherchent à utiliser la notation UML mais ces travaux s'intéressent essentiellement à l'aspect structurel alors que l'aspect dynamique est généralement quasi absent.

Les techniques formelles et plus précisément les travaux menés sur les transformations de graphe permettent de décrire l'aspect dynamique d'une architecture logicielle mais ces travaux exigent toujours un certain niveau d'aptitude mathématique.

D'autres travaux cherchent à surmonter les problèmes des ADLs et de la notation UML en les combinant avec les transformations de graphe. Cependant, ces travaux ne manquent pas de problèmes. Les notations proposées sont très liées aux transformations de graphe, difficiles à appréhender alors que l'aspect coordination est toujours absent.

Nous avons noté également que peu de travaux présentent un processus clair, un environnement complet et convivial fait défaut même si des prototypes existent.

2.6 Présentation des méthodes agiles

Créer des applications répondant aux attentes des utilisateurs est un des enjeux de la conception des architectures logicielles. Ces architectures doivent être analysées, conçues, mises en œuvre et doivent être flexibles et aisément maintenables au rythme des changements des besoins. Une description visuelle commune de l'architecture aide également les analystes, les architectes et les développeurs à rester alignés lors d'une activité de réutilisation ou de reconfiguration. Les méthodes agiles [103] sont des méthodes de gestion de projets informatiques et de développement qui entrent dans ce contexte et cherchent à proposer des solutions de modélisation. Elles ont pour priorité la satisfaction réelle du besoin du client, et non d'un contrat établi préalablement. Elles rendent le travail de développement plus facile et portent principalement leur attention sur les ressources humaines [103]. Les méthodes agiles connaissent un intérêt croissant depuis quelques années car elles apportent des solutions itératives, incrémentales et basées sur l'accepation du changement.

Comme le montre la figure 2.19, il existe une variété de méthodes agiles. Les principales méthodes sont : "Dynamic Software Development Method" (DSDM) [134], "Rapid Application Development" (RAD), "eXtreme Programming" (XP) [71, 43, 35], "Rational Unified Process" (RUP) [74, 75], "Two Tracks Unified Process" (2TUP) [127]. Ces méthodes sont globalement similaires et la plupart des techniques qu'elles préconisent sont communes. Une étude des principes proposés révèle un tronc commun issu des racines du RAD. Seules des techniques complémentaires les unes aux autres ou mieux adaptées à des typologies et à des tailles de projets spécifiques les différencient. Nous présentons dans cette section les méthodes les plus récentes et les plus utilisées.

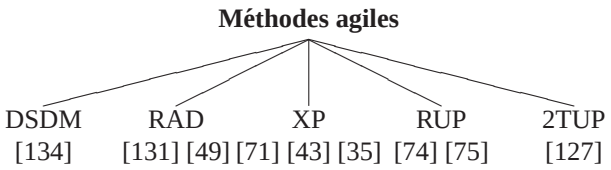

FIG. 2.19 – Méthodes agiles

2.6.1 UP - Unified Process

Le processus unifié UP (Unified Process) est un processus de développement logiciel construit sur UML pour la description de l'architecture logicielle. Il est itératif et incrémental, centré sur l'architecture, conduit par les cas d'utilisation et orienté vers la diminution des risques. C'est un patron de processus pouvant être adapté à une large classe de systèmes logiciels et à différents domaines d'application [65]. La gestion d'un tel processus est organisée suivant quatre phases : pré-étude (inception), élaboration, construction et transition. Ses activités de développement sont définies par cinq disciplines fondamentales qui décrivent la spécification des besoins, l'analyse et la conception, l'implémentation, le test et le déploiement [127]. Un processus UP doit être :

– Itératif et incrémental.
– Piloté par les risques.
– Construit autour de la création et de la maintenance d'un modèle, plutôt que de la production massive de documents.
– Orienté composant.
– Orienté utilisateur.

Il existe plusieurs travaux qui adaptent le processus unifié comme démarche de développement. Parmi les plus connues, nous citons RUP et 2TUP.

2.6.2 RUP - Rational Unified Process

RUP (Rational Unified Process) est un processus de développement par itérations, promu par la société Rational Software. Il se caractérise par une approche globale nommée "Vue 4+1". Les cinq composants de cette vue sont : la vue des cas d'utilisation, la vue logique, la vue d'implémentation, la vue de processus et la vue de déploiement [74]. RUP offre un processus itératif, incrémental et tient compte des besoins des utilisateurs. Il peut être vu comme une implémentation de la démarche générique UP, complétée par une série d'outils informatiques ainsi qu'une documentation des processus décrivant ces outils. Mais ce processus reste très procédural et lié à des outils bien particuliers qui ont tendance à l'alourdir [138].

2.6.3 2TUP - 2 Tracks Unified Process

2TUP (2 Tracks Unified Process) est un processus UP, dédié à la modélisation des systèmes d'information. Il apporte une réponse aux contraintes de changement continuel imposé aux systèmes d'information. En ce sens, il renforce le contrôle sur les capacités d'évolution et de correction de tels systèmes. Il intègre également la notion de composant selon différentes granularités : composants métier, composants logiciels, etc. [127, 130].

L'axiome fondateur de ce processus est de diviser la démarche en deux branches : fonctionnelle (approche par les fonctionnalités) et technique (étude de leur mise en œuvre). Le bénéfice attendu est de pouvoir réutiliser l'aspect fonctionnel en cas de changement d'architecture technique, et de pouvoir réutiliser l'aspect technique en cas d'évolution du fonctionnel (ou tout simplement pour développer une autre application présentant les mêmes contraintes d'architecture).

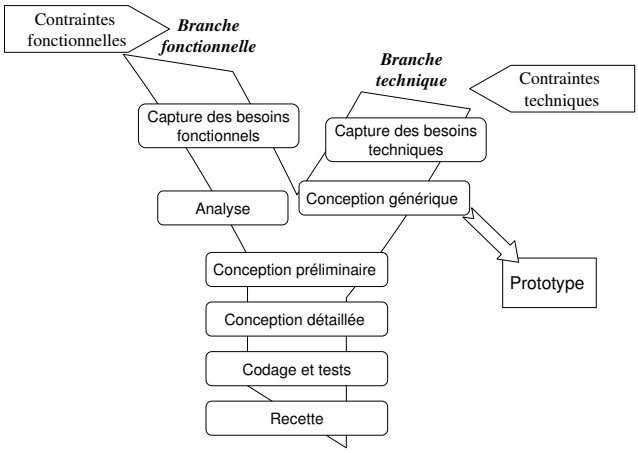

FIG. 2.20 – Le processus 2TUP

A l'issue des évolutions du modèle fonctionnel et de l'architecture technique, la réalisation du système consiste à fusionner les résultats des deux branches. Cette fusion conduit à l'obtention d'un processus de développement en forme de Y qui dissocie les aspects techniques des aspects fonctionnels. Illustré par la figure 2.20, le processus en Y s'articule autour de 3 phases : technique, fonctionnelle et de réalisation.

2TUP est construit sur UML. Il est itératif, centré sur l'architecture et conduit par les cas d'utilisation.

2.6.4 MDA : Model Driven Architecture

MDA [67] tend à produire une approche standard de l'utilisation d'UML en tant que langage de modélisation. Le standard MDA fait partie des travaux de l'OMG au même titre que CORBA (Common Object Request Broker Architecture) et UML et a fait l'objet, ces deux dernière années, d'une forte activité d'innovation. L'objectif de MDA est donc de définir une plateforme capable d'exécuter un modèle UML, assorti de règles OCL et d'aspects, et ainsi rendu exécutable. MDA propose une architecture en trois sous-modèles qui sont le CIM (Computation Independent Model), le PIM (Platform Independant Model) et le PSM (Platform Specific Model). Le premier permet de modéliser les exigences du client sans entrer dans les détails de réalisation de l'application ni sur les traitements. Le deuxième permet de modéliser un comportement cible sans se soucier des technologies sous-jacentes et particulièrement de l'outil MDA utilisé afin de garantir à terme un premier niveau d'interopérabilité entre les outils. Le troisième modèle est facultativement décrit en UML, car il représente l'implémentation des directives de conception vis-à-vis de la plateforme visée. Suivant la stratégie adoptée par l'éditeur de l'outil MDA, le PSM peut générer du code intermédiaire ou bien rendre le modèle directement exécutable au travers de moteurs d'interprétation. Le courant MDA représente un potentiel de développement important pour UML, car il apporte à ce dernier une valorisation encore plus significative aux projets informatiques. C'est pourquoi UML 2.0 apporte un soutien particulier à ce projet en renforçant la formalisation du méta-modèle et du langage de description de règles OCL [127].

2.7 Conclusion

Nous avons présenté, dans ce chapitre, l'état de l'art des travaux relatifs aux différentes approches permettant la description des architectures logicielles. Nous avons présenté leurs avantages ainsi que leurs limites. Afin de combler ce manque, nous proposerons, dans le chapitre qui suit, notre approche qui consiste à apporter des solutions pour la modélisation de la dynamique des architectures logicielles.

3

Profil UML pour décrire la dynamique des architectures logicielles

Après l'étude des différentes techniques utilisées pour décrire l'architecture logicielle, nous présentons dans ce qui suit, notre approche qui cherche à combiner plusieurs formalismes en profitant de leurs avantages et en essayant d'apporter des solutions à leurs limites. Notre approche est inspirée des ADLs qui constituent la base de tout travail sur l'architecture logicielle. Elle tire profit de la puissance des grammaires de graphe pour décrire la dynamique structurelle et elle offre une notation unifiée graphique basée sur la notation UML 2.0.

Dans nos travaux de thèse, nous nous concentrons sur la modélisation de la dynamique de l'architecture logicielle au niveau conceptuel (design time). Nous traitons essentiellement le problème de l'évolution des exigences représentant un changement dans l'activité soutenue par le système modélisé. Nous associons des opérations de reconfiguration architecturale à ces situations pour adapter le système à ces changements.

Notre approche, basée sur un profil UML [59, 60], permet de décrire la dynamique des architectures logicielles. Elle est basée sur trois méta-modèles, comme le montre la figure 3.1. Le premier (Quadrillage en pointillé avec couleur rouge) permet de décrire le style architectural d'une architecture logicielle. Il décrit l'ensemble des types de composants qui constituent le système, les types de connexions entre eux, ainsi que les contraintes d'ordre architectural. Le deuxième (Rayures verticales avec couleur vert) permet de décrire la dynamique et l'évolution de l'architecture en termes d'opérations de reconfiguration que peuvent avoir une architecture tout au long de son cycle de vie. Dans notre approche, nous nous intéressons uniquement à la dynamique structurelle (ajout et/ou suppression de composants et/ou de connexions). Le troisième méta-modèle (Briques diagonales avec couleur

bleu) permet de décrire la coordination. Il décrit l'enchaînement et l'ordre d'exécution des différentes opérations de reconfiguration.

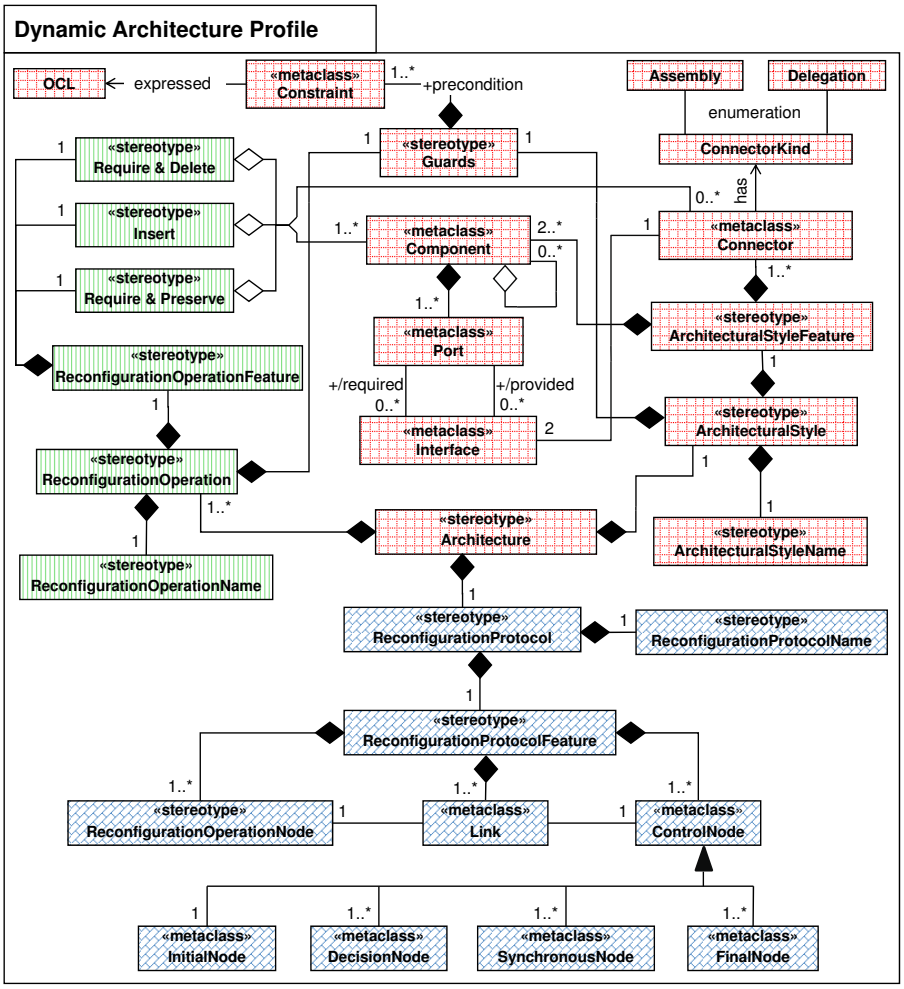

FIG. 3.1 – Le profil de l'architecture dynamique

3.1 Style architectural

La prise en compte de la complexité croissante des systèmes distribués, dynamiques et évolutifs et les contraintes inhérentes de ces systèmes font qu'il est nécessaire de pouvoir disposer d'un support permettant une maîtrise fine du processus de développement et une gestion sûre des différents éléments utilisés par le système. Pour cela, nous proposons un style architectural, à base de composants, pour la définition des types de composants pouvant intervenir dans le système et des connexions entre ces composants. Il définit aussi l'ensemble des propriétés architecturales qui doivent être satisfaites par toutes les configu-

rations appartenant à ce style. Le méta-modèle du style architectural étend le diagramme de composants d'UML 2.0. Il est décrit par un ensemble de concepts caractérisant la structure d'une architecture logicielle.

3.1.1 Méta-modèle

La structure du méta-modèle de *style architectural* est décrite par la figure 3.2.

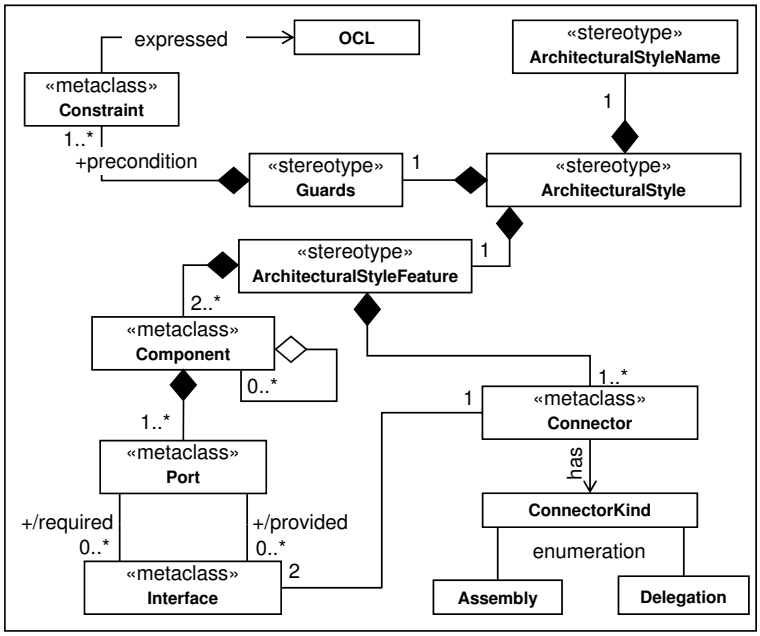

FIG. 3.2 – Le méta-modèle du style architectural

Le stéréotype «*ArchitecturalStyle*» est composé de trois stéréotypes «*ArchitecturalStyleName*», «*Guards*» et «*ArchitecturalStyleFeature*». Le stéréotype «*ArchitecturalStyleName*» décrit le nom du système à spécifier. Le stéréotype «*Guards*» décrit les contraintes de l'architecture que le système doit respecter durant son évolution. Ces contraintes sont exprimées avec le langage OCL (Object Constraint Language). Le stéréotype «*ArchitecturalStyleFeature*» est composé par les méta-classes "*Component*" et "*Connector*". Il spécifie l'ensemble des types de composants et de connecteurs qui constituent le style architectural d'un système. Cette spécification est décrite en utilisant la notation UML 2.0.

La méta-classe "*Component*", éventuellement composée de plusieurs composants, représente une partie modulaire d'un système. Chaque "*Component*" a une ou plusieurs interfaces fournies et/ou requises exposées par l'intermédiaire de ports. La méta-classe "*Port*" représente le point d'interaction pour un composant. La cardinalité [1..*] entre

"*Component*" et "*Port*" exprime qu'un composant peut avoir un ou plusieurs ports. La méta-classe "*Interface*" représente l'interface d'un composant. Une interface peut être soit de type fourni $+/$*provided* ou requis $+/$*required*. La méta-classe "*Connector*" définit un lien qui rend possible la communication entre deux ou plusieurs composants. La méta-classe "*ConnectorKind*" spécifie deux types de connecteurs. Les connecteurs de délégation "*Delegation Connector*" et les connecteurs d'assemblage "*Assembly Connector*". Un connecteur de type "*Delegation*" exprime un lien entre deux composants partant d'une interface requise vers une interface requise ou d'une interface fournie vers une interface fournie. Un connecteur de type "*Assembly*" exprime un lien entre deux composants partant d'une interface requise vers une interface fournie.

3.1.2 Modèle

Pour décrire le style architectural, nous proposons une nouvelle notation graphique décrite dans la figure 3.3. Cette notation sera détaillée avec un exemple dans la section 3.4.1.

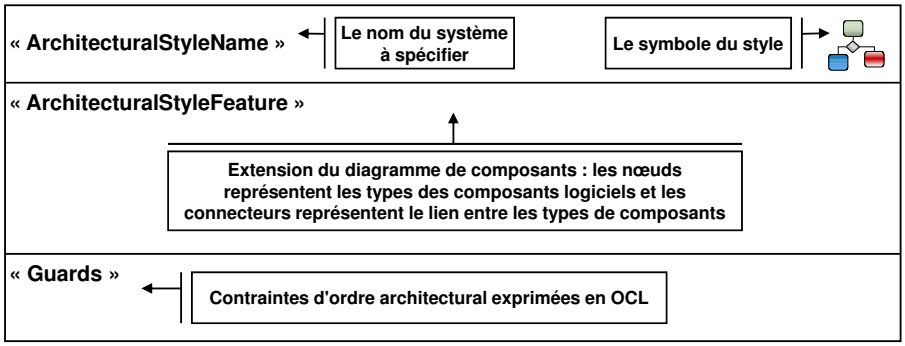

FIG. 3.3 – Le modèle du style architectural

Cette notation est composée de trois parties :

– *ArchitecturalStyleName* : pour indiquer le nom du système à spécifier.

– *ArchitecturalStyleFeature* : pour décrire le style architectural. Nous spécifions l'ensemble des types de composants et des connecteurs entre ces types. La spécification est faite en utilisant la notation UML 2.0.

– *Guards* : pour préciser les contraintes d'ordre architectural que le système doit respecter durant son évolution. Ces contraintes sont exprimées dans le langage OCL.

3.2 Opération de reconfiguration

Tout au long de son cycle de vie, l'architecture logicielle peut évoluer en exécutant des opérations de reconfiguration. Ces opérations font évoluer l'architecture d'une configuration à une autre. L'évolution de l'architecture doit être réalisée sans perturber le fonctionnement du système. Pour cela, nous proposons un deuxième méta-modèle permettant de décrire les différentes opérations de reconfiguration assurant l'évolution de l'architecture d'une application en termes d'ajout et/ou de suppression de composants et/ou de connexions. Ce méta-modèle qui étend le méta-modèle du style architectural, est caractérisé par un ensemble de concepts permettant de décrire la dynamique structurelle d'une architecture logicielle.

3.2.1 Méta-modèle

Les concepts décrits dans le méta-modèle du style architectural sont pris en considération dans le méta-modèle opération de reconfiguration. Dans ce dernier, nous ajoutons d'autres stéréotypes permettant de décrire la dynamique structurelle. La structure du méta-modèle *opération de reconfiguration* est décrite par la figure 3.4.

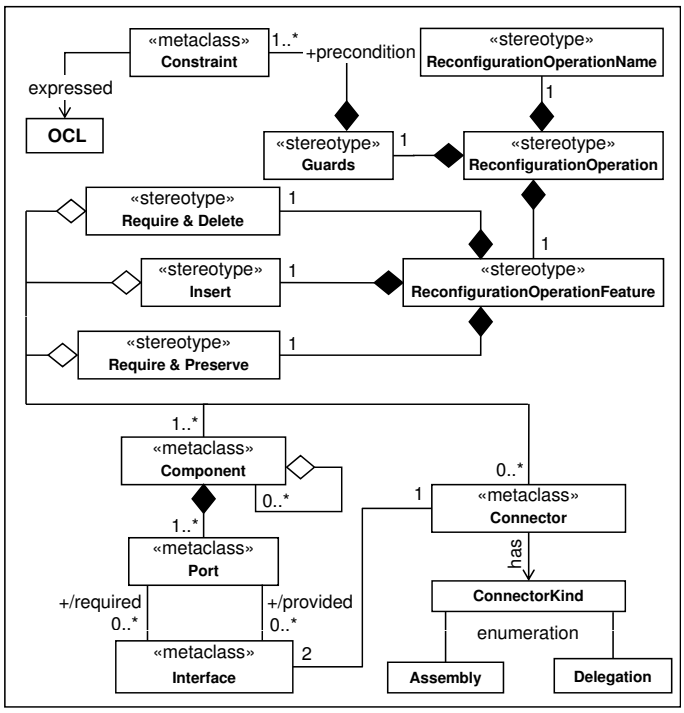

FIG. 3.4 – Le méta-modèle des opérations de reconfiguration

Le stéréotype «*ReconfigurationOperation*» décrit les différentes opérations de reconfiguration. Il est composé des stéréotypes «*ReconfigurationOperationName*», «*ReconfigurationOperationFeature*» et «*Guards*». Le stéréotype «*ReconfigurationOperationName*» décrit le nom de l'opération de reconfiguration à spécifier. Le stéréotype «*Guards*» décrit les contraintes de l'architecture que le système doit respecter durant l'exécution de l'opération de reconfiguration. Ces contraintes sont exprimées avec le langage OCL. Le stéréotype «*ReconfigurationOperationFeature*» est composé de trois stéréotypes :

– Le stéréotype «*require & preserve*» décrit la partie non modifiée durant l'opération de reconfiguration.

– Le stéréotype «*Insert*» décrit la partie à ajouter dans le système durant l'opération de reconfiguration.

– Le stéréotype «*require & delete*» décrit la partie à supprimer durant l'opération de reconfiguration.

3.2.2 Modèle

Pour décrire l'aspect dynamique, nous proposons, comme le montre la figure 3.5, une nouvelle notation. Cette notation sera présentée avec plus de détail dans la section 3.4.2.

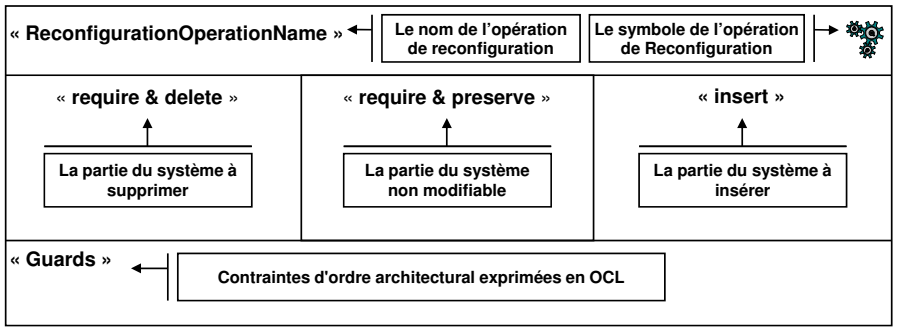

FIG. 3.5 – Le modèle d'une opération de reconfiguration

Cette notation est composée de cinq parties :

– «*ReconfigurationOperationName*» : pour énoncer le nom de l'opération de reconfiguration à exécuter.

– «*Require & delete*» : pour préciser la partie du système à supprimer durant l'opération de reconfiguration.

– «*Insert*» : pour préciser la partie à ajouter dans le système durant l'opération de reconfiguration.

– «*Require & preserve*» : pour préciser la partie non modifiée durant l'opération de reconfiguration.

– «*Guards*» : pour caractériser les contraintes architecturales que le système doit respecter durant son évolution. Ces contraintes sont exprimées dans le langage OCL.

La représentation dynamique que nous avons proposée est inspirée des transformations de graphe et plus précisément de la notation △. La figure 3.6 et le tableau 3.1 présente une correspondance entre les deux notations.

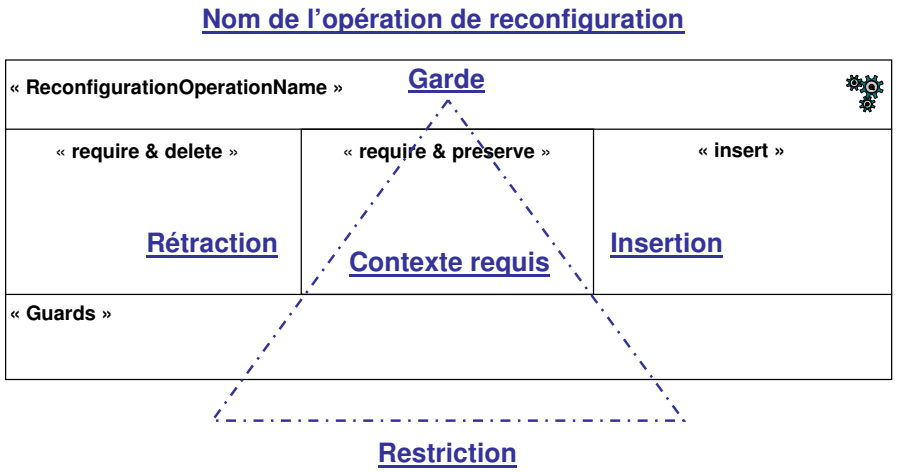

FIG. 3.6 – La correspondance entre la notation △ et la notation proposée

TAB. 3.1 – Tableau de correspondance entre la notation Δ et la notation proposée

Notation Δ	**Notre notation**
Nom de l'opération de reconfiguration	ReconfigurationOperationName
Rétraction	Require & delete
Contexte requis	Require & preserve
Insertion	Insert
Restriction + Garde	Guards

3.3 Protocole de reconfiguration

La description du style architectural et des opérations de reconfiguration est nécessaire mais insuffisante pour spécifier l'évolution d'une architecture logicielle. Pour cela, nous ajoutons le protocole de reconfiguration afin de décrire l'organisation et l'enchaînement

entre les différentes opérations de reconfiguration décrites au niveau dynamique. Ce profil est basé sur la notation UML 2.0 et étend le diagramme d'activités.

3.3.1 Méta-modèle

La structure du *protocole de reconfiguration* est décrite par la figure 3.7.

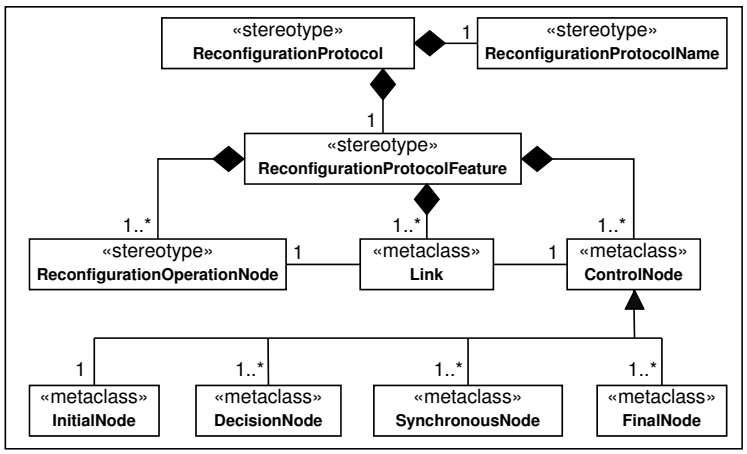

FIG. 3.7 – Le méta-modèle de protocole de reconfiguration

Le stéréotype «*ReconfigurationProtocol*» est composé des stéréotypes «*ReconfigurationProtocolName*» et «*ReconfigurationProtocolFeature*». Le stéréotype «*ReconfigurationProtocolName*» décrit le nom du système à spécifier. Le stéréotype «*ReconfigurationProtocolFeature*» est composé par le stéréotype «*ReconfigurationOperationNode*» et les deux méta-classes "*ControlNode*" et "*Link*". La méta-classe "*ControlNode*" est utilisée pour coordonner les flots entre les différentes opérations de reconfiguration. Nous utilisons quatre nœuds : "*InitialNode*", "*DecisionNode*", "*SynchronousNode*" et "*FinalNode*". La méta-classe "*Link*" établit le lien entre "*ControlNode*" et «*ReconfigurationOperationNode*».

3.3.2 Modèle

Pour décrire le protocole de reconfiguration, nous proposons une nouvelle notation telle que présentée dans la figure 3.8. Cette notation permet de décrire l'enchaînement entre les opérations de reconfiguration. Les nœuds de contrôle "*ControlNode*" sont utilisés pour synchroniser entre les différentes opérations de reconfiguration «*ReconfigurationOperationNode*» et les liens "*Link*" permettent d'établir le lien entre les opérations de reconfi-

guration et les nœuds de contrôle. Cette notation sera détaillée avec un exemple dans la section 3.4.3.

