

République Tunisienne
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique



Université de Sfax
Faculté des Sciences Economique
et de Gestion de Sfax

MEMOIRE

en vue de l'obtention du diplôme de
MASTER en Systèmes d'Information et Nouvelles Technologies

Une Approche pour la Modélisation et la Vérification des Politiques d'Adaptation pour le Style P/S

Réalisé par
Imen Tounsi

Soutenu le 15 mai 2010 devant le jury composé de :

Mr. Rafik Bouaziz
Mr. Tarak Chaari
Mr. Ahmed Hadj Kacem
Mr. Mohamed Hadj Kacem
Mme. Imen Loulou

Président
Membre
Encadreur
Invité
Invitée

Année Universitaire
2009-2010

Résumé

Les travaux de recherche menés dans le domaine des systèmes distribués couvrent une gamme étendue d'applications. L'architecture de tels systèmes est sujet d'un ensemble de défaillances telles que la panne de composants, la perte de connexions, etc. et ce en raison de la dynamique et la complexité de ces systèmes. Pour remédier à cette faiblesse, l'ajout des plans d'adaptation semble être une bonne solution. Toutefois, la difficulté principale de l'auto-adaptation architecturale surgit en considérant la fiabilité, la sécurité de l'adaptabilité et la préservation des contraintes stylistiques du système.

Dans nos travaux de recherche notre objectif est de proposer une approche pour la modélisation et la vérification des politiques d'adaptation centrées sur l'architecture pour le style Publier/Souscrire. Une politique d'adaptation permet de passer l'architecture d'une configuration biaisée à une configuration correcte. Pour que l'adaptation réussisse, il faut que les contraintes violées soient rétablies dans la nouvelle configuration. Pour cette raison nous proposons de vérifier la fiabilité et la consistance des politiques d'adaptation.

Nous proposons un profil UML pour la modélisation des politiques d'adaptation. Ce profil est basé sur une notation visuelle. Pour vérifier la fiabilité et la consistance des politiques définies nous avons utilisé des bases formelles. Nous proposons de transformer le model UML vers la notation formelle Z en utilisant le langage XSLT. De cette manière, nous nous assurons qu'une fois exécutée, la politique produira une configuration correcte stylistiquement. Les théorèmes de preuve sont définis formellement dans la notation Z et implémentés sous le système de preuve Z-EVES. Un environnement de développement supportant les différents aspects de notre approche est intégré comme un plug-in dans Eclipse. Notre approche est validée à travers deux études de cas.

Mots-clés : Auto-réparation, politiques d'adaptation, profil UML, style architectural, spécification formelle

Abstract

Title : An approach for modeling and checking adaptation policies for the P/S style

Abstract :

Distributed systems researches covers a wide spectrum of applications. The architecture of such systems are subject of some failures such as component failure, connection going down, etc. these failures comes from the dynamicity and the complexity of these systems. To cure this weakness, adding adaptation plans seems to be a good solution. However, the main difficulty of the architectural auto-adaptation emerges while considering the soundness, the safety of the adaptation and the preservation of stylistics constraints of the system.

In our work, our object is to propose an approach centric architecture for modeling and checking adaptation policies for the Publish/Subscribe style. A policy allows the transformation of an incorrect configuration of an architecture w.r.t the architectural style to a correct one. To guarantee the success of the adaptation, it is necessary that violated constraints will be restored in the new configuration. For this reason we propose to check the soundness and the consistency of adaptation policies.

We propose a UML profile for modeling adaptation policies. This profile is based on a visual notation. To verify the soundness and the consistency of defined policies we have used formal bases. We propose to transform the UML model to *Z* notation using XSLT language. Of this manner, we ensure us that once executed, the policy will produce a stylistic correct configuration. Proof theorems are formally defined in *Z* notation and implemented under the *Z/Eves* theorem prover. A software environment supporting the different features of this approach has been developed and integrated as a plug-in in Eclipse. Our approach is validated through two case studies.

Keywords : Self-repair, adaptation policies, UML profile, architectural style, formal specification

A mon Dieu.
A mes parents pour leur patience et leur amour.
A mon mari Mohamed.
A mes petites filles Mariem et Zeineb.
A mes beaux parents pour leur amour et leur encouragement.
A toute ma famille.
A vous tous !

Remerciements

Je loue Dieu tout puissant de m'avoir donné la vie, la santé et d'avoir fait de moi ce que je suis aujourd'hui. C'est grâce à lui que ce présent travail a vu le jour.

Je tiens, à exprimer ma gratitude et mes remerciements à mon encadreur Mr. Ahmed Hadj Kacem, maître de conférences à la Faculté des Sciences Économiques et de Gestion de Sfax pour le soutien qu'il m'a offert, le temps précieux dont il m'a fait part ainsi que ses encouragements et sa gentillesse qui ont été déterminants pour les résultats obtenus.

J'exprime également ma gratitude la plus profonde pour Mr. Mohamed Hadj Kacem, maître assistant à l'Institut Supérieur d'Informatique et de Multimedia de Sfax, qui n'a pas épargné un effort dans l'encadrement de ce projet et qui a été toujours disponible pour écouter, discuter et m'orienter à entreprendre les bonnes décisions.

Je remercie également Mme. Imen Loulou, assistante à la Faculté de Sciences de Gabes, pour l'encadrement, l'encouragement, ses conseils et orientations tout au long de ce travail.

Mes remerciements vont aux membres de jury qui ont accepté d'évaluer mon travail de master. Je remercie vivement Mr. Rafik Bouaziz, maître de conférences à la Faculté des Sciences Économiques et de Gestion de Sfax et Mr. Tarak Chaari, maître assistant à l'Institut Supérieur d'Électronique et de Communication de Sfax, qui ont accepté de juger mon travail.

Je remercie, les étudiants de PFE Mr. Jassim Awini et Mr. Idriss Chafroud pour leur travail sérieux et leurs aides.

J'exprime mes reconnaissances envers les membres de l'unité de recherche ReDCAD pour l'atmosphère amical et l'ambiance chaleureuse qui règne au sein de l'équipe.

Table des matières

Introduction Générale	1
1 Architectures logicielles : Concepts de base	3
1.1 Architecture logicielle	3
1.1.1 Composant	4
1.1.2 Connecteur	5
1.1.3 Port	5
1.1.4 Interface	5
1.1.5 Contraintes	6
1.1.6 Configuration	6
1.2 Style architectural	6
1.2.1 Style “client-serveur”	7
1.2.2 Style “publier-souscrire”	8
1.2.3 Style “pipes and filters”	9
1.3 Architecture à quatre niveaux	10
1.4 UML : Unified Modeling Language	11
1.4.1 UML et l’architecture logicielle	11
1.4.2 Profil UML	11
1.4.3 OCL : Object Constraint Language	12
1.5 Le langage Z	12
1.5.1 Schéma	13
1.5.2 Relation	14
1.6 Contraintes de Qualité de Service	15
1.6.1 Latence	15
1.6.2 Charge	15
1.6.3 Bande passante	15
1.7 Conclusion	15
2 Adaptation des architectures logicielles : Etat de l’art	17
2.1 Adaptation logicielle	17
2.1.1 Niveaux d’adaptation	18
2.1.2 Raisons d’adaptation	18
2.1.3 Processus d’adaptation	19
2.1.4 Système auto-adaptable	20
2.2 Architecture logicielle dynamique	20

2.2.1	Niveau implémentation	21
2.2.2	Niveau conception	22
2.3	Dynamique des systèmes “Publier/Souscrire”	24
2.4	Politiques d’adaptation	25
2.4.1	E-C-A	26
2.4.2	Notion de contexte	26
2.5	Langage UML et transformation vers un langage formel	27
2.5.1	Extension du langage UML	27
2.5.2	Transformation de UML vers un langage formel	28
2.6	Conclusion	28
3	Politiques d’adaptation : Approche proposée	29
3.1	Profil UML pour la modélisation des politiques d’adaptation	30
3.2	Transformation de UML vers Z	33
3.3	Style architectural	35
3.4	Vérification formelle	37
3.4.1	Fiabilité	37
3.4.2	Consistance	38
3.5	Etude de cas : <i>Follow me</i>	39
3.5.1	Modélisation des politiques d’adaptation	42
3.5.2	Vérification formelle	46
3.6	Conclusion	49
4	Un plug-in Eclipse pour la modélisation des politiques d’adaptation	51
4.1	Création d’un plug-in Eclipse	51
4.1.1	Plug-ins Eclipse pour la modélisation	52
4.1.2	Étapes de réalisation d’un diagramme éditeur avec GMF	53
4.2	Éditeur graphique pour la politique d’adaptation	54
4.2.1	Définition d’un modèle du domaine (<i>.ecore</i>)	54
4.2.2	Définition des dessins graphiques (<i>.gmfgraph</i>)	56
4.2.3	Définition de la palette (<i>.gmftool</i>)	56
4.2.4	Définition du mapping (<i>.gmfmap</i>)	57
4.2.5	Création du fichier modèle de génération (<i>.gmfgen</i>)	58
4.2.6	Génération du code de diagramme éditeur (<i>.diagram</i>)	59
4.2.7	Génération du plug-in	59
4.3	Description de l’utilisation du plug-in	60
4.4	Transformation d’un document XML vers une spécification Z	62
4.4.1	XSLT : eXtensible Stylesheet Language Transformations	62
4.4.2	Règles de transformation	62
4.5	Conclusion	64
	Conclusion Générale	65
	Bibliographie	66

Table des figures

1.1	Les éléments d'une architecture logicielle	4
1.2	Modélisation graphique d'un composant	5
1.3	Le style "client-serveur"	7
1.4	Le style "publier-souscrire"	8
1.5	Architecture avec service d'événement centralisé	8
1.6	Architecture avec service d'événement distribué	9
1.7	Le style "pipes and filters"	10
1.8	Architecture à quatre niveaux	10
2.1	Processus d'adaptation dynamique	19
2.2	Niveaux d'adaptation des architectures logicielles	21
2.3	Niveau implémentation	21
2.4	Niveau conception	22
2.5	Description de l'adaptation architecturale	25
2.6	Description de la politique d'adaptation	26
2.7	UML	27
3.1	Profil UML des politiques d'adaptation	30
3.2	Modèle général pour la description d'une politique d'adaptation	33
3.3	Règle de transformation d'un <i>ActionPlan</i>	34
3.4	Règle de transformation d'une <i>Solution</i>	34
3.5	Exemple d'un style architectural	36
3.6	Espace des états valides et invalides	37
3.7	Etude de cas : <i>Follow me</i>	40
3.8	Le style architectural <i>Follow me</i>	40
3.9	Configuration initiale	42
3.10	La configuration après l'application de la <i>Solution1</i>	43
3.11	La configuration après l'application de la <i>Solution2</i>	43
3.12	La configuration après l'application de la <i>Solution3</i>	43
3.13	Modèle de la politique d'adaptation	44
3.14	Preuve du théorème de fiabilité	47
3.15	Preuve du théorème de consistance	48
4.1	Étapes de transformation d'un modèle de la politique vers le langage Z	52
4.2	Dépendances entre les plug-ins de Eclipse	53
4.3	Étapes de réalisation d'un projet GMF	53

4.4	Diagramme éditeur sous Eclipse	54
4.5	Le modèle du domaine (<i>.ecore</i>)	55
4.6	Visualisation de l'ecore sous la forme d'un diagramme de classe (<i>.ecorediag</i>)	55
4.7	Le modèle graphique (<i>.gmfgraph</i>)	56
4.8	Le modèle d'outils (<i>.gmftool</i>)	57
4.9	Le modèle du mapping (<i>.gmfmap</i>)	58
4.10	Le modèle de génération (<i>.gmfgen</i>)	58
4.11	Génération du plug-in	59
4.12	Éditeur graphique	60
4.13	Forme arborescence	61
4.14	Fichier XML	61
4.15	Principe de fonctionnement d'un processus XSLT	62

Liste des tableaux

2.1	Techniques utilisées pour chaque phase d'adaptation	20
3.1	Description du profil	31
3.2	Notations de stéréotypes	32
3.3	Notations de la méta classe PseudoState	32

Introduction Générale

Les architectures des systèmes logiciels actuels sont devenues de plus en plus larges, dynamiques et complexes. Le bon fonctionnement de ces systèmes requiert des propriétés liées à l'adaptabilité, la réutilisabilité et la mobilité. L'auto-adaptation est proposée comme une approche utile pour réduire la complexité associée à la gestion de l'architecture de ces systèmes.

Un système auto-adaptable est un système qui a la possibilité de s'adapter en cours d'exécution pour faire face aux erreurs inattendues (panne de composants, perte de connexions, etc...) ou aux changements dans l'environnement d'exécution. Comme l'adaptation de l'architecture de ces systèmes est une tâche difficile, elle ne doit pas être ad-hoc, elle doit être guidée par des plans d'adaptation. Toutefois, la difficulté principale de l'auto-adaptation architecturale surgit en considérant la fiabilité et la sécurité de l'adaptabilité et la préservation des contraintes stylistiques du système. Pour que l'adaptation réussisse, il faut s'assurer de la conformité de l'état du système après l'adaptation vis-à-vis du style architectural défini. Un état conforme au style est un état qui satisfait toutes les contraintes définies par le style. Un style architectural est défini comme un ensemble de types de composants et de connexions avec la définition des propriétés invariables.

Nos travaux de recherche entrent dans ce contexte et consistent à proposer une approche d'auto-adaptation des architectures logicielles basée sur les politiques d'adaptation. Nous nous intéressons uniquement à la phase de conception (design time).

Notre approche est basée sur un profil UML pour la modélisation formelle de l'adaptation architecturale. Le profil offre une notation visuelle permettant de décrire les politiques d'adaptation. Cette approche permet aussi la génération automatique des spécifications formelles des politiques ainsi que la vérification de leur fiabilité et de leur consistance. Pour la vérification formelle, nous utilisons la démarche de vérification proposée par Mme. Imen LOULOU [LTH⁺09]. Le processus de vérification est codé dans la notation Z [WD96] et implémenté sous le système de preuve Z-EVES [MS97]. Nous avons instancié notre approche en utilisant le style architectural Publier/Souscrire. La définition formelle de ce style a été proposée par Mme. Imen Loulou [LJDK10]. Notre approche a été validée à travers deux études de cas "Opérations d'Intervention d'Urgence" [LTH⁺09, LTHK⁺09] et "Follow me".

Ce rapport est organisé comme suit :

Le chapitre 1 présente une étude sur les concepts de base des architectures logicielles. Il

introduit la définition des éléments constituant une architecture. Une section est réservée à l'introduction et à l'étude de quelques styles architecturaux. Nous nous concentrons sur le style Publier/Souscrire. Les trois dernières sections sont consacrées respectivement à l'étude du langage UML, la présentation de quelques concepts de base relatifs au langage Z et la définition des contraintes de qualité de service.

Le chapitre 2 présente quelques concepts sur l'adaptation logicielle. Une étude de synthèse sur les architectures logicielles dynamiques. A ce niveau, nous avons identifié deux niveaux d'abstraction : le niveau implémentation et le niveau conceptuel. Une section particulière est consacrée à la dynamique des systèmes Publier/Souscrire. Ensuite, nous avons introduit la notion de politiques d'adaptation. Une dernière section est consacrée à la présentation des extensions faites dans le langage UML et les transformations vers un langage formel.

Le chapitre 3 propose un profil UML pour la modélisation des politiques d'adaptation. Le profil est formé par un méta-modèle qui étend le diagramme d'activité de UML2.0. Nous proposons ensuite le processus de transformation de UML vers le langage formel Z à travers les règles de transformation. Nous utilisons les théorèmes proposés par Mme. Imen LOULOU pour vérifier la fiabilité et la consistance des politiques d'adaptation. La dernière section est consacrée à l'application de l'approche sur l'étude de cas : Follow me.

Le chapitre 4 présente la réalisation d'un plug-in Eclipse pour la modélisation des politiques d'adaptation. Ce chapitre comporte une description de l'utilisation du plug-in et la transformation d'un document XML vers une spécification Z en utilisant les règles qui sont exprimées avec le langage XSLT.

Ce manuscrit se termine par une conclusion générale et des perspectives.

1

Architectures logicielles : Concepts de base

L'architecture logicielle se définit comme une spécification abstraite en termes de composants logiciels qui le constituent et des connecteurs entre ces composants [CN01]. Les composants encapsulent typiquement l'information ou la fonctionnalité, tandis que les connecteurs coordonnent la communication entre les composants.

Pour aller plus loin et capturer l'expertise de conception pour un domaine particulier, la notion de style architectural est mise en priorité. 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 les architectures logicielles et ses différents concepts. Nous étudions quelques styles architecturaux. Nous présentons également une étude sur UML et les différentes utilisations et extensions proposés pour qu'il supporte l'architecture logicielle. Finalement, nous introduisons les contraintes de qualité de service.

1.1 Architecture logicielle

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 des logiciels complexes et de grande taille.

L'objectif de la description des architectures et d'avoir l'abstraction nécessaire pour

modéliser les systèmes logiciels complexes durant leur développement, déploiement et évolution.

Dans la littérature nous trouvons plusieurs définitions pour l'architecture logicielle.

Selon Kruchten et al. [KOS06], "L'architecture logicielle implique la structure et l'organisation par lesquelles des composants interagissent pour créer de nouvelles applications, possédant la propriété de pouvoir être les mieux conçues et analysées au niveau système".

Selon IEEE [Hil], "L'architecture est définie par la pratique comme l'organisation fondamentale d'un système, intégrée par ses composants, leurs relations entre-eux et avec l'environnement, et les principes qui guident sa conception et son évolution ».

Quant à nous, nous retenons la définition suivante : L'architecture logicielle, comme le montre la figure 1.1, 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 [Boa95, CLB⁺01, Gar00, IBRZ00, LVM95]. 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.

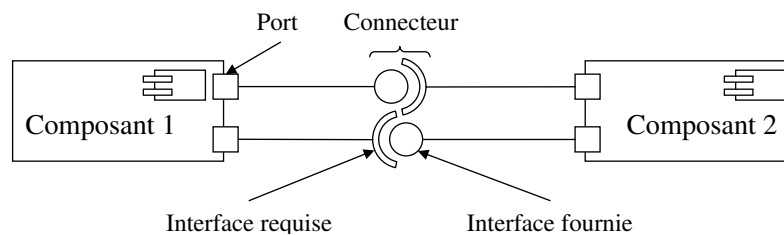


FIG. 1.1 – Les éléments d'une architecture logicielle

1.1.1 Composant

Dans la littérature, plusieurs définitions ont été proposées pour le terme composant logiciel. Même si ces définitions sont différentes selon le contexte et les technologies dans lesquelles elles ont été annoncées, elles ont le même point de convergence : la réutilisabilité [Ezr99, Som92]. Par réutilisation, on entend la possibilité de construire une nouvelle application en récupérant le code et les bouts de programmes développés auparavant [RL04].

Selon Allen et al. [ADG98], "un composant est une unité exécutable ayant la forme d'une boîte noire encapsulant les services qu'elle fournit. Ces services ne sont accessibles que par les interfaces publiées correspondantes et ce à travers un standard d'interaction".

Pour Olivier [Oli05], "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”.

1.1.2 Connecteur

Le connecteur, appelé aussi connexion, correspond à un élément d’architecture logicielle qui représente un moyen d’interaction entre les composants [Car03, Oli05]. Il permet également d’assembler des composants en utilisant leurs interfaces fournies et requises. Afin de fixer les conditions d’utilisation des connecteurs, des contraintes sont également déclarées.

1.1.3 Port

Un port permet de spécifier les points d’interactions d’un composant avec son environnement. Les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Un port représente un point d’accès à certains services du composant [Car03]. Le comportement interne d’un composant n’est visible et accessible qu’à travers ses ports.

1.1.4 Interface

C’est le point de communication qui permet d’interagir avec l’environnement [Oli05]. Pour un composant, il existe deux types d’interfaces :

- Les interfaces fournies décrivent les services proposés par le composant (figure 1.2).
- Les interfaces requises décrivent les services que les autres composants doivent fournir pour le bon fonctionnement du composant (figure 1.2).

Ces interfaces sont exprimées par l’intermédiaire des ports (figure 1.2).

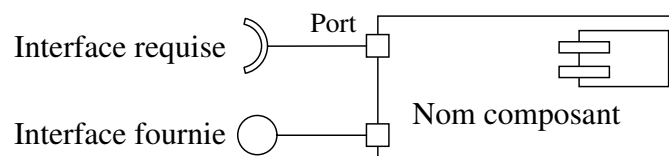


FIG. 1.2 – Modélisation graphique d’un composant

1.1.5 Contraintes

Les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou une de ces parties. Si celle-ci est violée, le système est considéré comme un système incohérent.

1.1.6 Configuration

Une configuration décrit l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que leurs connexions. Elle décrit la structure architecturale caractéristique du style c'est à dire la manière dont doivent interagir les composants et les connecteurs. Elle est définie par un schéma décrivant le comportement général d'une spécification, la composition des composants et des connecteurs [RL00].