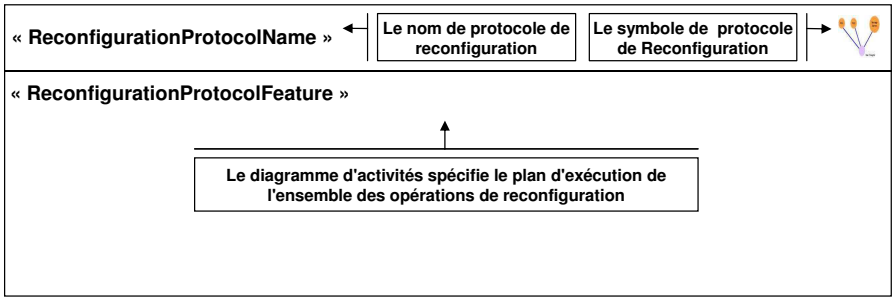

FIG. 3.8 – Le modèle de protocole de reconfiguration

3.4 Illustration

Afin d'illustrer notre travail, nous avons choisi un exemple qui a également servi pour illustrer les formalismes proposés dans [78] et [83]. Il s'agit d'un système logiciel de contrôle de patients *PMS* (Patient Monitoring System) permettant aux infirmières de contrôler leurs patients à distance au sein d'une clinique. A chaque patient, un contrôleur doit être attaché à son lit permettant de prendre des mesures. Grâce à ce contrôleur, chaque infirmière peut vérifier l'état de ses patients en demandant à distance les informations concernant la tension, la température, etc. En contre partie, lorsque l'état d'un patient devient anormal, c'est à dire quand l'une des données sort de sa limite, le contrôleur de lit envoie un signal d'alarme à l'infirmière responsable. Après avoir expliqué le principe sous-jacent de ce système, nous donnons maintenant la description informelle de son architecture. A chaque service de la clinique (pédiatrie, cardiologie, maternité, etc.) est associé un service d'événement (*EventService*) pour gérer les communications entre les infirmières (*Nurses*) et les contrôleurs de lit (*Patients*) rattachés au service en question. Chaque infirmière demande des informations relatives à ses patients en envoyant une requête au service d'événement auquel elle est liée. Ce service prend en charge cette demande et la transmet aux patients concernés ou plutôt à leur contrôleur de lit. Lorsque l'état d'un patient devient anormal, son contrôleur de lit envoie un signal d'alarme au service d'événement auquel il est rattaché. Ce service transmet par conséquent ce signal à l'infirmière responsable.

Le PMS est un système évolutif possédant les propriétés architecturales suivantes :

– Le nombre maximal de services est de 3.
– Un service contient au maximum 5 infirmières.
– Un service contient au maximum 15 patients.
– Un patient doit être toujours affecté à un service unique.

– Le service auquel est affecté le patient doit contenir au moins une infirmière.
– Une infirmière doit être attachée à un seul service.

Dans ce qui suit, nous présentons une modélisation possible de ce système selon le profil que nous avons proposé. Les contraintes OCL que nous présentons, dans les différents exemples, sont écrites et validées avec l'outil USE [55].

3.4.1 Modélisation du style architectural

En se basant sur la notation proposée pour décrire le style architectural, nous présentons dans la figure 3.9 le style architectural du système PMS. Le nom du style «*ArchitecturalStyleName*» est PMS. Dans la partie «*ArchitecturalStyleFeature*» et selon la notation UML 2.0, notre système contient trois types de composants (*EventService*, *Patient* et *Nurse*).

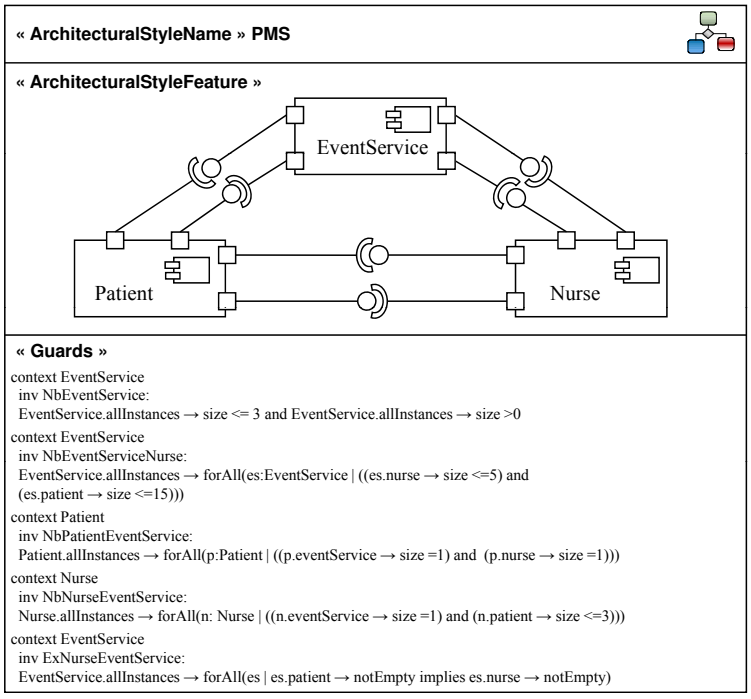

FIG. 3.9 – Le style architectural du PMS

Chaque type de composant contient quatre interfaces : deux fournies et deux requises. Le type *Patient* et le type *Nurse* communiquent à travers le type *EventService*.

Dans la partie «*Guards*» nous spécifions avec le langage OCL les propriétés architecturales. Le système peut contenir au maximum trois instances de type *EventService*. Chaque instance de *EventService* peut contenir entre zéro et cinq instances de type *Nurse* et peut

inclure entre zéro et quinze instances de type *Patient*. Une instance de type *Nurse* peut s'occuper au plus de trois instances de type *Patient*.

La figure 3.10 décrit une configuration possible de l'architecture PMS. Cette configuration contient une instance de composant de type *EventService*, une instance de composant de type *Patient* et une instance de composant de type *Nurse*. Ces instances sont interconnectées par des instances de type connecteur.

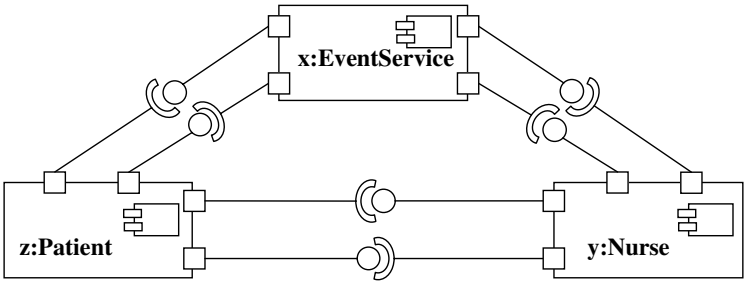

FIG. 3.10 – Une configuration possible du système PMS

3.4.2 Modélisation des opérations de reconfiguration

En se basant sur la notation proposée pour décrire le modèle dynamique, nous présentons, dans cette section, la spécification des différentes opérations de reconfiguration permettant de faire évoluer notre système PMS tout en tenant compte des propriétés décrites dans le style architectural.

3.4.2.1 Insertion d'un service d'événement

Cette opération de reconfiguration permet d'insérer une instance de composant de type *EventService*. La modélisation de cette opération de reconfiguration, avec notre nouvelle notation, est donnée par la figure 3.11.

Le nom de l'opération de reconfiguration «*ReconfigurationOperationName*» est *Insert_EventService(x)*

Dans la partie «*insert*» nous présentons l'instance de *EventService* à ajouter dans le système. Dans cette opération nous n'avons rien à supprimer, c'est pourquoi la partie «*require & delete*» est vide. En plus, pour l'insertion de cette instance de composant nous n'avons besoin d'aucune instance de composant dans la partie «*require & preserve*».

Pour exécuter correctement l'insertion d'un *EventService*, il faut vérifier la contrainte OCL : `EventService.allInstances → size < 3` donnée dans la partie

«*Guards*» qui traduit que le nombre d'instance de type *EventService* doit être strictement inférieur à 3.

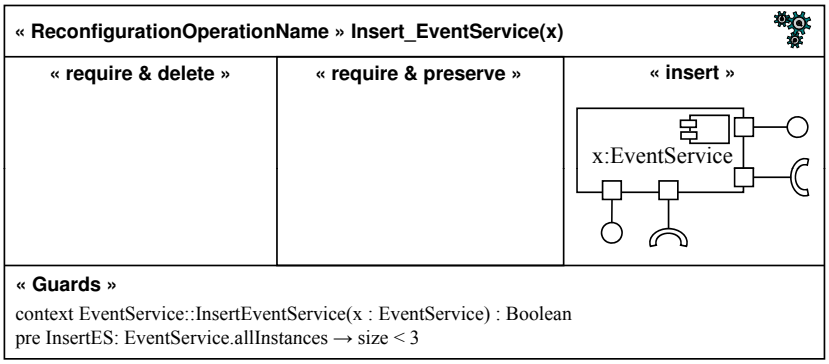

FIG. 3.11 – Insertion d'un *EventService*

La figure 3.12 décrit une configuration possible de l'architecture PMS après l'exécution de l'opération de reconfiguration *Insert_EventService*.

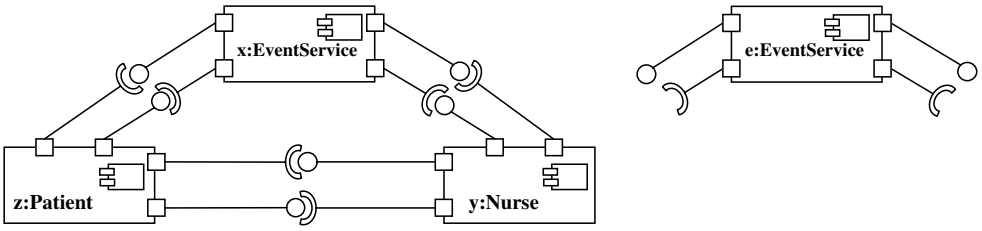

FIG. 3.12 – Une configuration possible après l'insertion d'un *EventService*

3.4.2.2 Insertion d'un patient

Cette opération de reconfiguration permet d'insérer une instance de composant de type *Patient* et de la lier à une instance de composant de type *EventService*. Pour exécuter cette opération, il faut mettre une instance de type *Patient* dans la partie «*insert*». L'insertion d'un patient nécessite l'existence d'une instance de type *Nurse* connectée à une instance de type *EventService*. Ces instances doivent être représentées dans la partie «*require & preserve*».

L'exécution de cette opération est conditionnée par trois conditions présentées dans la partie «*Guards*» : il faudrait d'abord qu'il y ait au moins une infirmière appartenant à ce service pour pouvoir s'occuper du nouveau patient : `x.nurse → size > 0` . En

plus, il faudrait vérifier que le service en question ne contient pas déjà quinze patients : `x.patient → size < 15` et que le nombre de patient pour l'infirmière en question est strictement inférieur à 3 : `y.patient → size < 3` .

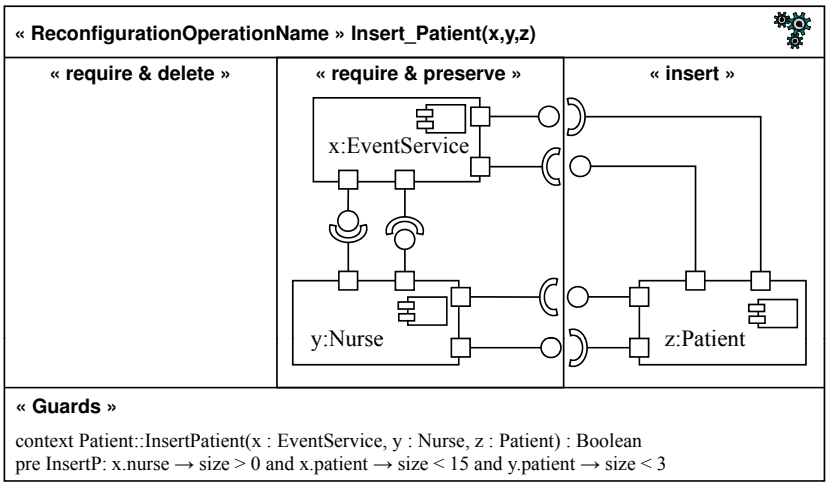

FIG. 3.13 – Insertion d'un *Patient*

La figure 3.14 décrit une configuration possible de l'architecture PMS après l'exécution de l'opération de reconfiguration *Insert_Patient*.

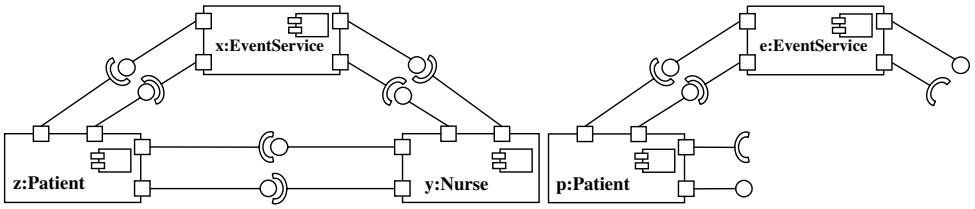

FIG. 3.14 – Une configuration possible après l'insertion d'un *Patient*

3.4.2.3 Insertion d'une infirmière

Cette opération de reconfiguration permet d'insérer une instance de composant de type *Nurse* et de la lier à une instance de composant de type *EventService* quelconque (devant exister dans le système) comme précisé dans la partie «*Guards*». Pour appliquer cette règle, on doit vérifier que l'instance de composant *EventService* en question ne contient pas déjà cinq instances de *Nurse* : `x.nurse → size < 5` .

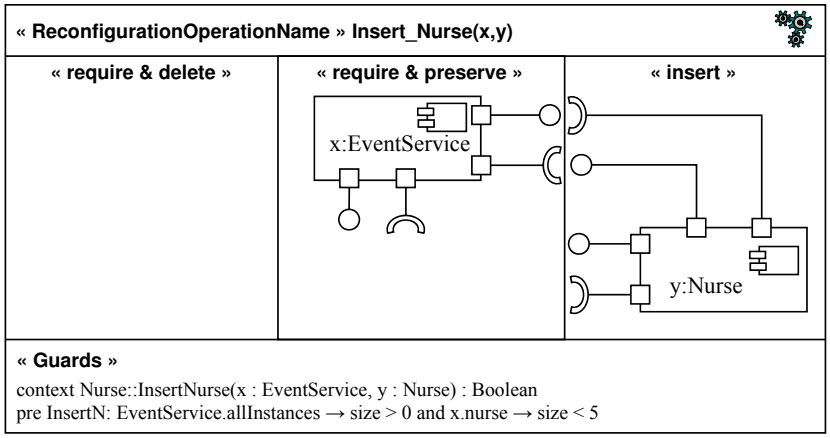

FIG. 3.15 – Insertion d'une *Infirmière*

La figure 3.16 décrit une configuration possible de l'architecture PMS après l'exécution de l'opération de reconfiguration *Insert_Nurse*.

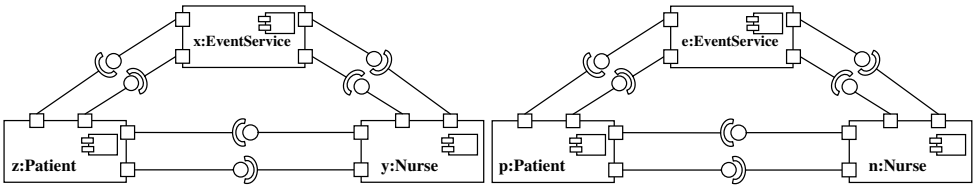

FIG. 3.16 – Une configuration possible après l'insertion d'une *Infirmière*

3.4.2.4 Transfert d'un patient

Un patient peut quitter un service vers un autre (sans pour autant être supprimé du système). Cette opération de reconfiguration permet de transférer une instance de composant de type *Patient* vers une instance de composant de type *EventService*. Pour ce faire, nous supprimons une connexion représentée dans la partie «*require & delete*» et nous ajoutons une nouvelle connexion dans la partie «*insert*».

Le transfert d'une instance de type *Patient* est exécuté seulement s'il y a une instance *Nurse* dans le nouveau *EventService* (`y :Nurse`) dont le nombre d'instance de type *Patient* en charge est strictement inférieur à trois : `y.patient → size < 3` . En plus, il faut vérifier que le nombre d'instance de type *Patient* connecté au nouveau *EventService* est strictement inférieur à 15 : `w.patient → size < 15` . La représentation de cette opération de reconfiguration est donnée par la figure 3.17.

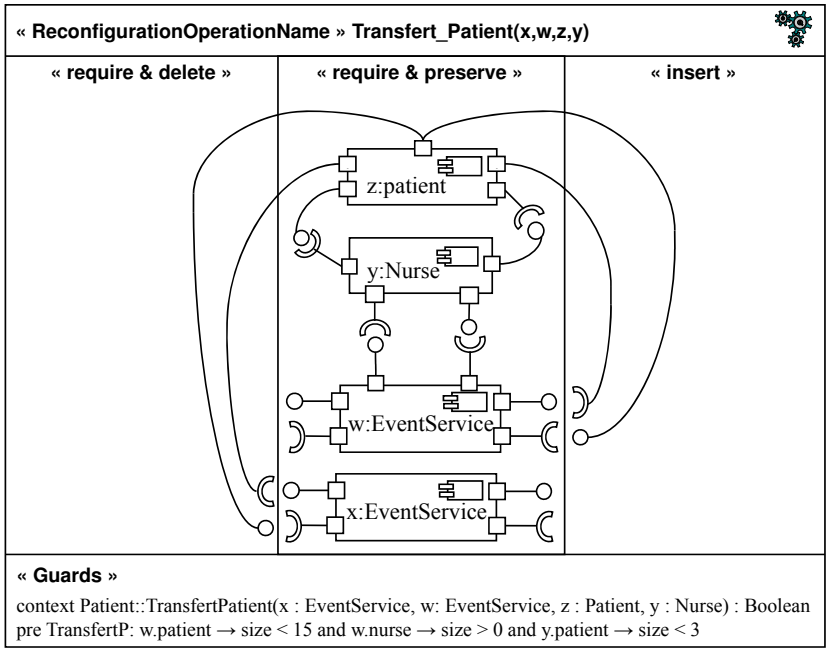

FIG. 3.17 – Transfert d'un *Patient*

3.4.2.5 Transfert d'une infirmière

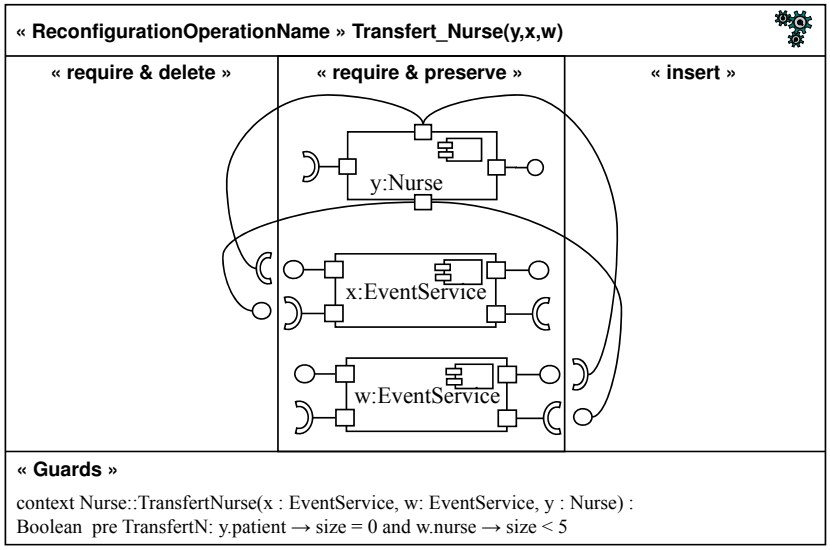

FIG. 3.18 – Transfert d'une *Infirmière*

Une infirmière peut quitter un service vers un autre (sans pour autant être supprimée du système). Comme le montre la figure 3.18, cette opération de reconfiguration per-

met de transférer une instance de composant de type *Nurse* vers une instance de composant de type *EventService* seulement si elle n'a pas une instance *Patient* en occupation `y.patient` $\rightarrow$ `size = 0` et que la nouvelle instance de *EventService* contient moins de cinq instances de type *Nurse* : `w.nurse` $\rightarrow$ `size` $<$ `5` .

3.4.3 Modélisation du protocole de reconfiguration

En se basant sur la notation proposée pour décrire le protocole de reconfiguration, nous décrivons, comme le montre la figure 3.19, un cas possible de l'enchaînement et de l'ordre d'exécution des différentes opérations de reconfiguration.

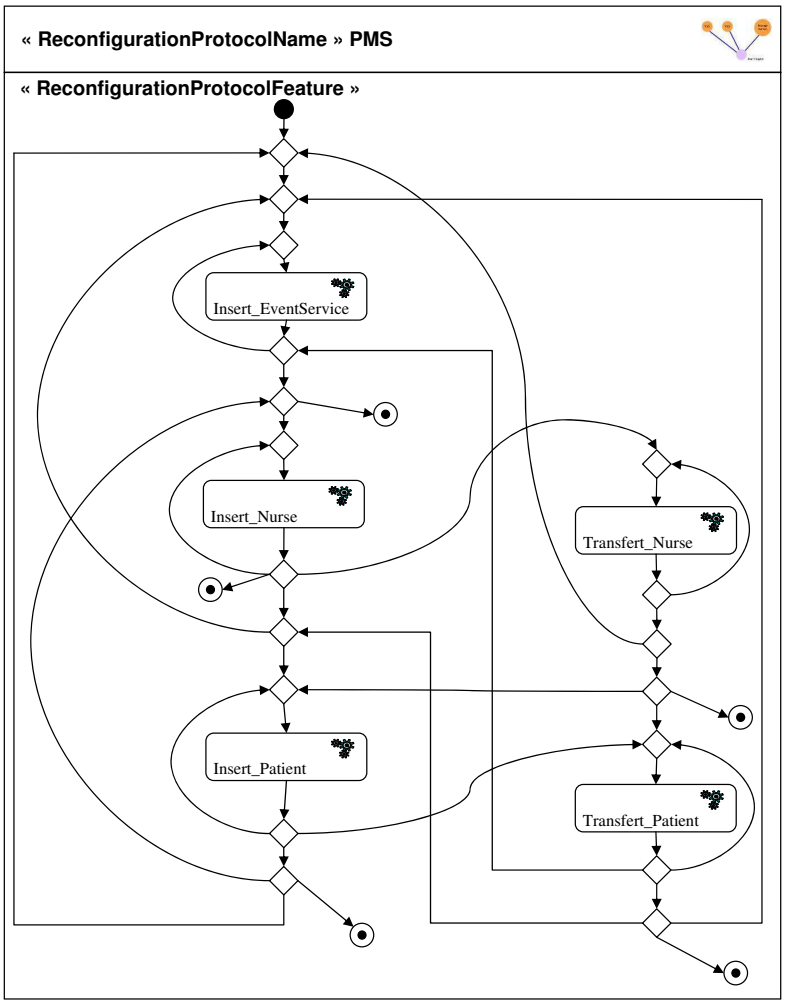

FIG. 3.19 – Le protocole de reconfiguration du système PMS

En effet, et selon l'exemple du PMS, la première opération de reconfiguration à exécuter

est "*Insert_EventService*". L'opération de reconfiguration "*Insert_Patient*" ne peut être exécutée que si l'opération de reconfiguration "*Insert_Nurse*" à été déjà exécutée. L'exécution de l'opération "*Transfert_Patient*" ne peut être exécutée que si l'opération "*Insert_Patient*" à été déjà exécutée. L'exécution de l'opération "*Transfert_Nurse*" est exécutée après son insertion. Ces opérations de reconfiguration peuvent être exécutées une ou plusieurs fois tant que les contraintes OCL mentionnées dans le style architectural et dans les opérations de reconfiguration sont vérifiées.

3.5 Conclusion

Nous avons proposé, dans ce chapitre, une approche de modélisation basée sur un profil UML. Notre approche traite la modélisation de la dynamique de l'architecture logicielle au niveau conceptuel (design time). Nous avons traité essentiellement l'évolution des exigences représentant un changement dans l'activité du système modélisé. Nous avons associé des opérations de reconfiguration architecturales à ces situations pour adapter le système à ces changements.

Le profil, que nous avons proposé, comporte trois méta-modèles. Le premier étend le diagramme de composants et décrit la structure de l'architecture en termes de type de composants et de connexions. Le deuxième étend le premier méta-modèle et décrit la dynamique structurelle de l'architecture en termes d'opérations de reconfiguration. Le troisième méta-modèle étend le diagramme d'activités et permet de décrire l'enchaînement et l'ordre d'exécution des opérations de reconfiguration.

Notre approche a été validée par plusieurs études de cas. Nous avons détaillé dans cette thèse le cas du système logiciel de contrôle de patients PMS (Patient Monitoring System). Les contraintes OCL que nous avons présenté, dans les différents exemples, ont été écrites et validées avec l'outil USE.

Durant la phase de modélisation, le concepteur peut tomber facilement dans l'erreur. L'objectif du chapitre suivant est de proposer une approche de validation basée sur deux parties. La première, appelée intra-validation, permet de vérifier la cohérence entre chaque méta-modèle et son modèle. La deuxième, appelée inter-validation, permet de vérifier le passage d'un modèle vers un autre. Ces deux validations sont utilisées pour faciliter l'identification des éventuelles incohérences et pour détecter toute utilisation incorrecte des concepts de notre profil.

4

Approche de Validation

Vu la complexité et le coût croissant de développement des architectures logicielles, une nouvelle orientation des travaux de recherches consiste à exécuter des tâches de vérification et de validation des modèles conceptuels avant d'avancer profondément dans le processus de développement [24, 11]. En effet, toute erreur dans la modélisation de l'architecture logicielle peut nuire au bon fonctionnement du système. Ainsi, la validation consiste essentiellement à contrôler si la modélisation de l'architecture logicielle est exempte d'erreurs.

Dans notre approche [60, 59], la modélisation de l'architecture est guidée par un profil UML basé sur trois méta-modèles et trois modèles utilisés pour décrire l'architecture logicielle. La question qui se pose est donc : les spécifications générées à partir des modèles sont-elles conformes à leurs méta-modèles ?

Pour répondre à cette question, nous proposons une approche de validation. Cette validation offre deux scénarios. Le premier, appelé *intra-validation*, est basé sur des règles de validation permettant de vérifier la cohérence et la conformité entre le modèle et son méta-modèle. Ces règles sont utilisées pour faciliter l'identification des éventuelles incohérences et pour détecter et corriger des erreurs de spécification. Le second, appelé *inter-validation*, est basé sur des règles de validation inter-modèles permettant d'assister et de guider le passage d'un modèle à un autre. Ces règles permettent de réduire les erreurs et d'assurer la cohérence entre les modèles.

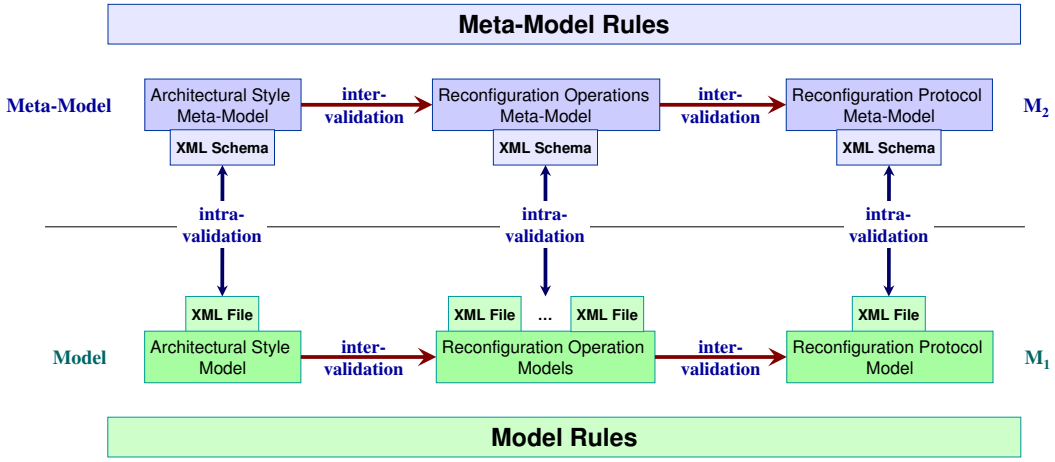

FIG. 4.1 – Approche de validation

4.1 Intra-Validation

L'intra-validation consiste à contrôler la conformité et la cohérence entre le modèle (le niveau M_1) et son méta-modèle correspondant (le niveau M_2). Comme il est décrit dans la figure 4.1, au niveau M_1, nous définissons nos trois modèles : Style Architectural, Opérations de Reconfiguration et Protocol de Reconfiguration. Pour chaque modèle, nous définissons un ensemble de règles (règles selon le modèle). Au niveau M_2, nous définissons notre profil exprimé par les trois méta-modèles. Pour chaque méta-modèle, nous définissons un ensemble de règles (règles selon le méta-modèle). Les modèles sont décrits dans des descriptions XML et les méta-modèles sont décrits dans des descriptions XML schéma (XSD). En se basant sur la technologie XML nous validons les descriptions XML par rapport à leurs XML schémas.

Nous définissons pour chaque modèle et chaque méta-modèle l'ensemble de ses règles. Celles-ci sont exprimées avec le langage XML.

4.1.1 Règles pour le style architectural

Le méta-modèle du style architectural, «*ArchitecturalStyleM-M*» est composé d'un «*ArchitecturalStyleName*», d'une «*ArchitecturalStyleFeature*» et d'un «*Guards*».

```
1   <xs:element name="ArchitecturalStyleM-M">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="ArchitecturalStyleName"/>
5            <xs:element ref="ArchitecturalStyleFeature"/>
6            <xs:element ref="Guards"/>
7         </xs:sequence>
8      </xs:complexType>
9   </xs:element>
```

Un «*ArchitecturalStyleName*» permet d'identifier le style architectural.

```
1   <xs:element name="ArchitecturalStyleName">
2      <xs:complexType>
3         <xs:attribute name="ArchitecturalStyleName" type="xs:ID" use="required"/>
4      </xs:complexType>
5   </xs:element>
```

Une «*ArchitecturalStyleFeature*» est composée de deux ou plusieurs «*Components*» et d'un ou de plusieurs «*Connectors*».

```
1   <xs:element name="ArchitecturalStyleFeature">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="Component" minOccurs="2" maxOccurs="unbounded"/>
5            <xs:element ref="Connector" maxOccurs="unbounded"/>
6         </xs:sequence>
7      </xs:complexType>
8   </xs:element>
```

Un «*Component*» est identifié par un nom unique (ligne 11). Il contient un ou plusieurs «*Ports*».

```
1   <xs:element name="Component">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="NameComponent"/>
5            <xs:element ref="Port" maxOccurs="unbounded"/>
6         </xs:sequence>
7      </xs:complexType>
8   </xs:element>
9   <xs:element name="NameComponent">
10     <xs:complexType>
11        <xs:attribute name="NameComponent" type="xs:ID" use="required"/>
12     </xs:complexType>
13  </xs:element>
```

Un «*Port*» est identifié par un nom unique, au sein du même composant (ligne 11). Il possède une «*Interface*».

```
1   <xs:element name="Port">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="NamePort"/>
5            <xs:element ref="Interface"/>
6         </xs:sequence>
```

```
7        </xs:complexType>
8     </xs:element>
9     <xs:element name="NamePort">
10       <xs:complexType>
11          <xs:attribute name="NamePort" type="xs:ID" use="required"/>
12       </xs:complexType>
13    </xs:element>
```

Une «*Interface*» est identifiée par un nom unique, au sein du même composant (ligne 3). Une «*Interface*» peut être de type `Required` (ligne 14) et/ou de type `Provided` (ligne 15).

```
1     <xs:element name="NameInterface">
2        <xs:complexType>
3           <xs:attribute name="NameInterface" type="xs:ID" use="required"/>
4        </xs:complexType>
5     </xs:element>
6     <xs:element name="Interface">
7        <xs:complexType>
8           <xs:sequence>
9              <xs:element ref="NameInterface"/>
10          </xs:sequence>
11          <xs:attribute name="Type" use="required">
12             <xs:simpleType>
13                <xs:restriction base="xs:NMTOKEN">
14                   <xs:enumeration value="Required"/>
15                   <xs:enumeration value="Provided"/>
16                </xs:restriction>
17             </xs:simpleType>
18          </xs:attribute>
19       </xs:complexType>
20    </xs:element>
```

Un «*Connector*» est identifié par un nom unique (ligne 3). Il peut être de type `Assembly` (ligne 13) ou de type `Delegation` (ligne 14).

```
1     <xs:element name=”NameConnector”>
2        <xs:complexType>
3           <xs:attribute name=”NameConnector” type=”xs:ID” use=”required”/>
4        </xs:complexType>
5     </xs:element><xs:element name=”Connector”>
6        <xs:complexType>
7           <xs:sequence>
8              <xs:element ref=”NameConnector”/>
9           </xs:sequence>
10          <xs:attribute name=”Type” use=”required”>
11             <xs:simpleType>
12                <xs:restriction base=”xs:NMTOKEN”>
13                   <xs:enumeration value=”Assembly”/>
14                   <xs:enumeration value=”Delegation”/>
15                </xs:restriction>
16             </xs:simpleType>
17          </xs:attribute>
18       </xs:complexType>
19    </xs:element>
```

Un «*Guards*» est composé par un ou plusieurs contraintes. Ces contraintes seront exprimées avec le langage OCL lors de la modélisation.

```
<xs:element name="Guards">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="Constraint" type="xs:string" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="Constraint" type="xs:string"/>
```

4.1.2 Règles pour l'opération de reconfiguration

Le méta-modèle «*ReconfigurationOperationM-M*» est composé d'un «*ReconfigurationOperationName*», d'une «*ReconfigurationOperationFeature*» et d'un «*Guards*».

```
<xs:element name="ReconfigurationOperationM−M">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="ReconfigurationOperationName"/>
         <xs:element ref="ReconfigurationOperationFeature"/>
         <xs:element ref="Guards"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Un «*ReconfigurationOperationName*» permet d'identifier une opération de reconfiguration.

```
<xs:element name="ReconfigurationOperationName">
   <xs:complexType>
      <xs:attribute name="Name" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
```

Une «*ReconfigurationOperationFeature*» est composée des trois parties «*Require*&*Delete*», «*Require*&*Preserve*» et «*Insert*».

```
<xs:element name="ReconfigurationOperationFeature">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="Require_Delete"/>
         <xs:element ref="Require_Preserve"/>
         <xs:element ref="Insert"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Chacune des trois parties «*Require*&*Delete*», «*Require*&*Preserve*» et «*Insert*» est composée de zéro ou plusieurs instances de «*Components*» (ligne 4) et de zéro ou

plusieurs instances de «*Connectors*» (ligne 5).

```
1  <xs:element name="Require_Delete">
2     <xs:complexType>
3        <xs:sequence>
4           <xs:element ref="Component" minOccurs="0" maxOccurs="unbounded"/>
5           <xs:element ref="Connector" minOccurs="0" maxOccurs="unbounded"/>
6        </xs:sequence>
7     </xs:complexType>
8  </xs:element>
9  ...
```

Une instance de composant est identifiée par un nom unique (ligne 11). Elle contient un ou plusieurs «*Ports*». Un port est identifié par un nom unique au sein du même composant (ligne 22) et il possède une «*Interface*». Egalement, une interface est identifiée par un nom unique, au sein du même port. Une interface peut être de type `Required` (ligne 27) ou de type `Provided` (ligne 28).

```
1  <xs:element name="Component">
2     <xs:complexType>
3        <xs:sequence>
4           <xs:element ref="NameComponent"/>
5           <xs:element ref="Port" maxOccurs="unbounded"/>
6        </xs:sequence>
7     </xs:complexType>
8  </xs:element>
9  <xs:element name="NameComponent">
10    <xs:simpleType>
11       <xs:restriction base="xs:ID"/>
12    </xs:simpleType>
13 </xs:element>
14 <xs:element name="Port">
15    <xs:complexType>
16       <xs:sequence>
17          <xs:element ref="NamePort"/>
18          <xs:element ref="Interface"/>
19       </xs:sequence>
20    </xs:complexType>
21 </xs:element>
22 <xs:element name="NamePort" type="xs:ID"/>
23 ...
24 <xs:element name="TypeInterface">
25    <xs:simpleType>
26       <xs:restriction base="xs:string">
27          <xs:enumeration value="Required"/>
28          <xs:enumeration value="Provided"/>
29       </xs:restriction>
30    </xs:simpleType>
31 </xs:element>
```

Un «*Connector*» est identifié par un nom unique (ligne 9). Il peut être de type `Assembly` (ligne 13) ou de type `Delegation` (ligne 14).

```
1  <xs:element name="Connector">
2     <xs:complexType>
3        <xs:sequence>
4           <xs:element ref="NameConnector"/>
5           <xs:element ref="TypeConnector"/>
6        </xs:sequence>
```

```
7        </xs:complexType>
8     </xs:element>
9     <xs:element name="NameConnector" type="xs:ID"/>
10    <xs:element name="TypeConnector">
11       <xs:simpleType>
12          <xs:restriction base="xs:string">
13             <xs:enumeration value="Assembly"/>
14             <xs:enumeration value="Delegation"/>
15          </xs:restriction>
16       </xs:simpleType>
17    </xs:element>
```

Un «*Guards*» est composé par un ou plusieurs contraintes. Ces contraintes seront exprimées par le langage OCL.

```
1     <xs:element name="Guards">
2        <xs:complexType>
3           <xs:sequence>
4              <xs:element name="Constraint" type="xs:string" maxOccurs="unbounded"/>
5           </xs:sequence>
6        </xs:complexType>
7     </xs:element>
8     <xs:element name="Constraint" type="xs:string"/>
```

4.1.3 Règles pour le protocole de reconfiguration

Le méta-modèle «*ReconfigurationProtocolM-M*» est composé d'un «*ReconfigurationProtocolName*» et d'une «*ReconfigurationProtocolFeature*».

```
1    <xs:element name="ReconfigurationProtocolM−M">
2        <xs:complexType>
3           <xs:sequence>
4              <xs:element ref="ReconfigurationProtocolName"/>
5              <xs:element ref="ReconfigurationProtocolFeature"/>
6           </xs:sequence>
7        </xs:complexType>
8     </xs:element>
```

Un «*ReconfigurationProtocolName*» permet d'identifier le protocole de reconfiguration.

```
1     <xs:element name="ReconfigurationProtocolName">
2        <xs:complexType>
3           <xs:attribute name="ReconfigurationProtocolName" type="xs:ID" use="required"/>
4        </xs:complexType>
5     </xs:element>
```

Une «*ReconfigurationProtocolFeature*» est composée d'un ou de plusieurs «*ReconfigurationOperationNode*» et d'un ou de plusieurs «*ControlNode*».

```
1     <xs:element name="Collection">
2        <xs:complexType>
3           <xs:sequence>
4              <xs:element ref="ReconfigurationOperationNode" maxOccurs="unbounded"/>
```

```
5            <xs:element ref="ControlNode" maxOccurs="unbounded"/>
6         </xs:sequence>
7      </xs:complexType>
8   </xs:element>
```

Un «*ReconfigurationOperationNode*» est identifié par un nom unique (ligne 11) et composé d'un ou de plusieurs «*Link*».

```
1   <xs:element name="ReconfigurationOperationNode">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="ReconfigurationOpertionName"/>
5            <xs:element ref="Link" maxOccurs="unbounded"/>
6         </xs:sequence>
7      </xs:complexType>
8   </xs:element>
9   <xs:element name="ReconfigurationOpertionName">
10     <xs:complexType>
11        <xs:attribute name="ReconfigurationOpertionName" type="xs:ID" use="required"/>
12     </xs:complexType>
13  </xs:element>
```

Un «*ControlNode*» possède un nom «*ControlNodeName*», un type «*ControlNodeType*» et un ou plusieurs «*Link*». Un «*ControlNodeType*» peut avoir un type parmi ces quatre (de la ligne 24 jusqu'à la ligne 27) : "`InitialNode`", "`DecisionNode`", "`SynchronousNode`" et "`FinalNode`".

```
1   <xs:element name="ControlNode">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element ref="ControlNodeName"/>
5            <xs:element ref="ControlNodeType"/>
6            <xs:element ref="Link" maxOccurs="unbounded"/>
7         </xs:sequence>
8      </xs:complexType>
9   </xs:element>
10  <xs:element name="ControlNodeName">
11     <xs:complexType>
12        <xs:attribute name="ControlNodeName" use="required">
13           <xs:simpleType>
14              <xs:restriction base="xs:string"/>
15           </xs:simpleType>
16        </xs:attribute>
17     </xs:complexType>
18  </xs:element>
19  <xs:element name="ControlNodeType">
20     <xs:complexType>
21        <xs:attribute name="ControlNodeType" use="required">
22           <xs:simpleType>
23              <xs:restriction base="xs:string">
24                 <xs:enumeration value="InitialNode"/>
25                 <xs:enumeration value="DecisionNode"/>
26                 <xs:enumeration value="SynchronousNode"/>
27                 <xs:enumeration value="FinalNode"/>
28              </xs:restriction>
29           </xs:simpleType>
30        </xs:attribute>
31     </xs:complexType>
32  </xs:element>
```

Un «*Link*» possède un type «*Type*» et une désignation «*Designation*». Un «*Link*» peut avoir un type parmi ces quatre (de la ligne 23 jusqu'à la ligne 26) : "`LinkToFinalNode`", "`LinkToDecisionNode`", "`LinkToOperationNode`" et "`LinkToSynchronousNode`".

```
1  <xs:element name="Link">
2     <xs:complexType>
3        <xs:sequence>
4           <xs:element ref="Type"/>
5           <xs:element ref="Designation"/>
6        </xs:sequence>
7     </xs:complexType>
8  </xs:element>
9  <xs:element name="Designation">
10    <xs:complexType>
11       <xs:attribute name="Designation" use="required">
12          <xs:simpleType>
13             <xs:restriction base="xs:string"/>
14          </xs:simpleType>
15       </xs:attribute>
16    </xs:complexType>
17 </xs:element>
18 <xs:element name="Type">
19    <xs:complexType>
20       <xs:attribute name="Type" use="required">
21          <xs:simpleType>
22             <xs:restriction base="xs:string">
23                <xs:enumeration value="LinkToFinalNode"/>
24                <xs:enumeration value="LinkToDecisionNode"/>
25                <xs:enumeration value="LinkToOperationNode"/>
26                <xs:enumeration value="LinkSynchronousNode"/>
27             </xs:restriction>
28          </xs:simpleType>
29       </xs:attribute>
30    </xs:complexType>
31 </xs:element>
```

4.1.4 Validation

Pour l'intra-validation, nous avons utilisé, comme illustré dans la figure 4.1, l'approche de validation proposée par MOF (Meta-Object Facility). Cette approche est utilisée afin de valider les modèles par rapport à leurs méta-modèles. Le niveau M_2 est composé de trois méta-modèles définissant la structure de chaque modèle. Le niveau M_1 est composé par les trois modèles résultants. Dans le MOF, la validation d'un modèle se fait avec le modèle du niveau supérieur. Le niveau M_1 est donc validé par le niveau M_2.

Les règles que nous avons définies sont modélisées et implémentées dans des descriptions XML schémas (niveau M_2). La modélisation de l'évolution d'un système est réalisé avec notre profil et les modèles résultants sont automatiquement transformés vers des descriptions XML (niveau M_1). La validation des descriptions XML se fait à l'aide de la description XML schéma correspondante en utilisant des outils XML tel que XMLSpy de Altova.

La validation consiste à contrôler que la description XML est bien formée et quelle est une

instance du son XML Schéma. La figure 4.2 représente la validation du style architectural.

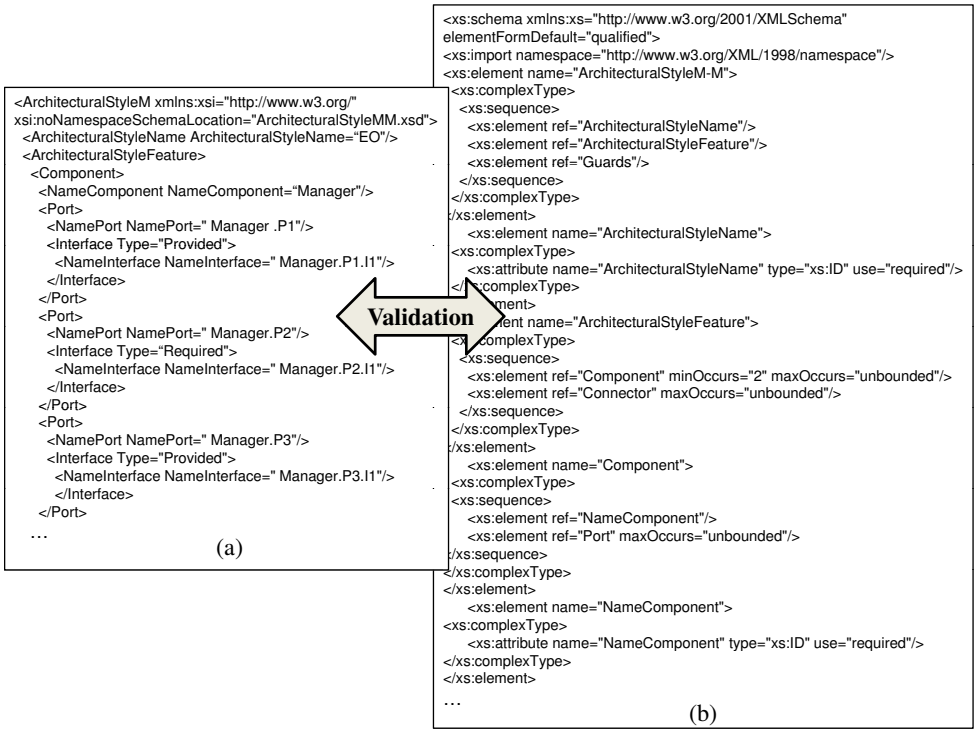

FIG. 4.2 – Validation de la description XML (a) par rapport à son XML Schéma (b)

4.2 Inter-Validation

De même, nous définissons des règles de validation inter-modèles. Ces règles, comme l'illustre la figure 4.1, permettent de guider et de valider le passage d'un modèle vers un autre.

Nous définissons deux types de validation. Le premier permet de guider le passage du style architectural vers la modélisation des opérations de reconfiguration. Le deuxième permet de guider le passage des opérations de reconfiguration vers la modélisation du protocole de reconfiguration. Ces règles sont implémentées avec le langage Java.

4.2.1 Style architectural vers operation de reconfiguration

Les règles suivantes permettent de guider le passage du style architectural vers la modélisation des opérations de reconfiguration.

– Chaque instance *Component* dans la partie *ReconfigurationOperationFeature* lui correspond un *Component* dans la partie *ArchitecturalStyleFeature*.
– Chaque instance *Connector* dans la partie *ReconfigurationOperationFeature* lui correspond un *Connector* dans la partie *ArchitecturalStyleFeature*.
– Chaque instance *Port* dans la partie *ReconfigurationOperationFeature* lui correspond un *Port* dans la partie *ArchitecturalStyleFeature*.
– Chaque instance *Interface* dans la partie *ReconfigurationOperationFeature* lui correspond une *Interface* dans la partie *ArchitecturalStyleFeature*.

4.2.2 Operation de reconfiguration vers protocole de reconfiguration

Les règles suivantes permettent de guider le passage des opérations de reconfiguration vers la modélisation du protocole de reconfiguration.

– Chaque *ReconfigurationOperationNode* dans la partie *ReconfigurationProtocolFeature* lui correspond une *ReconfigurationOperation* dans la partie *ReconfigurationOperationFeature* après élimination des noms des paramètres.
– Chaque *ReconfigurationControlNode* dans la partie *ReconfigurationProtocolFeature* doit regrouper une ou plusieurs contraintes OCL de la partie *Guards* de *ReconfigurationOperationFeature*.

4.3 Conclusion

Nous avons proposé dans ce chapitre une approche de validation à base de règles. Ces règles permettent d'identifier les éventuelles incohérences et de détecter toute utilisation incorrecte du profil proposé. La validation proposée permet de contrôler que les modèles générés, représentés dans des descriptions XML, sont conformes à leurs méta-modèles, représentés dans des descriptions XML schéma. La validation que nous avons proposée est structurelle et à base du langage XML. Cependant, cette approche souffre de quelques limites. Elle ne permet pas de vérifier si le style architectural est consistant ou non et elle ne permet pas de vérifier si l'exécution d'une opération de reconfiguration préserve le style architectural déjà défini. Dans le chapitre suivant, nous proposons une approche de vérification qui apporte des solutions à ces limites.

5

Approche de Vérification

Le profil que nous avons proposé est basé essentiellement sur le langage semi-formel UML. Les modèles générés à partir du profil peuvent être ambigus et peu précis. Cette génération informelle est un frein à la réutilisation et plus largement à la qualité de l'architecture logicielle produite du point de vue de sa correction ou encore de son évolution. Ceci est dû à l'absence d'une sémantique formelle précise pour UML, qui n'offre pas des outils rigoureux de vérification et de preuve. Toute erreur ou mauvaise conception de l'architecture logicielle d'une application peut causer des problèmes graves qui peuvent avoir de mauvaises répercussions. Ainsi, assurer une fiabilité et une cohérence de l'architecture reste un objectif ambitieux du génie logiciel. Ceci permet d'éviter des erreurs potentielles dans le fonctionnement de l'application.

Afin de pallier à ces inconvénients, nous faisons recours aux techniques formelles pour analyser et vérifier l'architecture logicielle. En effet, les techniques formelles ont été élaborées afin d'assurer un certain niveau de précision et de cohérence. Nous adoptons une approche basée sur la transformation des modèles semi-formels vers des spécifications formelles. L'intérêt majeur de cette approche est de surmonter le manque de précision du langage UML.

L'approche de vérification, décrite dans la figure 5.1, est composée de deux parties. Nous proposons, dans la première partie, une approche assurant une transformation automatique du style architectural et de chaque opération de reconfiguration vers le langage formel *Z* [133, 141]. Nous adoptons, dans la deuxième partie, une approche de vérification. Cette approche est développée, dans notre unité de recherche ReDCAD, dans le cadre des travaux de la thèse de Me. Imen LOULOU [84]. Nous vérifions la consistance du style architectural et la conformité de l'évolution d'une architecture par rapport à son style architectural. Nous utilisons le système de preuve Z/EVES [95] pour vérifier ces deux

propriétés.

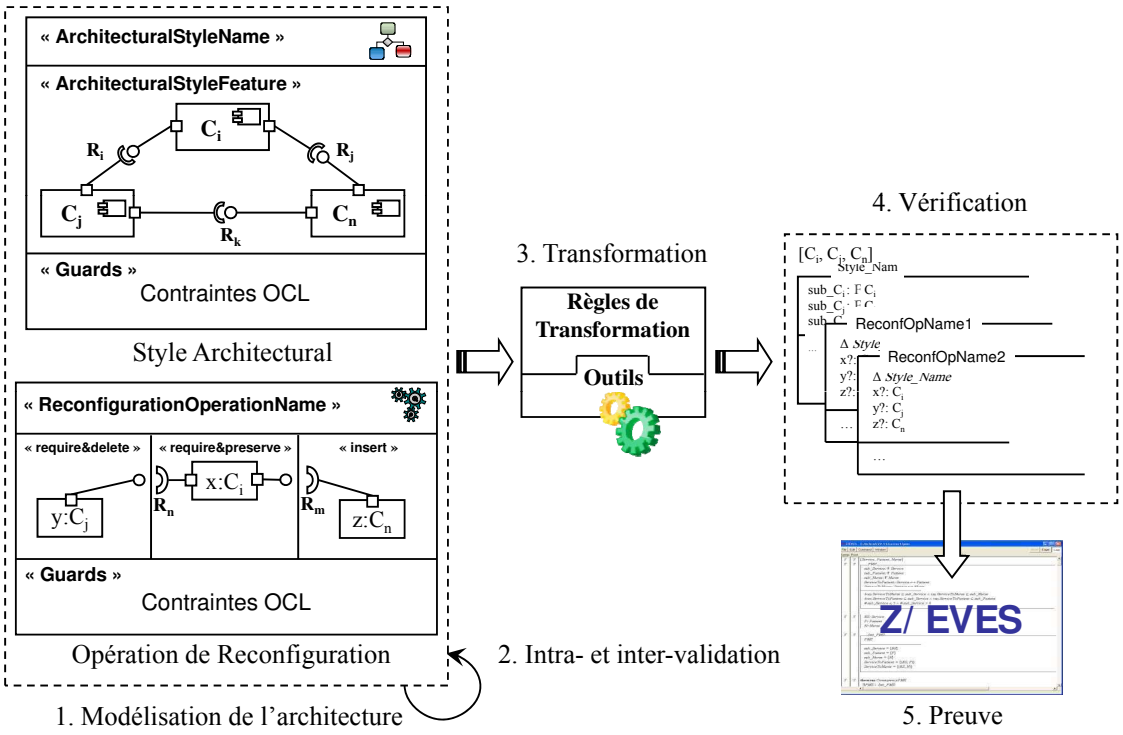

FIG. 5.1 – Approche de transformation et de vérification

5.1 Transformation vers *Z*

La transformation de la notation UML vers un langage formel n'est pas nouvelle. Cependant, il y a peu de travaux qui ont cherché à vérifier la conformité du style architectural après l'exécution d'une opération de reconfiguration. Ainsi, pour combler cette lacune et pour mener des raisonnements rigoureux sur l'architecture, nous proposons une approche qui transforme d'une façon automatique le style architectural et chaque opération de reconfiguration vers des spécifications formelles en *Z*. Les spécifications résultantes sont utilisées afin de mener des raisonnements et des preuves formelles.

Cette transformation est réalisée en deux étapes et basée sur des règles de transformation. La première étape est basée sur des règles permettant la transformation des notations graphiques du style architectural et de chaque opération de reconfiguration vers la notation *Z*. La deuxième étape est basée sur des règles permettant la transformation des contraintes architecturales exprimées avec le langage OCL vers la notation *Z*.

L'approche de transformation que nous proposons est inspirée des travaux de recherche de [50, 1, 2]. En effet, la notation *Z* se présente comme un formalisme pouvant décrire les

composants du système et les interconnexions entre eux ainsi que les propriétés architecturales. Un style architectural est décrit par le schéma *Z* suivant :

Nom_Style

Type de composants
Type de connexions

---

proprietes architecturales

Une architecture logicielle peut évoluer en modifiant la structure de son architecture. La configuration résultante représente une instance du style architectural présentée par un schéma d'opération *Z* exprimant les pré- et les post-conditions.

ReconfigurationOperation

Δ*Nom_Style*
$Par_1?, Par_2?, ..., Par_n?$

Pre_Conditions
Post_Conditions

Avec :

- Δ *Nom_Style* indique que l'opération de reconfiguration peut changer l'état du système.
- Par_1 ?, Par_2 ?, ..., Par_n ? représentent les paramètres d'entrée de l'opération de reconfiguration.
- Les pré-conditions et les post-conditions traduisent des conditions exprimées textuellement et qui doivent être évaluées en vrai pour que la production ait lieu. Ces conditions définissent des contraintes sur les valeurs des attributs, le nombre d'instances d'un composant, etc.

5.1.1 Transformation de la partie graphique vers *Z*

Dans ce qui suit, nous introduisons les règles de transformation du style architectural et des opérations de reconfiguration vers la notation *Z*.

5.1.1.1 Style architectural

La transformation du style architectural vers la notation *Z* cherche à préserver les types de composants, les types de connexions et les contraintes architecturales. Cette transformation, comme le montre la figure 5.2, obéit à un ensemble de règles.

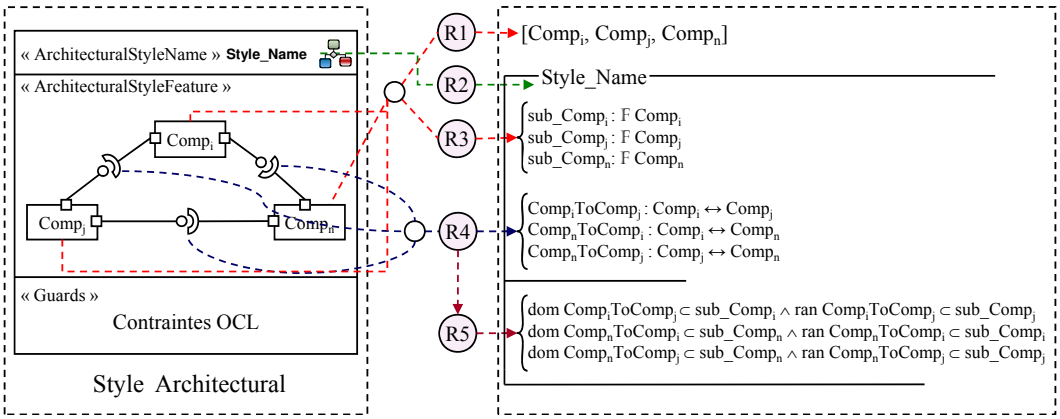

FIG. 5.2 – Transformation du style architectural vers la notation *Z*

■ Règle1 : Définition des types basiques de composants

Les types basiques de composants sont déterminés à partir des noms des types des composants figurant dans la partie « ArchitecturalStyleFeature ».

$$[Comp_i, Comp_j, ..., Comp_n]$$

■ Règle2 : Définition du nom du schéma *Z*

Cette règle génère le squelette du schéma *Z*. Le nom du schéma est déterminé à partir du nom du style donné par la partie « ArchitecturalStyleName ».

Style_Name

■ Règle 3 : Définition des ensembles de composants

Chaque type de composant présenté dans la partie « ArchitecturalStyleFeature » est traduit en un ensemble fini $\mathbb{F}$ de composants. Le type de chaque ensemble de composant généré correspond à un type basic déterminé à partir de la règle 1. Le nom de chaque ensemble de composant est pré-fixé par "*sub_*" suivi par le nom de type de composant. Ces ensembles sont déclarés dans la partie déclarative du schéma *Z*.

Style_Name

$sub\_Comp_i : \mathbb{F}\ Comp_i$

$sub\_Comp_j : \mathbb{F}\ Comp_j$

$sub\_Comp_n : \mathbb{F}\ Comp_n$

■ Règle 4 : Définition des relations

Chaque connexion présentée dans la partie « ArchitecturalStyleFeature » est transformée en une relation, $Comp_iToComp_j : Comp_i \leftrightarrow Comp_j$, dans la partie déclarative du schéma *Z*. La source de la relation, $Comp_i$, porte le nom du type de composant ayant l'interface requise et la cible de la relation, $Comp_j$, porte le nom du type de composant ayant l'interface fournie. Le nom de la relation est la clause "To" suffixée par la source et préfixée par la cible.

Style_Name

$sub\_Comp_i : \mathbb{F}\ Comp_i$
$sub\_Comp_j : \mathbb{F}\ Comp_j$
$sub\_Comp_n : \mathbb{F}\ Comp_n$
$Comp_iToComp_j : Comp_i \leftrightarrow Comp_j$
$Comp_nToComp_i : Comp_n \leftrightarrow Comp_i$
$Comp_nToComp_j : Comp_n \leftrightarrow Comp_j$

■ Règle 5 : Définition des contraintes sur les relations

Pour exprimer des contraintes sur les relations, le langage *Z* offre les deux fonctions `dom` (domaine) et `ran` (image). `Dom` représente l'ensemble de départ et `ran` représente l'ensemble d'arrivée. Le domaine d'une relation `R : T ↔ U` est l'ensemble de tous les éléments dans `T` qui sont liés au moins à un élément dans `U`. L'image de la relation `R` est l'ensemble de tous les éléments dans `U` liés au moins à un élément dans `T` [141].

Chaque relation a un domaine et une image [141]. En se basant sur la troisième et la quatrième règle, cette règle définit, dans la partie prédicative du schéma *Z*, les propriétés d'invariance sur l'ensemble de départ (domaine) et l'ensemble d'arrivée (image) de chaque relation. Chaque propriété exprime l'inclusion du domaine, respectivement de l'image, de la relation dans le sous-ensemble correspondant déjà défini dans la partie déclarative.