Une configuration représente en fait une instance possible d'un style architectural. Plus précisément, une configuration décrit les instances de composants intervenants et les connexions qu'elles entretiennent entre elles [MDK94].

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).

1.2 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 [NR99], 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 [MKMG97].

Un style architectural est une technique générique facilitant l'expression des solutions structurelles des systèmes. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle [SDK⁺95, wCGS⁺02, LPR03, MOO07].

Un style n'est pas une architecture mais une aide générique à l'élaboration de structures architecturales [CN01]. 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) [GAO94].

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

1.2.1 Style "client-serveur"

C'est le style le plus connu de tous [BN84, Uma97]. Comme le montre la figure 1.3, il se base sur deux types de composants : un composant de type serveur, écoutant des demandes et fournit un ensemble de services. Un composant de type client, désirent 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.

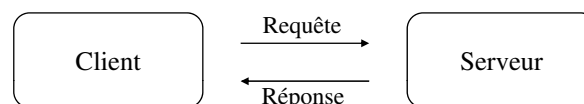


FIG. 1.3 – Le style "client-serveur"

1.2.2 Style “publier-souscrire”

Ce style émerge pour les applications distribuées à grande échelle. Il fournit un découplage des entités en terme de temps et d’espace. Ce découplage est assuré par le service d’événement qui est généralement représenté par un réseau de dispatchers responsables du routage des événements à partir de ceux qui les produisent vers ceux qui veulent les recevoir. Comme le montre la figure 1.4, ce style 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.

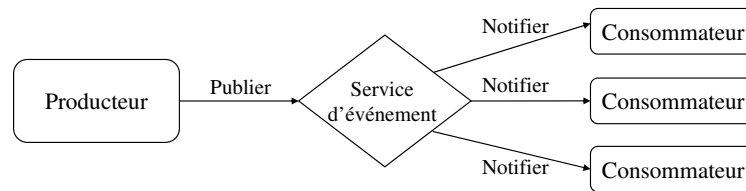


FIG. 1.4 – Le style “publier-souscrire”

Nous distinguons deux types d’architectures, les architectures utilisant un service d’événement centralisé et les architectures utilisant un service d’événement distribué. Dans le premier cas 1.5, le service d’événement utilise un dispatcher pour diffuser les événements. Dans le deuxième cas 1.6, le service d’événement utilise un nombre de dispatchers interconnectés qui fournissent des points d’accès aux différents clients (producteurs et consommateurs). Les dispatchers coopèrent ensemble pour transmettre les informations à partir des producteurs vers les consommateurs souscrits.

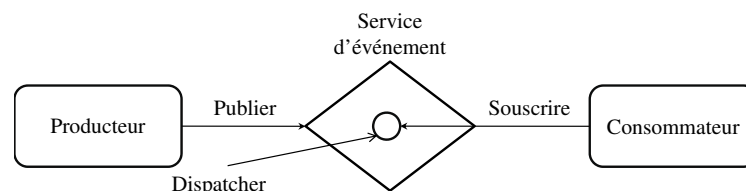


FIG. 1.5 – Architecture avec service d’événement centralisé

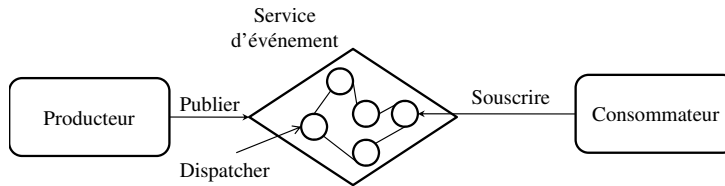


FIG. 1.6 – Architecture avec service d'événement distribué

Ce style est marqué par une diversité de topologies d'interconnexion régissant les dispatchers. Elle peut être sous forme hiérarchique, acyclique P2P et générale P2P.

- **Topologie hiérarchique** : Dans la topologie hiérarchique, un ensemble de dispatchers sont interconnectés et maintiennent une relation de “Maître/Esclave” avec d'autres dispatchers.
- **Topologie acyclique P2P** : Dans la topologie acyclique P2P, les dispatchers communiquent entre eux comme des pairs. Dans cette topologie, il existe toujours un et un seul chemin qui relie un dispatcher à un autre.
- **Topologie générale P2P** : Si nous éliminons cette contrainte d'acyclicité, nous obtiendrons la topologie générale P2P.

S'ajoute à ces spécificités, la notion de modèle de souscription qui décrit la façon avec laquelle les intérêts des consommateurs sont exprimés. Les systèmes Publier/Souscrire sont également caractérisés par une forte dynamique de leur architecture. Pour mes travaux de recherche, on s'intéresse pour le style publier-souscrire

1.2.3 Style “pipes and filters”

Dans ce style, comme le montre la figure 1.7, 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 [GAO94]. 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.

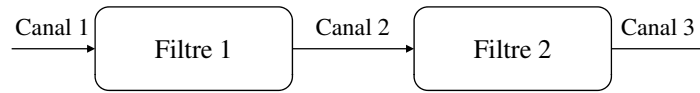


FIG. 1.7 – Le style “pipes and filters”

1.3 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 [Bla05].

Comme le montre la figure 1.8, 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.

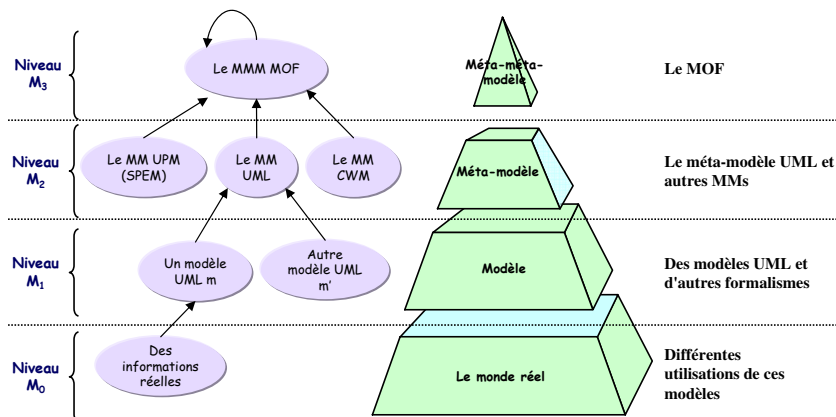


FIG. 1.8 – Architecture à quatre niveaux

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 .

1.4 UML : Unified Modeling Language

UML [Fow04, MG04] est un langage qui offre une notation graphique, unifiée et visuelle. Il est utilisé comme un langage commun et les différents architectes sont familiers, à un certain niveau, à ce langage, et savent manipuler la majorité de ses diagrammes [PC06].

1.4.1 UML et l'architecture logicielle

UML apporte des améliorations considérable avec la version 2.0 pour représenter les architectures logicielles en introduisant le concept de composant (component) et le concept 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 un ensemble de notation intéressante pour la modélisation de l'architecture logicielle [Bru06, Had08].

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 [OMG03a].

1.4.2 Profil UML

Un profil UML est une adaptation du langage UML à un domaine particulier [Bla05, Lop05]. Il est constitué de stéréotypes, de contraintes et de valeurs marquées [Nas05]. 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. Il s'agit d'un mécanisme d'extensibilité du méta-modèle d'UML permettant alors d'étendre la sémantique des éléments de modélisation.

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.

Les valeurs marquées ajoutent des informations sur les éléments de modèle. Une valeur marquée est une paire nom, valeur qui ajout une nouvelle propriété à un élément de modélisation. Cette propriété peut représenter une information d'administration (auteur, date de modification, état, version, etc.), de génération de code ou une information sémantique utilisée par un stéréotype.

L'OMG a standardisé certains profils. Nous citons dans ce qui suit quelques uns :

- 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.

1.4.3 OCL : Object Constraint Language

UML tout seul, ne permet pas 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 [Bla05] : 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 representable 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 [OMG03b, Baa05, OMG03d, AP03] en utilisant des outils tels que USE (UML-based Specification Environment) [GBR07] et Octopus (OCL Tool for Precise UML Specifications) [End98].

1.5 Le langage Z

Les problèmes qui surgissent dans les systèmes informatiques découlent souvent d'erreurs et d'insuffisance dans leurs spécifications de départ [HL94]. Ces problèmes sont dûs à la complexité croissante des applications. Toutefois, l'utilisation des méthodes formelles, qui tirent profit des principes mathématiques, peut résoudre ces problèmes en écartant toute ambiguïté et imprécision dans la spécification du système.

Le langage Z est créé par Jean-Raymond Abrial et développé par une équipe du "Programming Research Group (PRG)" de l'université d'Oxford [HL94]. Ce langage se fonde sur la théorie des ensembles et la logique des prédicats. Le langage Z est structuré en un ensemble de schémas.

Afin de spécifier les politiques d'adaptation, nous avons utilisé plusieurs concepts à savoir les schémas d'opération, les schémas d'état, les relations, etc [WD96].

Dans ce qui suit, nous présentons quelques concepts de base, relatifs au langage Z, que nous jugeons utile pour la compréhension de notre approche.

1.5.1 Schéma

Un schéma est identifié par un nom et il est divisé en deux parties : une partie déclarative et une autre prédictive. La première partie définit les différentes données qui caractérisent le schéma. Ces données doivent vérifier les contraintes spécifiées dans la deuxième partie. En Z, un schéma est représenté comme suit :

<i>Nom_schema</i>
<i>Partie_declarative</i>
<i>Partie_predicative</i>

Nous pouvons également écrire un schéma sous une forme linéaire :

$$\text{Nom_Schema} \hat{=} [\text{Partie_declarative} \mid \text{Partie_predicative}]$$

Schéma d'état : Ce schéma sert à déclarer les différentes variables du système, ainsi que les contraintes qui portent sur ces variables.

<i>Etat</i>
<i>Declaration</i>
<i>Predicat</i>

Schéma d'opération : Ce schéma sert à décrire les changements dans l'état du système, ou à extraire les données du système. Pour décrire une opération sur un état, nous employons deux copies de l'état : la première représente l'état avant l'opération et la deuxième représente l'état après.

<i>Operation</i>
Δ <i>Schema</i>
$in_1?, \dots, in_n? : INPUT$
$out_1!, \dots, out_m! : OUTPUT$
<i>Predicat</i>

Dans notre travail, les schémas sont utilisés pour décrire les états du système (les configurations), les opérations de reconfigurations et la politique d'adaptation.

Décorations schématiques : La présence de décorations dans un schéma indique la nature de celui ci.

- *Décorations postfixées* : Soit n un membre d'un schéma donné. On notera :
 - n La variable dans son état initial
 - n' La variable dans son état final (modifiée par affectation)
 - $n?$ La variable est un paramètre d'entrée du schéma dans son état initial
 - $n!$ La variable est un paramètre de sortie du schéma dans son état final
- *Décorations préfixées* : Soit $Schema$ un schéma donné. On note, dans la partie déclarative :

$\Delta Schema$ (delta Schema) : Si $Schema$ décrit l'état d'un système, alors $\Delta Schema$ est un schéma qui inclut les deux schémas : $Schema$ et $Schema'$ (l'état avant la reconfiguration et l'état d'après respectivement)

Les schémas peuvent être considérés comme des unités que nous pouvons les manipuler et les combiner par le moyen des opérateurs de composition séquentielle représentés par “ \circ ” et l'opérateur de piping “ \gg ”.

1.5.2 Relation

Une relation permet de modéliser la structure des interactions entre des composants de différentes natures.

- *Couple ordonné* : Soit x et y deux éléments distincts. On appellera couple ordonné x vers y la paire ayant comme premier élément x (élément de départ) et comme dernier élément y (élément d'arrivée). On la note : $x \mapsto y$ ou encore (x,y) .
- *Relation* : Un ensemble de paires ordonnées telles que leurs éléments de départ et d'arrivée font respectivement partie d'un même ensemble. Elle représente également un sous-ensemble d'un produit cartésien.
Si X et Y sont deux ensembles, alors $X \leftrightarrow Y$ dénote l'ensemble de toutes les relations entre X et Y . Le domaine de R est l'ensemble d'éléments dans X relié à un élément dans Y : $\text{dom}R$. L'image de R est l'ensemble d'éléments de Y avec lequel certains éléments de X sont reliés : $\text{ran}R$. Nous disons également que X et Y sont les ensembles source et cible de R .

- *Relation inverse* : Les relations sont directionnelles : elles relient les objets d'un ensemble aux objets d'un autre. Il est possible d'inverser cette direction. L'opérateur d'inverse \sim fait exactement ceci. La source et la cible sont échangées ainsi que les éléments de chaque paire.

Soit R un élément de l'ensemble $X \leftrightarrow Y$. La relation R^\sim relie y à x exactement quand R relie x à y .

1.6 Contraintes de Qualité de Service

Dans la littérature, il n'existe pas de consensus sur la définition de la qualité de service QoS. La recommandation ITU-X.902 définit la QoS comme un ensemble d'exigences dans le comportement collectif d'un ou de plusieurs objets. Vogel et al. définissent la QoS comme un ensemble de caractéristiques quantitatives et qualitatives d'un système, nécessaires pour atteindre la fonctionnalité requise par l'application [ABvBGJ95]. Nous pouvons aussi dire que la qualité de service représente l'aptitude d'un système à répondre de manière adéquate à des exigences qui visent à satisfaire ses usagers. Ces exigences peuvent être liées à plusieurs aspects, par exemple : le temps de latence (latency), la charge d'un composant (load) et la valeur de la bande passante entre deux composants (bandwidth), etc.

1.6.1 Latence

désigne un délai entre le moment où une information est envoyée et celui où elle est reçue. De façon plus générale, la latence peut aussi désigner l'intervalle entre la fin d'un événement et le début de la réaction à celui-ci, par exemple : le délai entre une requête d'un client et la réponse du serveur.

1.6.2 Charge

La charge d'un composant est une technique utilisée en informatique pour distribuer une tâche entre plusieurs composants afin d'assurer une haute disponibilité des services. Pour chaque composant (dispatcher) on lui associe une charge `maxLoad` pour que le dispatcher ne soit pas surchargé.

1.6.3 Bande passante

La bande passante indique un débit d'informations entre deux composants. Ce débit ne doit pas être inférieure à un seuil minimal.

1.7 Conclusion

Dans ce chapitre, 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 composants. 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 l'adaptabilité 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, l'adaptation 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'adaptation dynamique des architectures logicielles.

2

Adaptation des architectures logicielles : Etat de l'art

L'architecture logicielle est dite dynamique quand elle modélise un système qui interagit avec certains événements pendant son exécution. Ceci se traduit par la reconfiguration de l'architecture de ce système. Cette reconfiguration se fait par le biais des opérations de reconfiguration. Les opérations de reconfiguration de base consistent à ajouter et à supprimer des composants et à ajouter et à supprimer des connexions.

Dans ce chapitre, nous traitons le problème de reconfiguration des architectures logicielles dans le cadre des applications à base de composants. Nous mettrons l'accent sur les différents travaux réalisés dans ce domaine.

Ce chapitre est organisé en trois grandes parties. La première partie présente les concepts fondamentaux liés à l'adaptation logicielle. La deuxième partie discute les différents travaux visant l'adaptation des architectures logicielles et le dynamisme des systèmes "Publier-souscrire". Nous discutons dans la troisième partie notre approche et nous présentons l'extension de UML et la transformation vers un langage formel.

2.1 Adaptation logicielle

L'adaptation logicielle consiste à apporter des modifications à un logiciel ou à un système informatique dans le but d'assurer ses fonctions et, si possible, d'améliorer ses performances dans un environnement d'utilisation [Cha07].

Nous distinguons deux types d'adaptations logicielles [KBC02]. La première est dite sta-

tique dans le cas où l'adaptation nécessite l'arrêt de l'application. La deuxième est dite dynamique dans le cas où l'adaptation se fait pendant l'exécution de l'application.

L'adaptation statique est suffisante dans de nombreux cas. L'adaptation dynamique, s'avère néanmoins nécessaire dans les applications critiques dont l'environnement change constamment (certaines applications distribuées où le nombre de nœuds disponibles et la capacité de la bande passante évoluent de façon imprévisible), les applications dont l'arrêt est coûteux ou difficile à mettre en œuvre (certaines applications industrielles de très grande taille), les applications qui ne peuvent redémarrer que rarement par obligation contractuelle (les routeurs de télécommunication) et les applications critiques dont l'arrêt est interdit (certaines applications de gestion de centrales nucléaires) [KBC02].

2.1.1 Niveaux d'adaptation

Selon notre étude, nous pouvons classer l'adaptation logicielle selon trois niveaux [Kac08, CZL09] :

- Adaptabilité architecturale : elle permet de modifier les composants ainsi que leurs connexions pour ajuster l'architecture logicielle.
- Adaptabilité comportementale : elle permet de redéfinir dynamiquement le comportement interne de l'application et de ces composants sans changer sa structure ou son architecture logicielle.
- Adaptabilité des interfaces : elle permet de modifier les services fournis par un composant au biais de ses interfaces ou générer une interface utilisateur fonctionnelle dans le contexte de son utilisation . Ceci, se traduit soit par la modification de l'ensemble des services fournis soit par la modification de la signature d'un service

Dans nos travaux de recherche, nous nous intéressons au niveau architectural.

2.1.2 Raisons d'adaptation

L'adaptation logicielle peut être réalisée pour différentes raisons. Ces raisons peuvent être classées en quatre catégories [Kac08, NByCC02] :

- Adaptation curative : dans le cas où l'application ne se comporte pas correctement, une adaptation est nécessaire pour corriger les défauts. Une solution consiste à remplacer le composant défaillant par un autre supposé correct. Cette nouvelle configuration fournit les mêmes fonctionnalités que l'ancienne. Elle se contente simplement de corriger ses défauts.
- Adaptation adaptative : consiste à adapter l'architecture de l'application, dans le cas où

l'environnement d'exécution, quelques composants matériels ou d'autres applications ou ressources dont un élément dépend changent.

- Adaptation évolutive : permet d'étendre l'application avec de nouvelles fonctionnalités qui n'ont pas été prises en compte au moment du développement de l'application. Cette extension peut être réalisée par l'ajout ou la modification d'un ou de plusieurs composants tout en gardant l'architecture de l'application.
- Adaptation perfective : permet d'améliorer les performances du système. Si un composant, par exemple, reçoit beaucoup de requêtes et n'arrive pas à les satisfaire et afin d'éviter la dégradation des performances de l'application, une solution consiste à installer un autre composant qui lui partage sa tâche.

2.1.3 Processus d'adaptation

L'adaptation dynamique est le processus par lequel une application logicielle est modifiée afin de prendre en compte un changement [NByCC02], que ce soit au niveau de l'environnement ou de l'application elle-même. Il s'agit d'un processus en quatre temps. Il faut tout d'abord (i) observer l'environnement d'exécution, (ii) décider de l'opportunité de l'adaptation et de la stratégie appropriée à la situation détectée pour l'adapter, puis (iii) planifier les actions à réaliser pour adopter la stratégie décidée, et enfin (iiii) réaliser les traitements décidés. La Figure 2.1 représente un schéma simplifié de ce processus.

Dans nos travaux de recherche, nous visons l'adaptation curative dans la phase de planification qui, étant donné des plans d'adaptation, décidera quelle solution d'adaptation sera déclenchée et exécutée. Ce qui nous intéresse dans cette phase, ce n'est pas comment décider mais plutôt comment formaliser des plans d'adaptation.

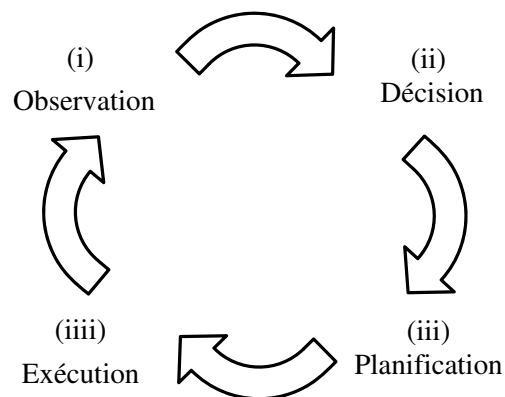


FIG. 2.1 – Processus d'adaptation dynamique

Pour chaque phase d'adaptation, plusieurs techniques possibles sont utilisées [BAP06]. Un résumé de ces techniques est présenté dans le tableau 2.1.

Phases d'adaptations	Techniques utilisées
- Observation	- Sensors (détecteurs) - Manuelle (observation) - Monitoring
- Décision	- Systèmes de règle - Diagnostic basé sur un modèle - Optimisation sous contraintes - Modèles probabilistes - Apprentissage automatique
- Planification	- Système de règles - Programmation logique - Recherche d'un chemin dans le graphe des configurations
- Exécution	- Programmation par aspect - Programmation alternative

TAB. 2.1 – Techniques utilisées pour chaque phase d'adaptation

2.1.4 Système auto-adaptable

Un système auto-adaptable est un système qui change d'une manière autonome son comportement ou son architecture suite à des événements dynamiques. Ces événements peuvent être internes tel que les propriétés du système ou le comportement interne du système, ou externes tel que les changements au niveau de l'environnement d'exécution.