Style_Name

$sub\_Comp_i : \mathbb{F}\ Comp_i$
$sub\_Comp_j : \mathbb{F}\ Comp_j$
$sub\_Comp_n : \mathbb{F}\ Comp_n$
$Comp_iToComp_j : Comp_i \leftrightarrow Comp_j$
$Comp_nToComp_i : Comp_n \leftrightarrow Comp_i$
$Comp_nToComp_j : Comp_n \leftrightarrow Comp_j$

$dom\ Comp_iToComp_j \subset sub\_Comp_i \land ran\ Comp_iToComp_j \subset sub\_Comp_j$
$dom\ Comp_nToComp_i \subset sub\_Comp_n \land ran\ Comp_nToComp_i \subset sub\_Comp_i$
$dom\ Comp_nToComp_j \subset sub\_Comp_n \land ran\ Comp_nToComp_j \subset sub\_Comp_j$

Afin de mieux illustrer l'approche de transformation, nous revenons à notre exemple et nous traduisons la partie graphique du style architectural du système PMS vers la notation

Z. L'application des règles de transformation de [R1] jusqu'à [R5] donne le schéma *Z* suivant.

$$[EventService, Patient, Nurse] \quad \text{[R1]}$$

PMS **[R2]**

$$sub\_EventService : \mathbb{F}\, EventService \quad \text{[R3]}$$
$$sub\_Patient : \mathbb{F}\, Patient$$
$$sub\_Nurse : \mathbb{F}\, Nurse$$
$$EventServiceToPatient : EventService \leftrightarrow Patient \quad \text{[R4]}$$
$$EventServiceToNurse : EventService \leftrightarrow Nurse$$
$$PatientToEventService : Patient \leftrightarrow EventService$$
$$NurseToEventService : Nurse \leftrightarrow EventService$$
$$PatientToNurse : Patient \leftrightarrow Nurse$$
$$NurseToPatient : Nurse \leftrightarrow Patient$$

$$\mathrm{dom}\, EventServiceToPatient \subseteq sub\_EventService \quad \text{[R5]}$$
$$\wedge\ \mathrm{ran}\, EventServiceToPatient \subseteq sub\_Patient$$
$$\mathrm{dom}\, EventServiceToNurse \subseteq sub\_EventService$$
$$\wedge\ \mathrm{ran}\, EventServiceToNurse \subseteq sub\_Nurse$$
$$\mathrm{dom}\, PatientToEventService \subseteq sub\_Patient$$
$$\wedge\ \mathrm{ran}\, PatientToEventService \subseteq sub\_EventService$$
$$\mathrm{dom}\, NurseToEventService \subseteq sub\_Nurse$$
$$\wedge\ \mathrm{ran}\, NurseToEventService \subseteq sub\_EventService$$
$$\mathrm{dom}\, PatientToNurse \subseteq sub\_Patient$$
$$\wedge\ \mathrm{ran}\, PatientToNurse \subseteq sub\_Nurse$$
$$\mathrm{dom}\, NurseToPatient \subseteq sub\_Nurse$$
$$\wedge\ \mathrm{ran}\, NurseToPatient \subseteq sub\_Patient$$

FIG. 5.3 – Transformation du style architectural vers un schéma *Z*

5.1.1.2 Opération de reconfiguration

Nous décrivons les étapes de transformation d'une opération de reconfiguration vers la notation *Z*. Cette transformation, comme le montre la figure 5.4, est basée sur un ensemble de règles de [R6] jusqu'à [R17] permettant de générer automatiquement un schéma d'opération *Z*.

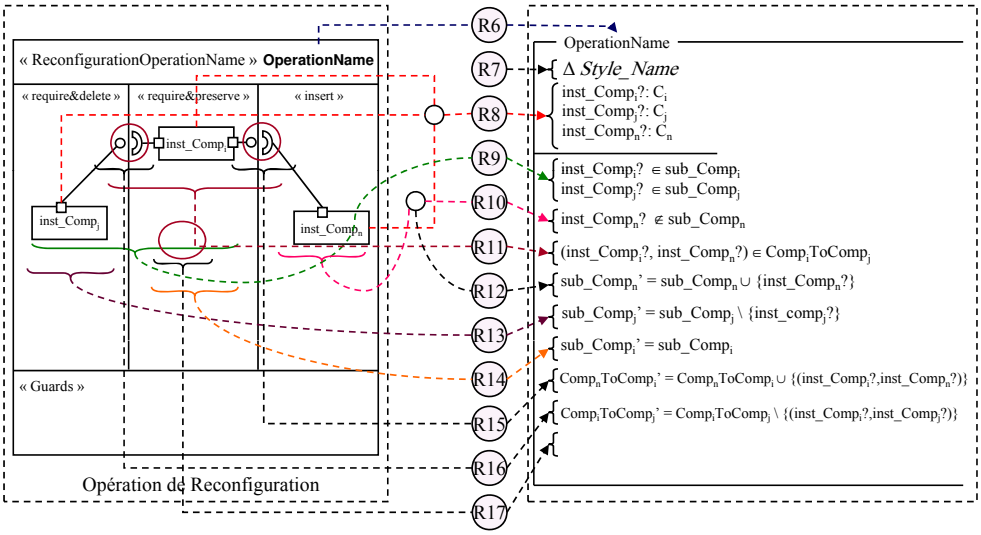

FIG. 5.4 – Transformation d'une opération de reconfiguration vers un schéma d'opération *Z*

■ Règle 6 : Définition du nom de l'opération de reconfiguration

Cette règle génère le squelette du schéma *Z*. Le nom du schéma est déterminé à partir de « ReconfigurationOperationName ».

ReconfigurationOperationName

■ Règle 7 : Définition du changement de l'état de l'architecture

Une architecture logicielle peut évoluer après l'exécution d'une opération de reconfiguration. Cette évolution est exprimée en *Z* par le caractère Δ. Le changement de l'état de l'architecture est présenté alors, dans la partie déclarative du schéma d'opération *Z*, par Δ suivi par le nom du style donné par « ArchitecturalStyleName ».

ReconfigurationOperationName

Δ*Style_Name*

■ Règle 8 : Définition des paramètres d'entrée

Cette règle définit les instances de composants présentées dans la partie « Require & Delete », « Require & Preserve » et « Insert » en tant que paramètres d'entrée pour le schéma d'opération *Z*.

Chaque instance de composant est transformée, dans la partie déclarative du schéma d'opération *Z*, vers le nom de l'instance de composant suffixé par " ?". Le type de chaque composant est le type abstrait associé au composant qui lui correspond.

ReconfigurationOperationName

Δ*Style_Name*
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

■ Règle 9 : Définition des pré-conditions sur l'appartenance des composants

Cette règle permet d'ajouter des pré-conditions exprimant l'appartenance des instances de composants de la partie « Require & Delete » et « Require & Preserve » à l'ensemble des composants de l'architecture. Ces pré-conditions sont ajoutées dans la partie prédicative du schéma d'opération *Z* sous la forme "$inst\_Comp \in sub\_Comp$". Avec *inst_Comp* désigne une instance de composant et *sub_Comp* désigne l'ensemble de type de composant.

ReconfigurationOperationName

Δ*Style_Name*
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$

■ Règle 10 : Définition des pré-conditions sur la restriction des composants

Cette règle permet d'ajouter des pré-conditions exprimant la non appartenance des instances de composants de la partie « Insert » à l'ensemble des composants de l'architecture. Ces pré-conditions sont ajoutées dans la partie prédicative du schéma d'opération *Z* sous la forme "$inst\_Comp \notin sub\_Comp$".

ReconfigurationOperationName

Δ*Style_Name*
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$

■ Règle 11 : Définition des pré-conditions sur les relations

Cette règle permet de transformer les connexions existantes entre la partie « Require & Delete » et « Require & Preserve » vers des relations exprimées avec la notation *Z*. Ces relations sont représentées en *Z* sous la forme suivante : "$(inst\_Comp_i, inst\_Comp_j) \in Comp_iToComp_j$". Avec $(inst\_Comp_i, inst\_Comp_j)$ est la relation entre les composants $inst\_Comp_i$ et $inst\_Comp_j$ et $Comp_iToComp_j$ définit l'ensemble des relations reliant les composants de type $Comp_i$ et $Comp_j$.

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$

■ Règle 12 : Définition des post-conditions liées à l'insertion des composants

L'insertion d'un composant en *Z* est traduite par la règle suivante : "$sub\_Comp' = sub\_Comp \cup inst\_Comp$". Avec sub_Comp' décrit l'ensemble des composants résultants après l'exécution de l'opération de reconfiguration. *sub_Comp* représente l'ensemble des composants avant l'exécution de l'opération de reconfiguration et *inst_Comp* désigne une instance de composant à insérer dans le système.

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$
$sub\_Comp'_n = sub\_Comp_n \cup \{inst\_Comp_n?\}$

■ Règle 13 : Définition des post-conditions liées à la suppression des composants

La suppression d'un composant en *Z* est traduite par la règle suivante : "$sub\_Comp' = sub\_Comp \setminus inst\_Comp$" où *inst_Comp* désigne une instance de composant à supprimer.

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$
$sub\_Comp'_n = sub\_Comp_n \cup \{inst\_Comp_n?\}$
$sub\_Comp'_j = sub\_Comp_j \setminus \{inst\_Comp_j?\}$

■ Règle 14 : Définition des post-conditions liées au non changement des composants

Si un ensemble de composants, figurant dans l'architecture n'a subi aucun changement après l'exécution de l'opération de reconfiguration, alors ces composants seront exprimés par la règle suivante : "$sub\_Comp' = sub\_Comp$".

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$
$sub\_Comp'_n = sub\_Comp_n \cup \{inst\_Comp_n?\}$
$sub\_Comp'_j = sub\_Comp_j \setminus \{inst\_Comp_j?\}$
$sub\_Comp'_i = sub\_Comp_i$

■ Règle 15 : Définition des post-conditions liées à l'insertion des connexions

L'insertion d'une connexion en Z est traduite par la règle suivante : "$Comp_iToComp'_j = Comp_iToComp_j \cup (inst\_Comp_i, inst\_Comp_j)$". Avec $Comp_iToComp'_j$ décrit la relation qui définit l'ensemble des connexions reliant les composants de type $Comp_i$ et $Comp_j$ et $(inst\_Comp_i, inst\_Comp_j)$ décrit la relation à insérer entre l'instance du composant $inst\_Comp_i$ et $inst\_Comp_j$.

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$
$sub\_Comp'_n = sub\_Comp_n \cup \{inst\_Comp_n?\}$
$sub\_Comp'_j = sub\_Comp_j \setminus \{inst\_Comp_j?\}$
$sub\_Comp'_i = sub\_Comp_i$
$Comp_nToComp'_i = Comp_nToComp_i \cup \{(inst\_Comp_n?, inst\_Comp_i?)\}$

■ Règle 16 : Définition des post-conditions relatives à la suppression des connexions

La suppression d'une relation en *Z* est traduite par la règle suivante : "$Comp_iToComp'_j = Comp_iToComp_j \setminus (inst\_Comp_i, inst\_Comp_j)$". Avec $(inst\_Comp_i, inst\_Comp_j)$ décrit la relation à supprimer suite à une opération de suppression d'une connexion. $inst\_Comp_i$ et $inst\_Comp_j$ représentent les instances des composants que nous désirons déconnecter.

ReconfigurationOperationName

$\Delta Style\_Name$
$inst\_Comp_i? : type\_Comp_i$
$inst\_Comp_j? : type\_Comp_j$
$inst\_Comp_n? : type\_Comp_n$

$inst\_Comp_i? \in sub\_Comp_i$
$inst\_Comp_j? \in sub\_Comp_j$
$inst\_Comp_n? \notin sub\_Comp_n$
$(inst\_Comp_i?, inst\_Comp_j?) \in Comp_iToComp_j$
$sub\_Comp'_n = sub\_Comp_n \cup \{inst\_Comp_n?\}$
$sub\_Comp'_j = sub\_Comp_j \setminus \{inst\_Comp_j?\}$
$sub\_Comp'_i = sub\_Comp_i$
$Comp_nToComp'_i = Comp_nToComp_i \cup \{(inst\_Comp_n?, inst\_Comp_i?)\}$
$Comp_iToComp'_j = Comp_iToComp_j \setminus \{(inst\_Comp_i?, inst\_Comp_j?)\}$

■ Règle 17 : Définition des post-conditions relatives au non changement des connexions

Si une connexion, figurant dans l'architecture, n'a subi aucun changement au cours de l'opération de reconfiguration, alors nous exprimons ce cas par la règle suivante : "$Comp_iToComp'_j = Comp_iToComp_j$". Ce cas n'est pas traité dans l'exemple présenté dans la figure 5.4. Nous n'avons pas des connexions qui n'ont pas subi des modifications.

Nous revenons à l'exemple du PMS et nous traduisons la partie graphique de l'opération de reconfiguration *Inser_Patient* vers la notation *Z*. L'application des règles de transformation successivement de [R6] jusqu'à [R17] donne le schéma d'opération *Z* présenté dans la figure 5.5 suivante.

Insert_Patient **[R6]**

ΔPMS	[R7]
$x? : EventService$	[R8]
$y? : Nurse$	
$z? : Patient$	
$x? \in sub\_EventService$	[R9]
$y? \in sub\_Nurse$	
$z? \notin sub\_Patient$	[R10]
$(x?, y?) \in EventServiceToNurse$	[R11]
$(y?, x?) \in NurseToEventService$	
$sub\_Patient' = sub\_Patient \cup \{z?\}$	[R12]
$sub\_EventService' = sub\_EventService$	[R14]
$sub\_Nurse' = sub\_Nurse$	
$EventServiceToPatient' = EventServiceToPatient \cup \{(x?, z?)\}$	[R15]
$PatientToEventService' = PatientToEventService \cup \{(z?, x?)\}$	
$NurseToPatient' = NurseToPatient \cup \{(y?, z?)\}$	
$PatientToNurse' = PatientToNurse \cup \{(z?, y?)\}$	
$NurseToEventService' = NurseToEventService$	[R17]
$EventServiceToNurse' = EventServiceToNurse$	

FIG. 5.5 – Transformation de l'opération *Insert_Patient* vers un schéma d'opération *Z*

5.1.2 Transformation des contraintes OCL vers *Z*

Outre la transformation des notations graphiques, exprimées avec la notation UML, vers la notation *Z*, nous proposons également une transformation des contraintes OCL, exprimées dans les parties « Guards », vers des prédicats *Z*.

Dans cette perspective, nous définissons une passerelle entre OCL et *Z*. La passerelle que nous proposons implémente un ensemble de règles de passage. Cette passerelle correspond à une compilation des termes OCL en termes exprimés en *Z*. Ainsi, une description précise de la syntaxe de OCL et de *Z* est nécessaire afin d'accomplir cette compilation. Avant de décrire la passerelle, nous présentons donc la grammaire du langage OCL et celle du langage *Z*. Les deux grammaires ne représentent pas tous les concepts utilisés dans les deux langages. Elles représentent seulement les concepts que nous avons intégrés dans notre profil.

Typiquement, la transformation d'une expression OCL se fait en trois étapes. La première sert à segmenter l'expression OCL en différents segments selon la grammaire OCL définie (figure 5.6). Cette segmentation permet d'identifier les types de composants, les types de connexions, les noms des instances, les fonctions, les constantes et les opérateurs logiques et relationnels. La deuxième étape sert à traduire les segments générés en des prédicats *Z* selon la grammaire *Z* définie (figure 5.7). Cette transformation est faite selon le type des segments. La troisième étape consiste à faire le passage entre OCL et *Z* en utilisant la passerelle (figure 5.8).

Pour les grammaires que nous proposons, nous utilisons les conventions suivantes :
– Les choix sont séparés par des barres verticales : |
– Une partie optionnelle est limitée par deux crochets : [ ]
– Le symbole "*" indique la répétition zéro ou plusieurs fois du fragment concerné.
– Les symboles en majuscule indiquent les symboles non-terminaux, représentent les mots clés.
– Les symboles en minuscule indiquent les symboles terminaux.
– Un ensemble de règles de production, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux.

5.1.2.1 Grammaire OCL$^-$

Les règles suivantes représentent la grammaire du langage OCL. Ces règles constituent un sous-ensemble de la syntaxe de OCL que nous avons utilisée pour la définition des règles de transformation de OCL vers *Z*.

```
EXPOCL := [CONTEXT_EXP] [Fixe] EXPRESSION_OCL
EXPRESSION_OCL := RELATION_EXP | RELATION_EXP [OPLOG_OCL
RELATION_EXP]*
RELATION_EXP := ADDIDITIVE_EXP OPREL_OCL CONST | "("
ADDIDITIVE_EXP OPREL_OCL ( Const | ADDIDITIVE_EXP ) ")" |
ADDIDITIVE_EXP [RELATION_EXP]*
ADDIDITIVE_EXP := FONCT_APPEL POSTFIX ( FONCT | CONTIF_OCL
ADDIDITIVE_EXP ) | ε
OPLOG_OCL := "and" | "or" | "implies" | "not"
OPREL_OCL := "=" | "<" | ">" | "<=" | ">=" | "<>"
CONST := "0"-"9" | ("0"-"9")*
POSTFIX := "→"
FONCT_APPEL := NOM_TYPE"."NOM_TYPE | Nom"."NOM_TYPE | NOM_TYPE
CONTIF_OCL := "forAll" | "exists"
FONCT := "size()" | "isEmpty()" | "notEmpty()" | "includes(NOM)" |
"excludes(NOM)"
CONTEXT_EXP := "Context" CONTEXT_DEC ":" TYPE_RETOUR
CONTEXT_DEC := NOM_TYPE "::" NOM_REOP "("PARAMETRE_LISTE")"
PARAMETRE_LISTE := PARAMETRE ["," PARAMETRE]*
```

```
PARAMETRE := NOM
TYPE_RETOUR := "Boolean"
FIXE := "Pre" | "Post"
NOM := ["a"-"z"] | ["A"-"Z"] (["a"-"z"] | ["0"-"9"] | ["A"-"Z"])*
NOM_TYPE := ["a"-"z"] | ["A"-"Z"] (["a"-"z"] | ["0"-"9"] |
["A"-"Z"])*
NOM_REOP := ["a"-"z"] | ["A"-"Z"] (["a"-"z"] | ["0"-"9"] |
["A"-"Z"])*
```

FIG. 5.6 – Grammaire OCL^{-}

5.1.2.2 Grammaire Z^{-}

Les règles suivantes représentent la grammaire du langage *Z*. Ces règles constituent un sous-ensemble de la syntaxe de *Z* que nous avons utilisée pour la définition des règles de passage.

```
EXP_Z := EXPRESSION_Z [OPLOG_Z EXPRESSION_Z]*
EXPRESSION_Z := EXPRESSION1 | EXPRESSION2 | EXPRESSION3
EXPRESSION1 := Card NomType OPREL_Z Const
EXPRESSION2 := PARTIE1 "•" PARTIE2 OPREL_Z Const
EXPRESSION3 := PARTIE1 "•" PARTIE3
PARTIE1 := CONTIF_Z NOM "•" NomType
PARTIE2 := Card "(" NomConnexion "(|" "{" NOM "}" "|)" ")"
PARTIE3 := EXP OPLOG_Z EXP
EXP := NOM_CONNEXION "(|" "{" NOM "}" "|)" OPREL_Z Const
NOM_CONNEXION := NOM_TYPE "To" NOM_TYPE
CARD := "#"
CONTIF_Z := "∀" | "∃"
OPLOG_Z := "∧" | "∨" | "⇒" | "¬"
OPREL_Z := "=" | "<" | ">" | "≤" | "≥" | "≠"
CONST := "0"-"9" | ("0"-"9")*
NOM := ["a"-"z"] | ["A"-"Z"] (["a"-"z"] | ["0"-"9"] | ["A"-"Z"])*
NOM_TYPE := ["a"-"z"] | ["A"-"Z"] (["a"-"z"] | ["0"-"9"] |
["A"-"Z"])*
```

FIG. 5.7 – Grammaire Z^{-}

5.1.2.3 Grammaire de la passerelle

Dans cette section, nous décrivons une grammaire pour décrire les règles de transformation de OCL vers *Z*. La partie gauche d'une règle représente un symbole de la grammaire OCL qui se traduit par sa partie droite dans le langage *Z*.

```
EXP_OCL := Exp_Z
EXPRESSION_OCL := EXPRESSION_Z
RELATION_EXP := EXPRESSION1 | EXPRESSION2 | EXPRESSION3
ADDIDITIVE_EXP := PARTIE1 | PARTIE2 | PARTIE3
OPLOG_OCL := OPLOG_Z
OPREL_OCL := OPREL_Z
CONTIF_OCL := CONTIF_Z
CONST := CONST
NOM := NOM
NOMTYPE := NOMTYPE
```

FIG. 5.8 – Grammaire de la passerelle

5.1.2.4 Exemple de transformation

Pour mieux illustrer la transformation d'une expression OCL vers un prédicat *Z* nous détaillons dans ce qui suit l'exemple suivant.

$$\texttt{EventService}\rightarrow\texttt{size()} \leq 3 \texttt{ and EventService}\rightarrow\texttt{size()} > 0$$

Cette expression OCL exprime que le système peut contenir entre zéro et trois instances de composants de type EventService.

Cette expression est décomposée en différents segments selon la grammaire OCL définie.

```
EXP_OCL := EXPRESSION_OCL
EXPRESSION_OCL := RELATION_EXP1 OPLOG_OCL RELATION_EXP2
RELATION_EXP1 := ADDIDITIVE_EXP OPREL_OCL CONST
ADDIDITIVE_EXP := FONCT_APPEL POSTFIX FONCT
FONCT_APPEL := NOM_TYPE
NOM_TYPE := "EventService"
POSTFIX := "→"
FONCT := "Size()"
OPREL_OCL := "<="
CONST := "3"
OPLOG_OCL := "And"
RELATION_EXP2 := ADDIDITIVE_EXP OPREL_OCL CONST
ADDIDITIVE_EXP := FONCT_APPEL POSTFIX FONCT
FONCT_APPEL := NOM_TYPE
NOM_TYPE := "EventService"
```

```
POST_FIX := "→"
FONCT := "Size()"
OPREL_OCL := ">"
CONST := "0"
```

FIG. 5.9 – Décomposition de l'expression OCL

La transformation de l'expression précédente vers *Z* génère le prédicat suivant :

$$\#\ \texttt{sub\_EventService} \leq 3 \wedge \#\ \texttt{sub\_EventService} > 0$$

Ce prédicat est décomposé en différents segments selon la grammaire *Z* définie.

```
ExpZ := ExpressionZ1 OpLogZ ExpressionZ2
ExpressionZ1 := Expression1
ExpressionZ2 := Expression1
Expression1 := Card NomType OpRelZ Const
Card := "#"
NomType := "EventService"
OpRelZ := "≤"
Const := "3"
OpLogZ := "∧"
Card := "#"
NomType := "EventService"
OpRelZ := ">"
Const := "0"
```

FIG. 5.10 – Transformation de l'expression OCL vers *Z*

Nous revenons à notre exemple et nous traduisons la partie graphique et la partie guards du style architectural du système PMS vers la notation *Z*. L'application des règles de transformation de [R1] jusqu'à [R5] et l'utilisation des grammaires de transformation donnent le schéma *Z* suivant (figure 5.11).

$[EventService, Patient, Nurse]$

PMS

$sub\_EventService : \mathbb{F}\ EventService$
$sub\_Patient : \mathbb{F}\ Patient$
$sub\_Nurse : \mathbb{F}\ Nurse$
$EventServiceToPatient : EventService \leftrightarrow Patient$
$EventServiceToNurse : EventService \leftrightarrow Nurse$
$PatientToEventService : Patient \leftrightarrow EventService$
$NurseToEventService : Nurse \leftrightarrow EventService$
$PatientToNurse : Patient \leftrightarrow Nurse$
$NurseToPatient : Nurse \leftrightarrow Patient$

$\mathrm{dom}\ EventServiceToPatient \subseteq sub\_EventService$
$\wedge\ \mathrm{ran}\ EventServiceToPatient \subseteq sub\_Patient$
$\mathrm{dom}\ EventServiceToNurse \subseteq sub\_EventService$
$\wedge\ \mathrm{ran}\ EventServiceToNurse \subseteq sub\_Nurse$
$\mathrm{dom}\ PatientToEventService \subseteq sub\_Patient$
$\wedge\ \mathrm{ran}\ PatientToEventService \subseteq sub\_EventService$
$\mathrm{dom}\ NurseToEventService \subseteq sub\_Nurse$
$\wedge\ \mathrm{ran}\ NurseToEventService \subseteq sub\_EventService$
$\mathrm{dom}\ PatientToNurse \subseteq sub\_Patient$
$\wedge\ \mathrm{ran}\ PatientToNurse \subseteq sub\_Nurse$
$\mathrm{dom}\ NurseToPatient \subseteq sub\_Nurse$
$\wedge\ \mathrm{ran}\ NurseToPatient \subseteq sub\_Patient \# sub\_EventService \leq 3 \wedge \# sub\_EventService > 0$
$\forall x : sub\_EventService \bullet \#(EventServiceToNurse(\!|\ \{x\}\ |\!)) \leq 5$
$\forall x : sub\_EventService \bullet \#(EventServiceToPatient(\!|\ \{x\}\ |\!)) \leq 15$
$\forall x : sub\_Patient \bullet \#(PatientToEventService(\!|\ \{x\}\ |\!)) = 1$
$\forall x : sub\_Patient \bullet \#(PatientToNurse(\!|\ \{x\}\ |\!)) = 1$
$\forall x : sub\_Nurse \bullet \#(NurseToEventService(\!|\ \{x\}\ |\!)) = 1$
$\forall x : sub\_Nurse \bullet \#(NurseToPatient(\!|\ \{x\}\ |\!)) \leq 3$
$\forall x : sub\_EventService \bullet EventServiceToPatient(\!|\ \{x\}\ |\!) \neq \varnothing \Rightarrow EventServiceToNurse(\!|\ \{x\}\ |\!) \neq \varnothing$

FIG. 5.11 – Transformation du style architectural de l'exemple PMS vers la notation *Z*

De même, nous traduisons la partie graphique et la partie guards de l'opération de reconfiguration *Inser_Patient* vers la notation *Z*. L'application des règles de transformation successivement de `[R6]` jusqu'à `[R17]` et l'utilisation des grammaires de transformation donnent le schéma d'opération *Z* présenté dans la figure 5.12 suivante.

Insert_Patient

ΔPMS
$x? : EventService$
$y? : Nurse$
$z? : Patient$

$x? \in sub\_EventService$
$y? \in sub\_Nurse$
$z? \notin sub\_Patient$
$(x?, y?) \in EventServiceToNurse$
$(y?, x?) \in NurseToEventService$
$sub\_Patient' = sub\_Patient \cup \{z?\}$
$sub\_EventService' = sub\_EventService$
$sub\_Nurse' = sub\_Nurse$
$EventServiceToPatient' = EventServiceToPatient \cup \{(x?, z?)\}$
$PatientToEventService' = PatientToEventService \cup \{(z?, x?)\}$
$NurseToPatient' = NurseToPatient \cup \{(y?, z?)\}$
$PatientToNurse' = PatientToNurse \cup \{(z?, y?)\}$
$NurseToEventService' = NurseToEventService$
$EventServiceToNurse' = EventServiceToNurse$
$\#(EventServiceToNurse(\!| \{x?\} |\!)) > 0 \land$
$\#(EventServiceToPatient(\!| \{x?\} |\!)) < 15 \land$
$\#(NurseToPatient(\!| \{y?\} |\!)) < 3$

FIG. 5.12 – Transformation de l'opération *Insert_Patient* vers la notation *Z*

5.2 Vérification

L'objectif de ce chapitre n'est pas la transformation en elle-même mais plutôt la vérification et le raisonnement formel. En effet, la transformation vers *Z* n'est qu'une phase intermédiaire avant d'entamer la partie vérification et preuve de théorèmes. La définition de cette sémantique formelle offre deux avantages.

– Elle permet de vérifier et de confirmer, sans ambiguïté, la consistance de tout style architectural.

– Elle permet de vérifier la préservation du style architectural après l'exécution d'une opération de reconfiguration.

Pour vérifier la consistance et la conformité de l'architecture par rapport à son style architectural, nous adoptons l'approche proposée par Me. Imen LOULOU et al. [84].

5.2.1 Consistance

Un schéma *Z* est considéré comme inconsistant ou incohérent s'il contient un prédicat non satisfaisant et/ou une contradiction entre les prédicats. Si ce schéma décrit un système alors, dans ce cas, le système est dit non réalisable [141].

Vérifier la consistance d'une spécification revient à prouver qu'au moins une instance valide existe [141]. Dans *Z*, si nous traitons un schéma qui décrit l'état du système, il s'agirait alors de fournir un schéma qui l'instancierait. Dans notre contexte, un schéma décrit un style architectural. Il s'agit donc de prouver l'existence d'une configuration (état initial) qui appartient au style en question.

En *Z*, la spécification d'un système est définie par un ensemble de composants, de relations et des contraintes architecturales. Dans l'exemple du schéma *InitSystem*, comme le montre la figure 5.13, component_i représente un type de composant, c_i représente une instance de composant, la relation_{ij} représente une relation entre le composant c_i et le composant c_j et contrainte_i représente une contrainte architecturale.

InitSystem

ArchitectureStyle

$component_i = \{c_i\}$

$relation_{ij} = \{(c_i, c_j)\}$

$contrainte_i$

...

FIG. 5.13 – Schéma d'initialisation

Supposons que *ArchitectureStyle* représente le schéma du système et que *InitSystem* représente un schéma *Z* qui décrit l'état initial du système. Si nous prouvons le théorème suivant :

Theorem *ConsistentTheorem*

$\exists$ *ArchitectureStyle* . *InitSystem*

alors nous avons montré qu'un état initial existe, et donc les exigences demandées sur le système sont consistantes. Ce résultat est appelé le théorème d'initialisation [141].