2.2 Architecture logicielle dynamique

Plusieurs travaux de recherche ont essayé d'apporter des solutions aux défaillances des architectures logicielles à base de composants. L'adaptation dynamique est l'une des solutions possibles. Ce type d'adaptation est dit auto-adaptation ou bien auto-réparation dans le cas de réparation d'un problème. Les contributions existantes liées à ce domaine peuvent être classées principalement, comme le montre la figure 2.2, en deux axes de recherches. Le premier axe concerne le niveau implémentation et le second concerne le niveau conceptuel. Nous présentons, dans ce qui suit, une étude de chaque axe et nous nous concentrons sur le niveau conceptuel.

Ensuite, nous présentons, les principaux travaux réalisés dans le but de proposer des méthodes d'adaptation des architectures logicielles.

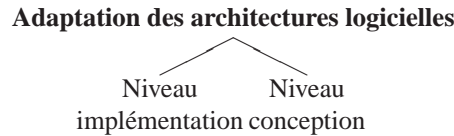


FIG. 2.2 – Niveaux d’adaptation des architectures logicielles

2.2.1 Niveau implémentation

Les travaux réalisés à ce niveau ont proposé des algorithmes et des politiques de reconfiguration. Ces travaux sont classifiés en deux axes, certains se basent sur la notion du style architectural dans leurs processus d’adaptation [CMPC03, PMJ06, Jae05] et d’autres ne le sont pas [FGI06] (figure 2.3).

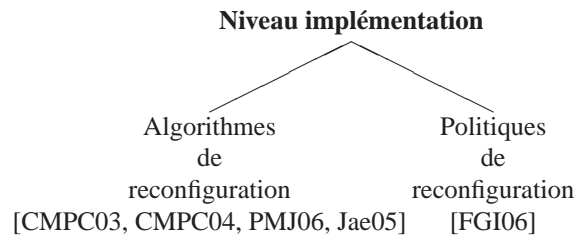


FIG. 2.3 – Niveau implémentation

Parmi les travaux qui ont proposé des algorithmes nous citons le travail de Jaeger et al. [PMJ06]. Ce travail permet l’adaptation dynamique des systèmes Publier/Souscrire en se basant sur les middlewares basés contenu. Jaeger et al. proposent des algorithmes pour empêcher la perte ou la duplication des notifications et maintenir l’ordre des messages.

Dans le même contexte, Cugola et al. abordent le problème de fiabilité dans les middlewares basés contenu pour les systèmes Publier/Souscrire. Ils proposent des algorithmes dits “Epidemic Algorithms” [CMPC03] puis ils les évaluent à travers une simulation [CMPC04].

Parmi les travaux qui ont proposé des politiques de reconfiguration, nous citons le travail de Fredj et al.. Ce travail vise, l’adaptation dynamique des architectures logicielles suite à une perte de connexion dans les environnements diffus [FGI06]. L’adaptation architecturale se fait au niveau middleware par la conception et implémentation des services middleware qui réalisent des solutions adaptées à ce problème. Pour gérer la perte de connexion quatre techniques ont été mises à la disposition. Deux pour l’anticipation à la perte de connexion (la réplication et le transfert d’état) et deux pour la tolérance à la perte de connexion (le rollback et le rejouer).

2.2.2 Niveau conception

Au niveau conceptuel, les travaux de recherches sont classifiés en deux axes, certains n'utilisent pas des plans d'adaptation pour la reconfiguration [YYX05], et d'autres utilisent des plans d'adaptation (figure 2.4). Nous citons les modèles de reconfiguration, les politiques, les contrats d'application et les stratégies. Le style architectural constitue la base de plusieurs travaux.

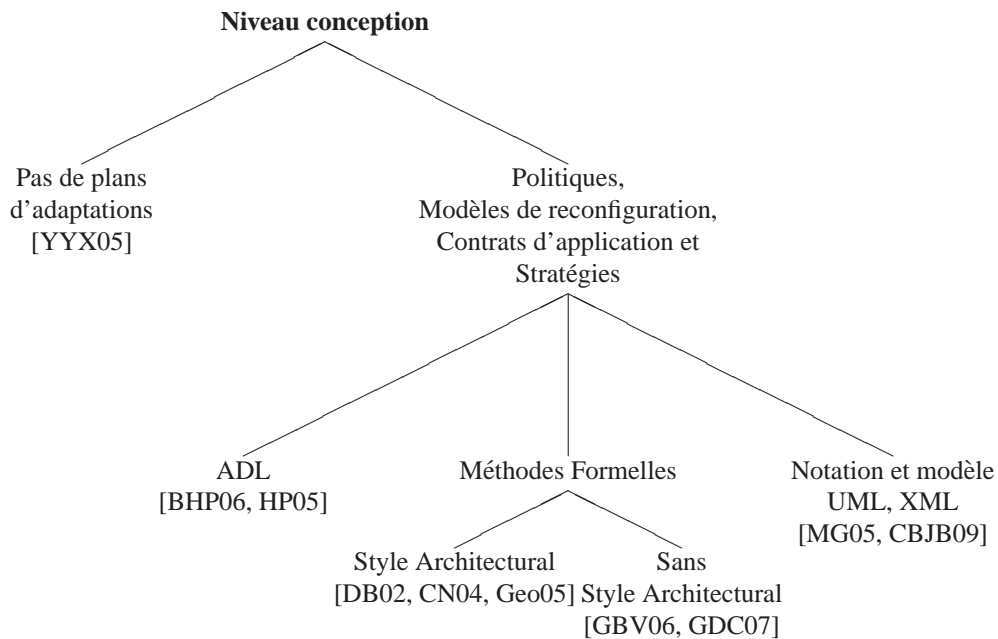


FIG. 2.4 – Niveau conception

Les travaux menés à ce niveau, comme le montre la figure 2.4, ont utilisé différents formalismes pour réaliser les plans d'adaptation. Certains de ces travaux ont utilisé les Langages de Description des Architectures (ADLs) telle que la plateforme SOFA 2.0 [BHP06] qui utilise ArchJava comme langage de description de ces politiques.

D'autres travaux, comme CASA "Contract-based Adaptive Software Architecture", ont utilisé des modèles spécifiés avec XML [MG05]. Pour réaliser l'adaptation, le framework CASA se base sur des politiques d'adaptation définies par un contrat d'application. Ces politiques sont spécifiées dans des fichiers XML séparés de l'application. Cette séparation permet leur modification en cours d'exécution. L'inconvénient de cette approche réside dans le fait qu'elle place une grande responsabilité sur le développeur de l'application. Ceci concerne le développement de toutes les configurations possibles et la génération du contrat d'application conséquent [MG03].

D'autres travaux ont utilisé les méthodes formelles pour spécifier l'auto-adaptation. Ces travaux peuvent être subdivisés en deux axes de recherche : le premier axe concerne les

travaux qui n'ont pas utilisé la notion de style architectural dans leur processus d'adaptation. Le second axe s'intéresse aux travaux qui ont utilisé la notion de style architectural dans leur processus d'adaptation.

Dans le premier axe nous trouvons les travaux de Guennoun et al.. Ils ont défini un méta-modèle relatif à la description et à la gestion automatique des architectures dynamiques. Ce méta-modèle permet de décrire le protocole de gestion de l'architecture. Pour l'adaptation architecturale, ils utilisent des règles de reconfiguration basées sur des techniques orientées règles (règles de transformation de graphes). Ces règles spécifient les actions de transformation élémentaires. Les adaptations architecturales sont appliquées dans le domaine d'intervention d'urgence et de qualité de service (QoS). Cependant, nous précisons que ce travail n'utilise pas les politiques de reconfiguration mais il passe directement aux protocoles de reconfiguration des architectures (PRA) qui sont définies par un ensemble d'événements impliquant une reconfiguration et les règles de transformation à appliquer [GDC07].

MADCAR [GBV06] est un modèle abstrait pour les moteurs d'assemblage qui a pour objectif la reconfiguration dynamique et automatique d'applications à base de composants. Ce modèle est dit abstrait car il est indépendant de tout modèle de composant. Dans ce travail, une politique d'adaptation est utilisée pour la reconfiguration dynamique. Elle est séparée de la description de l'application. MADCAR a été appliqué au modèle de composants Fractal et il est implémenté avec le langage de contraintes Smalltalk.

Le second axe de recherche s'intéresse aux travaux qui ont utilisé la notion de style architectural dans leur processus d'adaptation.

Dans ce contexte, les travaux de Garlan et al. présente une approche basée sur l'architecture pour développer les systèmes auto-adaptables [DB02]. Dans ces travaux, le style architectural est utilisé pour décider quand faut-il appliquer les modification. En effet, quand une violation d'une contrainte stylistique est détectée, la stratégie d'adaptation la plus appropriée sera déclenchée. Pour chaque propriété stylistique est associée une stratégie d'adaptation qui est formée par un ensemble de tactiques [CGS⁺02]. Pour valider leur approche, ils examinent les résultats obtenus après l'adaptation. Si le problème, pour lequel la stratégie est exécutée est résolu, l'adaptation est supposée correcte. Cependant, aucun test n'est appliqué afin de vérifier la conformité de la nouvelle configuration au style architectural. Par exemple, s'ils tentent de corriger un problème de haute latence, ils appliquent la stratégie appropriée puis ils mesurent la valeur de la latence après l'adaptation, si cette dernière ne dépasse pas un seuil prédéfini l'adaptation est supposée correcte.

Les travaux de Taylor et al. se concentrent sur l'utilisation des politiques auto-adaptables dites "Politiques basées connaissances". Chaque politique est identifiée par un identifiant unique et peut inclure une ou plusieurs observations (les informations pertinentes à l'adaptation sont représentées sous forme d'un ensemble d'observations) permettant l'évolution des politiques d'adaptation en cours d'exécution [CN04]. Les opérations de reconfiguration de base ont été enrichies par l'introduction d'autres opérations dites opérations d'évolution (AddObservation, AddPolicy, RemoveObservation, RemovePolicy) pour la modification des événements déclencheurs et des règles d'adaptation [Geo05]. Pour la

réalisation de ce travail, Taylor et al. se basent sur le style architectural Publier/Souscrire hiérarchique via l'ADL C2 [OMT98, TMA⁺96].

Les travaux de Yang et al. proposent une approche pour l'auto-adaptation des architectures logicielles [YYX05]. Dans cette approche, la réflexion de l'architecture est utilisée pour que l'architecture soit observable et contrôlable. De plus, le style architectural est utilisé pour aider à la sélection de la stratégie de réparation la plus appropriée et pour assurer la consistance et la validité des changements. Cependant, la sûreté de la reconfiguration doit être vérifiée chaque fois que la stratégie de réparation est exécutée. Après le choix de la stratégie de réparation, la reconfiguration est exécutée en transformant le graph de l'architecture. Dans ces travaux, un prototype a été implémenté en adoptant le style architectural client-serveur.

Récemment, Perry et al. ont examiné les fonctions de base nécessaires pour les systèmes auto-adaptables et ont étudié un certain nombre de problèmes liés à la conception architecturale pour intégrer l'exécution de réflexion (runtime reflection) et l'adaptation dans les systèmes logiciels [HP05]. Ils ont également proposé des améliorations aux langages de description des architectures (ADL) pour que ces ADLs supportent l'auto-adaptation architecturale.

Comme ces travaux de recherches, notre travail utilise le style architectural comme une base pour l'auto-adaptation du système logiciel. La contribution vitale de notre travail par rapport à ces travaux consiste à vérifier formellement la propriété de préservation du style architectural pour chaque politique d'adaptation définie. Nous n'avons pas besoin d'examiner la sûreté de l'adaptation chaque fois que nous voulons exécuter une politique d'adaptation puisque cette politique a été déjà vérifiée. Notre approche est applicable pour tous les styles architecturaux. Elle est instancié pour le style Publier/Souscrire.

Notre approche est indépendante du modèle de composant. Elle assure l'adaptation non anticipée dans la mesure où l'architecture de l'application et ses composants peuvent être changés pendant l'exécution. Les politiques d'adaptation sont exprimées avec un langage formel afin de garantir leur validité, leur consistance et leur cohérence.

2.3 Dynamique des systèmes “Publier/Souscrire”

Les systèmes Publier/Souscrire sont caractérisés par une forte dynamique de leur architecture [JMWP06, ZS06, LBZ05]. Grâce au découplage fourni par ce type d'architecture (découplage dans l'espace et dans le temps), il est possible de connecter ou de déconnecter un composant sans affecter directement les autres composants. Ceci rend l'architecture logicielle hautement *adaptable et reconfigurable*.

Des recherches considérables ont été faites dans le domaine d'adaptation dynamique des systèmes Publier/Subscriber au niveau implémentation [CMPC03, HMA06, Jae05]. Peu de travaux visent le dynamisme au niveau conceptuel.

Les travaux de Molina et al. [HGM04] décrivent comment un système Publier/Souscrire

peut être étendu pour fonctionner dans un environnement mobile, où les événements peuvent être produits par le déplacement des capteurs ou des utilisateurs. Ils décrivent également comment le système peut être replié pour faire face aux échecs, perte de messages et déconnexions. La solution de la replication peut augmenter la fiabilité mais elle peut poser des problèmes. Par exemple, un utilisateur peut recevoir une séquence d'événements qui sont contradictoires.

Khelifi et al. [Kkk08] présentent une autre approche pour concevoir et valider les architectures logicielles basées sur le style Publier/Souscrire. Cette approche permet la description de la dynamique de l'architecture du système logiciel en utilisant les automates. Pour vérifier le comportement et la structure du modèle du système, ils utilisent le "model checker SPIN".

2.4 Politiques d'adaptation

Comme il a été déjà mentionné, notre travail de recherche aborde la problématique d'adaptation architecturale pour assurer la survie des systèmes logiciels face aux variations des environnements d'exécution. Nous proposons une approche permettant l'adaptation architecturale afin de garantir l'auto-réparation. L'adaptation architecturale, comme le montre la figure 2.5, fait passer l'architecture de l'application d'une configuration (n) à une configuration (n+1) selon un contexte bien déterminé et en suivant des politiques d'adaptation.

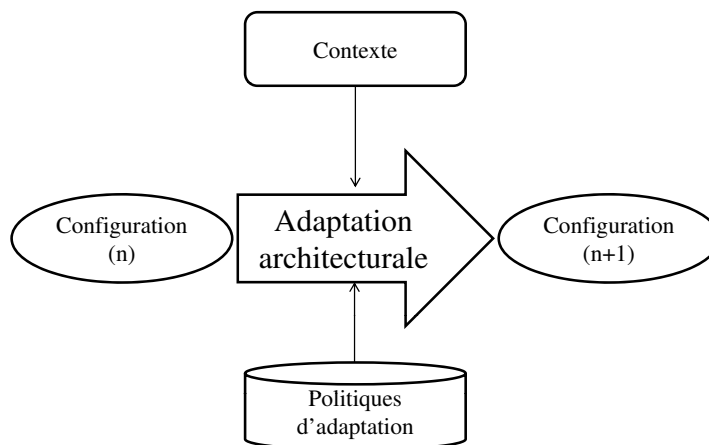


FIG. 2.5 – Description de l'adaptation architecturale

Une politique d'adaptation est un ensemble de règles et de stratégies conçues pour accomplir un ensemble particulier de buts. Elle peut être définie comme une fonction qui transforme une série d'événements qui représentent le contexte à un ensemble d'actions en respectant un ensemble de conditions et en partant d'un état initial [LBN99] (Figure 2.6).

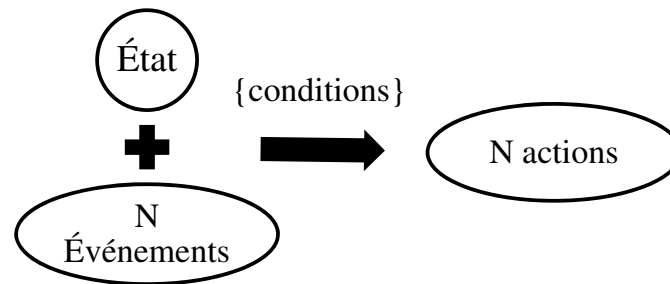


FIG. 2.6 – Description de la politique d’adaptation

Les politiques d’adaptation sont décrites sous la forme E-C-A (en Événement si Condition alors Action). Ces politiques utilisent un ensemble d’opérations de reconfiguration (ajout et suppression de composants et ajout et suppression de connexions) pour modifier la configuration d’un système logiciel.

2.4.1 E-C-A

Généralement les politiques d’adaptation sont décrites sous la forme du schéma E-C-A (Événement, Condition, Action) [ANC96]. Il s’agit de spécifier les sondes (logicielles ou matérielles) pertinentes à l’application et les conditions de déclenchement d’une adaptation (par exemple, une déconnexion réseau).

La signification d’une telle règle est la suivante : “Quand un événement se produit, vérifiez la condition et si elle est satisfaite, exécutez l’action”. L’événement est dit déclencheur de la règle d’adaptation (eng. Trigger).

2.4.2 Notion de contexte

Quand nous parlons de l’adaptation, la notion de contexte est prise en compte comme un critère important. Cette notion a été exploitée dans plusieurs domaines informatiques. Les définitions dans la littérature sont souvent orientées selon le domaine, elles sont par conséquent trop limitées. Néanmoins, nous pouvons définir le contexte comme suit : “Le contexte est l’ensemble d’informations externes pouvant être utilisées pour caractériser la situation d’une application. Ces informations peuvent être dynamiques et peuvent donc changer durant l’utilisation de l’application” [ADB⁺99]. Comme exemple, nous pouvons citer : l’état de l’application, la disponibilité CPU, la valeur de la bande passante, la connexion réseau, les changements des ressources disponibles, etc.

“le contexte couvre toutes les informations pouvant être utilisées pour caractériser la situa-

tion d’une entité. Une entité est une personne, un lieu, ou un objet qui peut être pertinent pour l’interaction entre l’utilisateur et l’application, y compris l’utilisateur et l’application eux-mêmes”.

2.5 Langage UML et transformation vers un langage formel

L’utilisation de UML dans le domaine des architectures logicielles peut être classée principalement en deux axes. Le premier, cherche à modéliser les architectures logicielles avec UML. Différents types d’extensions ont été proposés afin de personnaliser UML pour décrire certains aspects liés aux architectures logicielles, comme la description des plans d’adaptation. Le deuxième, cherche à proposer des règles de transformation de UML vers un langage formel.

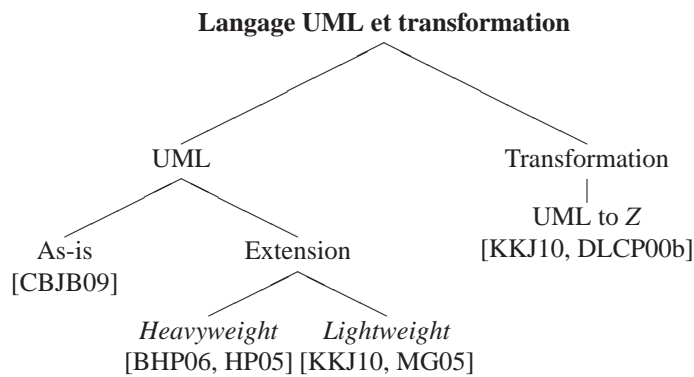


FIG. 2.7 – UML

2.5.1 Extension du langage UML

D’une façon générale, UML a été utilisé pour décrire les architectures logicielles. Nous remarquons qu’il existe principalement trois axes : l’utilisation de UML comme il est “As-is”, l’extension légère “Lightweight” et l’extension lourde “Heavyweight”.

Le premier axe consiste à utiliser UML comme il est sans aucune modification.

L’extension “lightweight” [SX03, MNN02] permet d’adapter des méta-modèles à des buts spécifiques par l’extension des méta-classes de base. L’adaptation est définie par des stéréotypes groupés dans un profil.

L’extension “heavyweight” [KKJ10, PMSA04, PM03, KS00] permet d’ajouter de nouveaux modèles ou remplacer la sémantique existante en modifiant le méta-modèle de UML.

Selon l'étude de l'état de l'art que nous avons mené, nous avons constaté que peu de travaux ont utilisé UML pour décrire leurs plans d'adaptation. A titre d'exemple nous citons les travaux de CHAUVEL et al. [CBJB09].

CHAUVEL et al. proposent un processus de développement basé sur les modèles, combinant l'architecture logicielle et ses capacités d'adaptation [CBJB09]. Pour la modélisation des politiques d'adaptation, ils utilisent des machines à états. Ils utilisent UML comme il-est sans aucune extension.