Nous revenons à notre exemple et nous vérifions la consistance du schéma du système PMS. Comme le montre la figure 5.14, pour vérifier le théorème de la consistance, nous définissons un état initial *PMS_Init*. L'état initial se compose d'une instance *EventService ES*, d'une instance *Patient P* et d'une instance *Nurse N* inter-reliées ensemble ((P, ES), (N, P), (N, ES)...). La preuve du théorème *ConsistentPMS* vérifie que la spécification du système est consistante et elle ne contient aucune contradiction. La spécification est décrite comme suit :

PMS_Init

PMS

$sub\_EventService = \{ES\}$
$sub\_Patient = \{P\}$
$sub\_Nurse = \{N\}$
$PatientToEventService = \{(P, ES)\}$
$NurseToPatient = \{(N, P)\}$
$NurseToEventService = \{(N, ES)\}$
$PatientToNurse = \{(P, N)\}$
$EventServiceToPatient = \{(ES, P)\}$
$EventServiceToNurse = \{(ES, N)\}$

FIG. 5.14 – Une configuration possible du système PMS

Le théorème d'initialisation que nous avons prouvé avec le système de preuve de l'outil Z/EVES est par conséquent le suivant :

Theorem *ConsistentPMS*
$\exists PMS \bullet PMS\_Init$

En choisissant la commande *prove by reduce*, Z/EVES vient à bout de la démonstration et affiche *true*.

5.2.2 Préservation du style architectural

Afin de prouver que l'évolution de l'architecture préserve toujours le style architectural, il faudrait vérifier pour chaque opération de reconfiguration que les contraintes (invariants)

définies dans le style architectural sont préservées à la fin de leur application.

Dans cette étape, les opérations de reconfiguration sont spécifiées formellement. Chaque opération de reconfiguration est spécifiée par un schéma d'opération *Z*, qui définit les paramètres d'entrée (c_i ?) et les pré- et post-conditions. Ces conditions sont essentielles pour vérifier que l'évolution de l'architecture préserve les invariants. Les opérations de reconfiguration sont exécutées seulement si leurs conditions préalables sont satisfaites. Dans le schéma d'opération ci-dessous, $PreCond_i$ et $PostCond_i$ dénotent une pré- et une post-condition de l'opération de reconfiguration $Operation_i$.

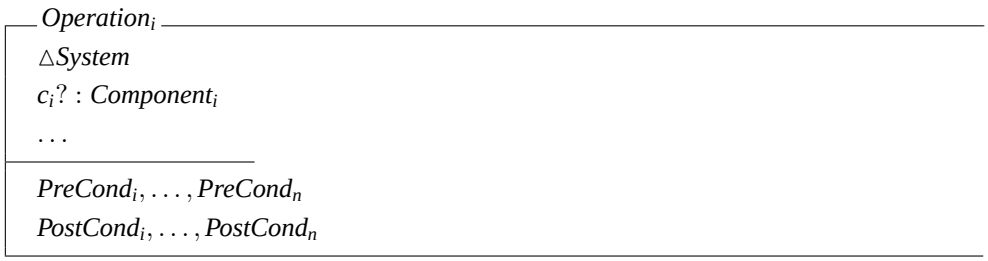

FIG. 5.15 – Template de schéma d'opération *Z*

Après avoir spécifié les opérations de reconfiguration formellement, ces opérations doivent être vérifiées. Pour évaluer l'impact d'une opération de reconfiguration sur une contrainte, nous définissons et nous prouvons le théorème *PreCondTheorem*. Ce théorème énonce les pre-conditions qui doivent être satisfaites pour garantir que les contraintes sont préservées après l'exécution de l'opération et vérifient que l'exécution de l'opération de reconfiguration préserve le style architectural.

Theorem *PreCondTheorem*
$\forall System \wedge c? : Component_i$
$\mid preConditions$ **.** pre $Operation_i$

Illustrons maintenant l'approche de spécification des opérations de reconfiguration en utilisant l'exemple de PMS. Nous avons spécifié formellement les opérations de reconfiguration telles que l'insertion et la suppression d'un service d'événement, d'un patient et d'une infirmière. Pour l'illustration, le schéma suivant indique l'opération de reconfiguration *InsertPatient*. Le schéma *Z* de l'opération déclare qu'il faut au moins une infirmière *y*

appartenant au service en question *x* pour pouvoir s'occuper du nouveau patient *z*. En plus, il faut que le service en question ne contient pas déjà quinze patients et que le nombre de patients pour l'infirmière en question *y* est strictement inférieur à 3. Afin de valider l'opération d'insertion d'un nouveau patient, nous utilisons l'outil Z/EVES pour prouver le théorème *PreInsertPatient*, qui assure que l'insertion d'un patient est conforme aux contraintes du système décrites dans le schéma du système *PMS*.

Theorem *Pre_Insert_Patient*
$\forall PMS, x? : EventService, y? : Nurse, z? : Patient$
$| \; y? \in sub\_Nurse$
$\land \; x? \in EventService$
$\land \; x? \in sub\_EventService$
$\land \; z? \notin sub\_Patient$
$\land \; (x?, y?) \in EventServiceToNurse$
$\land \; (y?, x?) \in NurseToEventService$
$\land \; \#(EventServiceToNurse(\!| \; \{x?\} \; |\!)) > 0$
$\land \; \#(EventServiceToPatient(\!| \; \{x?\} \; |\!)) < 15$
$\land \; \#(NurseToPatient(\!| \; \{y?\} \; |\!)) < 3$
$\land \; (\forall x : sub\_EventService \bullet \#(EventServiceToPatient \cup \{(x?, z?)\}(\!| \; \{x\} \; |\!)) \leq 15)$
$\land \; (\forall x : sub\_Patient \cup \{z?\} \bullet \#(PatientToEventService \cup \{(z?, x?)\}(\!| \; \{x\} \; |\!)) = 1)$
$\land \; (\forall x : sub\_Patient \cup \{z?\} \bullet \#(PatientToNurse \cup \{(z?, y?)\}(\!| \; \{x\} \; |\!)) = 1)$
$\land \; (\forall x : sub\_Nurse \bullet \#(NurseToPatient \cup \{(y?, z?)\}(\!| \; \{x\} \; |\!)) \leq 3)$
$\land \; (\forall x : sub\_EventService \bullet EventServiceToNurse(\!| \; \{x\} \; |\!) \neq \{\}) \bullet \text{pre } Insert\_Patient$

Z/Eves réussit à démontrer ce théorème si on lui suggère la commande *prove by reduce*. On obtient le prédicat suivant à prouver.

PMS
$\land \; x? \in Event\_Service$
$\land \; y? \in Nurse$
$\land \; z? \in Patient$
$\land \; y? \in sub\_Nurse$
$\land \; x? \in sub\_Event\_Service$
$\land \; z? \notin sub\_Patient$
$\land \; (x?, y?) \in Event\_ServiceToNurse$
$\land \; (y?, x?) \in NurseToEvent\_Service$
$\land \; \#(Event\_ServiceToNurse(\!| \; \{x?\} \; |\!)) > 0$
$\land \; \#(Event\_ServiceToPatient(\!| \; \{x?\} \; |\!)) < 15$
$\land \; \#(NurseToPatient(\!| \; \{y?\} \; |\!)) < 3$

$$
\begin{array}{l}
\land\ (\forall\, x : \mathit{sub\_Event\_Service} \bullet \#(\mathit{Event\_ServiceToPatient} \cup \{(x?, z?)\}(\!|\ \{x\}\ |\!)) \leq 15) \\
\land\ (\forall\, x\_0 : \mathit{sub\_Patient} \cup \{z?\} \\
\quad \bullet \#(\mathit{PatientToEvent\_Service} \cup \{(z?, x?)\}(\!|\ \{x\_0\}\ |\!)) = 1) \\
\land\ (\forall\, x\_1 : \mathit{sub\_Patient} \cup \{z?\} \\
\quad \bullet \#(\mathit{PatientToNurse} \cup \{(z?, y?)\}(\!|\ \{x\_1\}\ |\!)) = 1) \\
\land\ (\forall\, x\_2 : \mathit{sub\_Nurse} \\
\quad \bullet \#(\mathit{NurseToPatient} \cup \{(y?, z?)\}(\!|\ \{x\_2\}\ |\!)) \leq 3) \\
\land\ (\forall\, x\_3 : \mathit{sub\_Event\_Service} \bullet \mathit{Event\_ServiceToNurse}(\!|\ \{x\_3\}\ |\!) \neq \{\}) \\
\Rightarrow\ (\exists\, \mathit{Event\_ServiceToNurse}' : \mathbb{P}\,(\mathit{Event\_Service} \times \mathit{Nurse})\,; \\
\quad \mathit{Event\_ServiceToPatient}' : \mathbb{P}\,(\mathit{Event\_Service} \times \mathit{Patient})\,; \\
\quad \mathit{NurseToEvent\_Service}' : \mathbb{P}\,(\mathit{Nurse} \times \mathit{Event\_Service})\,; \\
\quad \mathit{NurseToPatient}' : \mathbb{P}\,(\mathit{Nurse} \times \mathit{Patient})\,; \\
\quad \mathit{PatientToEvent\_Service}' : \mathbb{P}\,(\mathit{Patient} \times \mathit{Event\_Service})\,; \\
\quad \mathit{PatientToNurse}' : \mathbb{P}\,(\mathit{Patient} \times \mathit{Nurse})\,; \\
\quad \mathit{sub\_Event\_Service}' : \mathbb{P}\ \mathit{Event\_Service}\,; \\
\quad \mathit{sub\_Nurse}' : \mathbb{P}\ \mathit{Nurse}\,; \\
\quad \mathit{sub\_Patient}' : \mathbb{P}\ \mathit{Patient} \bullet \mathit{Insert\_Patient})
\end{array}
$$

5.3 Conclusion

Dans ce chapitre, nous avons proposé une approche de vérification basée sur la transformation des modèles semi-formels vers des spécifications formelles. Ce chapitre s'articule autour de deux parties. Dans la première, nous avons proposé une approche de transformation permettant de transformer un style architectural vers un schéma *Z* et chaque opération de reconfiguration vers un prédicat *Z*. Nous avons également présenté une passerelle permettant de transformer une expression OCL vers un prédicat *Z*. La passerelle est composée d'un ensemble de règles de passage qui définissent l'équivalent de chaque élément du langage OCL dans le langage *Z*. Dans la deuxième partie, nous avons adopté une approche de vérification [84]. Nous avons vérifié la consistance du style architectural et nous avons prouvé que l'évolution de l'architecture est toujours conforme à son style architectural. Ces théorèmes ont été prouvés en utilisant le système de preuve de l'outil Z/EVES.

Comme perspective à ce chapitre, nous prévoyons d'améliorer l'approche de vérification en y ajoutant d'autres propriétés. Pour la partie protocole de reconfiguration, nous prévoyons d'achever la partie transformation permettant de traduire la description XML générée vers un langage formel. Nous prévoyons d'utiliser les réseaux de pétri pour vérifier le non blocage des différentes opérations de reconfiguration et que chaque opération de reconfiguration sera exécutée au moins une fois.

6

Démarche de modélisation

Jusqu'à maintenant, nous avons présenté les étapes nécessaires pour la modélisation de la dynamique des architectures logicielles en utilisant le Profil UML, pour la validation des modèles résultants et pour la vérification et la preuve de la consistance du style architectural et de la conformité de l'évolution de l'architecture par rapport à son style. Il reste cependant à définir une démarche entre ces différentes étapes et de capitaliser des règles permettant de guider et d'assister les architectes à modéliser l'architecture d'une application dynamique tout en suivant le profil proposé.

La démarche X (démarche de modélisation des architectures logicielles dynamiques), que nous proposons permet de décrire les différentes étapes pour modéliser l'architecture logicielle (Style architectural, Opération de reconfiguration et Protocole de reconfiguration) et de définir les différentes règles de passage d'un modèle à un autre. Elle automatise le processus de validation et assiste l'utilisateur pour vérifier des propriétés architecturales. La démarche que nous proposons est inspirée de l'approche MDA [10, 67] (Model Driven Architecture) et 2TUP [127] (2 Tracks Unified Process) du processus UP (Unified Process). Notre démarche est subdivisée en deux parties : une partie PIM (Platform Independant Model) et une partie PSM (Platform Specific Model). La première, modélise l'architecture logicielle sans se soucier des technologies sous-jacentes. La seconde, modélise l'implementation des directives de conception vis-à-vis de la plateforme visée (Java/EJB, C #/.Net, etc.). La démarche que nous présentons est itérative, centrée sur l'architecture et incrémentale.

6.1 Description de la démarche *X*

La démarche *X* que nous proposons est basée sur l'approche MDA et le processus UP et apporte une réponse aux contraintes de changement continuel imposées aux architectures logicielles dynamiques. Elle renforce le contrôle de l'adaptation et la correction de telles architectures. Comme le montre la figure 6.1, la démarche que nous proposons est basée sur quatre branches et a la forme de *X*.

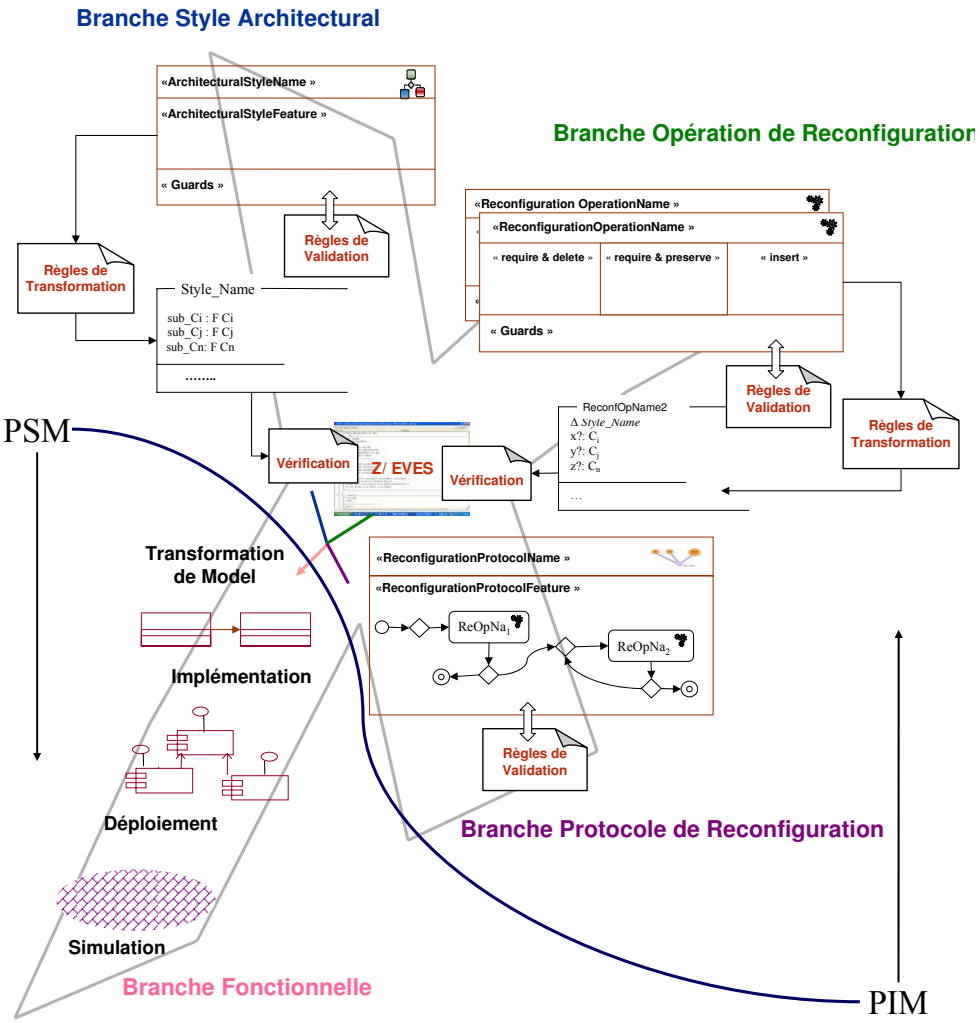

FIG. 6.1 – La démarche proposée

6.1.1 Branche style architectural

La capture des besoins architecturaux est la première étape de la démarche *X*. Elle formalise et détaille les besoins du niveau statique. Cette étape, comme le montre la figure 6.2, produit, par itération, un style architectural valide et consistant.

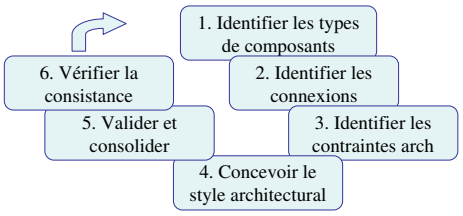

FIG. 6.2 – La branche style architectural

La branche style architectural se fait à travers les étapes suivantes :
– La première étape consiste à identifier les types de composants. Pour chaque composant, il faut identifier la liste de ses ports et de ses interfaces.
– La deuxième étape consiste à identifier la liste des connexions. Pour chaque connexion, il faut identifier le type de composant source et le type de composant destinataire.
– La troisième étape consiste à définir la liste des contraintes architecturales, exprimées avec le langage OCL, que le système doit respecter.
– La quatrième étape consiste à modéliser le style architectural en utilisant les résultats des trois étapes précédentes.
– La cinquième étape consiste à valider le modèle du style architectural résultant par rapport à son méta-modèle. Cette validation consiste donc à transformer d'une façon automatique le modèle du style architectural vers une description XML qui sera elle même validée par rapport à son méta-modèle représenté par une description XSD (XML Schéma). La traduction du modèle UML vers XML et la phase de validation sont automatiques grâce au plug-in FUJABA que nous avons développé.
– La sixième étape consiste à vérifier la consistance. La description XML du style architectural est transformée, en utilisant des règles de transformation, d'une façon automatique, grâce au plug-in FUJABA vers le langage *Z*. La spécification résultante est analysée et le théorème de consistance est vérifié avec l'outil Z/EVES.

Les résultats de cette branche appartiennent à la partie PIM. Ils ne dépendent d'aucune technologie particulière.

6.1.2 Branche opération de reconfiguration

Cette branche inclue la capture des besoins adaptatifs, qui recense toutes les évolutions possibles de l'architecture. Elle consiste, comme le montre la figure 6.3, à identifier pour chaque opération de reconfiguration les étapes suivantes :

– La première étape consiste à identifier la liste des composants et des connexions à ajouter dans la partie «*Insert*».
– La deuxième étape consiste à identifier la liste des composants et des connexions à supprimer dans la partie «*require & delete*».
– La troisième étape consiste à identifier la liste des composants et des connexions non modifiables dans la partie «*require & preserve*».

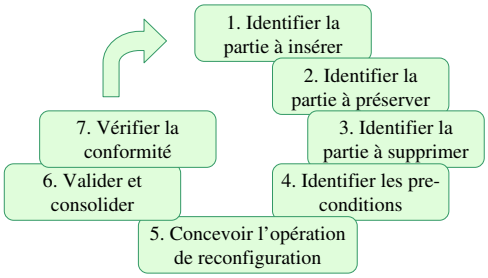

FIG. 6.3 – La branche opération de reconfiguration

– La quatrième étape consiste à identifier la liste des pré-conditions, exprimées avec le langage OCL, pour chaque opération de reconfiguration.
– La cinquième étape consiste à modéliser l'opération de reconfiguration en utilisant les résultats des quatre étapes précédentes.
– La sixième étape consiste à valider le modèle UML résultant de chaque opération de reconfiguration par rapport à son méta-modèle. Chaque modèle est transformé vers une description XML qui sera elle même validée par rapport à son méta-modèle représenté par une description XSD (XML schema). La traduction du modèle UML vers XML et la phase de validation sont automatiques grâce au plug-in FUJABA que nous avons développé.
– La septième étape consiste à vérifier la conformité de l'évolution d'une architecture après l'exécution d'une opération de reconfiguration vis-à-vis de son style architectural. La description XML est transformée d'une façon automatique, grâce à des règles de transformation implémentée et intégrée dans le plug-in FUJABA, vers le langage *Z*. La spécification résultante est analysée et le théorème de conformité est vérifié avec l'outil Z/EVES.

De même, les résultats de cette branche appartiennent à la partie PIM. Ils ne dépendent d'aucune technologie particulière.

6.1.3 Branche protocole de reconfiguration

Cette branche, comme le montre la figure 6.4, recense l'enchaînement et l'ordre d'exécution des différentes opérations de reconfiguration. Les étapes se déroulent comme suit :

– La première étape consiste à identifier les différentes opérations de reconfiguration.
– La deuxième étape consiste à identifier les différents nœuds de contrôle afin de synchroniser les différentes opérations de reconfiguration.
– La troisième étape consiste à identifier les différents liens de coordination afin de relier les différentes opérations de reconfiguration.
– La quatrième étape consiste à modéliser l'enchaînement de l'exécution des opérations de reconfiguration en utilisant les résultats des trois étapes précédentes.

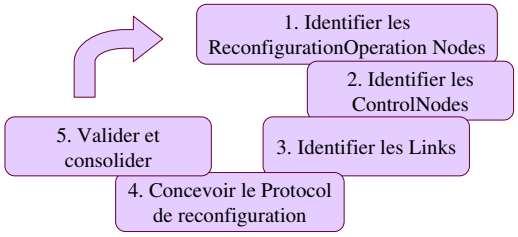

FIG. 6.4 – La branche protocole de reconfiguration

– La cinquième étape consiste à valider le modèle UML résultant du protocole de reconfiguration par rapport à son méta-modèle. Chaque modèle est transformé vers une description XML qui sera validée par rapport à son méta-modèle représenté par une description XSD (XML schema). La traduction du modèle UML vers XML et la phase de validation sont automatiques grâce au plug-in FUJABA que nous avons développé.

Les résultats de cette branche appartiennent à la partie PIM. Ils ne dépendent d'aucune technologie particulière.

6.1.4 Branche fonctionnelle

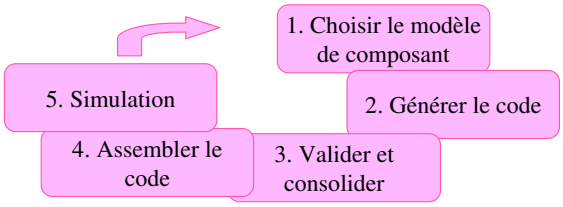

FIG. 6.5 – La branche fonctionnelle

La finalité de cette branche, comme le montre la figure 6.5, est de transformer les modèles PIM modélisés vers un modèle PSM choisi. La transformation de modèles comporte essentiellement deux étapes. La première consist à spécifier les règles de transformation exprimant la correspondance entre les concepts du méta-modèle qui décrit le modèle source et les concepts du méta-modèle qui décrit le modèle cible. La deuxième a pour but d'appliquer ces règles au modèle source pour produire le modèle cible. Nous pouvons ainsi, par transformation de modèles, générer plusieurs PSM spécifiques à des plateformes différentes à partir d'un PIM unique (figure 6.6) [67].

Au cours des phases de développement d'un système dans l'architecture MDA, la plupart de ces transformations, comme le montre la figure 6.7, ont souvent besoin d'être réitérées pour affiner les divers modèles obtenus avant de passer à la génération de code.

La branche fonctionnelle inclut les étapes suivantes :

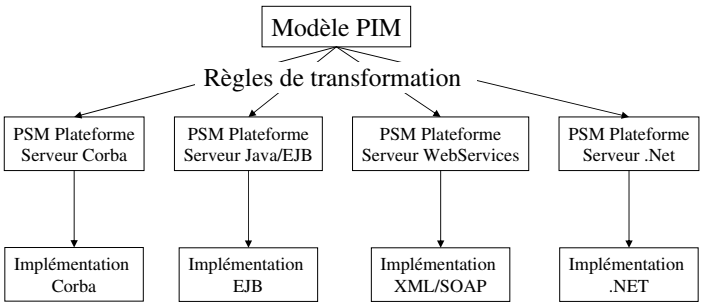

FIG. 6.6 – Transformation d'un modèle PIM vers différentes plateformes

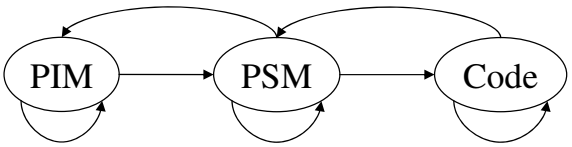

FIG. 6.7 – Différentes opérations de transformation sur les modèles MDA

– La première étape consiste à choisir le modèle de composant. Nous traçons donc la cartographie des composants du système à développer et nous étudions ensuite comment réaliser chaque composant.
– La deuxième étape consiste à faire le codage, qui produit les composants en y intégrant l'aspect métier et teste au fur et à mesure les unités de code réalisées.
– La troisième étape consiste à valider les fonctions de chaque composant développé.
– La quatrième étape consiste à assembler les composants développé au sein d'une même application.
– La cinquième étape c'est l'étape finale qui consiste à simuler le fonctionnement global du système développé.

Pour la branche fonctionnelle, nous avons fait quelques réflexions et nous avons présenté quelques éléments de base. Des travaux plus avancés sont en cours de développement au sein de notre unité de recherche ReDCAD.

6.2 Caractéristiques de la démarche *X*

6.2.1 *X* produit des modèles réutilisables

Les trois branches de la partie PIM capitalisent la description de la dynamique de l'architecture logicielle de l'application à développer. Elle constitue généralement un investissement pour le moyen et le long terme. L'architecture du système d'information est en effet indépendante des technologies utilisées.

Comme le montre la figure 6.8, les différents modèles produits de la partie PIM sont

les mêmes pour les différentes technologies de la partie PSM. Il suffit de “greffer” une nouvelle architecture technique pour mettre à jour un système existant.

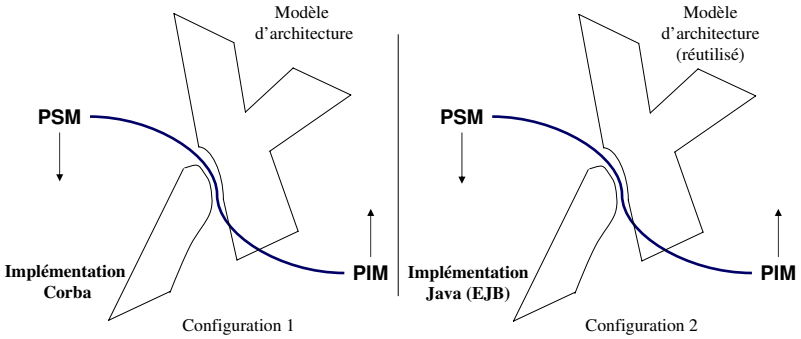

FIG. 6.8 – Réutilisation du modèle PIM

6.2.2 Démarche itérative

Une itération est une séquence distincte d'activités, destinée à être exécutée plusieurs fois et autant de fois qu'on peut en avoir besoin. Elle permet de faire évoluer le système d'une configuration à une autre en produisant un changement du niveau architectural. Le contenu d'une itération est porteur de corrections, d'améliorations et d'évolutions du système [127].

Dans notre démarche nous proposons deux types d'itérations : itération spécifique et itération générale. L'itération spécifique consiste à affiner et à améliorer un modèle d'une branche donnée. L'itération générale est corrective et consiste à retourner et à corriger les erreurs détectées d'un modèle.

Nous présentons les différentes étapes d'une itération possible.

– La première itération conçoit le style architectural de l'application à développer.
– La deuxième itération valide le style architectural et vérifie sa consistance. Elle commence la modélisation des opérations de reconfiguration.
– La troisième itération valide chaque opération de reconfiguration et vérifie sa conformité par rapport au style architectural. Elle commence à concevoir le protocole de reconfiguration.
– La quatrième itération valide le protocole de reconfiguration. Elle avance dans le choix du modèle de composant et la génération de codes de manière à présenter une première version de déploiement pour les utilisateurs. Elle permet entre-temps de corriger et d'améliorer les itérations précédentes.
– La cinquième itération avance dans la réalisation des fonctions jusqu'à l'obtention complète du système initialement envisagé.

Chaque itération passe en revue toutes les activités de la démarche X. Il est évident que

l'effort consacré à chaque activité n'est pas identique suivant la phase de développement du projet. En cas d'erreurs, le retour est possible pour apporter les corrections nécessaires.

6.2.3 Démarche incrémentale

Un incrément est la différence entre deux évolutions produites à la fin de deux itérations successives. Une ou plusieurs itérations s'inscrivent dans chacune des phases de modélisation du système et servent en conséquence à réaliser l'objectif propre à chaque phase [127].

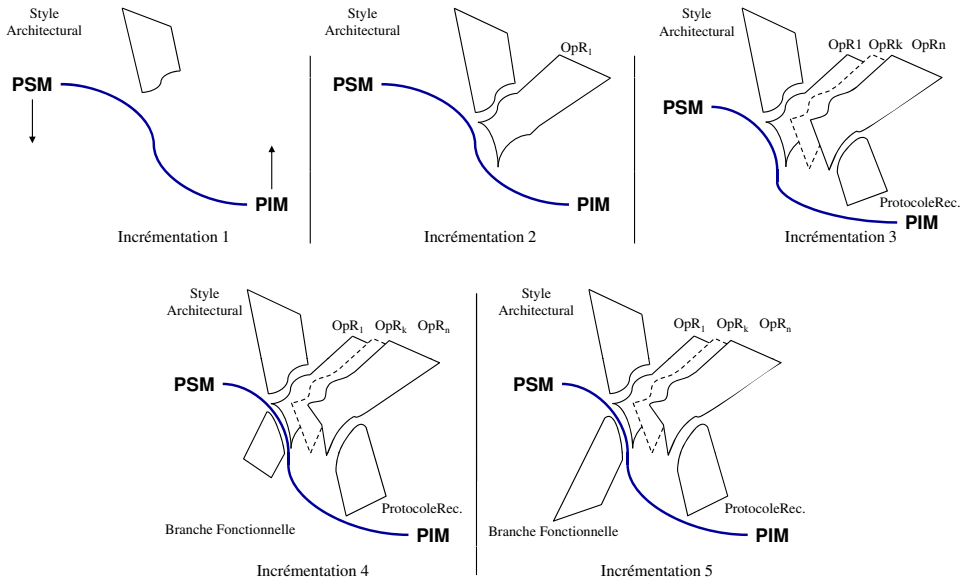

FIG. 6.9 – Un exemple d'incrémentation

Comme le montre la figure 6.9, le système évolue par incrément. A chaque itération, de nouvelles fonctionnalités sont ajoutées et ce jusqu'à ce que toutes les fonctionnalités demandées soient présentes. Le retour en arrière et très coûteux, c'est pourquoi le style architectural doit être bien pensé dès la première phase.