Dans notre travail, nous avons proposé un profil UML pour modéliser les politiques d'adaptation. Ce profil étend légèrement le langage UML2.0. Cette extension consiste à étendre la syntaxe et la sémantique de la notation UML par des stéréotypes.

2.5.2 Transformation de UML vers un langage formel

Plusieurs travaux combinent le langage UML avec un langage formel dans le but de spécifier la dynamique des architectures logicielles [FBLPS97, KC02]. L'objectif de ce domaine de recherche est de proposer une fondation formelle pour les modèles UML [LPU02]. Pour utiliser un langage formel, certains travaux ont proposé de traduire UML vers un langage formel [KC02, KKJ10, DLCP00a].

Hadj Kacem et al. adoptent une approche basée sur la transformation de UML vers des spécifications formelles [KKJ10]. Cette approche permet une transformation automatique du style architectural et de chaque opération de reconfiguration vers le langage Z afin de vérifier la consistance et la conformité de l'architecture.

Dupuy et al. proposent de traduire les diagrammes de classes de UML vers le langage Z par le biais des règles de traduction [DLCP00b]. Ils ont développé un outil (*RoZ*) et l'ont intégré dans l'environnement Rational Rose [DLCP00a].

2.6 Conclusion

Nous avons présenté, dans ce chapitre dédié à l'état de l'art, les différents travaux de la littérature qui se situent dans le domaine d'adaptation des architectures logicielles. Deux niveaux ont été identifiés, le niveau implémentation et le niveau conception.

Certains travaux utilisent des plans d'adaptations pour la reconfiguration et d'autres ne l'utilisent pas. Pour la réalisation de ces plans d'adaptation de multiples techniques ont été utilisées. Nous avons énuméré ces techniques et nous les avons étudié en détail.

Nous avons présenté également la notion des politiques d'adaptation qui s'appuient sur le schéma E-C-A et qui nécessitent un contexte pour leurs déclenchement.

Dans les chapitres suivants, nous présentons notre approche pour la modélisation et la vérification des politiques d'adaptation.

3

Politiques d'adaptation : Approche proposée

Après l'étude des différentes techniques utilisées pour décrire l'adaptation des architectures logicielles, nous présentons dans ce qui suit notre approche. Notre approche se concentre sur la modélisation et la vérification des politiques d'adaptation des architectures logicielles au niveau conceptuel (design time).

Dans cette approche, nous proposons un profil UML permettant de modéliser les politiques d'adaptation. Ce profil est basé sur le langage semi-formel UML et il étend le diagramme d'activité de UML2.0.

La politique générée à partir du profil peut être ambiguë et peu précise. 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. Afin de pallier à ces inconvénients, nous faisons recours au langage formel Z pour analyser et vérifier la politique d'adaptation. 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.

Notre travail de master s'inscrit dans le cadre des travaux de thèse de Mme. Imen Loulou [LTH⁺09]. Elle a proposé une approche de spécification et de vérification des politiques d'adaptation en utilisant le langage formel Z.

Dans nos travaux de recherche, nous avons apporté des extensions à cette approche. Nous avons enrichi le style architectural avec les propriétés de QdS. Cette extension est motivée du fait que les raisons primordiales à l'origine des échecs des architectures logicielles viennent généralement des propriétés de QdS. Les politiques d'adaptations ont été,

également, enrichies par l'ajout des alternatives pour le choix de la solution la plus appropriée. De même, nous avons opté à spécifier des solutions plus développées et complexes.

Notre approche est appliquée sur les systèmes Publier/Souscrire et elle a été validée à travers deux études de cas. Chaque étude de cas traite un problème de défaillance différent de l'autre. Dans la première étude de cas appelée "Opération d'intervention d'urgence" nous avons traité le cas de perte de connexion entre un dispatcher et un producteur. Dans la deuxième étude de cas appelée "Follow me" nous avons traité le cas de haute latence entre deux managers.

3.1 Profil UML pour la modélisation des politiques d'adaptation

Pour modéliser les politiques d'adaptation, nous proposons un profil UML. Ce profil offre une notation graphique et visuelle basée sur la notation UML2.0. Le profil proposé étend le diagramme d'activité de UML2.0 [OMG03c].

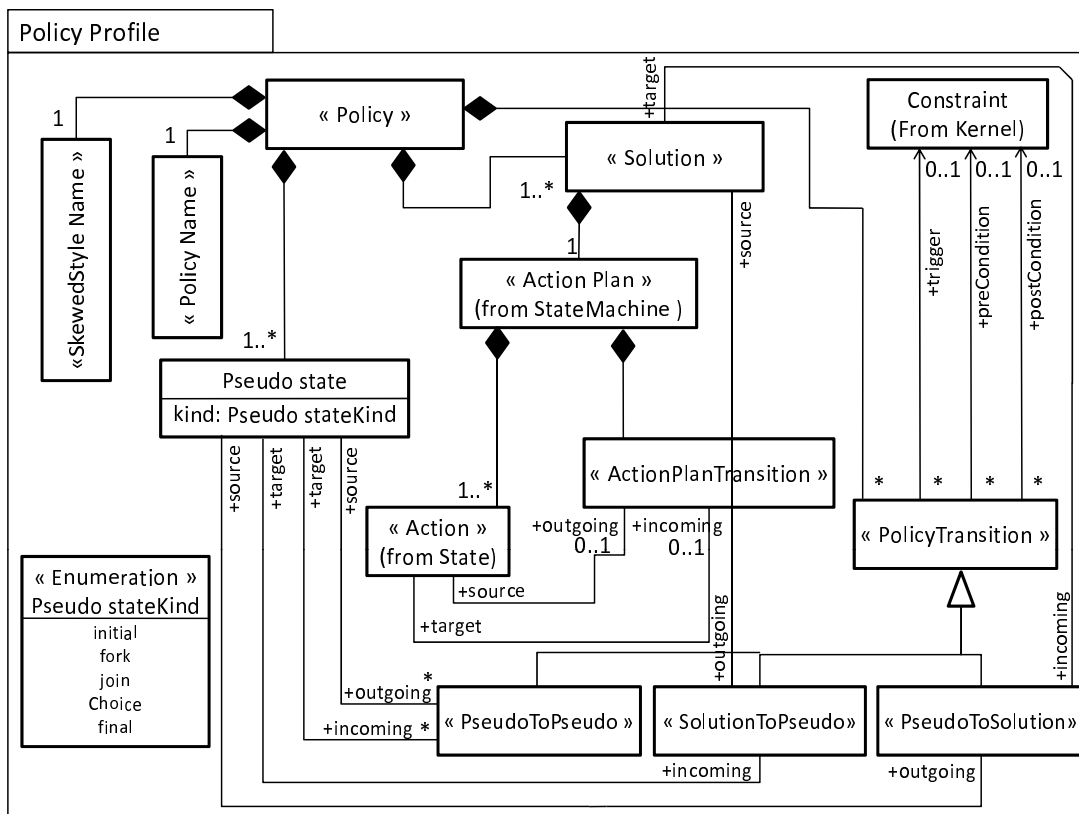


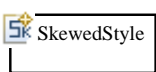
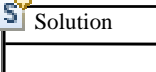
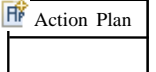
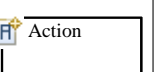
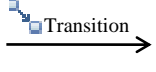
FIG. 3.1 – Profil UML des politiques d'adaptation

Le profil UML, représenté dans la figure 3.1, définit un ensemble de concepts utilisés pour modéliser la structure des politiques d'adaptation.





Le profil est composé d'un méta-modèle enrichi avec de nouvelles notations. Le méta-modèle est décrit par un ensemble de méta-classes spécifiées en notation UML et de stéréotypes que nous définissons. Le tableau 3.1 décrit les différents concepts utilisés et les tableaux 3.2 et 3.3 décrivent les différentes notations utilisées.

Méta classe ou stéréotype	Description	Méta classe de Base	Contrainte
« Policy »	- Décrit une politique d'adaptation. - Composée d'un « PolicyName », un « SkewedStyleName », un ou plusieurs « Solution », plusieurs « PolicyTransition » et un ou plusieurs PseudoState.	Activity	-
« PolicyName »	- Décrit un nom pour identifier la politique d'adaptation à modéliser.	Object	- Une politique contient un est un seul PolicyName.
« SkewedStyleName »	- Décrit un nom pour identifier le style architectural biaisé.	Object	- Une politique contient un est un seul « SkewedStyleName ».
« Solution »	- Décrit une solution pour résoudre les problèmes identifiés. - Composée d'un et un seul « ActionPlan ». - Peut être la source d'un type spécial de « PolicyTransition » nommée « SolutionToPseudo » et la cible de « PseudoToSolution ».	Activity	-
« ActionPlan »	- Décrit le changement de l'état résultant quand les « Action » sont exécutées séquentiellement. - Composé de plusieurs « Action » et « ActionPlanTransition » reliant les « Action ».	State Machine	- Un « ActionPlan » ne peut exister qu'à l'intérieur d'une « Solution ».
« Action »	- Modélise les changements structurels comme l'ajout et la suppression de composant et de connexion et interroge l'état du système.	State	- Une « Action » ne peut exister qu'à l'intérieur d'un « ActionPlan ».
« PolicyTransition »	- Est une généralisation de trois transitions « PseudoToPseudo », « PseudoToSolution » et « SolutionToPseudo ». - « PseudoToPseudo » est une connexion dirigée entre deux PseudoStates. - « PseudoToSolution » connecte un PseudoState à une Solution. - « SolutionToPseudo » connecte une Solution à un PseudoState.	Control flow	- Contient une contrainte appelée <i>trigger</i> pour déclencher une « Policy » ou une <i>postCondition</i> pour déclencher une « Solution ».
PseudoState	- Utilisé pour connecter plusieurs transitions entre elles. - L'attribut <i>kind</i> définit le type du <i>PseudoState</i> et il peut prendre une des valeurs suivantes :	-	-
	Initial	-	- Zéro transition entrante et une et une seule transition sortante.
	Join	-	- A une seule transition sortante.
	Fork	-	- A une seule transition entrante.
	Choice	-	- A une seule transition entrante et plusieurs transitions sortantes.
Final	-	- A une seule transition entrante et zéro transition sortante.	

TAB. 3.1 – Description du profil

Stéréotype	« SkewedStyleName »	« Solution »	« ActionPlan »	« Action »	« PolicyTransition »
Notation					

TAB. 3.2 – Notations de stéréotypes

Méta classe	PseudoState			
	Initial	Join et Fork	Choice	Final
Notation				

TAB. 3.3 – Notations de la méta classe PseudoState

Pour instancier la politique, nous proposons un modèle général pour la politique d'adaptation. Comme décrit dans la figure 3.2, la politique d'adaptation a un nom (*Policy*) et possède un *SkewedStyle* (*Sk_Style*). Ce concept sera détaillé dans la section 3.3. Après la détection d'un événement déclencheur, un ensemble de conditions seront testées pour exécuter la *Solution* la plus appropriée. Dans une *Solution*, nous spécifions le *ActionPlan* et dans le *ActionPlan* nous décrivons les *Actions* et leurs connexions. L'ensemble de toutes les solutions convergent avec le connecteur de jointure pour former une nouvelle architecture (*NewArchitecture*). Ce modèle explicite la forme E-C-A (Événement, Condition, Action) de la politique d'adaptation. Il sera détaillé avec une étude de cas dans la section 3.5.

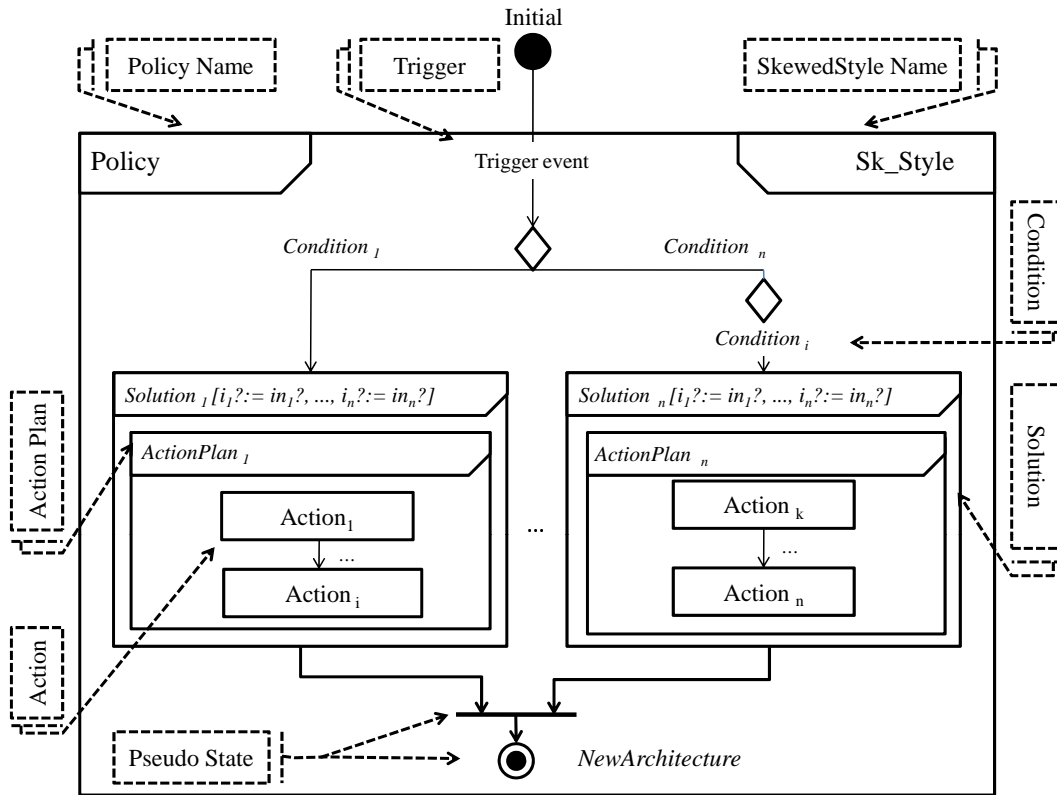


FIG. 3.2 – Modèle général pour la description d'une politique d'adaptation

3.2 Transformation de UML vers Z

La politique générée à partir du profil proposé peut être ambiguë et peu précise. Ceci est dû à l'absence d'une sémantique formelle précise pour UML. En effet, toute erreur ou mauvaise conception de la politique d'adaptation peut causer des problèmes d'adaptation qui peuvent avoir de mauvaises répercussions.

Afin de pallier à ces inconvénients, nous faisons recours au langage formel Z pour analyser et vérifier les politiques modélisées. Nous proposons une transformation automatique de la politique d'adaptation vers le langage Z afin de mener des preuves de vérification. Nous cherchons à prouver les propriétés de fiabilité et de consistance.

La transformation proposée est basée sur des règles de transformation. Ces règles sont utilisées pour transformer le modèle UML de la politique d'adaptation vers des schémas Z.

Dans le profil UML le *ActionPlan* décrit le changement de l'état du système qui résulte de

l'exécution séquentielle des actions. Un *ActionPlan* est transformé vers le langage Z de la manière suivante :

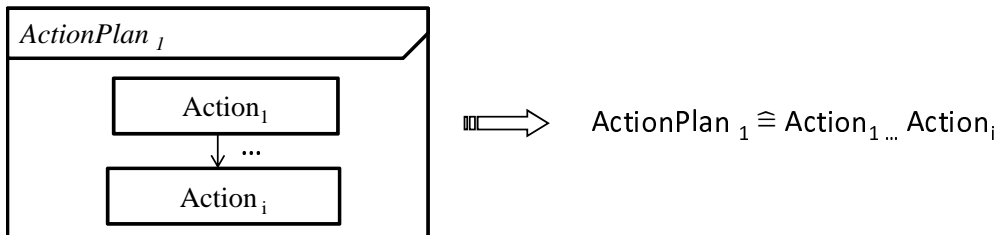


FIG. 3.3 – Règle de transformation d'un *ActionPlan*

Les liens qui relient les actions peuvent être de type “*piping*” ou bien “*sequential composition*” ceci dépend du type des actions. Si c'est une composition séquentielle elle est représentée avec “ \circ ” et si elle est de type *piping* elle est représentée avec l'opérateur de *pinping* “ \gg ”.

Une “*Solution*” est spécifiée avec le langage Z comme suit :

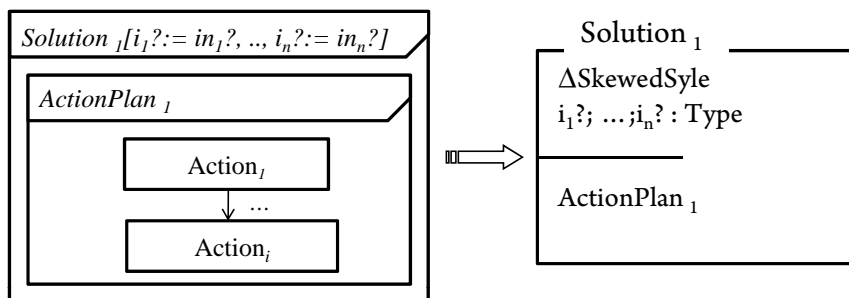


FIG. 3.4 – Règle de transformation d'une *Solution*

Comme nous avons expliqué, une *Solution* décrit le “*ActionPlan*” nécessaire si les conditions déclarées (si elles existent) sont vérifiées.

Une politique d'adaptation, comme elle est représentée dans la figure 3.2, est transformée vers le langage Z comme suit :

<i>Policy</i> <hr/> $\Delta State$ $in_1? ; in_2? ; \dots ; in_n? : Type$ $out_1! ; out_2! ; \dots ; out_m! : Type$ <hr/> <i>Pre – conditions</i> <i>Triggering Event</i> if <i>condition</i> ₁ then <i>Solution</i> ₁ [$i_1? := in_1?, \dots, i_n? := in_n?$] else ... <i>Solution</i> _n [$i_1? := in_1?, \dots, i_n? := in_n?$] <hr/>
--

Avec :

- $\Delta State$ indique que la politique va changer l'état du système.
- $in_i?$ est un paramètre d'entrée pour la politique.
- $out_i!$ est un paramètre de sortie pour la politique. Les politiques peuvent ne pas avoir des paramètres de sortie.
- *Pre – conditions* permettent de surveiller la politique. Elles interdisent l'exécution de la politique si elles ne sont pas satisfaites.
- *Triggering Event* décrit l'erreur détectée qui va déclencher la politique.
- *Solution_i* définit une solution d'adaptation.

Spécifier des politiques d'adaptation est une solution qui n'est pas suffisante. Il faut s'assurer que les politiques définies génèrent un état de système correct vis-à-vis du style architectural. Pour le faire, nous devons prouver les propriétés de fiabilité et de consistance des politiques définies.

3.3 Style architectural

Le style architectural joue un rôle très important dans le processus d'auto-adaptation. Plus précisément, nous l'utilisons pour la détection des anomalies du système. En effet, lorsqu'une contrainte stylistique est violée, la politique d'adaptation correspondante se déclenche pour corriger le problème généré. Nous l'utilisons également pour décider quelle solution faut-il adopter pour corriger le problème. Le style architectural est utilisé également pour s'assurer que les adaptations réalisées sur l'architecture ne violent pas les contraintes architecturales.

Les raisons primordiales à l'origine des échecs de l'architecture logicielle viennent

généralement des propriétés de performance. Pour ces raisons, nous avons enrichi le style architectural par les propriétés de QdS. Nous nous concentrons principalement sur trois propriétés de performance : la charge d'un composant, la latence entre deux composants et la valeur de la bande passante entre deux composants.

La spécification formelle du style architectural Publier/Souscrire est proposée par Mme Imen Loulou dans le cadre de sa thèse [LJDK10].

Le style architectural décrit par la figure 3.5, modélise un réseau de managers de type dispatcher et un ensemble de clients de type consommateurs. Les liens de communication sont modélisés par des liens. Pour un lien, les événements sont propagés du composant fourni vers le composant requis. Par exemple, pour le lien `PushMC`, les événements sont propagés du dispatcher vers le composant consommateur. Un noeud est décrit avec trois étiquettes : le *Rôle*, le *Type*, et le *Nom* de la variable. Ainsi, un noeud marqué (NDisp, X, P) dénote que `P` est un réseau de dispatchers de type `X` (dans notre cas de type `Manager`). Le type et les noms des variables seront instanciés par l'architecte tandis que le rôle est maintenu inchangé. « `NetworkDispatchers` » et « `Client` » sont des stéréotypes dénotant respectivement un réseau de dispatchers et un ensemble de clients.

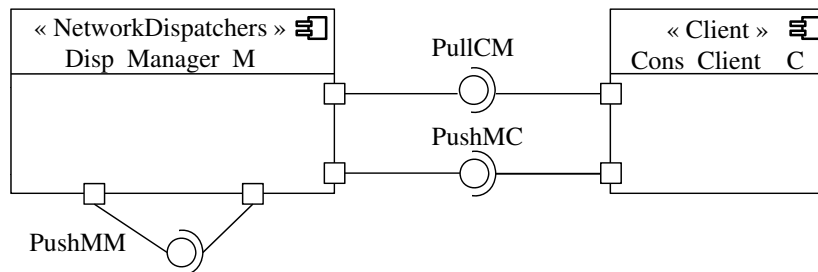


FIG. 3.5 – Exemple d'un style architectural

En utilisant le langage `Z`, ce style est spécifié avec le schéma d'état *ArchStyle* présenté ci-dessous :

<i>ArchStyle</i>	
$M : \mathbb{F} \text{ Manager}$	
$C : \mathbb{F} \text{ Client}$	
$\text{PushMC} : \text{Manager} \leftrightarrow \text{Client}$	
$\text{PullCM} : \text{Manager} \leftrightarrow \text{Client}$	
$\text{PushMM} : \text{Manager} \leftrightarrow \text{Manager}$	
$\text{PushMC} = \text{PullCM} \sim$	[P1]
$\text{PushMM} = \text{PushMM} \sim$	[P2]
$\text{dom PushMC} = M$	[P3]
$\text{ran PushMC} = C$	[P4]

Le premier prédicat [P1] stipule que la communication entre un manager et un client peut être établie par les deux modes de connexion `Push` et `Pull`. Le deuxième prédicat [P2] stipule que la communication entre les managers doit être bidirectionnelle. Le prédicat [P3] stipule que le domaine de `PushMC` correspond à l'ensemble de tous les managers (M). Le prédicat [P4] stipule que l'image de `PushMC` correspond à l'ensemble de tous les clients (C).

Dans ce qui suit, nous introduisons la notion de style biaisé. Comme le montre la figure 3.6, quand une erreur est détectée, la configuration du système devient non conforme au style architectural, elle appartient plutôt à un style biaisé appelé “*Skewed Style*”. Ce style est moins restrictif, puisque la configuration invalide satisfait toutes les contraintes stylistiques sauf celle (celles) qui est (sont) violée (s) (*ViolConst*).

La figure montre l'espace des états valides (resp. invalides) qui représente l'ensemble des configurations correctes (resp. incorrectes) par rapport au style architectural. Chaque configuration (C_i) qui est conforme au style est aussi conforme au style biaisé.

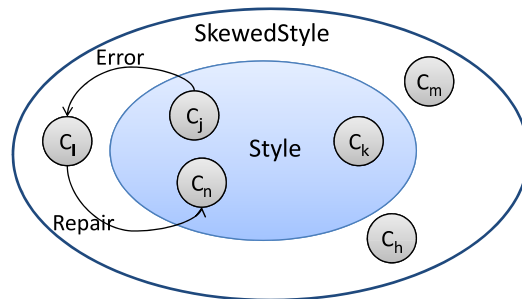


FIG. 3.6 – Espace des états valides et invalides

3.4 Vérification formelle

La transformation vers Z est une phase intermédiaire pour mener des raisonnements formels afin de prouver les propriétés de fiabilité et de consistance. La démarche de vérification a été proposée par Mme. Imen Loulou dans le cadre de sa thèse. Nous détaillons dans ce qui suit ces deux propriétés.

3.4.1 Fiabilité

Une fois déclenchée, la politique placera le système dans une nouvelle configuration représentée par *SkewedStyle'*. Si la politique *Policy* est fiable, la nouvelle configuration

SkewedStyle' doit être une instance du style architectural *Style*. Afin de vérifier que quelque soit l'exécution de la politique, la configuration résultante est conforme au style architectural, nous proposons de prouver le théorème suivant :

Theorem Soundness

$$\forall Policy \bullet SkewedStyle' \Leftrightarrow Style'$$

Le théorème stipule que si la politique est exécutée, la configuration générée est correcte vis-à-vis du style défini. Cependant, nous n'avons aucune garantie que la politique sera exécutée. L'exécution peut ne pas avoir lieu. Ceci pour deux raisons :

- La première est au niveau de la spécification de la politique. Si cette spécification ne préserve pas les contraintes définies par le style (si les solutions proposées effectuent des changements qui violent les contraintes définies).
- La deuxième est au niveau de la configuration initiale elle-même (*SkewedStyle*). Si cette configuration n'appartient pas à l'ensemble des états pour lesquels les changements proposés sont permis.

3.4.2 Consistance

Si nous pouvons montrer que chaque *Solution* proposée par la *Politique* peut être exécutée au moins une fois, nous pouvons conclure que la politique sera toujours exécutée et donc génère une configuration correcte. Pour le faire, nous proposons de tester chaque *Solution* sur une configuration incorrecte et évaluer le résultat d'essai.

La configuration définie est une instance de “*SkewedStyle*”. Formellement, elle est définie par le schéma *TestArchitecture*. Dans la notation *Z*, un état initial est habituellement caractérisé par un schéma décoré, représentant l'état après initialisation. Dans la partie prédictive, nous devons initialiser toutes les variables définies dans la partie déclarative, celles déclarées dans “*SkewedStyle*”.

<i>TestArchitecture</i>
<i>SkewedStyle'</i>
<i>Variable Initialization</i>

Ensuite, nous pouvons spécifier le résultat d'exécution de la politique définie. *PolicyExec* décrit le changement de l'état qui résulte quand la *politique* est exécutée sur *TestArchitecture*.

$$PolicyExec \hat{=} TestArchitecture \S Policy$$

Si l'exécution est effectuée, ceci devrait générer une nouvelle configuration qui doit être correcte si la solution appliquée est fiable. Nous spécifions cette nouvelle configuration avec le schéma d'état *NewArchitecture*. Ce schéma stipule que cette architecture est une instance du style d'origine. Dans la partie prédicative, "*PolicyExec*" exprime les nouvelles valeurs des variables obtenues après l'adaptation.

<i>NewArchitecture</i>
<i>Style'</i>
<i>PolicyExec</i>

Cependant, nous avons spécifié dans la partie déclarative que *NewArchitecture* est une instance de *Style*. Ceci n'est pas encore prouvé. Pour cette raison, le théorème d'initialisation suivant doit être prouvé :

Theorem Consistency
 $\exists \textit{Style}' \bullet \textit{NewArchitecture}$

Si ce théorème est prouvé, ceci assure que la solution testée a été exécutée et elle a généré une nouvelle architecture (*NewArchitecture*) conforme au style architectural.

Pour résumer la procédure de vérification, nous proposons de vérifier le théorème de fiabilité *Soundness*. Après la preuve de ce théorème, nous testons chaque solution sur une configuration incorrecte avec le théorème *Consistency*. Si la preuve réussit, nous pouvons affirmer que la politique définie est fiable et consistante.

Pour illustrer notre approche, nous l'avons testé sur deux études de cas : "Opération d'intervention d'urgence" et "Follow me". Pour la première nous avons spécifié le cas de perte de connexion entre un dispatcher et un producteur [LTH⁺09]. Pour la deuxième, nous avons spécifié le cas de haute latence entre deux managers. Cette étude de cas sera détaillée dans ce qui suit.

3.5 Etude de cas : *Follow me*

Pour illustrer notre approche, nous avons choisi l'étude de cas "*Follow me*" [KN05]. "*Follow me*" est une application audio qui se rend compte de l'endroit du client, et l'endroit des haut-parleurs autour de la salle (voir figure 3.7). Dès son entrée dans un local (hall, réception, restaurant, etc.), le PDA du client envoie un signal à son point d'accès (Manager). Suite à ce signal, un Sink sera informé qu'il devra produire de la musique.

Une fois informé, il produira la musique à son point d'accès (Manager) qui va coopérer avec l'ensemble des autres managers pour faire propager la musique au client.

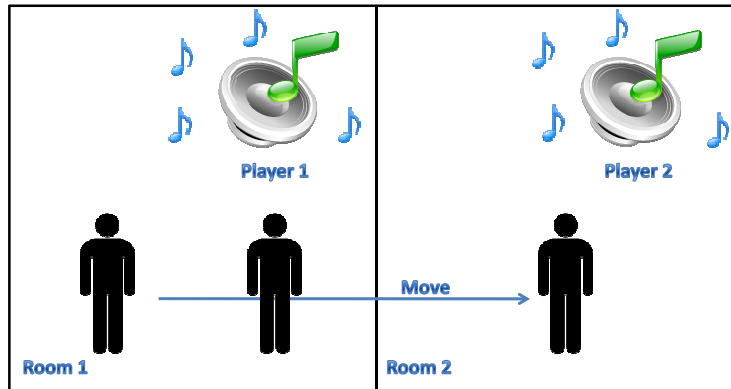


FIG. 3.7 – Etude de cas : *Follow me*

Nous modélisons cette application en utilisant le style architectural basé-événement Publier/Souscrire. La propagation des événement entre les participants est faite à travers un réseau de dispatchers nommé *Managers*. Le *Client* et le *Sink* sont associés à des composants de type producteur-consommateur. Le style architectural de cette application est modélisé en utilisant la notation UML2.0 comme le montre la figure 3.8.

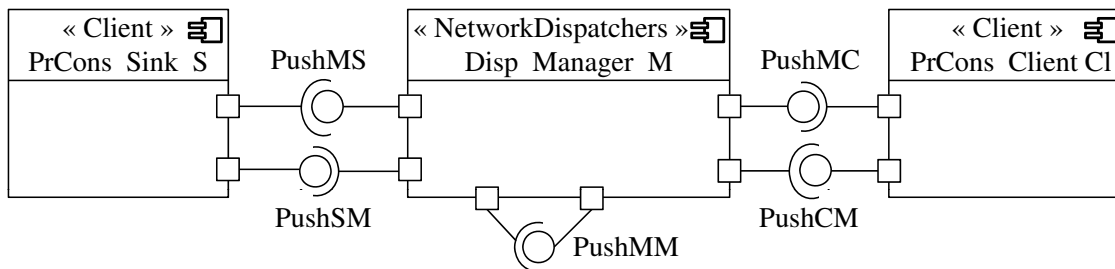


FIG. 3.8 – Le style architectural *Follow me*

La spécification du style architectural est décrite avec le schéma d'état *follow_me*. Les types de composants impliqués sont spécifiés avec le schéma d'état qui explicite le rôle joué par chacun d'eux. Le *Client* et le *Sink* sont deux entités de type Producteur-Consommateur (*PrCons*) et le *Manager* est une entité de type dispatcher (*Disp*).

<i>Client</i>
<i>role</i> : <i>ROLE</i>
<i>role</i> = <i>PrCons</i>

<i>Sink</i>
<i>role</i> : <i>ROLE</i>
<i>role</i> = <i>PrCons</i>

<i>Manager</i>
<i>role</i> : <i>ROLE</i>
<i>role</i> = <i>Disp</i>

ROLE est défini comme un type libre contenant tous les rôles possibles qui peuvent être joués par les entités de type Publier/Souscrire :

$ROLE ::= PrCons \mid Prod \mid Cons \mid Disp$

<i>follow_me</i>	
$Cl : \mathbb{F} Client$	
$S : \mathbb{F} Sink$	
$M : \mathbb{F} Manger$	
$PushCM : Client \leftrightarrow Manager$	
$PushMC : Manager \leftrightarrow Client$	
$PushSM : Sink \leftrightarrow Manager$	
$PushMS : Manager \leftrightarrow Sink$	
$PushMM : Manager \leftrightarrow Manager$	
$PushCM = PushMC^{\sim}$	[P1]
$PushSM = PushMS^{\sim}$	[P2]
$dom PushCM \subseteq Cl \wedge ran PushCM \subseteq M$	[P3]
$dom PushMS \subseteq M \wedge ran PushMS \subseteq S$	[P4]
$dom PushMM = M \wedge ran PushMM = M$	[P5]
$\forall x, y : M \mid x \neq y \wedge (x, y) \in PushMM \vee (y, x) \in PushMM$	
$\bullet latency(x, y) \leq maxLatency \vee latency(y, x) \leq maxLatency$	[P6]

Jusqu'à présent, nous avons présenté la conception du style de l'application "Follow me" en faisant abstraction de la topologie d'interconnexion gouvernant les dispatchers. Nous proposons donc de raffiner ce style afin de prendre en considération la topologie. Cette topologie est la topologie générale P2P.

De plus, nous augmentons le style architectural avec des propriétés de QdS ([P6]) car les raisons les plus importantes derrière les échecs de l'architecture de logiciel viennent généralement des propriétés de QdS.

Toutes les propriétés décrites dans la partie prédicative sont reliées aux règles de communication entre les différents composants. Elles doivent être respectées et préservées durant le processus d'adaptation. En conséquence, nous devons définir une liste de politiques d'adaptation qui seront déclenchées si une violation d'une contrainte stylistique est détectée.

3.5.1 Modélisation des politiques d'adaptation

Prenons un exemple d'une politique d'adaptation. Cette politique est déclenchée quand un événement de haute latence est détecté entre deux *Managers* (figure 3.9), c'est-à-dire quand la contrainte (1), représentée dans la figure 3.13 ([P6] de la spécification du style `follow_me`), est violée. Suite à cette violation de contrainte, le système passe à une configuration non conforme au style architectural `follow_me`.

La politique que nous proposons offre trois solutions possibles selon la cause de la défaillance.

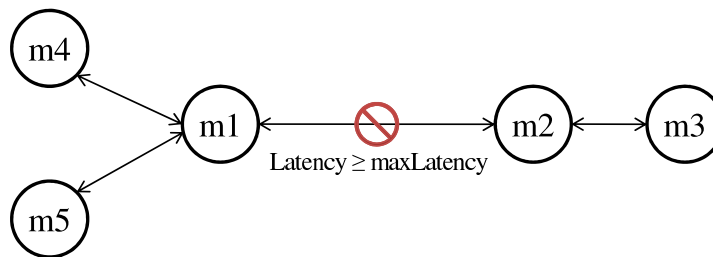


FIG. 3.9 – Configuration initiale

- Si la politique détecte un problème de haute latence et l'un des deux managers est surchargé, la solution proposée, comme elle est présentée dans la figure 3.10, consiste à répliquer le manager avec les liens qu'il admet pour alléger sa charge et relier le nouveau manager à ses voisins.

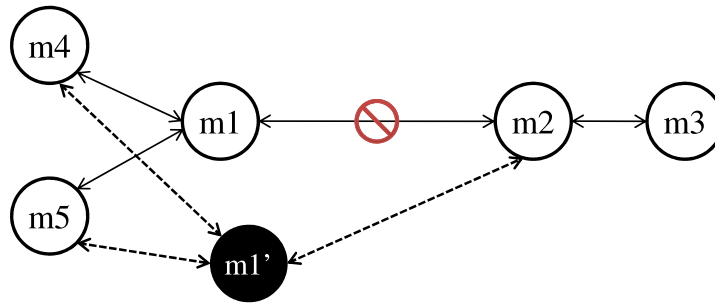


FIG. 3.10 – La configuration après l'application de la Solution1

- Si la politique détecte une dégradation de la valeur de la bande passante de la connexion et l'un des deux managers a des voisins, la solution proposée, comme elle est présentée dans la figure 3.11, consiste à rediriger le lien au manager le plus proche et le moins chargé parmi les voisins de ce manager.

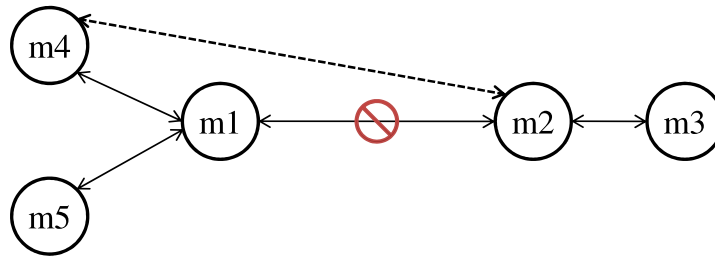


FIG. 3.11 – La configuration après l'application de la Solution2

- Si la politique détecte une dégradation de la valeur de la bande passante de la connexion et la configuration contient seulement deux managers, la solution proposée, comme elle est présentée dans la figure 3.11, consiste à créer un nouveau manager et le connecter à ces deux autres managers.

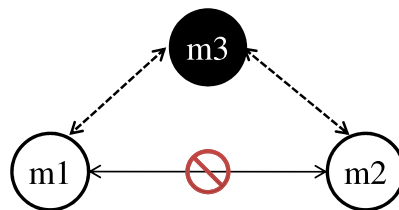


FIG. 3.12 – La configuration après l'application de la Solution3

La modélisation de la politique d'adaptation et de ses différentes solutions selon notre approche donne le modèle suivant illustré par la figure (3.13).

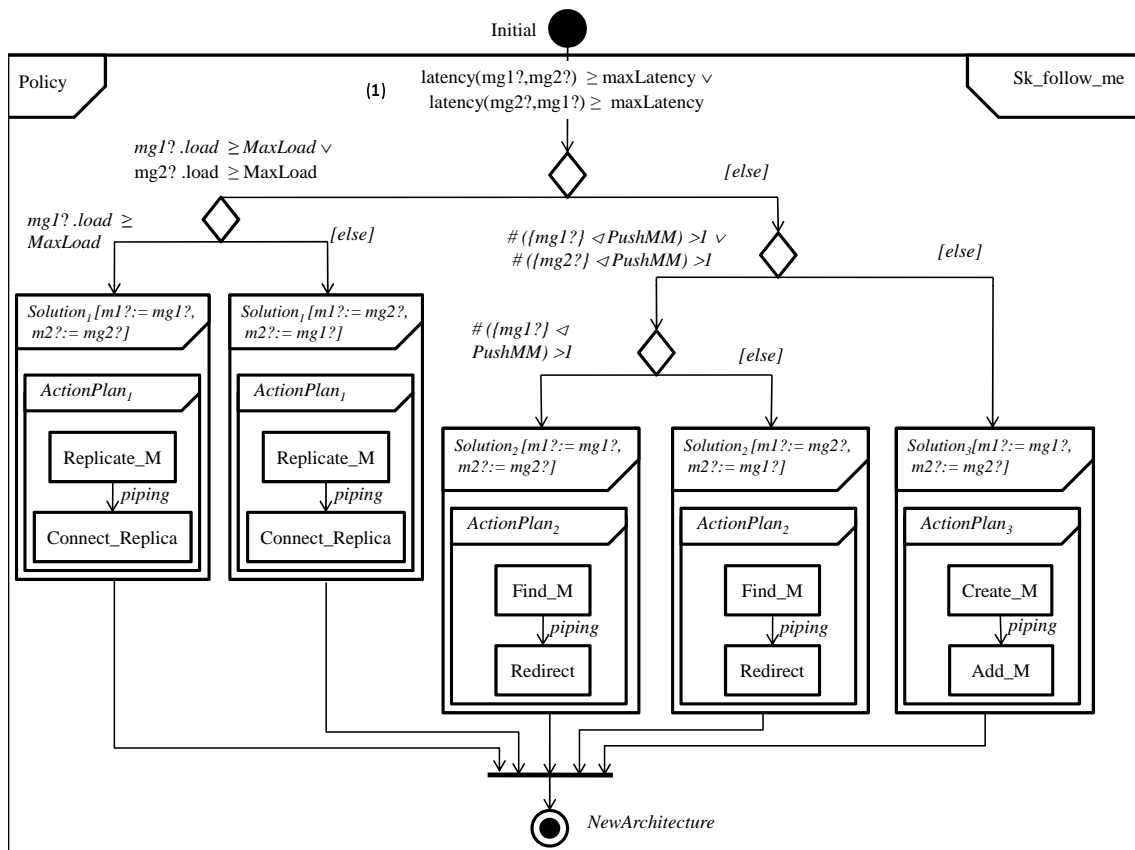


FIG. 3.13 – Modèle de la politique d'adaptation

Comme décrit dans le modèle, le nom de la politique est `Policy` et celui du style biaisé est `Sk_follow_me`.

A partir d'un état initial et avec un événement déclencheur (1) la politique d'adaptation est déclenchée. Après le test d'un nombre de conditions la solution la plus appropriée est déclenchée. Pour la première solution (*Solution₁*) nous spécifions le plan d'action *ActionPlan₁*. Dans *ActionPlan₁* nous spécifions deux actions, *Replicate_M* (Replicate Manager) et *Connect_Replica*, connectées avec le connecteur de piping.