Une ou plusieurs itérations s'inscrivent dans chacune des phases de gestion de projet et servent en conséquence à réaliser l'objectif propre à chaque phase.

– En phase de modélisation du style architectural, les itérations servent à bien définir les types des composants et la liste des connexions ainsi que la définition des contraintes OCL. Par raffinement, cette phase sert à livrer le style architectural le mieux approprié aux besoins des utilisateurs.
– En phase de modélisation des opérations de reconfiguration, les itérations servent à bien définir et pour chaque opération, la liste des composants et des connexions à ajouter, à supprimer et non modifiable ainsi que la définition des contraintes OCL. Elles

servent aussi à confirmer l'adéquation de chaque opération de reconfiguration au style architectural.
– En phase de modélisation du protocole de reconfiguration, les itérations servent à définir le séquencement et les différents enchaînements possibles entre les opérations de reconfiguration.
– En phase fonctionnelle, les itérations servent à implémenter, tester et livrer progressivement toutes les fonctionnalités du système.

La démarche de développement en X se reproduit à différents niveaux d'avancement en se terminant sur la livraison d'une nouvelle version. Elle offre la possibilité de revenir sur une ou plusieurs phases afin de corriger et de faire évoluer le système.

6.2.4 Démarche de modélisation à base d'UML

La démarche en X que nous proposons est basée sur UML. La majorité des concepts et des notations que nous avons présentés dans ce manuscrit sont basés sur le profil UML déjà défini. En plus, les concepts de modèle et de méta-modèle est inspiré du langage UML.

6.2.5 Démarche centrée sur l'architecture

Une démarche centrée sur l'architecture impose le respect du style architectural à chaque étape de construction du système [127]. Le choix et la modélisation du style architectural est une étape fondamentale et la condition qui préside les autres étapes de conception et de développement d'un système. Elle permet la structuration (style architectural), le suivi (opération de reconfiguration) et la cohérence (protocole de reconfiguration).

L'utilisation d'un style architectural cohérent permet d'établir des opérations précises de reconfiguration. L'architecte utilise le style architectural pour faire évoluer son système en modélisant et en exécutant des opérations de reconfiguration. Il contrôle et coordonne l'exécution de ces opérations en utilisant le protocole de reconfiguration. La démarche centrée sur l'architecture permet donc d'organiser et de contrôler la modélisation et l'évolution de l'architecture.

6.2.6 Démarche orientée vers les composants

La démarche que nous proposons manipule les composants, au sens UML du terme et au sens architectural du terme. En effet, elle modélise l'architecture logicielle en utilisant les composants et les connexions.

Une fois que l'architecture de l'application est modélisée, nous pouvons nous intéresser aux fonctionnalités propres du système. Le choix d'un style architectural influence la façon dont seront organisés et déployés les composants du système. Un composant logiciel

est une partie du système logiciel qui doit être connue, installée, déclarée et manipulée par les exploitants du système. Les composants logiciels sont les éléments que l'on déploie pour installer le système complet. Les composants sont des sous-parties du logiciel, de sorte que chaque composant correspond à un sous-système de configuration logicielle. Un composant doit être interchangeable entre différentes versions et peut être arrêté ou démarré séparément. Il assume des fonctions bien identifiées dans le système, de sorte qu'en cas de dysfonctionnement, le composant défaillant est facilement repérable [127].

6.3 Environnement de modélisation

Pour que notre approche soit expérimentée, elle doit offrir aux utilisateurs les moyens pour la manipuler. Pour cela, nous avons travaillé sur l'extension de l'outil FUJABA [28, 72] afin qu'il supporte notre approche. Nous avons ajouté deux Plug-ins Java. Le premier offre des outils pour la modélisation et la validation de la dynamique de l'architecture. Le deuxième offre des outils pour la transformation des modèles UML vers le langage *Z* et la vérification. La vérification et la preuve des théorèmes sont réalisées avec l'outil Z/EVES.

FUJABA est un atelier qui a retenu notre attention en raison de son approche intéressante. C'est un logiciel gratuit qui offre la possibilité de modifier le code source et d'y ajouter des plug-ins permettant ainsi d'intégrer notre profil, de valider et de vérifier nos spécifications.

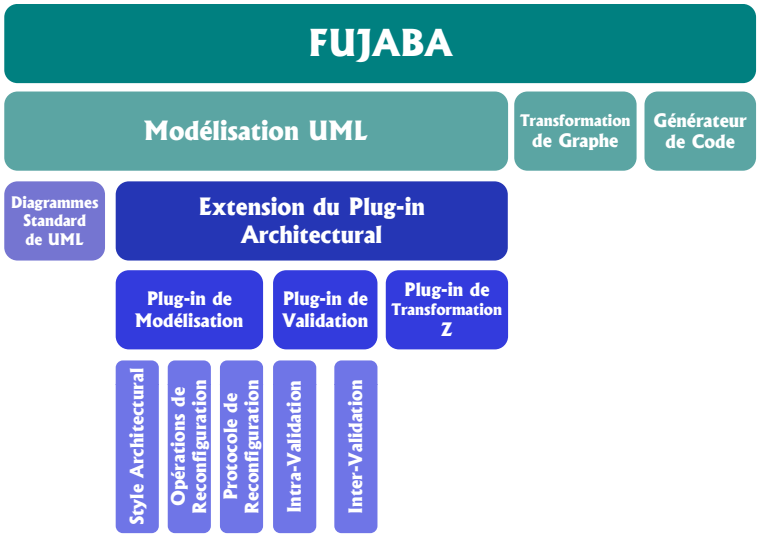

FIG. 6.10 – Extension de l'outil FUJABA

L'extension que nous avons proposée, comme le montre la figure 6.10, offre essentiellement trois fonctionnalités. La première permet de modéliser la dynamique de l'architecture logicielle. La deuxième permet de transformer les modèles UML vers le langage XML et de les valider par rapport à leurs méta-modèles correspondants. La troisième

permet de transformer les descriptions XML vers le langage *Z* afin de vérifier certaines propriétés.

6.3.1 Modélisation

Le profil proposé a fait l'objet d'une implémentation et d'une intégration dans l'atelier d'aide à la conception FUJABA. L'extension que nos avons réalisée, sous forme d'un Plug-in java, permet dans un premier temps la modélisation de la dynamique de l'architecture logicielle. Pour ce faire, le plug-in offre trois interfaces, équipée chacune par une barre à outil, permettant de modéliser le style architectural (figure 6.11), les opérations de reconfiguration (figure 6.12) et le protocole de reconfiguration (figure 6.13).

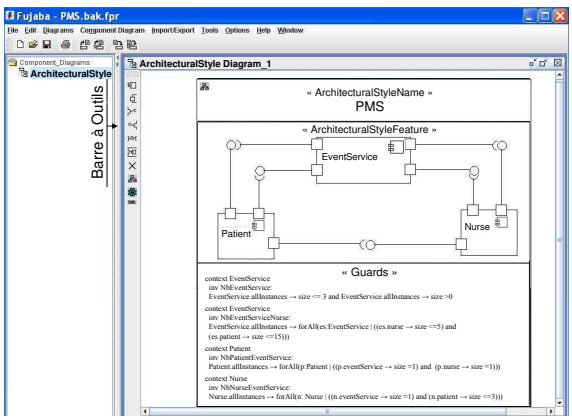

FIG. 6.11 – Interface pour la modélisation du style architectural

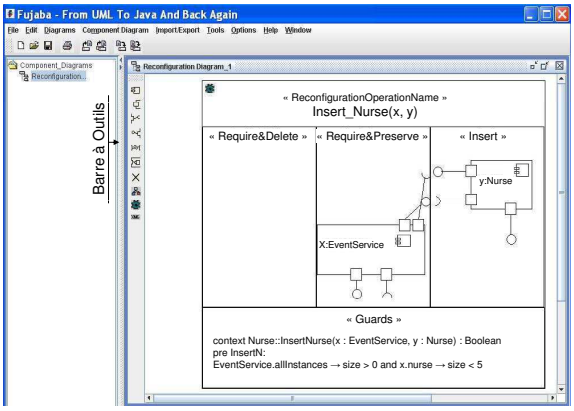

FIG. 6.12 – Interface pour la modélisation des opérations de reconfiguration

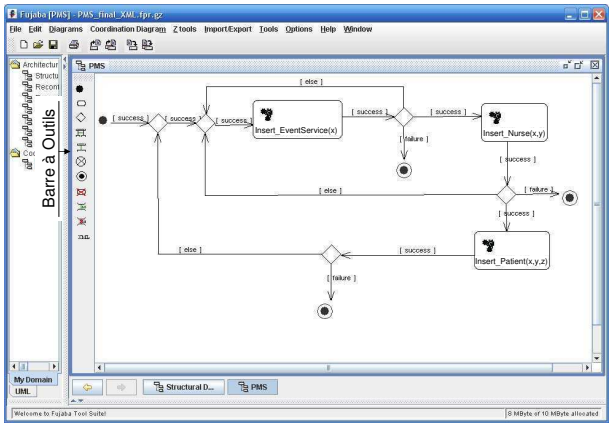

FIG. 6.13 – Interface pour la modélisation du protocole de reconfiguration

6.3.2 Validation

Le plug-in implémenté permet la transformation automatique des modèles UML vers le langage XML. Cette transformation est assurée en cliquant sur le bouton *Export To XML* (figure 6.14).

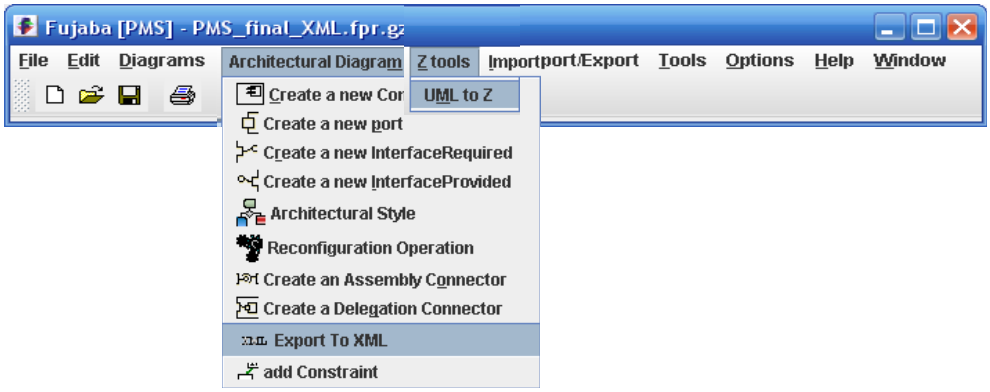

FIG. 6.14 – Transformation des modèles UML vers des descriptions XML

Nous présentons ici la description XML générée automatiquement à partir de notre plug-in. Le script XML ci-dessous est relatif à l'étude de cas PMS et décrit le style architectural que nous avons présenté dans la section 3.4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ArchitecturalStyleM-M
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ArchitecturalStyleMM.xsd">
   <ArchitecturalStyleName ArchitecturalStyleName="PMS"/>
   <ArchitecturalStyleFeature>
      <Component>
         <NameComponent NameComponent="EventService"/>
         <Port>
            <NamePort NamePort="EventService.P1"/>
            <Interface Type="Provided">
               <NameInterface NameInterface="EventService.P1.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="EventService.P2"/>
            <Interface Type="Provided">
               <NameInterface NameInterface="EventService.P2.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="EventService.P3"/>
            <Interface Type="Required">
               <NameInterface NameInterface="EventService.P3.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="EventService.P4"/>
            <Interface Type="Required">
               <NameInterface NameInterface="EventService.P4.I1"/>
```

```
            </Interface>
         </Port>
      </Component>
      <Component>
         <NameComponent NameComponent="Patient"/>
         <Port>
            <NamePort NamePort="Patient.P1"/>
            <Interface Type="Required">
               <NameInterface NameInterface="Patient.P1.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Patient.P2"/>
            <Interface Type="Provided">
               <NameInterface NameInterface="Patient.P2.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Patient.P3"/>
            <Interface Type="Required">
               <NameInterface NameInterface="Patient.P3.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Patient.P4"/>
            <Interface Type="Provided">
               <NameInterface NameInterface="Patient.P4.I1"/>
            </Interface>
         </Port>
      </Component>
      <Component>
         <NameComponent NameComponent="Nurse"/>
         <Port>
            <NamePort NamePort="Nurse.P1"/>
            <Interface Type="Required">
               <NameInterface NameInterface="Nurse.P1.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Nurse.P2"/>
            <Interface Type="Required">
               <NameInterface NameInterface="Nurse.P2.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Nurse.P3"/>
            <Interface Type="Provided">
               <NameInterface NameInterface="Nurse.P3.I1"/>
            </Interface>
         </Port>
         <Port>
            <NamePort NamePort="Nurse.P4"/>
            <Interface Type="Required">
               <NameInterface NameInterface="Nurse.P4.I1"/>
            </Interface>
         </Port>
      </Component>
      <Connector Type="Assembly">
         <NameConnector NameConnector="EventService.P1.I1_TO"/>
      </Connector>
      <Connector Type="Assembly">
         <NameConnector NameConnector="Patient.P1_TO_EventService.P1"/>
      </Connector>
      <Connector Type="Assembly">
         <NameConnector NameConnector="Nurse.P1_TO_EventService.P2"/>
      </Connector>
      <Connector Type="Assembly">
         <NameConnector NameConnector="Patient.P3_TO_Nurse.P3"/>
```

```
99        </Connector>
100       <Connector Type="Assembly">
101          <NameConnector NameConnector="EventService.P3_TO_Patient.P2"/>
102       </Connector>
103       <Connector Type="Assembly">
104          <NameConnector NameConnector="EventService.P4_TO_Nurse.P2"/>
105       </Connector>
106    </ArchitecturalStyleFeature>
107    <Guards>
108       <Constraint>context EventService
109         inv NbEventService:
110         EventService.allInstances->size <= 3 and EventService.allInstances->size >0
111       </Constraint>
112       <Constraint>context EventService
113         inv NbEventServiceNurse:
114         EventService.allInstances->forAll(es:EventService | ((es.nurse?size <=5) and
115         (es.patient->size <=15)))
116       </Constraint>
117       <Constraint>context Patient
118         inv NbPatientEventService:
119         Patient.allInstances->forAll(p:Patient | ((p.eventService->size =1) and
120         (p.nurse->size =1)))
121       </Constraint>
122       <Constraint>context Nurse
123         inv NbNurseEventService:
124         Nurse.allInstances->forAll(n: Nurse | ((n.eventService->size =1) and
125         (n.patient->size<=3)))
126       </Constraint>
127       <Constraint>context EventService
128         inv ExNurseEventService:
129         EventService.allInstances->forAll(es | es.patient->notEmpty implies
130         es.nurse->notEmpty)
131       </Constraint>
132    </Guards>
133 </ArchitecturalStyleM-M>
```

Après la transformation vers XML, nous passons à la phase de validation. Celle-ci consiste à tester que les modèles générés (description XML) sont conformes à leurs méta-modèle (XML schéma).

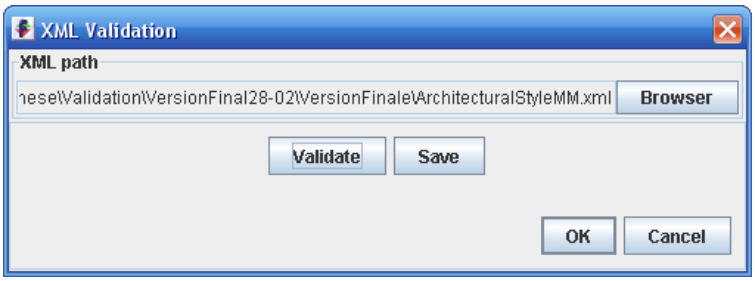

FIG. 6.15 – Validation des modèles XML par rapport à leur XML schéma

Les descriptions XML schéma (XSD) sont enregistrées dans un dossier de l'outil FUJABA. Pour montrer la validité d'une description XML par rapport à son XML schéma, nous utilisons des outils XML afin de vérifier la bonne formation de la description XML et pour montrer la conformité du modèle (description XML) par rapport à son méta-modèle

(description XML schéma). Il faut vérifier que la description XML est une instance de son XML schéma correspondant.

Dans l'entête de chaque description XML généré, le chemin permettant d'accéder au fichier XSD correspondant est ajouté d'une façon automatique dans l'attribut `xsi :noNamespaceSchemaLocation` (ligne 3).

```
1 <?xml version="1.0" encoding="UTF-8"?> <ArchitecturalStyleM-M
2 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xsi:noNamespaceSchemaLocation="ArchitecturalStyleMM.xsd">
4 ...
```

Le plug-in, grâce à des modules Java, permet aussi de vérifier les règles intra-modèle tels que l'unicité des noms des composants, des noms des connecteurs, des noms des ports, etc. Il automatise aussi les règles inter-modèles permettant le passage d'un modèle à un autre.

6.3.3 Transformation vers *Z*

Après la validation, nous passons à la phase de transformation des modèles XML vers le langage *Z*. Cette transformation est réalisée d'une façon automatique grâce à notre plug-in en cliquant sur le bouton *UML to Z*.

Afin d'implémenter le module *UML to Z* et de l'intégrer dans l'outil FUJABA, nous avons suivi les étapes suivantes :

– Développement des fichiers nécessaires : programmes java et classes d'action : "plug-in.xml" et "stable.xml".
  Le fichier "plug-in.xml" contient des informations générales sur le plug-in tels que son nom et sa version. Le fichier "stable.xml" définit les éléments à ajouter dans l'interface graphique de l'outil FUJABA.
– Déploiement des fichiers développés et création du fichier JAR qui encapsule les classes JAVA, ainsi que les classes d'action.
– Installation du plug-in.

Nous présentons ici, le fichier "stable.xml". Il permet de définir la barre des menus, les sous-menu, les menus flottants ainsi que les actions à exécuter.

```
1
2 <UserInterface>
3    <!-- ************************** List Actions here ******************** -->
4    <Action id="MyPlugin.newMyDiagram" class="de.upb.myplugin.actions.
5    NewArchitecturalDiagramAction" enabled="true">
6       <Name>New Architectural Diagram</Name>
7       <Mnemonic>w</Mnemonic>
8       <ToolTip>Create a new Architectural Diagram</ToolTip>
9       <Icon>de/upb/myplugin/images/none.gif</Icon>
10   </Action>
11   <Action id="MyPlugin.newMyComponent" class="de.upb.myplugin.actions.
12   NewMyComponentAction" enabled="true">
13      <Name>Create a new Component</Name>
```

```
      <Mnemonic>C</Mnemonic>
      <ToolTip>Create a new Component</ToolTip>
      <Icon>de/upb/myplugin/images/compo.gif</Icon>
   </Action>
   <Action id="MyPlugin.newMyConnection" class="de.upb.myplugin.actions.
   NewMyConnectionAction" enabled="true">
      <Name>Create an Assembly Connector</Name>
      <Mnemonic>O</Mnemonic>
      <ToolTip>Create an Assembly Connector</ToolTip>
      <Icon>de/upb/myplugin/images/assembly.gif</Icon>
   </Action>
   <Action id="MyPlugin.newMyDelegation" class="de.upb.myplugin.actions.
   NewMyDelegationAction" enabled="true">
      <Name>Create a Delegation Connector</Name>
      <Mnemonic>g</Mnemonic>
      <ToolTip>Create a Delegation Connector</ToolTip>
      <Icon>de/upb/myplugin/images/delegation.gif</Icon>
   </Action>
   ...
   <Action id="umltoz_id" class="de.upb.umltoz.actions.ShowUMLTOZDialog"
   enabled="true">
      <Name>UML to Z</Name>
      <Mnemonic>m</Mnemonic>
      <ToolTip>UML to Z</ToolTip>
   </Action>
   <!-- ************************** Define MenuBar here ******************** -->
   <MenuBar id="mainMenuBar">
      <MenuSection id="diagramsMenuSection">
         <Menu id="diagramsMenu">
            <MenuSection id="newDiagramSection">
               <MenuItem actionId="MyPlugin.newMyDiagram"/>
            </MenuSection>
         </Menu>
      </MenuSection>
      <MenuSection id="diagramSpecificSection">
         <Menu id="MyPlugin.myDiagramMenu" visible="false">
            <Name>Architectural Diagram</Name>
            <Mnemonic>M</Mnemonic>
            <MenuSection id="MyPlugin.myDiagramMenu.newSection">
               <MenuItem actionId="MyPlugin.newMyComponent"/>
               <MenuItem actionId="MyPlugin.newMyConnection"/>
               <MenuItem actionId="MyPlugin.newMyDelegation"/>
               ...
            </MenuSection>
         </Menu>
      <MenuSection id="diagramSpecificSection">
         <Menu id="Ztools_id" visible="true">
            <Name>Z tools</Name>
            <Mnemonic>z</Mnemonic>
            <MenuSection id="Ztools_id.umltoz">
               <MenuItem actionId="umltoz_id"/>
            </MenuSection>
         </Menu>
      </MenuSection>
   </MenuBar>
   <!-- ************************** List PopupMenus here ******************** -->
   <PopupMenu class="de.upb.myplugin.metamodel.MyComponent">
      <MenuSection id="de.upb.myplugin.menusection">
         <MenuItem actionId="MyPlugin.newMyComponent"/>
         <MenuItem actionId="MyPlugin.newMyPort"/>
         <MenuItem actionId="MyPlugin.newMyConstraint"/>
         <MenuItem actionId="MyPlugin.EditComponent"/>
      </MenuSection>
   </PopupMenu>
   <PopupMenu class="de.upb.myplugin.metamodel.MyConnection">
      <MenuSection id="de.upb.myplugin.bendmenusection">
         <MenuItem actionId="MyPlugin.newBend"/>
         <MenuItem actionId="MyPlugin.AddToInsert"/>
```