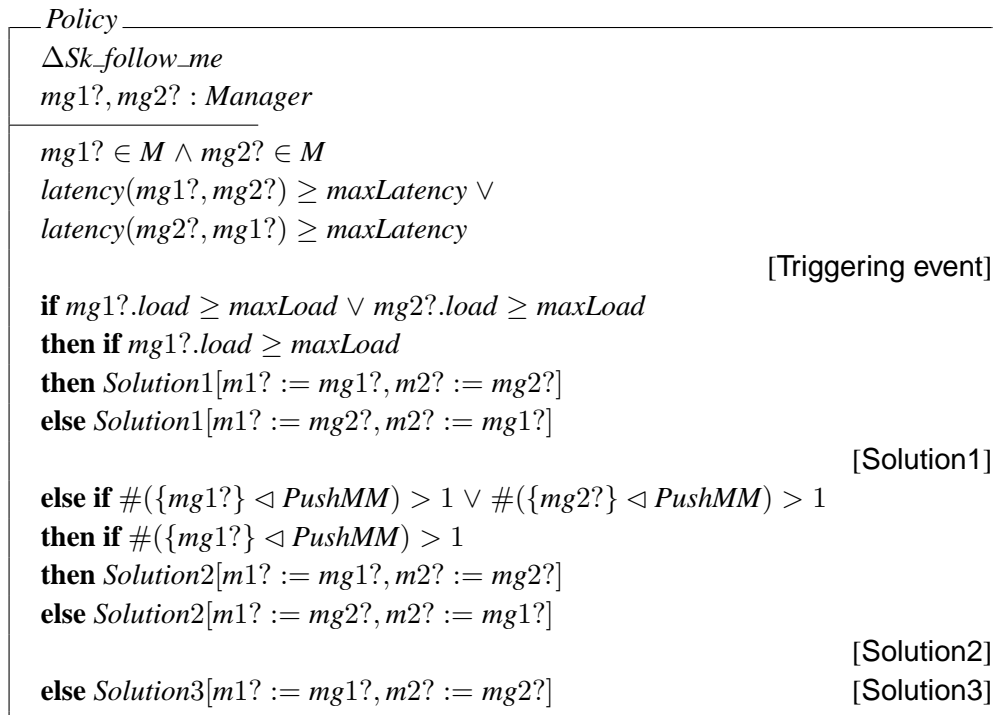
Pour la deuxième solution (*Solution₂*) nous spécifions le plan d'action *ActionPlan₂*. Dans *ActionPlan₂* nous spécifions deux actions, *Find_M* (Find Manager) et *Redirect*, connectées aussi avec le connecteur de piping.

Pour la troisième solution (*Solution₃*) nous spécifions le plan d'action *ActionPlan₃*. Dans *ActionPlan₃* nous spécifions deux actions, *Create_M* (Create Manager) et *Add_M* (Add Manager), connectées également avec le connecteur de piping.

La politique d'adaptation utilise une stratégie dans laquelle elle exécute l'une des solu-

tions. Si la première alternative réussit elle applique le plan d'action correspondant. Sinon elle teste la solution suivante. L'exécution de la solution appropriée ramène le système à une configuration résultante nommée *NewArchitecture*.

La transformation du modèle de la politique d'adaptation vers le langage Z est représentée via le schéma Z suivant :



Avec :

- $latency(mg1?, mg2?)$ représente la latence entre les deux managers $mg1$ et $mg2$.
- $maxLatency$ spécifie la latence maximale entre deux managers.
- $load$ décrit la charge d'un manager.
- $maxLoad$ représente la charge maximale d'un manager.

Dans ce qui suit, nous détaillons la *Solution₂*. Elle propose un *Action_Plan* appelé *ActionPlan₂* exprimé par une succession de deux actions *Find_M* (Find manager) et *Redirect*. Ce *Action_Plan* est utilisé pour rediriger le lien de communication vers un autre manager. Il déconnecte le manager surchargé de l'autre manager (le lien *Push_{MM}*) et le relie avec le manager le moins chargé parmi ses voisins les plus proches.

$\frac{\text{Solution2}}{\Delta Sk_follow_me}$ $m1?, m2? : Manager$ <hr/> $Action_Plan2[m1? := m1?, m2? := m2?]$

$$ActionPlan2 \hat{=} Find_M \ggg Redirect$$

La transformation formelle des actions `Find_M` et `Redirect` sont représentées respectivement dans les schémas Z suivants.

$\frac{Find_M}{Sk_follow_me}$ $m1?, m! : Manager$ <hr/> $find(m1?) \in M$ $m! = find(m1?)$

$\frac{Redirect}{\Delta Sk_follow_me}$ $m?, m2? : Manager$ <hr/> $PushMM' = PushMM \cup \{(m?, m2?), (m2?, m?)\}$ $M' = M$ $C' = C$ $S' = S$ $PushCM' = PushCM$ $PushMC' = PushMC$ $PushSM' = PushSM$ $PushMS' = PushMS$
--

3.5.2 Vérification formelle

Afin de prouver la fiabilité et la consistance de ces politiques, nous avons fait recours à l'outil de preuve Z-EVES [Saa99, Bol01].

Dans nos travaux, nous avons introduit les propriétés de QdS dans le style architectural. Ces propriétés n'entrent pas dans la processus de vérification. Elles sont utilisées dans le déclenchement de la politique et dans le choix de la bonne solution.

Nous avons appliqué la démarche présentée dans la section 3.4. Nous avons instancié le théorème de fiabilité. Avec ce théorème, nous avons démontré que chaque exécution de la politique placera le système dans un état conforme au *Structural_Style* (structural style est le style architectural contenant uniquement les propriétés structurelles). Nous avons prouvé le théorème *Sound_Policy* représenté dans la figure 3.14.

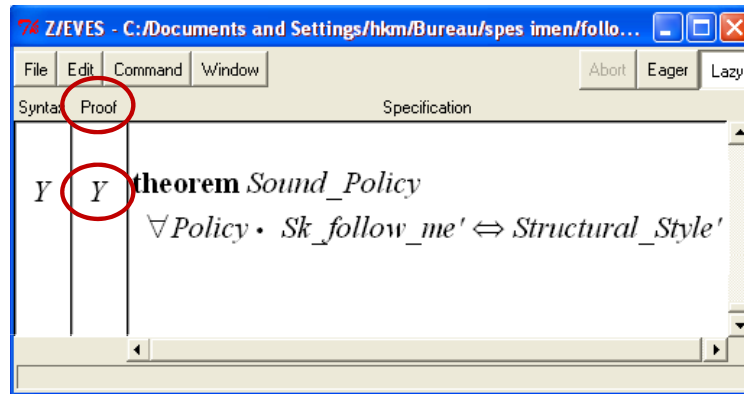


FIG. 3.14 – Preuve du théorème de fiabilité

Nous devons prouver également que chaque solution proposée par la politique peut être exécutée au moins une fois. Pour le faire, nous avons testé chaque solution sur une configuration incorrecte. Ensuite nous avons évalué le résultat du test.

Par exemple, pour tester $Solution_1$ il faut passer par trois étapes. La première consiste à spécifier un état initial invalide. La configuration contient deux managers, la latence entre ces deux managers est supérieure au seuil et le manager $m2$ est surchargé .

$Init_Sk_follow_me$ Sk_follow_me'
$M = \{m1, m2\}$ $Cl = \{c1, c2\}$ $S = \{s1, s2\}$ $PushMM = \{(m1, m2), (m2, m1)\}$ $PushCM = \{(c1, m1)\}$ $PushMC = \{(m1, c1)\}$ $PushSM = \{(s1, m2)\}$ $PushMS = \{(m2, s1)\}$

$m1, m2 : Manager$ $c1, c2 : Client$ $s1, s2 : Sink$
$maxLatency = 5$ $maxLoad = 5$ $latency(m1, m2) = 7$ $m1.load = 2$ $m2.load = 8$

La deuxième étape consiste à spécifier le résultat de l'exécution de *Policy*. Ce résultat est décrit comme suit :

$$Policy_Exec \hat{=} Init_Sk_follow_me' \circ Policy[mg1? := m1, mg2? := m2]$$

<i>NewArchitecture</i>
<i>Structural_Style'</i>
<i>Policy_Exec</i>

La troisième étape consiste à prouver un deuxième théorème nommé *Consistent_Policy*. Le but est d'affirmer que *NewArchitecture* est une instance de *Structural_Style'*.

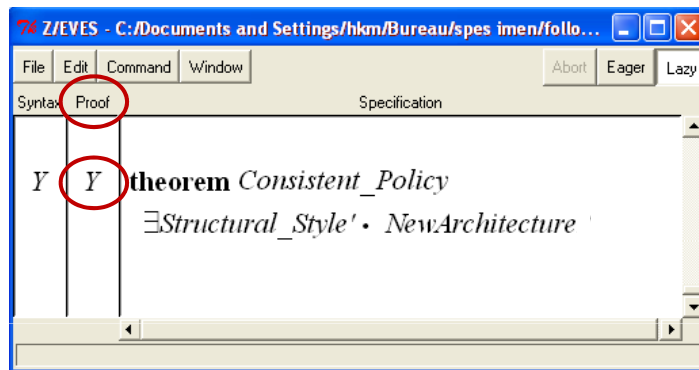


FIG. 3.15 – Preuve du théorème de consistance

Comme le montre la figure 3.15, la valeur Y dans la colonne *Proof* signifie que le théorème a été prouvé avec succès. Ceci signifie que la solution examinée a été exécutée et a généré une nouvelle configuration *NewArchitecture*. Cette configuration respecte toutes les propriétés structurelles des contraintes stylistiques définies par le style *follow_me* ([P1]...[P5]).

Pour prouver ce théorème, nous avons utilisé un ensemble de commandes de preuves exécutées séquentiellement dans l'outil Z-EVES. Ces commandes sont représentées dans l'Annexe A.

En prouvant ces théorèmes pour toutes les solutions, nous assurons que la politique définie est fiable et nous serons sûres que cette politique ramène le système dans un état correct vis-à-vis des propriétés structurelles du style architectural *follow_me*.

3.6 Conclusion

Nous avons présenté dans ce chapitre une approche visuelle et graphique pour la modélisation et la vérification des politiques d'adaptation. Nous avons présenté dans un premier temps un profil UML pour la modélisation des politiques d'adaptation. Dans un deuxième temps, nous avons présenté le processus de transformation d'un modèle UML vers le langage *Z* et nous avons adopté la démarche de vérification proposée par Mme. Imen Loulou dans le cadre de sa thèse visant à vérifier formellement les propriétés de fiabilité et de consistance. Dans un troisième temps, nous avons validé notre approche à travers une étude de cas "Follow me".

Dans le chapitre suivant, nous présentons un plug-in Eclipse que nous avons développé pour la modélisation des politiques d'adaptation.

4

Un plug-in Eclipse pour la modélisation des politiques d'adaptation

Pour que notre approche soit valide, elle doit offrir aux utilisateurs les moyens pour la manipuler. Pour cela, nous avons implémenté un plug-in Eclipse. Il offre un ensemble d'outils permettant, d'une part, la modélisation des politiques d'adaptation d'une façon graphique et visuelle et d'autre part, il permet la transformation des modèles générés vers le langage *Z*.

Notre choix de la plateforme Eclipse a été motivé puisqu'il intègre, en natif, un ensemble d'outils et de plug-ins appropriés permettant le développement des plug-ins qui offrent la possibilité de modéliser des applications.

Dans ce chapitre, nous détaillons dans un premier temps, les étapes de réalisation d'un diagramme éditeur avec l'outil GMF. Ensuite, nous décrivons l'éditeur graphique, les étapes de sa réalisation et la manière de son utilisation. Enfin, nous présentons l'implémentation des règles de transformation d'un modèle UML vers le langage *Z*.

4.1 Création d'un plug-in Eclipse

Le plug-in que nous avons développé inclut un éditeur graphique, offrant les outils nécessaires, pour la conception des politiques d'adaptation conformément au profil UML proposé. La conception de la politique d'adaptation est générée sous la forme d'un document XML. Le plug-in transforme le fichier XML, selon les règles de transformation, vers des spécifications *Z* enregistrées dans un fichier \LaTeX (.tex). Ces règles sont exprimées

avec le langage XSLT (eXtensible Styles Language Transformation). Le fichier \LaTeX sera par la suite importé sous l'environnement Z/Eves pour la visualisation des schémas Z et la vérification des théorèmes. Ces différentes étapes sont représentées par la figure 4.1.

1. Conception de la politique

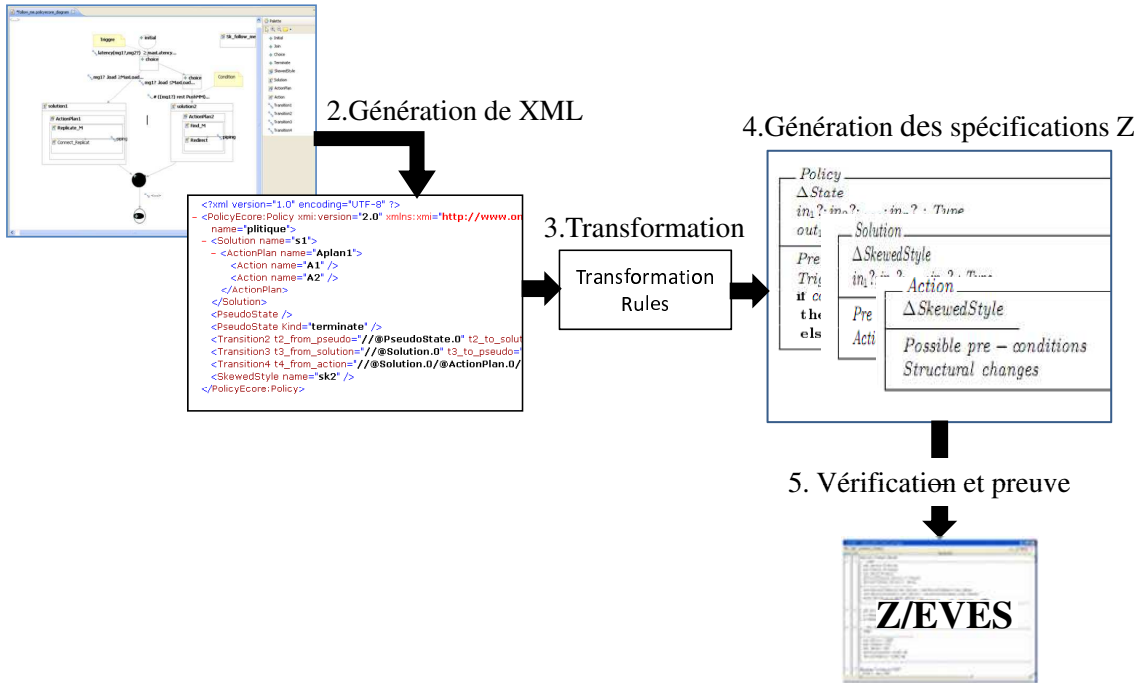


FIG. 4.1 – Étapes de transformation d'un modèle de la politique vers le langage Z

Pour le développement de ce plug-in, nous avons fait appel à une famille de solutions autour de la plate-forme Eclipse. En effet, dans le but de créer un éditeur graphique sous Eclipse, nous avons utilisé les plug-ins GMF [GMF] (Graphical Modeling Framework), EMF [EMF] (Eclipse Modeling Framework) et GEF [GEF] (Graphical Editing Framework).

Le plug-in GMF permet la génération d'un éditeur graphique. Il utilise EMF pour créer le méta-modèle et le GEF pour implémenter des dessins graphiques.

4.1.1 Plug-ins Eclipse pour la modélisation

GMF [GMF] est un plug-in Eclipse qui permet de réaliser un éditeur graphique pour un langage dont la syntaxe abstraite est donnée sous forme d'un méta-modèle (*ecore*). Il fournit une composante de génération et d'exécution des infrastructures qui permet de développer des éditeurs graphiques. GMF est basé sur les plug-ins EMF et GEF.

La figure 4.2 montre les dépendances entre l'éditeur graphique généré, GMF, EMF, GEF et la plateforme Eclipse.

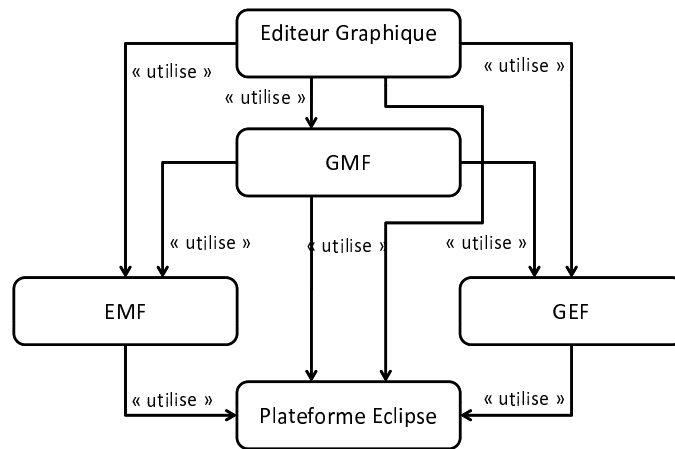


FIG. 4.2 – Dépendances entre les plug-ins de Eclipse

4.1.2 Étapes de réalisation d'un diagramme éditeur avec GMF

Pour créer un projet GMF, il faut suivre un ensemble d'étapes. La première consiste à créer un nouveau projet GMF. La deuxième est constituée de trois phases : la première phase est la définition d'un modèle du domaine (Domain Model) dans un fichier *ecore*. La deuxième phase sert à définir, dans un fichier *gmfgraph*, les différents dessins graphiques à utiliser dans l'éditeur. La troisième phase sert à définir, dans un fichier *gmftool*, la palette de l'éditeur. La troisième étape est le mapping. C'est l'étape la plus importante pour la création d'un projet GMF. Le mapping, génère un fichier *gmfmap*, et consiste à faire la correspondance entre les concepts du méta-modèle, les figures et les éléments de la palette. Afin de générer le code de l'éditeur, il est indispensable de créer le fichier *gmfgen*. Ce fichier synthétise les informations provenant des différents modèles.

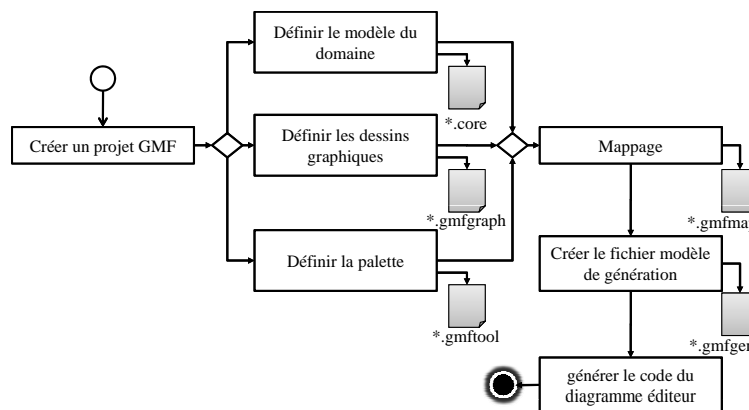


FIG. 4.3 – Étapes de réalisation d'un projet GMF

4.2 Éditeur graphique pour la politique d'adaptation

Nous présentons dans ce qui suit le plug-in Eclipse que nous avons développé. Ce plug-in est le résultat de l'application d'un ensemble d'étapes. Avant de les détailler, et afin de donner une idée sur la réalisation obtenue, nous présentons l'éditeur résultant par une illustration avec l'étude de cas "Follow me" (figure 4.4).

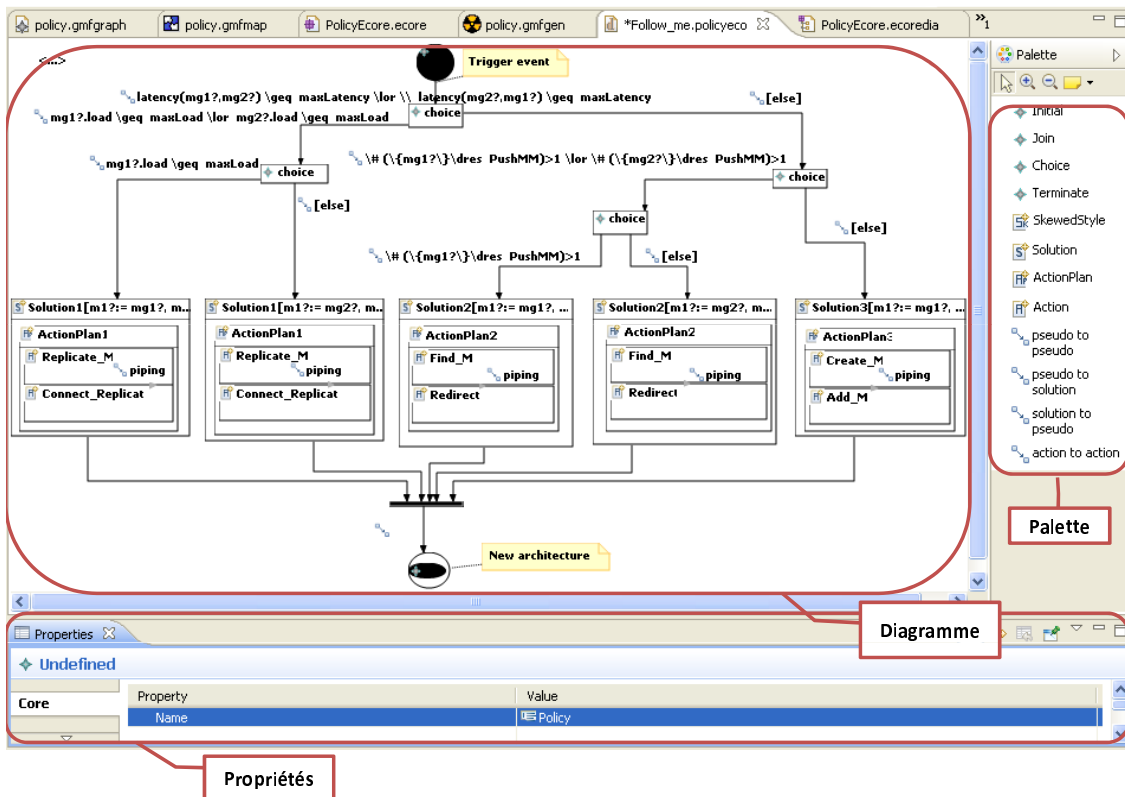


FIG. 4.4 – Diagramme éditeur sous Eclipse

4.2.1 Définition d'un modèle du domaine (.ecore)

Le profil UML que nous avons défini est implémenté, selon la technologie GMF, sous forme d'un modèle du domaine (figure 4.5). Ce modèle est une extension du diagramme d'activité de UML2.0. Il décrit les différentes entités et liens utilisés dans la politique ainsi que leurs propriétés.

Un fichier *ecore* peut être représenté sous la forme d'une arborescence comme le montre la figure 4.5, ou bien sous la forme d'un diagramme de classe (figure 4.6). Nous pouvons créer le deuxième type de fichier grâce au projet "Ecore Tools" d'Eclipse.

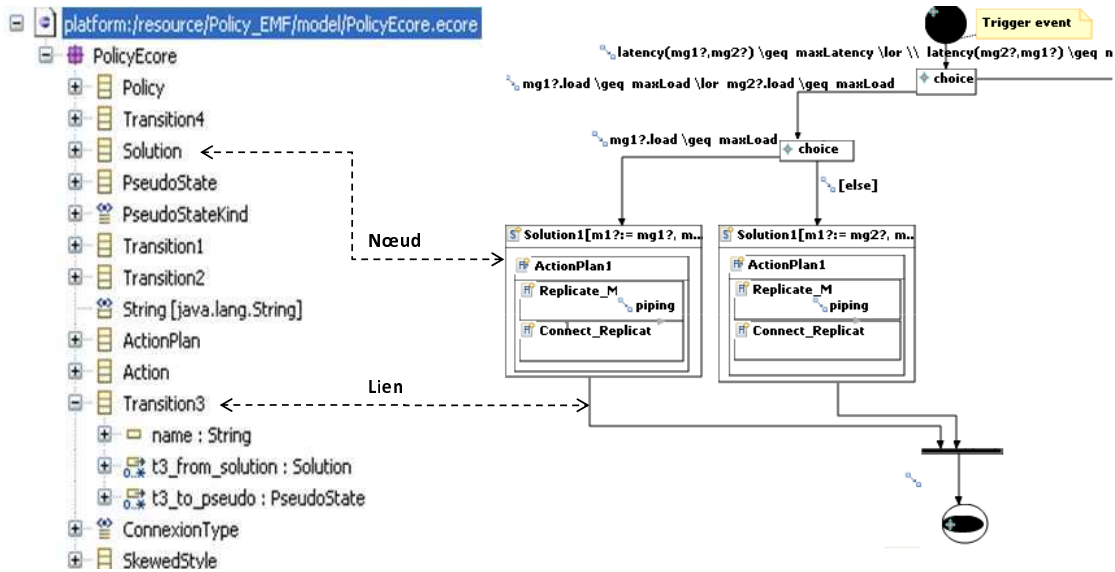


FIG. 4.5 – Le modèle du domaine (.ecore)

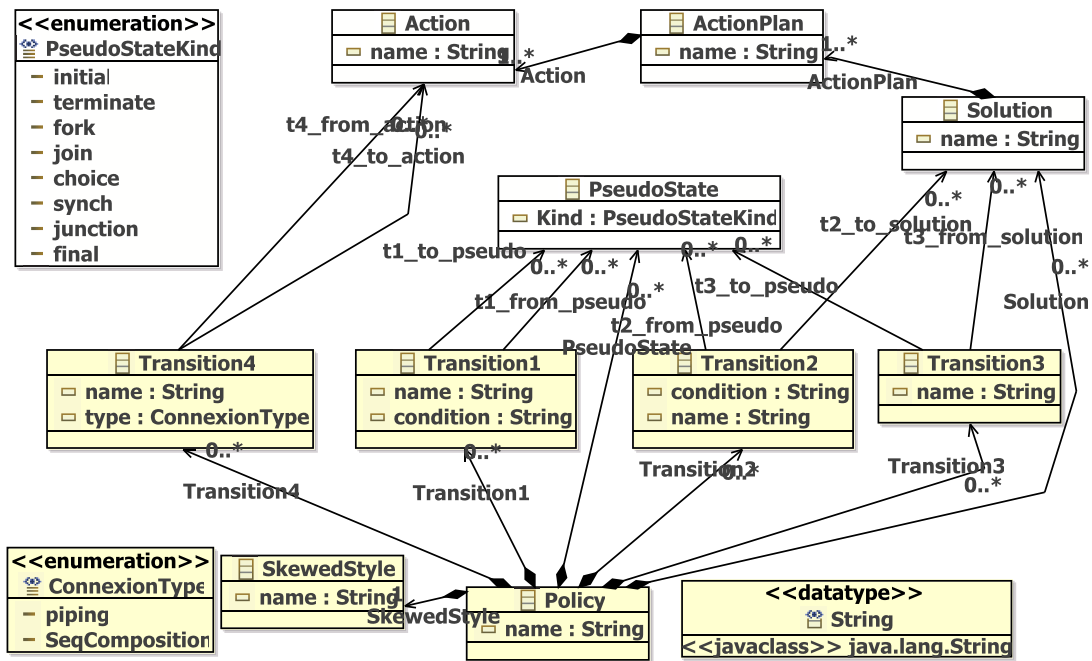


FIG. 4.6 – Visualisation de l'ecore sous la forme d'un diagramme de classe (.ecorediag)

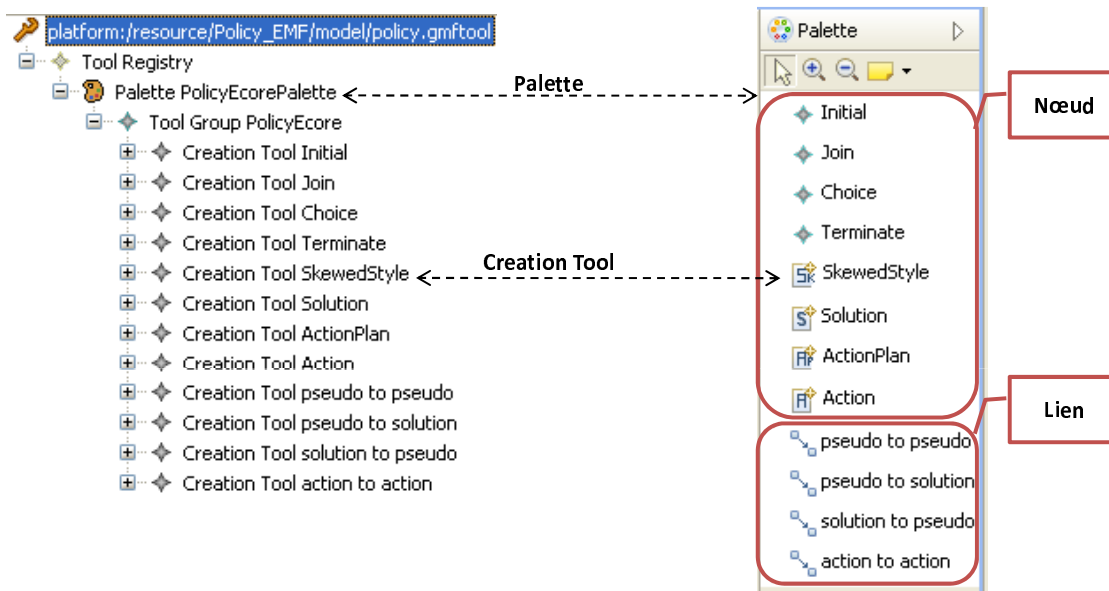
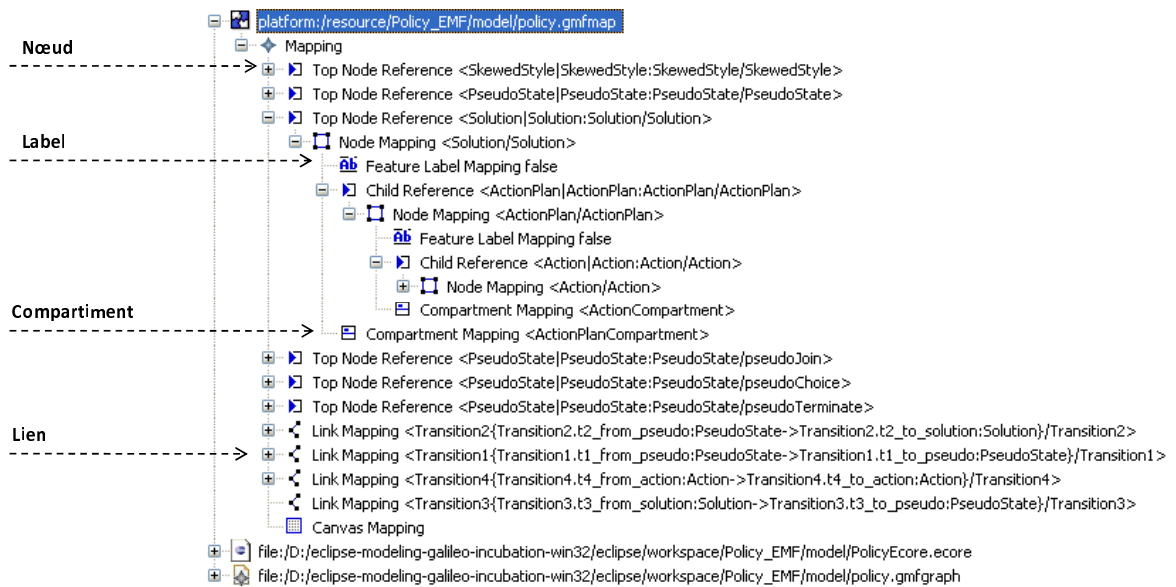


FIG. 4.8 – Le modèle d'outils (.gmftool)

4.2.4 Définition du mapping (.gmfmap)

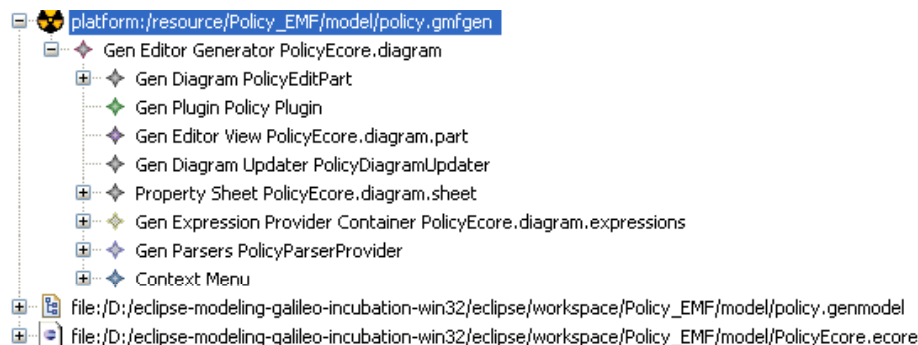
Jusqu'à présent, nous avons défini le modèle du domaine, le modèle graphique et le modèle d'outil. Dans ce qui suit, nous présentons le mapping (figure 4.9) qui fait le lien entre les entités du modèle du domaine, l'outil associé dans la palette et la figure correspondante dans le graphe.

Dans le fichier *gmfmap* nous définissons les compartiments (*SolutionCompartment* et *ActionPlanCompartment*) pour que les figures (*Solution* et *ActionPlan*) puissent contenir plusieurs autres figures (*ActionPlan* et *Action* respectivement). Nous avons introduit également les contraintes OCL qui permettent de dessiner différentes figures (les figures correspondant à *Initial*, *Join*, *Choice* et *Terminate*) pour une seule entité d'ecore (*PseudoState*).

FIG. 4.9 – Le modèle du mapping (*.gmfmap*)

4.2.5 Création du fichier modèle de génération (*.gmfgen*)

Le fichier *gmfgen* englobe toutes les informations nécessaires permettant de générer le code du diagramme éditeur (figure 4.10).

FIG. 4.10 – Le modèle de génération (*.gmfgen*)

4.2.6 Génération du code de diagramme éditeur (.diagram)

A partir du fichier *gmfgen*, le code de l'éditeur sera généré d'une façon automatique. A ce niveau là, nous sommes capables de concevoir un modèle de politique d'adaptation comme celui présenté dans la figure 4.4.

4.2.7 Génération du plug-in

Le déploiement de notre projet permet la génération d'un plug-in Eclipse. Ce plug-in englobe les différentes fonctionnalités dans des fichiers *jar*. Ces fichiers seront intégrés avec l'installation standard d'Eclipse. Notre éditeur de politiques d'adaptation sera accessible à travers les menus d'Eclipse.

La génération de ces fichiers est représentée dans la figure 4.11.

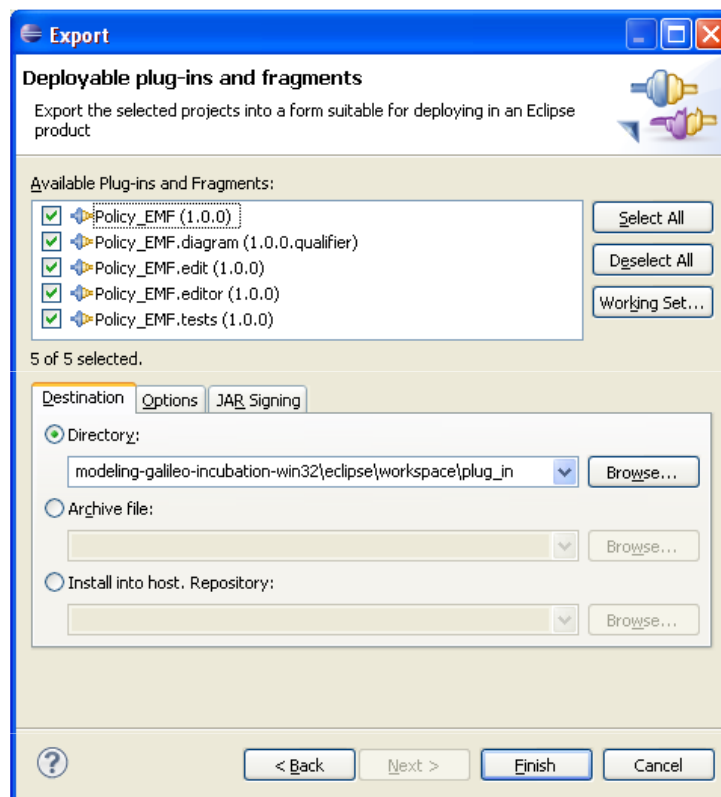


FIG. 4.11 – Génération du plug-in

4.3 Description de l'utilisation du plug-in

Notre plug-in permet de créer un diagramme éditeur pour la conception des politiques d'adaptation par l'éditeur représenté dans la figure 4.12. Pour dessiner un élément graphique, il suffit de le tirer de la palette d'outils et de le mettre dans le volet éditeur de diagramme selon la technologie "drag-and-drop".

Nous avons implémenté et intégré un ensemble de contrôles. A titre d'exemple, nous avons la possibilité de créer un et un seul style biaisé. Un plan d'action ne peut être créé qu'à l'intérieur d'une solution. Une action ne peut être créée qu'à l'intérieur d'un plan d'action.

Nous pouvons générer deux types de fichiers. Le premier est de type *policycore_diagram*. Dans ce fichier, le diagramme de la politique est représenté graphiquement, comme représenté dans la figure 4.4. Le deuxième est de type *policycore*. Nous pouvons lire ce fichier de deux manières différentes, soit comme une arborescence, comme le montre la figure 4.13, soit sous forme d'un fichier XML, comme le montre la figure 4.14.

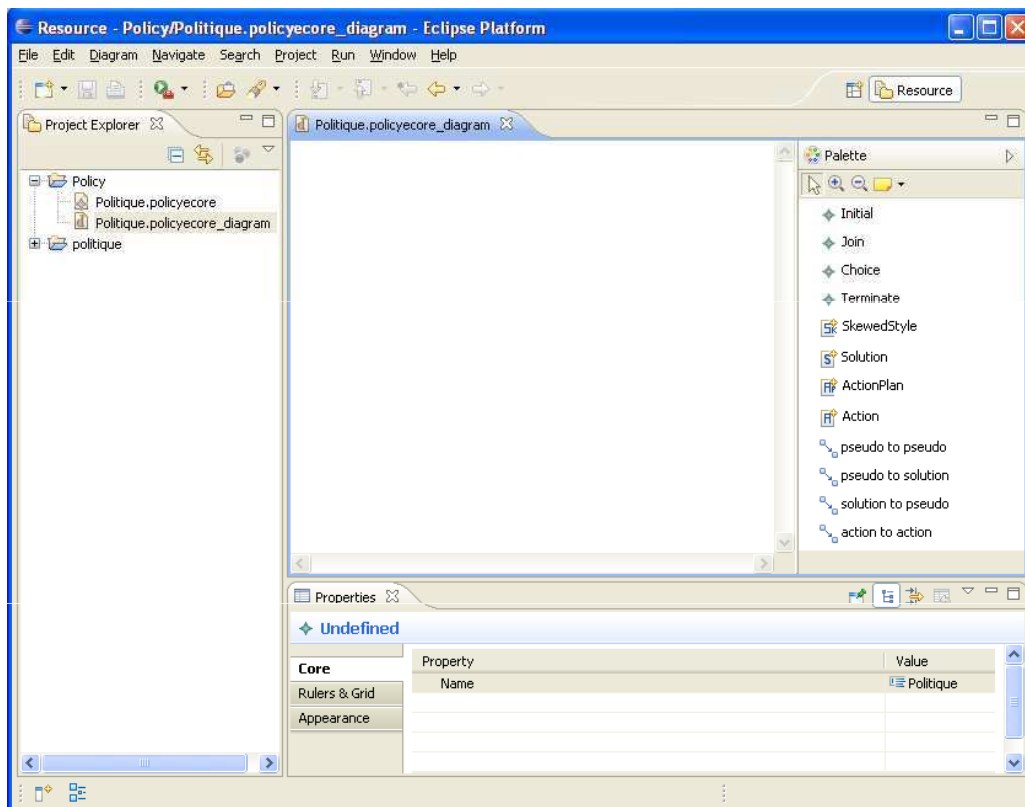


FIG. 4.12 – Éditeur graphique

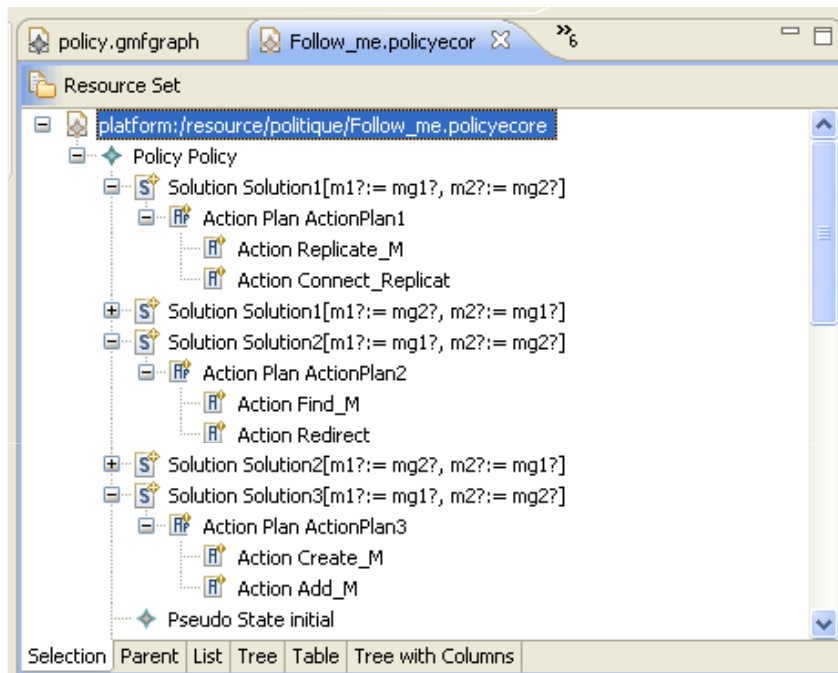


FIG. 4.13 – Forme arborescence

```

<?xml version="1.0" encoding="UTF-8"?>
<PolicyEcore:Policy xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  <Solution name="Solution1[m1?:= mg1?, m2?:= mg2?]">
    <ActionPlan name="ActionPlan1">
      <Action name="Replicate_M"/>
      <Action name="Connect_Replicat"/>
    </ActionPlan>
  </Solution>
  <Solution name="Solution1[m1?:= mg2?, m2?:= mg1?]">
    <ActionPlan name="ActionPlan1">
      <Action name="Replicate_M"/>
      <Action name="Connect_Replicat"/>
    </ActionPlan>
  </Solution>
  <Solution name="Solution2[m1?:= mg1?, m2?:= mg2?]">
    <ActionPlan name="ActionPlan2">
      <Action name="Find_M"/>
      <Action name="Redirect"/>
    </ActionPlan>
  </Solution>
  </PolicyEcore:Policy>

```

FIG. 4.14 – Fichier XML

4.4 Transformation d'un document XML vers une spécification Z

Après la modélisation de la politique d'adaptation et la génération d'un document XML, nous transformons ce document vers une spécification Z générée sous la forme d'un fichier \LaTeX . Les règles de transformation sont décrites avec le langage XSLT [XSL]. Cette transformation est réalisée dans le cadre d'un projet de fin d'études à l'Institut Supérieur d'Informatique et de Multimedia de Sfax par Mr. Jassim Awini et Mr. Idriss Chafroud.