```
82          <MenuItem actionId="MyPlugin.AddToDelete"/>
83          <MenuItem actionId="deleteBend"/>
84       </MenuSection>
85    </PopupMenu>
86    ...
87    <!-- ************************** List ToolBars here ********************** -->
88    <ToolBar id="MyPlugin.myDiagramToolBar" rolloverButtons="true" visible="false">
89       <ToolBarSection id="MyPlugin.myDiagramToolBar.newSection">
90          <Button actionId="MyPlugin.newMyComponent"/>
91          <Button actionId="MyPlugin.newMyPort"/>
92          <Button actionId="MyPlugin.newMyInterfaceRequired"/>
93          <Button actionId="MyPlugin.newMyInterfaceProvided"/>
94          <Button actionId="MyPlugin.newMyConnection"/>
95          <Button actionId="MyPlugin.newMyDelegation"/>
96          <Button actionId="exportoXML"/>
97          ...
98       </ToolBarSection>
99    </ToolBar>
100 </UserInterface>
```

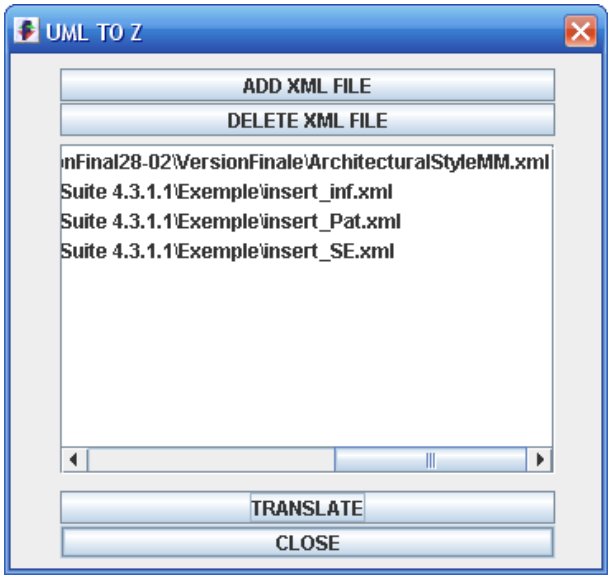

FIG. 6.16 – L'interface UML to *Z*

En cliquant sur le sous-menu *UML to Z*, l'interface "UML To Z" est affichée (figure 6.16). Cette fenêtre permet d'ajouter les descriptions XML (le style architectural et les différentes opérations de reconfiguration) afin de les transformer vers le langage *Z*.

Le module transformation vers *Z* reçoit en entrée une description XML. Il la transforme selon les règles de transformation et les grammaires définies pour générer un fichier LaTeX (.tex) selon une structure bien déterminée.

Nous avons choisi LaTeX comme format de sortie car il est supporté par l'outil de vérification Z/EVES. La figure 6.17 décrit la transformation du style architectural vers le langage *Z*.

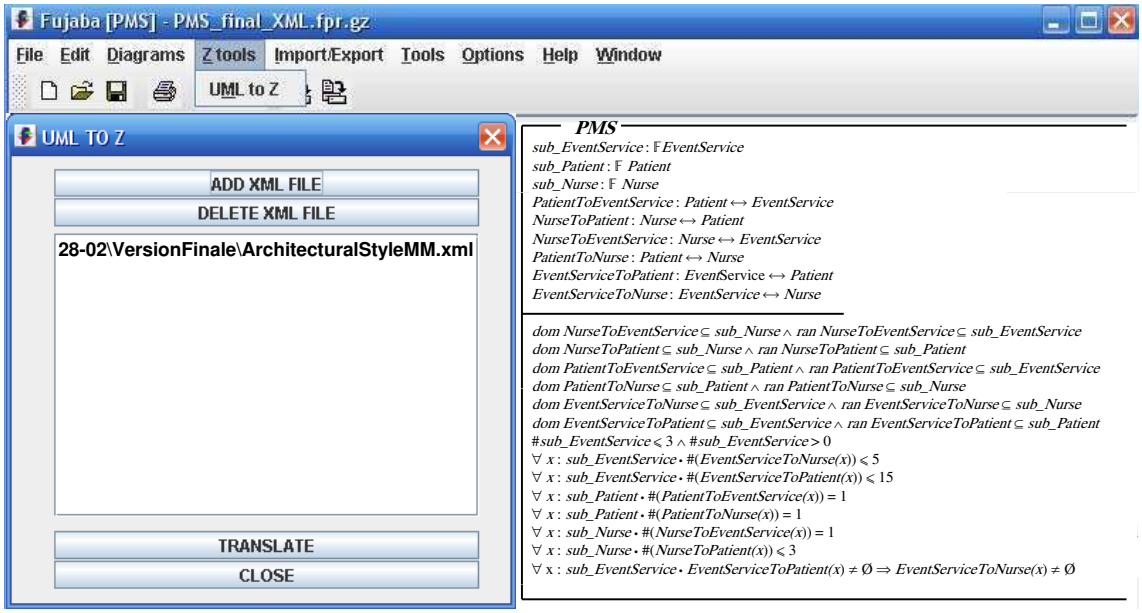

FIG. 6.17 – Transformation du style architectural vers le langage *Z*

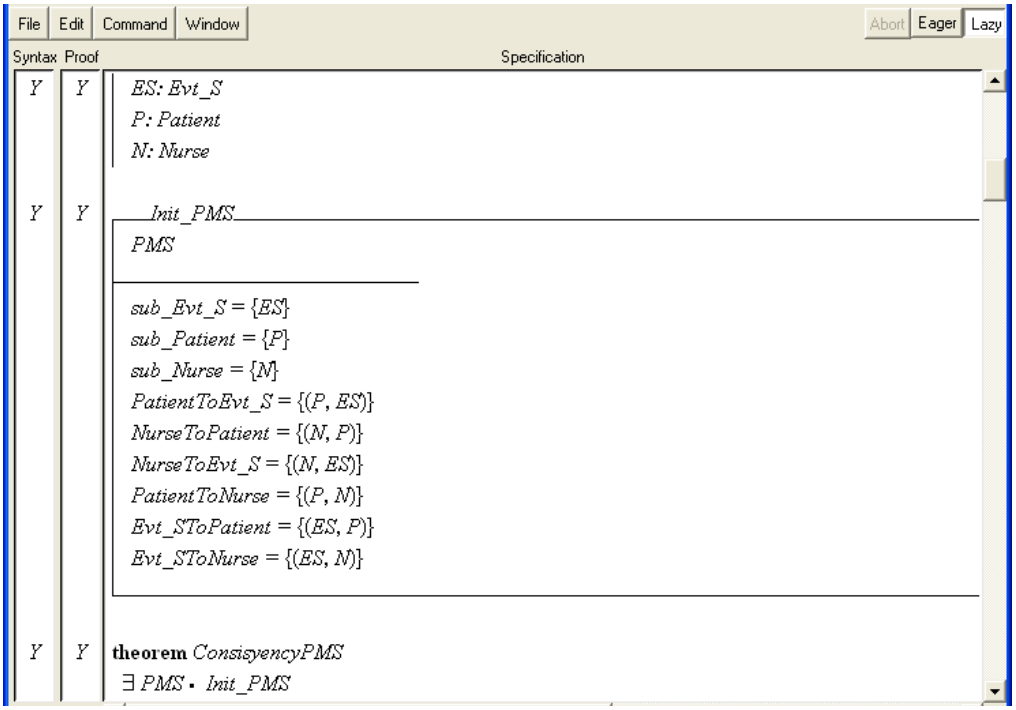

FIG. 6.18 – Vérification et preuve avec Z/EVES

Après la génération, les fichiers LaTeX seront reproduits pour y ajouter les théorèmes afin de les prouver en utilisant l'outil Z/EVES (figure 6.18).

6.4 Conclusion

Nous avons présenté dans ce chapitre, une démarche de modélisation des architectures logicielles dynamiques. Cette démarche appelée X permet de décrire les différentes étapes pour modéliser l'architecture logicielle. Elle propose des règles de passage d'un modèle à un autre. Elle automatise le processus de validation et assiste l'utilisateur pour vérifier la consistance du style architectural et la conformité de l'évolution de l'architecture vis-à-vis de son style architectural. La démarche proposée est inspirée de l'approche MDA et 2TUP du processus UP.

Pour que notre approche soit exploitée, elle doit offrir aux utilisateurs les moyens pour la manipuler. En effet, le profil proposé, l'approche de validation et de vérification ainsi que la démarche X ont fait l'objet d'une implémentation et d'une intégration, sous forme de plug-ins java, dans l'atelier d'aide à la conception FUJABA. Les plug-ins implémentés sont disponibles sur l'URL : http ://www.laas.fr/∼khalil/TOOLS/X.zip

Conclusion

Bilan des contributions

Dans ce mémoire, nous avons présenté l'ensemble des travaux réalisés dans le cadre de cette thèse. Nous avons passé en revue l'état de l'art et nous avons étudié la littérature concernant les domaines de recherche abordés. Nous avons introduit les concepts relatifs aux architectures dynamiques en nous basant sur les standards et les travaux de recherche. Nous avons présenté les langages de description des architectures les plus utilisés et les plus publiés. Nous avons mis l'accent sur les mécanismes de spécification définis, afin de mettre en relief les avantages et les limites de ces différents langages. Nous avons présenté également les travaux sur la reconfiguration des architectures, au moyen d'UML et des techniques de réécriture de graphes. Enfin, nous avons présenté un bilan des travaux qui proposent des approches diverses et combinées.

Grace à cette étude et afin de réduire les limites des autres approches et afin de proposer une solution en vue de gérer l'évolution dynamique des architectures logicielles, conformes à l'approche guidée par les modèles, nous avons proposé une approche qui étend la notation UML 2.0. C'est une approche de description orientée règles, basée sur la transformation de graphe, permettant, ainsi, de décrire la dynamique structurelle sous forme d'opérations de reconfiguration.

Le Profil UML proposé apporte plusieurs solutions aux différentes limites identifiées pour le traitement des architectures dynamiques. Ce profil, offre trois méta-modèles et utilise une notation unifiée et graphique basée sur la notation UML 2.0. Ce Profil utilise les grammaires de graphes offrant ainsi la possibilité de décrire l'aspect dynamique. Le premier méta-modèle étend le diagramme de composants et décrit la structure de l'architecture en termes de composants et de connexions entre les composants. Le deuxième étend le premier méta-modèle et décrit la dynamique structurelle de l'architecture en termes d'opérations de reconfiguration. Dans notre approche, nous nous sommes intéressés uniquement à la dynamique structurelle (ajout et/ou suppression de composants et/ou de connexions). Le troisième méta-modèle étend le diagramme d'activités et permet de décrire l'enchaînement et l'ordre d'exécution des opérations de reconfiguration.

Afin de valider les modèles construits nous avons proposé une approche de validation à base de règles intra-modèle et de règles inter-modèles. Ces règles sont utilisées pour faciliter l'identification des éventuelles incohérences afin de détecter toute utilisation in-

correcte du profil proposé. La validation que nous avons proposée est structurelle et à base du langage XML.

De même, nous avons élaboré une technique de vérification. Elle assure, dans une première étape, la transformation d'une spécification, à partir du méta-modèle Style Architectural et Opération de Reconfiguration, au langage *Z*. Les spécifications *Z* générées permettent, dans une deuxième étape, de vérifier la consistance et la conformité de l'architecture. Nous avons proposé également une démarche de modélisation permettant de décrire les différentes étapes pour modéliser la dynamique de l'architecture logicielle. Elle propose des règles de passage d'un modèle à un autre, elle automatise le processus de validation et assiste l'utilisateur pour vérifier certaines propriétés architecturales.

Le profil, l'approche de validation et de vérification ainsi que la démarche X proposés ont fait l'objet d'une implémentation et d'une intégration, sous forme de plug-in java, dans l'atelier d'aide à la conception FUJABA. Les plug-ins implémentés sont disponibles sur l'URL : http ://www.laas.fr/∼khalil/TOOLS/X.zip.

Perspectives

Les perspectives des travaux présentés dans ce mémoire suivent différents axes de réflexion et se situent dans les différents contextes de recherche.

A court terme, nous prévoyons d'améliorer l'approche de vérification en ajoutant d'autres propriétés à vérifier autres que la consistance et la conformité des opérations de reconfiguration par rapport au style architectural.

Pour la partie protocole de reconfiguration, nous prévoyons d'achever la partie transformation permettant de traduire la description XML générée vers un langage formel. Nous prévoyons d'utiliser les réseaux de pétri pour vérifier la vivacité des différentes opérations de reconfiguration et que chaque opération de reconfiguration sera exécutée au moins une fois.

Nous prévoyons de travailler sur la branche fonctionnelle de la démarche X, afin de générer par raffinement un code source conformément à un modèle de composant choisi. L'application sera déployée pour tester et gérer l'évolution de l'architecture d'une façon dynamique.

A moyen terme, nous prévoyons d'utiliser les politiques d'adaptation. Notre approche sera guidée par un modèle de contexte et des politiques d'adaptation (pour décrire les règles de reconfiguration). Ces politiques seront décrites sous la forme E-C-A (on Evénement if Condition then Action) et en utilisant un ensemble d'opérations de reconfiguration (ajout et suppression de composants et ajout et suppression de connexions). Ces politiques seront décrites dans une description XML séparée de l'application, ce qui permet sa modification dynamique (même pendant l'exécution de l'application). Les plans de reconfiguration seront exprimés avec un langage formel afin de garantir leurs validités, leurs consistances et leurs cohérences.

Afin de rester dans un cadre compatible avec la réutilisation des notations UML, nous prévoyons de confronter le profil proposé par l'utilisation du paquetage *Profiles* des versions récentes d'UML.

A long terme, nous prévoyons d'intégrer notre approche avec les autres travaux réalisés au sein de notre unité de recherche ReDCAD de Sfax et les autres travaux réalisés au sein de l'équipe OLC de LAAS-CNRS. Cette intégration permettra d'avoir une approche complète assurant la modélisation, la validation, la vérification, la génération de code exécutable, le déploiement de l'application et le contrôle (monitoring) dans le but d'assurer l'auto-réparation.

Publications de l'auteur

– Mohamed Hadj Kacem, Mohamed Jmaiel, and khalil Drira. Vers un environnement d'administration des applications coopératives distribuées. In *GEI'02 : Proceedings des* 2^{eme} *Journées Scientifiques des Jeunes Chercheurs en Génie Electrique et Informatique*, page 4p, Hammamet, Tunisia, 2002.

– Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and khalil Drira. Environnement pour l'administration des applications distribuées à base de composants. In *JSF'03 : Proceedings des Journées Scientifiques Francophones*, Tozeur, Tunisia, 2003.

– Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed hadj Kacem, and khalil Drira. Evaluation and comparison of adl based approaches for the description of dynamic software architectures. In *ICEIS'05 : Proceedings of the* 7^{th} *International Conference on Enterprise Information Systems*, pages 189–195, Miami, USA, 2005. INSTICC Press.

– Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and khalil Drira. Using UML2.0 and GG for describing the dynamic of software architectures. In *ICITA'05 : Proceedings of the Third International Conference on Information Technology and Applications*, pages 46–51, Sydney, Australia, 2005. IEEE Computer Society.

– Mohamed Hadj Kacem, Mohamed Nadhmi Miladi, Mohamed Jmaiel, Ahmed Hadj Kacem, and khalil Drira. Towards a UML profile for the description of dynamic software architectures. In *COEA'05 : Proceedings of the International Conference on Component-Oriented Enterprise Applications*, volume 70 of *Lecture Notes in Informatics*, pages 25–39, Erfurt, Germany, 2005.

– Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and khalil Drira. Describing dynamic software architectures using an extended UML model. In *SAC'06 : Proceedings of the* 21^{st} *Annual Symposium on Applied Computing, Track-Model Transformation*, pages 23–27, Dijon, France, 2006. ACM.

– Mohamed Hadj Kacem, Mohamed Jmaiel, , Ahmed Hadj Kacem, and khalil Drira. An UML-based approach for validation of software architecture descriptions. In *TEAA'06 : Proceedings of the* 2^{nd} *International Conference on Trends in Enterprise*

Application Architecture, volume 4473 of *Lecture Notes in Computer Science*, pages 158–171, Berlin, Germany, 2007. Springer.

– Mohamed Nadhmi Miladi, Mohamed Jmaiel, and Mohamed Hadj Kacem. A UML profile and a FUJABA plugin for modelling dynamic software architectures. In *MoDSE'07 : Proceedings of the Workshop on Model-Driven Software Evolution*, pages 20–23, Amsterdam, Netherlands, 2007. IEEE Computer Society.

– Mohamed Nadhmi Miladi, Mohamed Hadj Kacem, Achraf Bouchriss, Mohamed Jmaiel, and khalil Drira. A UML rule-based approach for describing and checking dynamic software architectures. In *AICCSA'08 : Proceedings of the* 6^{th} *ACS/IEEE International Conference on Computer Systems and Applications*, Doha, Qatar, 2008. IEEE Computer Society.

Bibliographie

[1] Gregory Abowd, Robert Allen, and David Garlan. Using + style to understand descriptions of software architecture. *SIGSOFT Software Engineering Notes*, 18(5) :9–20, 1993.

[2] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4) :319–364, 1995.

[3] Jean-Raymond Abrial. Programming as a mathematical exercise. In Charles Antony Richard Hoare and John C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 113–137. Prentice-Hall International, 1985.

[4] Accord. Assemblage de composants par contrats en environnement ouvert et réparti, etat de l'art sur les langages de description d'architecture (ADLs). Projet ACCORD, Technical Report Livrable 1.1-2, RNTL, France, Juin 2002.

[5] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *ASE'02 : Proceedings of the* 17^{th} *IEEE international conference on Automated software engineering, 23-27 September 2002, Edinburgh, Scotland, UK*, pages 271–274, Edinburgh, Scotland, UK, September 2002.

[6] David Akehurst and Octavian Patrascoiu. OCL 2.0 - implementing the standard for multiple metamodels. UML 2003 preliminary version, technical report, Computing Laboratory, University of Kent, Canterbury, UK, Janvier 2003.

[7] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *ECOOP'02 : Proceedings of the* 16^{th} *European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer.

[8] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *FASE'98 : Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering*, pages 21–37, Lisbon, Portugal, March 1998.

[9] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

[10] Sylvain Andre. MDA (model driven architecture) principes et états de l'art. Technical report, CNAM, 05 Novembre 2004.

[11] Egidio Astesiano and Gianna Reggio. Towards a well-founded UML-based development method. pages 102–115, Brisbane, Australia, 2003. IEEE Computer Society.

[12] Thomas Baar. OCL and graph-transformations - a symbiotic alliance to alleviate the frame problem. In *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 20–31. Springer, 2005.

[13] Olivier Barais. Approche statique, dynamique et globale de l'architecture d'applications réparties. Master report, Ecole Mine de Douai, Laboratoire d'Informatique fondamentale de Lille, Juillet 2002.

[14] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Modeling and validation of service-oriented architectures : application vs. style. In *ESEC/FSE-11 : Proceedings of the* 9^{th} *European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 68–77, New York, NY, USA, 2003. ACM Press.

[15] Luciano Baresi, Reiko Heckel, Sebastian Thone, and Daniel Varro. Style-based refinement of dynamic software architectures. In *WICSA'04 : Proceedings of the* 4^{th} *Working International IEEE/IFIP Conference on Software Architecture*, pages 155–164, Oslo, Norway, 2004. IEEE Computer Society.

[16] Luc Bellissard, Slim Ben Atallah, Alain Kerbrat, and Michel Riveill. Component-based programming and application management with Olan. In *OBPDC'95 : Object-Based Parallel and Distributed Computation*, pages 290–309, 1995.

[17] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2) :201–227, 1996.

[18] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

[19] Morgan Björkander and Cris Kobryn. Architecting systems with UML 2.0. *IEEE Software*, 20(4) :57–61, 2003.

[20] Xavier Blanc. *MDA en action : Ingénierie logicielle guidée par les modèles*. Eyrolles, 2005.

[21] Maarten Boasson. The artistry of software architecture. *IEEE Software*, 12(6) :13–16, 1995.

[22] Behzad Bordbar, Luisa Giacomini, and D.J. Holding. UML and petri nets for design and analysis of distributed systems. In *Proceedings of the 2000 IEEE International Conference on Control Applications*, pages 610–615, Alaska, USA, 2000.

[23] Lopez Pere Botella, Franch-Gutiérrez Xavier, and Josep M. Ribó-Balust. Software process modelling languages based on UML. *The European Journal for the Informatics Professional*, 5(5) :34–39, 2004.

[24] Thouraya Bouabana-Tebibel and Mounira Belmesk. Formalization of UML object dynamics and behavior. In *SMC'05 : Proceedings of the IEEE International Conference on Systems, Netherlands, 10-13 October*, pages 4971–4976. IEEE, 2004.

[25] Pearl Brereton and David Budgen. Component-based systems : A classification of issues. *IEEE Computer*, 33(11) :54–62, 2000.

[26] Jean-Michel Bruel. *UML et Modélisation de la Composition Logicielle*. Habilitation à diriger des recherches, Université de Pau et des Pays de l'Adour, Université de Pau et des Pays de l'Adour, Décembre 2006.

[27] Doug Bryan. Using rapide to model and specify inter-object behavior. In *OOPSLA'94 : Proceedings of the Workshop on Precise Behavioral Specifications in OO Information Modeling*, volume 29 de SIGPLAN Notices, pages 579–595. ACM Press, Mai 1994.

[28] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The fujaba real-time tool suite : Model-driven development of safety-critical, real-time systems. In *ICSE'05 : Proceedings of the* 27^{th} *International Conference on Software Engineering , St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[29] Jean Bézivin, Jean-Paul Bouchet, and Erwan Breton. Correspondance structurelle entre produits et procédés : un pattern classique, analysé avec des méta-modèles explicites. In *Journée GRACQ : Modèles, Objets et composants.*, pages 1–22, Juin 1999.

[30] Cyril Carrez. *Contrats Comportementaux pour Composants.* Phd thesis, l'école Nationale Supérieure des Télécommunications, Spécialité : Informatique et Réseaux, Décembre 2003.

[31] Christelle Chaudet. *P-Space : Langage et outils pour la description d'architectures évolutives à composants dynamiques. Formalisation d'architectures logicielles et industrielles.* Phd thesis, Université de Savoie, Décembre 2002.

[32] Christelle Chaudet and Flavio Oquendo. P-space : A formal architecture description language based on process algebra for evolving software systems. In *ASE'00 : Proceedings of the* 15^{th} *IEEE international conference on Automated software engineering*, pages 245–248, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Christelle Chaudet and Flavio Oquendo. P-space : Modeling evolvable distributed software architectures. In *PDPTA'01 : Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Monte Carlo Resort, Las Vegas, Juin 2001.

[34] B. Chen and S. Sadaoui. Lotos : Theories and tools. Technical report, Department of Computer Science, U. of Regina, November 2003.

[35] Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors. *Agile Processes in Software Engineering and Extreme Programming*, volume 4536 of *Lecture Notes in Computer Science*, Italy, June 2007. Springer.

[36] Steve Cook. The UML family : Profiles, prefaces and packages. In *UML'00 : Proceedings of the* 3^{rd} *International Conference on The Unified Modeling Language. Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 255–264, York, UK, 2000. Springer.

[37] V. Cortellessa, A. Di Marco, P. Inverardi, H. Muccini, and P. Pelliccione. Using uml for sa-based modeling and analysis. In *Proceedings of the Modeling Languages and Applications, UML Satellite Activities*, Lecture Notes in Computer Science, page 15. Springer, 2005.

[38] Thierry Coupaye, Romain Lenglet, Mikael Beauvois, Eric Bruneton, and Pascal Déchamboux. Composants et composition dans l'architecture des systèmes répartis. Octobre 2001.

[39] Joëlle Coutaz and Laurence Nigay. Architecture logicielle conceptuelle des systèmes interactifs. *Analyse et conception de l'IHM. Hermés*, chapitre 7 :207–246, 2001.

[40] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and M. Encarnación Beato. Coordination in a reflective architecture description language. In *COORDINATION'02 : Proceedings of the* 5^{th} *International Conference on Coordination Models and Languages*, pages 141–148, London, UK, 2002. Springer-Verlag.

[41] Hirsch Dan, Inverardi Paola, and Montanari Ugo. Modeling software architectures and styles with graph grammars and constraint solving. In *WICSA'99 : Proceedings of the* 1^{st} *Working IFIP Conference on Software Architecture*, pages 127–142, San Antonio, Texas, February 1999. Kluwer Academic Publishers.

[42] Desmond D'Souza, Aamond Sane, and Alan Birchenough. First-class extensibility for UML-profiles, stereotypes, patterns. volume 1723 of *Lecture Notes in Computer Science*, pages 265–277, Fort Collins, CO, USA, 28-30 October 1999. Springer.

[43] Yael Dubinsky. Teaching extreme programming in a project-based capstone course. In *XP'03 : Proceedings of the* 4^{th} *International Conference on Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 435–436. Springer, 2003.

[44] Domiczi Endre. OctoGuide - a graphical aid for navigating among octopus/UML artifacts. In *Proceedings of the ECOOP'98 : Workshop on Object-Oriented Technology*, page 560, London, UK, 1998. Springer-Verlag.

[45] Claudia Ermel, Roswitha Bardohl, and Julia Padberg. Visual design of software architecture and evolution based on graph transformation. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001.

[46] Cléver R. Guareis De Farias, Luis Ferreira Pires, Wanderley Lopes de Souza, and Célio Estevan Moron. Specification and validation of a real-time parallel kernel using LOTOS. In *MASCOTS '01 : Proceedings of the* 9^{th} *International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 7–14, Washington, DC, USA, 2001. IEEE Computer Society.

[47] Roy T. Fielding. Software architectural styles for network-based applications. Phase ii survey paper, draft 1.1, University of California, Irvine, July 15 1999.

[48] Martin Fowler. *UML 2.0.* CompusPress, Paris, 1^{ere} édition, 5^{eme} tirage edition, 2004.

[49] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Malcom Shroff. Exploring the semantics of UML type structures with *Z*. In *FMOODS'97 : Proceedings of the International workshop on Formal methods for open object-based distributed systems*, pages 247–257, London, UK, 1997. Chapman and Hall, Ltd.

[50] David Garlan. Software architecture : a roadmap. In *ICSE'00 : Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, USA, 2000. ACM Press.

[51] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT'94 : Proceedings of the* 2^{nd} *ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.

[52] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44(1) :23–49, 2002.

[53] David Garlan, Robert T. Monroe, and David Wile. Acme : architectural description of component-based systems. pages 47–67, 2000.

[54] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[55] Martin Gogolla, Fabian Büttner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 1(69) :27–34, 2007.

[56] Karin Guennoun, Khalil Drira, and Michel Diaz. Une approche orientée modèle pour la gestion des architectures logicielles distribuées dynamiques. *Information Sciences for Decision Making*, 1(19) :50–58, 2005.

[57] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Evaluation and comparison of ADL based approaches for the description of dynamic of software architectures. In *ICEIS'05 : Proceedings of the* 7^{th} *International Conference on Enterprise Information Systems.*, pages 189–195, Miami, USA, May 2005. INSTICC Press.

[58] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Using UML2.0 and GG for describing the dynamic of software architectures. In *ICITA'05 : Proceedings of the* 3^{rd} *International Conference on Information Technology and Applications*, volume 2, pages 46–51, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Describing dynamic software architectures using an extended UML model. In *SAC'06 : Proceedings of the* 21^{st} *Annual Symposium on Applied Computing, Track - Model Transformation*, volume 2, pages 1245–1249, Dijon, France, April 2006. ACM SIG Proceedings.

[60] Mohamed Hadj Kacem, Mohamed Nadhmi Miladi, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Towards a UML profile for the description of dynamic software architectures. In *COEA'05 : Proceedings of the International Conference on Component-Oriented Enterprise Applications*, pages 25–39, Erfurt, Germany, September 2005. Lecture Notes in Computer.

[61] Reiko Heckel, Alexey Cherchago, and Marc Lohmann. A formal approach to service specification and matching based on graph transformation. *Electronic Notes in Theoretical Computer Science*, 105 :37–49, 2004.

[62] Gerald H. Hilderink. Graphical modelling language for specifying concurrency based on CSP. *IEE Proceedings - Software*, 150(2) :108–120, 2003.

[63] Christine Hofmeister and Robert L. Nord. From software architecture to implementation with UML. In *COMPSAC '01 : Proceedings of the* 25^{th} *Annual International Computer Software and Applications Conference on Invigorating Software Development*, pages 113–114, Washington, DC, USA, October 2001. IEEE Computer Society.

[64] Vladimir Issarny, Luc Bellissard, Michel Riveill, and Apostolos Zarras. Component-based programming of distributed applications. *Journal of Distributed Systems*, 1752/2000 :327–353, 2000.

[65] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Le processus unifié de développement logiciel*. 1ère édition edition, juin 2000.

[66] Ackbar Joolia, Thais Batista, Geoff Coulson, and Antonio Tadeu A. Gomes. Mapping ADL specifications to an efficient and reconfigurable runtime component platform. In *WICSA'05 : Proceedings of the* 5^{th} *Working IEEE/IFIP Conference on Software Architecture*, pages 131–140, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Hubert Kadima. *MDA Conception orientée objet guidée par les modèles*. Dunod edition, Mars 2005.

[68] Mohamed Mancona Kandé, Valentin Crettaz, Alfred Strohmeier, and Shane Sendall. Bridging the gap between IEEE 1471, an architecture description language, and UML. *Software and System Modeling*, 1(2) :113–129, 2002.

[69] Mohamed Mancona Kandé and Alfred Strohmeier. Towards a UML profile for software architecture descriptions. In *UML'00 : Proceedings of the* 3^{rd} *International Conference on Unified Modeling Language. Advancing the Standard, York, UK*, volume 1939 of *LNCS*, pages 513–527. Springer, 2000.

[70] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Adaptation dynamique concepts et expérimentations. In *ICSSEA'02 : Proceedings of the* 15^{th} *International Conference Software, Systems and their Applications*, CNAM Paris, France, Décembre 2002.

[71] Jeremy Kivi, Darlene Haydon, Jason Hayes, Ryan Schneider, and Giancarlo Succie. Extreme programming : A university team design experience. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, volume 2, pages 816–820, 2000.

[72] Hans J. Kohler, Ulrich Nickel, Jorg Niere, and Albert Zundorf. Integrating UML diagrams for production control systems. In *ICSE'00 : Proceedings of the* 22^{nd} *international conference on Software engineering*, pages 241–251, New York, NY, USA, 2000. ACM Press.

[73] Jun Kong, Kang Zhang, Jing Dong, and Guanglei Song. A graph grammar approach to software architecture verification and transformation. In *COMPSAC'03 : Proceedings of the* 27^{th} *Annual International Computer Software and Applications Conference*, pages 492–497. IEEE Computer Society, 2003.

[74] Pierre Kroll and Philippe Kruchten. *Guide pratique du RUP*. CampusPress, 2003.

[75] Philippe Kruchten. *The Rational Unified Process : An Introduction*. Addison-Wesley, Boston, USA, second edition edition, 2000.

[76] Phipippe Kruchten, Bran Selic, Grant Larsen, and Alan Brown. Describing software architecture with UML. In *ICSE'01 : Proceedings of the* 23^{rd} *International Conference on Software Engineering*, pages 715–716, Washington, DC, USA, 2001. IEEE Computer Society.

[77] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jorg Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *IFM'02 : Proceedings of the* 3^{rd} *International Conference on Integrated Formal Methods*, pages 11–28, London, UK, 2002. Lecture Notes in Computer Science, Springer-Verlag.

[78] Daniel LeMétayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7) :521–533, 1998.

[79] Bass Len, Clements Paul, and Kazman Rick. *Software Architecture in Practice*. Addison-Wesley, second edition edition, April 2003.

[80] Fabien Leymonerie. *ASL : un langage et des outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques.* Phd thesis, Université de Savoie, Décembre 2004.

[81] Fabien Leymonerie, Sorana Cimpan, and Flavio Oquendo. Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux : Application à j2ee. In *ICSSEA'01 : Proceedings of the* 14^{th} *International Conference on Software and Software Engineering and their Applications*, Paris, Décembre 2001.

[82] Denivaldo Cicero Pavao Lopes. *Etude et applications de l'approche MDA pour des plates-formes de Services Web.* Phd thesis, Université de Nantes, Ecole doctorale sciences et technologies de l'information et des matériaux, 2005.

[83] Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Towards a unified graph-based framework for dynamic component-based architectures description in z. In *ICPS'04 : Proceedings of the IEEE/ACS International Conference on Pervasive Services*, pages 227–234. IEEE Computer Society, 2004.

[84] Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Formal design of structural and dynamic features of publish/subscribe architectural styles. In *ECSA'07 : Proceedings of the* 1^{st} *European Conference on Software Architecture,*

Aranjuez, Spain, September 24-26, volume 4758 of *Lecture Notes in Computer Science*, pages 44–59. Springer, 2007.

[85] David C Luckham. Rapide : A language and toolset for simulation of distributed systems by partial orderings of events. *IEEE Transactions on Software Engineering*, 1996.

[86] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, September 1995.

[87] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical Report CSL-TR-95-674, Stanford, CA, USA, 1995.

[88] Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed programs. In *IWCCS'94 : Proceedings of the IEEE Workshop on Configurable Distributed Systems*, North Falmouth, Massachusetts, USA, Mars 1994.

[89] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2) :73–82, Mars 1993.

[90] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Software Engineering Notes*, 21(6) :3–14, 1996.

[91] Nenad Medvidovic. ADLs and dynamic architecture changes. In *ISAW'96 : Proceedings of the* 2^{nd} *International software Architecture Workshop*, pages 24–27, New York, USA, 1996. ACM Press.

[92] Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *SIGSOFT Software Engineering Notes*, 22(3) :190–198, 1997.

[93] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11(1) :2–57, 2002.

[94] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, Janvier 2000.

[95] Irwin Meisels and Mark Saaltink. The Z/EVES Reference Manual (for Version 1.5). Reference manual, ORA Canada, 1997.

[96] Francisco José MOO MENA. *Modelisation Des Architectures Logicielles Dynamiques, Application à la gestion de la qualité de service des applications à base de services Web*. Phd thesis, Institut National Polytechnique de Toulouse, Avril 2007.

[97] Mohamed Nadhmi Miladi. Profil UML pour la modèlisation des architectures logicielles à base de composants. Master report, Université de Sfax, Ecole Nationale d'Ingénieurs de Sfax, Juillet 2005.

[98] Moda-Tel. *(Deliverable 3.3), MDA modelling and application principles*. MODA-TEL, en ligne edition, Mai 2004. Disponible sur :http ://www.modatel.org/ Modatel/pub/deliverables/D3.3-final.pdf.

[99] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Transactions on Software Engineering*, 14(1) :43–52, 1997.

[100] Ronald Morrison, Graham N. C. Kirby, Dharini Balasubramaniam, Kath Mickan, Flavio Oquendo, Sorana Cimpan, Brian Warboys, Bob Snowdon, and R. Mark Greenwood. Support for evolving software architectures in the archware adl. In *WICSA'04 : Proceedings of the* 4^{th} *Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 69–78, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, Paris, $2^{`eme}$ édition, $5^{`eme}$ tirage edition, 2004.

[102] Mahmoud Nassar. *Analyse/conception par points de vue : le profil VUML*. Phd thesis, Institut national polytechnique de Toulouse, école doctorale : Informatique et Télécommunications, Septembre 2005.

[103] Sylvain NICLOT. Alternative à RUP : dX. Technical report, Conservatoire National des Arts et Métiers Centre Enseignement Principal de Tours, 2005.

[104] Elisabetta Di Nitto and David Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *ICSE'99 : Proceedings of the* 21^{st} *international conference on Software engineering*, pages 13–22, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[105] Goaer Olivier. De l'adaptation des composants logiciels vers leur évolution. Mastersthesis, Laboratoire d'Informatique de Nantes Atlantique, 7 septembre 2005.

[106] OMG. Meta object facility (MOF) specification (version 1.3). OMG document, Object Management Group, ftp ://ftp.omg.org/pub/docs/formal/00-04-03.pdf, March 2000.

[107] OMG. MDA guide version 1.0.1, document number : omg/2003-06-01. OMG document, June 2003.

[108] OMG. UML 2.0 OCL specification. OMG document, url : http ://www.omg.org, Octobre 2003.

[109] OMG. UML 2.0 superstructure specification, final adopted specification. Omg document, August 2003.

[110] OMG. The unified modelling language 2.0 - object constraint language 2.0 proposal. OMG document, url : http ://www.omg.org, 2003.

[111] Flavio Oquendo, Brian Warboys, Ronald Morrison, Regis Dindeleux, Ferdinando Gallo, Hubert Garavel, and Carmen Occhipinti. Archware : Architecting evolvable software. In *EWSA'04 : Proceedings of* 1^{st} *European Workshop Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2004.

[112] Noel De Palma, Philippe Laumay, and Luc Bellissard. Ensuring dynamic reconfiguration consistency. In *WCOP'01 : Proceedings of the* 6^{th} *International Workshop on Component-Oriented Programming, ECOOP related Workshop*, Udapest - Hungary, Juin 2001.

[113] Virginia C. Carneiro De Paula, George R. Ribeiro-Justo, and P. R. F. Cunha. Specifying and verifying reconfigurable software architectures. In *PDSE*, pages 21–31, 2000.

[114] Jennifer Pérez, Nour Ali, José A. Carsí, and Isidro Ramos. Dynamic evolution in aspect-oriented architectural models. In *EWSA'05 : Proceedings of the* 2^{nd} *European Workshop Software Architecture, Pisa, Italy, June 13-14, 2005*, volume 3527 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 2005.

[115] Jorge Enrique Pérez-Martinez. Heavyweight extensions to the UML metamodel to describe the C3 architectural style. *SIGSOFT Software Engineering Notes*, 28(3) :5–5, 2003.

[116] Jorge Enrique Pérez-Martínez and Almudena Sierra-Alonso. UML 1.4 versus UML 2.0 as languages to describe software architectures. In *EWSA'04 : Proceedings of* 1^{st} *European Workshop Software Architecture, EWSA 2004, St Andrews, UK, May 21-22*, volume 3047 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2004.

[117] Robert G. Pettit and Hassan Gomaa. Improving the reliability of concurrent object-oriented software designs. In *WORDS'03 : Proceedings of the* 9^{th} *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 262–269, Italy, 2003. IEEE Computer Society.

[118] Manuele Kirsch Pinheiro, Tiago Lopes Telecken, Carlos Mario Dal Col Zeve, José Valdeni de Lima, and Nina Edelweiss. A cooperative environment for e-learning authoring. In *Document numérique*, volume 5 of *4*, pages 89–114, 2001.

[119] Vivien Quema. Configuration d'un middleware dirigée par les applications. Master report, Université Joseph Fourier, Institut National Polytechnique de Grenoble, Juin 2002.

[120] Rapide. Guide to the rapide 1.0 language reference manuals. Technical report, Group Computer Systems Lab Stanford University, Juillet 1997.

[121] George R. Ribeiro-Justo, Virginia C. Carneiro de Paula, and Paulo Roberto Freire Cunha. Formal specification of evolving distributed software architectures. In *DEXA'98 : Proceedings of the* 9^{th} *International Workshop on Database and Expert Systems Applications*, pages 548–553, Washington, DC, USA, 1998. IEEE Computer Society.

[122] Josep M. Ribo and Xavier Franch. A two-tiered methodology to extend the UML metamodel. Research report, Universitat Politecnica de Cataluna, Departament de Llenguatges i Sistemes Informàtics, 2002.

[123] Michel Riveill and Aline Senart. Aspects dynamiques des langages de description d'architecture logicielle. *L'Objet RTSI - L'objet. Coopération et systèmes à objets*, 8(3), 2002.

[124] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. In *ICSE'98 : Proceedings of the* 20^{th} *international conference on Software engineering*, pages 209–218, Washington, DC, USA, 1998. IEEE Computer Society.

[125] Etienne Roblet, Khalil Drira, and Michel Diaz. Formal design and development of a corba-based application for cooperative HTML group editing support. *Journal of Systems and Software*, 60(2) :113–127, 2002.

[126] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture modeling language based on UML2.0. In *APSEC'04 : Proceedings of the* 11^{th} *Asia-Pacific Software Engineering Conference*, pages 663–669. IEEE Computer Society, 2004.

[127] Pascal Roques and Franck Vallée. *UML 2 en action : De l'analyse des besoins à la conception J2EE*. Eyrolles, Juin 2004.

[128] Gwen Salaun, Michel Allemand, and Christian Attiogbe. A method to combine any process algebra with an algebraic specification language : the p-calculus example. In *COMPSAC'02 : Proceedings of the* 26^{th} *International Computer Software and Applications Conference on Prolonging Software Life : Development and Redevelopment*, pages 385–392, Washington, DC, USA, 2002. IEEE Computer Society.

[129] Petri Selonen and Jianli Xu. Validating uml models against architectural profiles. In *ESEC/FSE-11 : Proceedings of the* 9^{th} *European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 58–67, New York, NY, USA, 2003. ACM Press.

[130] Frédérick Seyler. *UGATZE : Méta-modélisation pour la réutilisation de composants hétérogènes distribues*. Phd thesis, L'Université de Pau et des pays de l'Adour, Décembre 2004.

[131] Malcolm Shroff and Robert B. France. Towards a formalization of UML class structures in Z. In *COMPSAC'97 : Proceedings of the* 21^{st} *International Computer Software and Applications Conference*, pages 646–651, Washington, DC, USA, 1997. IEEE Computer Society.

[132] Adel Smeda, Mourad Oussalah, and Tahar Khammaci. A multi-paradigm approach to describe complex software system. *WSEAS Transactions on Computers*, 3(4) :936–941, October 2003.

[133] Mike Spivey. The *z* notation (second edition). *Prentice Hall International*, 1992.

[134] Jennifer Stapleton. DSDM : Dynamic systems development method. In *TOOLS'99 : Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 406–406, Washington, DC, USA, 1999. IEEE Computer Society.

[135] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, Novembre 2002.

[136] Kenji Taguchi and Keijiro Araki. The state-based CCS semantics for concurrent Z specification. In *ICFEM'97 : Proceedings of the* 1^{st} *International Conference on Formal Engineering Methods*, pages 283–292, Washington, DC, USA, 1997. IEEE Computer Society.

[137] Amjad Umar. *Object-oriented client/server Internet environments*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1997.

[138] Jean-Pierre Vickoff. Agilité et performance en conduite de projet. *Dossier Méthodes*, 2(4) :18–24, Avril 2006.

[139] Thomas Weigert, David Garlan, John Knapman, Birger Moller-Pedersen, and Bran Selic. Modeling of architectures with UML (Panel). In *UML'00 : Proceedings of the* 3^{rd} *International Conference on Unified Modeling Language. Advancing the Standard, York, UK*, volume 1939 of *Lecture Notes in Computer Science*, pages 556–569. Springer, 2000.

[140] Michel Wermelinger. Specification of software architecture reconfiguration. Phd thesis, Université Nova de Lisbon, Juin 1999.

[141] Jim Woodcock and Jim Davies. *Using Z Specification, Refinement, and Proof*. University of Oxford, 1995.

[142] Apostolos Zarras, Valérie Issarny, Christos Kloukinas, and Viet Khoi Nguyen. Towards a base uml profile for architecture description. In *ICSE'01 : Proceedings of the* 23^{rd} *International Conference on Software Engineering*, pages 22–26, Washington, DC, USA, 2001. IEEE Computer Society.

[143] Yu Zhao, Yushun Fan, Xinxin Bai, Yuan Wang, Hong Cai, and Wei Ding. Towards formal verification of UML diagrams based on graph transformation. In *CEC-East'04 : Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business, September, Beijing, China*, pages 180–187, China, 2004. IEEE Press.

[144] Tewfik ZIADI. *Manipulation de Lignes de Produits en UML*. Phd thesis, Université de Rennes 1, Décembre 2004.

[145] Paul Ziemann, Karsten Hölscher, and Martin Gogolla. From UML models to graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 127(4) :17–33, 2005.

A

Annexe : Les XML schémas

Dans le quatrième chapitre, nous avons présenté un extrait des XML schéma, des trois méta-modèles, que nous avons proposés afin de décrire les règles de validation. Nous présentons, dans ce qui suit, les différents diagrammes et les descriptions correspondantes.

A.1 XML schéma : style architectural

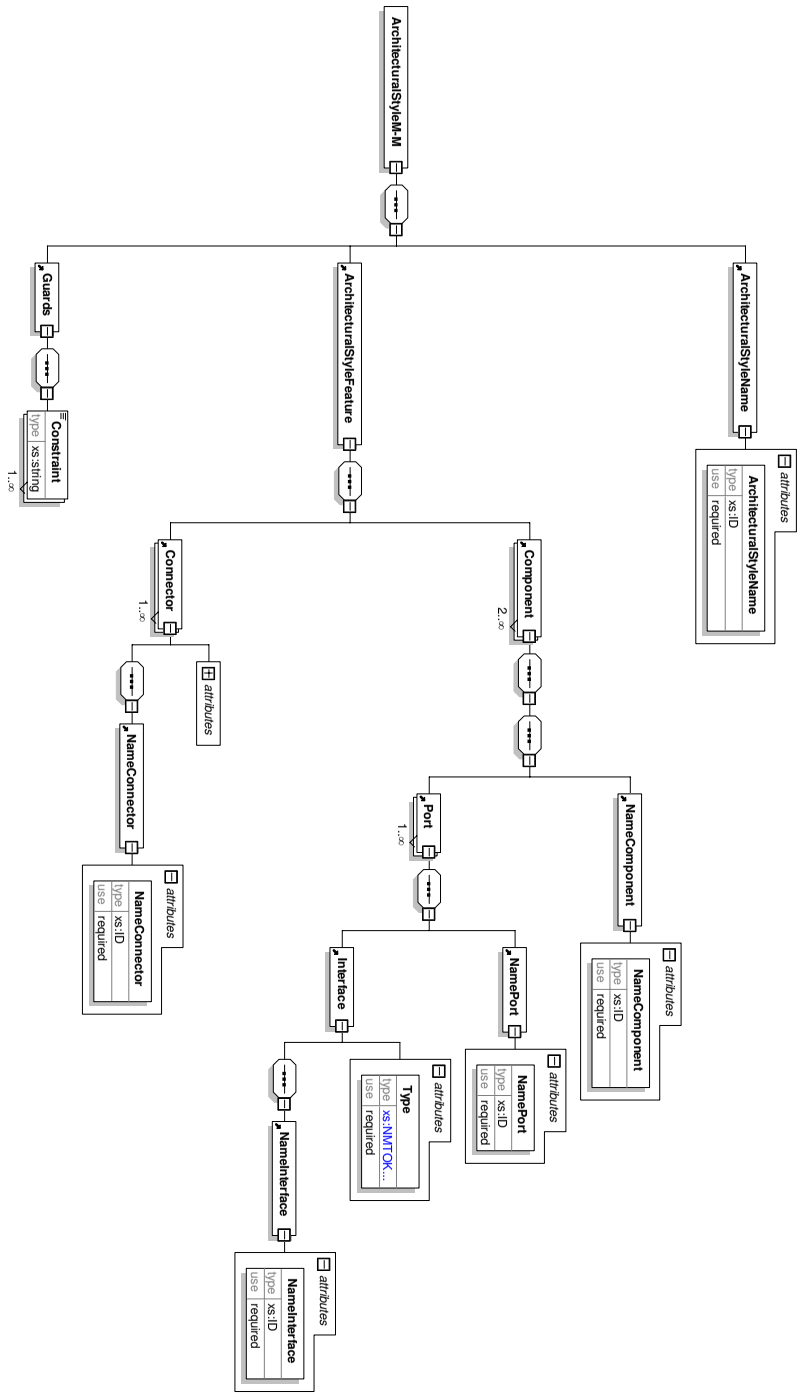

FIG. A.1 – Le diagramme XSD du style architectural

20 xmlns:xs="http://www.w3.org/2001/XMLSchema">

```
<xs:element name="ArchitecturalStyleM-M">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="ArchitecturalStyleName"/>
         <xs:element ref="ArchitecturalStyleFeature"/>
         <xs:element ref="Guards"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="ArchitecturalStyleName">
   <xs:complexType>
      <xs:attribute name="ArchitecturalStyleName" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="ArchitecturalStyleFeature">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="Component" minOccurs="2" maxOccurs="unbounded"/>
         <xs:element ref="Connector" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="Component">
   <xs:complexType>
      <xs:sequence>
         <xs:sequence>
            <xs:element ref="NameComponent"/>
            <xs:element ref="Port" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="NameComponent">
   <xs:complexType>
      <xs:attribute name="NameComponent" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="Port">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="NamePort"/>
         <xs:element ref="Interface"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="NamePort">
   <xs:complexType>
      <xs:attribute name="NamePort" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="NameInterface">
   <xs:complexType>
      <xs:attribute name="NameInterface" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="Interface">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="NameInterface"/>
      </xs:sequence>
      <xs:attribute name="Type" use="required">
         <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="Required"/>
               <xs:enumeration value="Provided"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
```

```
89       </xs:complexType>
90    </xs:element>
91    <xs:element name="Connector">
92       <xs:complexType>
93          <xs:sequence>
94             <xs:element ref="NameConnector"/>
95          </xs:sequence>
96          <xs:attribute name="Type" use="required">
97             <xs:simpleType>
98                <xs:restriction base="xs:NMTOKEN">
99                   <xs:enumeration value="Delegation"/>
100                  <xs:enumeration value="Assembly"/>
101               </xs:restriction>
102            </xs:simpleType>
103         </xs:attribute>
104      </xs:complexType>
105   </xs:element>
106   <xs:element name="NameConnector">
107      <xs:complexType>
108         <xs:attribute name="NameConnector" type="xs:ID" use="required"/>
109      </xs:complexType>
110   </xs:element>
111   <xs:element name="Guards">
112      <xs:complexType>
113         <xs:sequence>
114            <xs:element name="Constraint" type="xs:string" maxOccurs="unbounded"/>
115         </xs:sequence>
116      </xs:complexType>
117   </xs:element>
118   <xs:element name="Constraint" type="xs:string"/>
119 </xs:schema>
```

Listing A.1 – La description XSD du style architectural

A.2 XML schéma : opération de reconfiguration

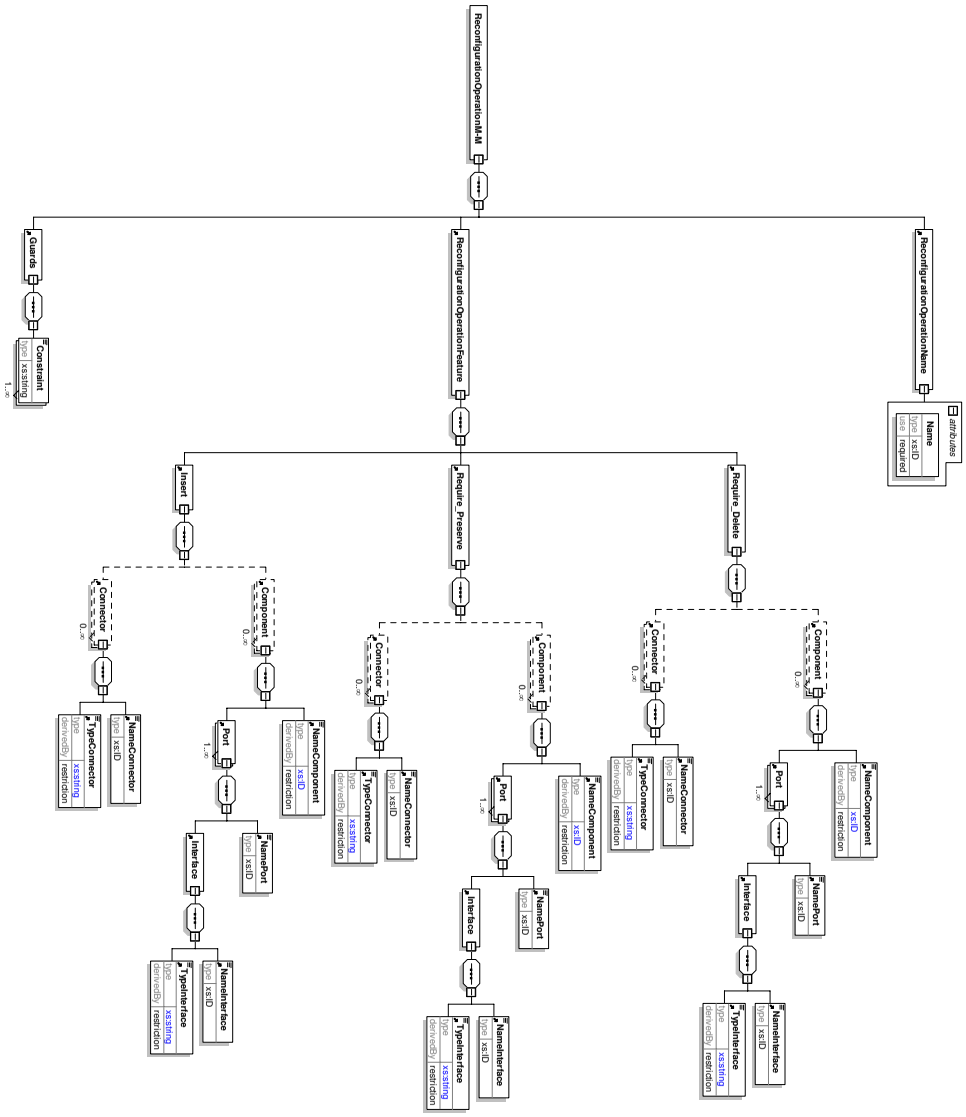

FIG. A.2 – Le diagramme XSD des opérations de reconfiguration

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ReconfigurationOperationM-M">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ReconfigurationOperationName"/>
            <xs:element ref="ReconfigurationOperationFeature"/>
            <xs:element ref="Guards"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationOperationName">
      <xs:complexType>
         <xs:attribute name="Name" type="xs:ID" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationOperationFeature">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Require_Delete"/>
            <xs:element ref="Require_Preserve"/>
            <xs:element ref="Insert"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Require_Delete">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Component" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Connector" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Require_Preserve">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Component" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Connector" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Insert">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Component" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Connector" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Component">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="NameComponent"/>
            <xs:element ref="Port" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="NameComponent">
      <xs:simpleType>
         <xs:restriction base="xs:ID"/>
      </xs:simpleType>
   </xs:element>
   <xs:element name="Port">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="NamePort"/>
            <xs:element ref="Interface"/>
         </xs:sequence>
```

```
68          </xs:complexType>
69       </xs:element>
70       <xs:element name="NamePort" type="xs:ID"/>
71       <xs:element name="Interface">
72          <xs:complexType>
73             <xs:sequence>
74                <xs:element ref="NameInterface"/>
75                <xs:element ref="TypeInterface"/>
76             </xs:sequence>
77          </xs:complexType>
78       </xs:element>
79       <xs:element name="NameInterface" type="xs:ID"/>
80       <xs:element name="TypeInterface">
81          <xs:simpleType>
82             <xs:restriction base="xs:string">
83                <xs:enumeration value="Provided"/>
84                <xs:enumeration value="Required"/>
85             </xs:restriction>
86          </xs:simpleType>
87       </xs:element>
88       <xs:element name="Connector">
89          <xs:complexType>
90             <xs:sequence>
91                <xs:element ref="NameConnector"/>
92                <xs:element ref="TypeConnector"/>
93             </xs:sequence>
94          </xs:complexType>
95       </xs:element>
96       <xs:element name="NameConnector" type="xs:ID"/>
97       <xs:element name="TypeConnector">
98          <xs:simpleType>
99             <xs:restriction base="xs:string">
100               <xs:enumeration value="Assembly"/>
101               <xs:enumeration value="Delegation"/>
102            </xs:restriction>
103         </xs:simpleType>
104      </xs:element>
105      <xs:element name="Guards">
106         <xs:complexType>
107            <xs:sequence>
108               <xs:element name="Constraint" type="xs:string" maxOccurs="unbounded"/>
109            </xs:sequence>
110         </xs:complexType>
111      </xs:element>
112      <xs:element name="Constraint" type="xs:string"/>
113   </xs:schema>
```

Listing A.2 – La description XSD de l'opération de reconfiguration

A.3 XML schéma : protocole de reconfiguration

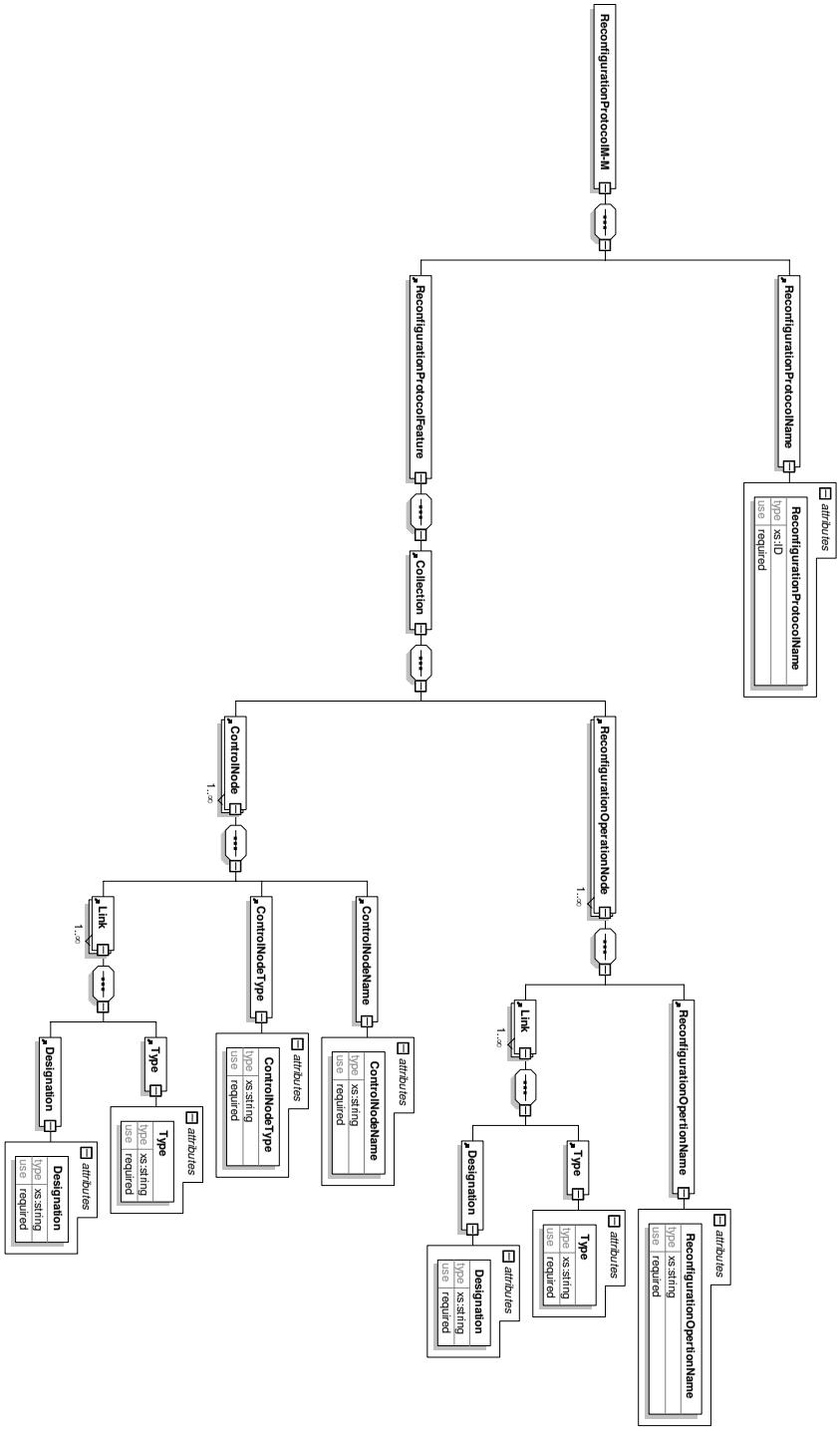

FIG. A.3 – Le diagramme XSD du protocole de reconfiguration

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ReconfigurationProtocolM-M">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ReconfigurationProtocolName"/>
            <xs:element ref="ReconfigurationProtocolFeature"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationProtocolName">
      <xs:complexType>
         <xs:attribute name="ReconfigurationProtocolName" type="xs:ID" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationProtocolFeature">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Collection"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Collection">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ReconfigurationOperationNode" maxOccurs="unbounded"/>
            <xs:element ref="ControlNode" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationOperationNode">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ReconfigurationOpertionName"/>
            <xs:element ref="Link" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ControlNode">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ControlNodeName"/>
            <xs:element ref="ControlNodeType"/>
            <xs:element ref="Link" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ReconfigurationOpertionName">
      <xs:complexType>
         <xs:attribute name="ReconfigurationOpertionName" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="Link">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Type"/>
            <xs:element ref="Designation"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="Type">
      <xs:complexType>
         <xs:attribute name="Type" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="LinkToFinalNode"/>
                  <xs:enumeration value="LinkToDecisionNode"/>
                  <xs:enumeration value="LinkToOperationNode"/>
```

```
                  <xs:enumeration value="LinkToSynchronousNode"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="Designation">
      <xs:complexType>
         <xs:attribute name="Designation" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:string"/>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="ControlNodeName">
      <xs:complexType>
         <xs:attribute name="ControlNodeName" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:string"/>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="ControlNodeType">
      <xs:complexType>
         <xs:attribute name="ControlNodeType" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="FinalNode"/>
                  <xs:enumeration value="DecisionNode"/>
                  <xs:enumeration value="InitialNode"/>
                  <xs:enumeration value="SynchronousNode"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="ProjectName">
      <xs:complexType>
         <xs:sequence minOccurs="0">
            <xs:element ref="ProjectName"/>
            <xs:element ref="ReconfigurationProtocolM-M"/>
         </xs:sequence>
         <xs:attribute name="ProjectName">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="PMS"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Listing A.3 – La description XSD du protocole de reconfiguration

B

Annexe : Les Règles de transformation

Dans le cinquième chapitre, nous avons présenté les règles de transformation d'une description XML vers le langage *Z*. Nous présentons dans ce qui suit ces règles.

```
version="1.0" encoding="UTF-8"?> <!-- edited with XMLSpy -->
<TranslateFunction>
   <function name="size" type="np">
      <cas sign="l(obj:typeName)r()">
         <translate>\# [sub:l1] </translate>
      </cas>
   <!--  <cas sign="l(con:typeName,typeName)r()">
         <translate>\#(\{ [dec:l1] \}\dres [con:l1,l2] )</translate>
      </cas>-->
      <cas sign="l(con:typeName,typeName)r()">
         <translate>\#( [con:l1,l2] \limg \{ [dec:l1] \}\rimg )</translate>
      </cas>
      <cas sign="l(con:name,typeName)r()">
         <translate>\#( [con:$l1,l2] \limg \{ [obj:l1] \}\rimg )</translate>
      </cas>
   </function>
   <function name="isEmpty" type="np">
      <cas sign="l(obj:typeName)r()">
         <translate> [sub:l1] =\emptyset</translate>
      </cas>
      <cas sign="l(con:typeName,typeName)r()">
         <translate>( [con:l1,l2] \limg \{ [dec:l1] \}\rimg )=\emptyset</translate>
      </cas>
      <cas sign="l(con:name,typeName)r()">
         <translate>( [con:$l1,l2] \limg \{ [obj:l1] \}\rimg )=\emptyset</translate>
      </cas>
   </function>
   <function name="notEmpty" type="np">
      <cas sign="l(obj:typeName)r()">
         <translate> [sub:l1] \neq\emptyset</translate>
      </cas>
      <cas sign="l(con:typeName,typeName)r()">
         <translate>( [con:l1,l2] \limg \{ [dec:l1] \}\rimg )\neq\emptyset</translate>
      </cas>
      <cas sign="l(con:name,typeName)r()">
```

```
155         <translate>( [con:$l1,l2] \limg \{ [obj:l1] \}\rimg )\neq\emptyset</translate>
156      </cas>
157   </function>
158   <function name="includes" type="wp">
159      <cas sign="l(obj:typeName)r(dec:name,typeName)">
160         <translate> [r1] \in [obj:l1] </translate>
161      </cas>
162      <cas sign="l(con:typeName,typeName)r(name.name)">
163         <translate>( [obj:r1] , [obj:r2] )\in(\{ [obj:r1] \}\dres [con:l1,l2] )</translate>
164      </cas>
165   </function>
166   <function name="excludes" type="wp">
167      <cas sign="l(obj:typeName)r(dec:name,typeName)">
168         <translate> [r1] \notin [obj:l1] </translate>
169      </cas>
170      <cas sign="l(con:typeName,typeName)r(name.name)">
171         <translate>( [obj:r1] , [obj:r2] )\notin(\{ [obj:r1] \}\dres [con:l1,l2] )</translate>
172      </cas>
173   </function>
174   <function name="including" type="wp">
175      <cas sign="l(obj:typeName)r(obj:name)">
176         <translate> [sub:l1]'= [sub:l1] \cup\{ [obj:r1]\}</translate>
177      </cas>
178      <cas sign="l(con:typeName,typeName)r(con:name,name)">
179         <translate> [con:l1,l2]'= [con:l1,l2] \cup\{( [obj:r1], [obj:r2])\}</translate>
180      </cas>
181   </function>
182   <function name="excluding" type="wp">
183      <cas sign="l(obj:typeName)r(obj:name)">
184         <translate> [sub:l1]'= [sub:l1] \setminus\{ [obj:r1]\}</translate>
185      </cas>
186      <cas sign="l(con:typeName,typeName)r(con:name,name)">
187         <translate> [con:l1,l2]'= [con:l1,l2] \setminus\{( [obj:r1], [obj:r2])\}</translate>
188      </cas>
189   </function>
190   <function name="includesAll" type="wp">
191      <cas sign="l(obj:typeName)r(obj:typeName)">
192         <translate> [sub:r2] \subset [sub:l1] </translate>
193      </cas>
194      <cas sign="l(con:typeName,typeName)r(con:typeName,typeName)">
195         <translate> [con:r1,r2] \subset [con:l1,l2] </translate>
196      </cas>
197   </function>
198   <function name="excludesAll" type="wp">
199      <cas sign="l(obj:typeName)r(obj:typeName)">
200         <translate>\disjoint\langle [sub:l1] , [sub:l2] \rangle</translate>
201      </cas>
202      <cas sign="l(con:typeName,typeName)r(con:typeName,typeName)">
203         <translate>\disjoint&lt; [con:l1,l2] , [con:r1,r2] &gt;</translate>
204      </cas>
205   </function>
206   <function name="forAll" type="wp">
207      <cas sign="l(obj:typeName)r(exp)">
208         <translate>\forall [dec:l1] @ [tra:r1] </translate>
209      </cas>
210      <cas sign="l(con:typeName,typeName)r(exp)">
211         <translate>\forall [dec:l1,l2] @ [tra:r1] </translate>
212      </cas>
213   </function>
214   <function name="exists" type="wp">
215      <cas sign="l(obj:typeName)r(exp)">
216         <translate>\exists [dec:l1] @ [tra:r1] </translate>
217      </cas>
218      <cas sign="l(con:typeName,typeName)r(exp)">
219         <translate>\exists [dec:l1,l2] @ [tra:r1] </translate>
220      </cas>
221   </function>
222   <function name="one" type="wp">
```

```
223       <cas sign="l(obj:typeName)r(exp)">
224          <translate>\exists_1 [dec:l1] @ [tra:r1] </translate>
225       </cas>
226       <cas sign="l(con:typeName,typeName)r(exp)">
227          <translate>\exists_1 [dec:l1,l2] @ [tra:r1] </translate>
228       </cas>
229    </function>
230    <function name="select" type="wp">
231       <cas sign="l(obj:typeName)r(exp)">
232          <translate>\{ [dec:l1] @ [tra:r1] \}</translate>
233       </cas>
234       <cas sign="l(con:typeName,typeName)r(exp)">
235          <translate>\{ [dec:l1,l2] @ [tra:r1] \}</translate>
236       </cas>
237    </function>
238 </TranslateFunction>
```

Listing B.1 – Règles de transformation d'une description XML vers *Z*