4.4.1 XSLT : eXtensible Stylesheet Language Transformations

XSL est une norme du W3C qui accompagne la norme XML. L'objectif majeur de cette spécification était de transformer des documents XML en des documents HTML. Ceci dans l'objectif de décrire la présentation de données définies par le formalisme XML.

De nos jours, XSL est décomposé en deux langages, un langage de transformation et un langage de formatage. Le premier permet de transformer un document XML vers un autre document (XML ou non), alors que le deuxième langage permet d'utiliser des balises pré-définies pour représenter l'aspect visuel d'un document XML. Ces deux langages peuvent être utilisés indépendamment l'un de l'autre.

Dans le cadre de nos travaux, nous nous sommes uniquement intéressés au langage XSLT [XSL].

4.4.2 Règles de transformation

Un processus XSLT permet de transformer un document XML vers un autre document (XML ou non). Pour cela, il applique les transformations décrites par une feuille de style XSLT à un document XML afin de produire un document correspondant aux transformations spécifiées. Ce principe est décrit par la figure 4.15.

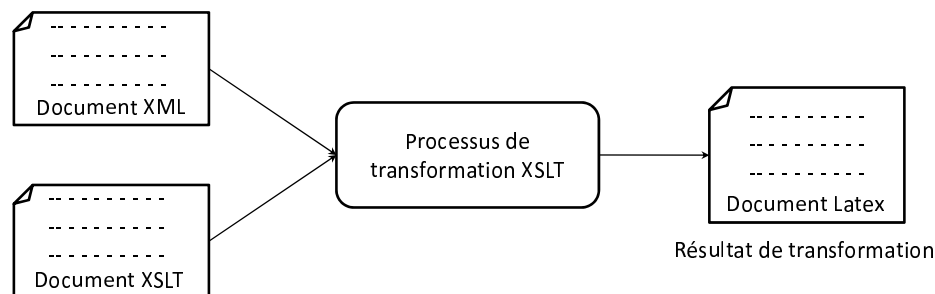


FIG. 4.15 – Principe de fonctionnement d'un processus XSLT

Chaque règle de transformation est composée de deux parties. La première partie indique quand la règle peut être exécutée. La deuxième partie indique l'information substituée à celle d'origine. La syntaxe générale d'une règle XSLT est représentée par le fragment XSLT ci-dessous.

```

1 <xsl:template match="informationà substituer" >
2   nouvelle information
3 </xsl:template>

```

Listing 4.1 – Règle XSLT de transformation

Les règles de transformation, décrites dans la section 3.2 du chapitre précédent, sont écrites avec le langage XSLT. Les fragments XSLT suivants représentent les règles de transformation d'un Plan d'Action, d'une Solution et d'une Politique.

```

1 ...
2 <xsl:variable name="a" select="../Solution[1]/ActionPlan/@name"/>
3 ...
4 <xsl:variable name="x1" select="concat(substring-before($x,'-'),'\\-',substring-after($x,'-'))"/>
5 ...
6 <xsl:variable name="y1" select="concat(substring-before($y,'-'),'\\-',substring-after($y,'-'))"/>
7 ...
8 \begin{zed}
9 <xsl:value-of select = "substring-before($a,[''])" />
10 \sdef
11 <xsl:value-of select="$x1"/>
12 \zpipe
13 <xsl:value-of select="$y1"/>
14 \end{zed}

```

Listing 4.2 – Règle XSLT pour transformer un *Plan d'Action*

```

1 ...
2 <xsl:variable name="c" select="../Solution[1]/@name"/>
3 ...
4 <xsl:variable name="p" select="substring-after($a,['='])"/>
5 <xsl:variable name="p1" select="substring-before($p,[''])"/>
6 ...
7 \begin{schema}
8 {<xsl:value-of select = "substring-before($c,[''])" />}
9 \Delta
10 <xsl:value-of select="$m1"/> \\
11 <xsl:value-of select="$p1"/>:Manager \\
12 \where
13 <xsl:value-of select="../Solution[1]/ActionPlan/@name"/>
14 \end{schema}

```

Listing 4.3 – Règle XSLT pour transformer une *Solution*

```

1 ...
2 <xsl:variable name="l1" select="substring-before($l,[':'])"/>
3 ...
4 <xsl:value-of select="$p1"/>,<xsl:value-of select="$l1"/>:Manager \\
5 ...
6 \begin{schema}{Policy}
7 \Delta <xsl:value-of select="$m1"/> \\
8 <xsl:value-of select="$p1"/>,<xsl:value-of select="$l1"/>: Manager\\
9 \where
10 <xsl:value-of select="../Transition1[1]/@condition"/>\\
11 if \ <xsl:value-of select="../Transition1[2]/@condition"/>\\

```

```
12  then\  
13  if\ <xsl:value-of select="../Transition2[1]/@condition"/>\\  
14  then\ <xsl:value-of select="../Solution[1]/@name"/>\\  
15  else\ <xsl:value-of select="../Solution[2]/@name"/>\\  
16  else\ <xsl:value-of select="../Solution[3]/@name"/>\\  
17  \end{schema}
```

Listing 4.4 – Règle XSLT pour transformer une *Politique*

4.5 Conclusion

Dans ce chapitre, nous avons détaillé les différentes étapes d'implémentation et de création d'un plug-in Eclipse. Ce plug-in est conçu pour la modélisation graphique des politiques d'adaptation. Il traduit le diagramme modélisé et génère un document XML. Puis, un programme écrit avec le langage XSLT transforme automatiquement le document XML en un fichier Z au format \LaTeX . Ceci pour faire les vérifications formelles nécessaires avec l'outil Z-EVES et s'assurer que les politiques générées sont fiables et consistantes.

Conclusion Générale

Dans ce rapport, nous avons tout d'abord présenté une étude sur les concepts de base des architectures logicielles et sur l'adaptation dynamique. Puis nous avons passé en revue l'état de l'art et nous avons étudié la littérature concernant les domaines de recherche abordés. Grâce à cette étude et afin de réduire les limites des autres approches et de proposer une solution en vue de gérer l'adaptation dynamique des architectures logicielles, nous avons proposé une approche basée sur un profil UML2.0 pour la modélisation formelle de l'adaptation architecturale. Le profil offre une notation visuelle permettant de décrire les politiques d'adaptation. Cette approche permet aussi la génération automatique des spécifications formelles des politiques ainsi que la vérification de leur fiabilité et de leur consistance. Le processus de vérification est codé dans la notation Z et implémenté sous le système de preuve Z/EVES. Nous avons instancié notre approche en utilisant le style architectural Publier/Souscrire et nous l'avons validée à travers deux études de cas. Afin d'offrir au concepteur un éditeur graphique pour la modélisation des politiques d'adaptation, nous avons implémenté un plug-in sous Eclipse. Ce plug-in permet aussi de transformer automatiquement le modèle graphique vers la notation formelle Z à travers des règles de transformation implémentées avec le langage XSLT.

Dans le contexte de l'auto-adaptabilité curative, le processus souvent suivi est formé de quatre phases à savoir : la phase de monitoring, la phase d'analyse, la phase de planification et la phase d'exécution. Dans nos travaux, on s'est intéressé uniquement à la phase de planification. Comme perspectives, nous prévoyons d'aborder l'adaptation dans toutes les phases du processus.

Nous prévoyons aussi que les politiques définies deviennent dynamiquement adaptables. L'adaptation de la politique inclut le changement dynamique des paramètres de la politique et le choix de la politique qui devrait être déclenchée parmi un ensemble prédéfini de politiques en temps d'exécution. Le choix de la politique peut être réalisé grâce au mécanisme d'apprentissage. Ce mécanisme nous offre la possibilité même de générer de nouvelles politiques si c'est nécessaire.

Bibliographie

- [ABvBGJ95] Vogel Andreas, Kerhervé Brigitte, von Bochmann Gregor, and Gecsei Jan. Distributed multimedia and QOS : A survey. *IEEE MultiMedia*, 2(2) :10–19, 1995.
- [ADB⁺99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99 : Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.
- [ADG98] 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.
- [ANC96] Corporate Act-Net Consortium. The active database management system manifesto : a rulebase of adbms features. *SIGMOD Rec.*, 25(3) :40–49, 1996.
- [AP03] 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.
- [Baa05] 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.
- [BAP06] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Afpac : Enforcing consistency during the adaptation of a parallel component. *Scalable Computing : Practice and Experience*, 7(3) :83–95, September 2006. electronic journal (<http://www.scpe.org/>).
- [BHP06] Tomas Bures, Peter Hntynka, and Frantisek Plastil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *Proc. of SERA 2006*, pages 40–48, 2006.
- [Bla05] Xavier Blanc. *MDA en action : Ingénierie logicielle guidée par les modèles*. Eyrolles, 1ère édition edition, Avril 2005.
- [BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

- [Boa95] Maarten Boasson. The artistry of software architecture. *IEEE Software*, 12(6) :13–16, 1995.
- [Bol01] Claude Bolduc. Les démonstrateurs automatiques de théorèmes. Technical report, Université Laval, Laval, Août 2001.
- [Bru06] 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.
- [Car03] Cyril Carrez. *Contrats Comportementaux pour Composants*. Phd thesis, l’école Nationale Supérieure des Télécommunications, Spécialité : Informatique et Réseaux, Décembre 2003.
- [CBB09] Franck Chauvel, Olivier Barais, Jean-Marc Jézéquel, and Isabelle Borne. Un processus à base de modèles pour les systèmes auto-adaptatifs. *Revue de l’Electricité et de l’Electronique (REE)*, pages 38–44, 2009.
- [CGS⁺02] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for grid computing. In *HPDC*, pages 389–398. IEEE Computer Society, 2002.
- [Cha07] Tarak Chaari. *Adaptation d’applications pervasives dans des environnements multi-contextes*. Thèse de doctorat en informatique, INSA de Lyon, September 2007.
- [CLB⁺01] Thierry Coupaye, Romain Lenglet, Mikael Beauvois, Eric Bruneton, and Pascal Déchamboux. Composants et composition dans l’architecture des systèmes répartis. Octobre 2001.
- [CMPC03] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *In : Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pages 1–8. ACM Press, 2003.
- [CMPC04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic algorithms for reliable content-based publish-subscribe : An evaluation. In *ICDCS ’04 : Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04)*, pages 552–561, Washington, DC, USA, 2004. IEEE Computer Society.
- [CN01] Joelle Coutaz and Laurence Nigay. Architecture logicielle conceptuelle des systèmes interactifs. *Analyse et conception de l’IHM.*, chapitre 7 :207–246, 2001.
- [CN04] Georgas John C. and Taylor Richard N. Towards a knowledge-based approach to architectural adaptation management. In *WOSS ’04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 59–63, New York, NY, USA, 2004. ACM.
- [CZL09] Tarak Chaari, Mohamed Zouari, and Frédérique Laforest. Ontology Based Context-Aware Adaptation Approach, April 2009. Context-Aware Mobile and Ubiquitous Computing for Enhanced Usability : Adaptive Technologies and Applications. Dragan Stojanovic (Ed) ISBN 978-1-60566-290-9. IGI publishing.

- [DB02] Garlan David and Schmerl Bradley. Model-based adaptation for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.
- [DLCP00a] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An overview of RoZ : A tool for integrating UML and Z specifications. In *CAiSE'00 : Proceedings of the 12th International Conference Advanced Information Systems Engineering, Stockholm, Sweden*, volume 1789 of *Lecture Notes in Computer Science*, pages 417–430. Springer, 2000.
- [DLCP00b] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z. *L'Objet, numéro thématique Méthodes formelles pour les objets*, 6(1), 2000.
- [EMF] Emf. <http://www.eclipse.org/modeling/emf/>.
- [End98] 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.
- [Ezr99] Michel Ezran. *Réutilisation logicielle*. Eyrolles, 1999.
- [FBLPS97] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Malcolm 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.
- [FGI06] Manel Fredj, Nikolaos Georgantas, and Valérie Issarny. Adaptation to connectivity loss in pervasive computing environments. In *Proceedings of the 4th MiNEMA workshop in Sintra*, pages 77–82, 2006.
- [Fow04] Martin Fowler. *UML 2.0*. CompusPress, Paris, 1^{ère} édition, 5^{ème} tirage edition, 2004.
- [GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT'94 : Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.
- [Gar00] 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.
- [GBR07] Martin Gogolla, Fabian Battner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 1(69) :27–34, 2007.
- [GBV06] Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. Madcar : an abstract model for dynamic and automatic (re-)assembling of component-based applications. In *9th Int. SIGSOFT Symposium on Component-Based Software Engineering (CBSE2006), No 4063, LNCS*, pages 360–367. Springer, 2006.

- [GDC07] Karim Guennoun, Khalil Drira, and Christophe Chassot. Architectural adaptability management for mobile cooperative systems. In *MUE '07 : Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering*, pages 1130–1135, Washington, DC, USA, 2007. IEEE Computer Society.
- [GEF] Gef. <http://www.eclipse.org/gef/>.
- [Geo05] John C. Georgas. Knowledge-based architectural adaptation management for self-adaptive systems. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 658–658, New York, NY, USA, 2005. ACM.
- [GMF] Gmf. <http://www.eclipse.org/modeling/gmf/>.
- [Had08] Mohamed Hadj Kacem. *Modélisation des applications distribuées à architecture dynamique : Conception et Validation*. Phd, Université Paul Sabatier, Toulouse III, en cotutelle avec l'Université de Sfax en Tunisie, Novembre 2008.
- [HGM04] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6) :643–652, 2004.
- [Hil] Rich Hilliard. IEEE-std-1471-2000 recommended practice for architectural description of software-intensive systems. website.
- [HL94] Henry Habrias and David Lightfoot. *La spécification formelle avec Z*. TEKNEA, Toulouse, Janvier 1994.
- [HMA06] Parzyjegla Helge, Gero Muhl, and Michael A.Jaeger. Reconfiguring publish/subscribe overlay topologies. In *ICDCSW '06 : Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 29, Washington, DC, USA, 2006. IEEE Computer Society.
- [HP05] Matthew J. Hawthorne and Dewayne E. Perry. Architectural styles for adaptable self-healing dependable systems. (ICSE05), May 2005.
- [IBRZ00] Vladimir Issarny, Luc Bellissard, Michel Riveill, and Apostolos Zarras. Component-based programming of distributed applications. *Journal of Distributed Systems*, 1752/2000 :327–353, 2000.
- [Jae05] Michael A. Jaeger. Self-organizing publish/subscribe. In *DSM '05 : Proceedings of the 2nd international doctoral symposium on Middleware*, pages 1–5, New York, NY, USA, 2005. ACM.
- [JMWP06] Michael A. Jaeger, Gero Mühl, Matthias Werner, and Helge Parzyjegla. Reconfiguring self-stabilizing publish/subscribe systems. In *DSOM*, pages 233–238, 2006.
- [Kac08] Mohamed Hadj Kacem. *Modelling of distributed applications with dynamic architecture : design and validation*. PhD thesis, LAAS - Toulouse, nov 2008.
- [KBC02] Abdelmadjid Ketfi, Nouredine Belkhatir, and Pierre-Yves Cunin. Adaptation dynamique concepts et expérimentations. In *ICSSEA'02 : Proceedings of the 15th International Conference Software, Systems and their Applications*, CNAM Paris, France, Décembre 2002.

- [KC02] Soon-Kyeong Kim and David A. Carrington. A formal model of the UML metamodel : The UML state machine and its integrity constraints. In *ZB'02 : Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 497–516, London, UK, 2002. Springer.
- [KKJ10] Slim Kallel, Mohamed Hajd Kacem, and Mohamed Jmaiel. Modeling and enforcing invariants of dynamic software architectures. *Software and Systems Modeling*, page 23, April 2010.
- [Kkk08] Siwar Khelifi, Hatem Hadj kacem, and Ahmed Hadj kacem. Specification and verification of the structural and behavioural properties of publish/subscribe architectures. In *Second International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2008)*,, 2008.
- [KN05] Robin Kirk and Jan Newmarch. A location-aware, service-based audio system. *IEEE Consumer Communications and Networking Conference*, 2005.
- [KOS06] Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE Softw.*, 23(2) :22–30, 2006.
- [KS00] Mohamed Mancona Kandé and Alfred Strohmeier. Towards a UML profile for software architecture descriptions. In *UML'00 : Proceedings of the 3rd International Conference on Unified Modeling Language. Advancing the Standard, York, UK*, volume 1939 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2000.
- [LBN99] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *AAAI '99/IAAI '99 : Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.
- [LBZ05] Carlo Ghezzi Luciano Baresi and Luca Zanolin. Modeling and validation of publish/subscribe architectures. In Springer Berlin Heidelberg, editor, *Testing Commercial-off-the-Shelf Components and Systems*, number 978-3-540-21871-5 (Print) 978-3-540-27071-3 (Online), pages 273–291, 2005.
- [LJDK10] Imen Loulou, Mohamed Jmaiel, Khalil Drira, and Ahmed Hadj Kacem. P/S-CoM : Building correct by design publish/subscribe architectural styles with safe reconfiguration. *Journal of Systems and Software*, 83(3) :412–428, March 2010.
- [Lop05] 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.
- [LPR03] Bass Len, Clements Paul, and Kazman Rick. *Software Architecture in Practice*. Addison-Wesley, second edition edition, April 2003.

- [LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In *FME'02 : Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 21–40, London, UK, 2002. Lecture Notes in Computer Science Springer-Verlag.
- [LTH⁺09] Imen Loulou, Imen Tounsi, Mohamed Hadj Kacem, Ahmed Hadj Kacem, and Mohamed Jmaiel. Making sound policies for self-healing systems. In *9^{ème} Conférence Internationale sur les Nouvelles Technologies de REpartition (NOTERE'09)*, pages 178–185, Montreal - Canada, July 2009.
- [LTHK⁺09] Imen Loulou, Imen Tounsi, Mohamed Hadj-Kacem, Ahmed Hadj-Kacem, and Mohamed Jmaiel. A formal architecture-centric approach for safe self-repair. In *In Second International Conference on Web and Information Technologies (ICWIT'09)*, Kerkennah Island, Sfax, Tunisia, June 2009.
- [LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical Report CSL-TR-95-674, Stanford, CA, USA, 1995.
- [MDK94] 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.
- [MG03] Arun Mukhija and Martin Glinz. Casa – a contract-based adaptive software. In *Proc. of 3rd IEEE Workshop on Applications and Services in Wireless Networks*, pages 275–286, 2003.
- [MG04] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, Paris, 2^{'ème} édition, 5^{'ème} tirage edition, 2004.
- [MG05] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of 18th International Conference on Architecture of Computing Systems*, 2005.
- [MKMG97] 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.
- [MNN02] Mikic-Rakic Marija, Mehta Nikunj, and Medvidovic Nenad. Architectural style requirements for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 49–54, New York, NY, USA, 2002. ACM.
- [MOO07] Francisco José MOOMENA. *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.
- [MS97] Irwin Meisels and Mark Saaltink. The Z/EVES Reference Manual (for Version 1.5). Reference manual, ORA Canada, 1997.
- [Nas05] 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.

- [NByCC02] Abdelmajid Ketfi Nouredine, Nouredine Belkhatir, Pierre yves Cunin, and Adele Team Bat C. Adapting applications on the fly. In *ASE'02 : Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 313. IEEE Computer Society, 2002.
- [NR99] Elisabetta Di Nitto and David Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *ICSE'99 : Proceedings of the 21st international conference on Software engineering*, pages 13–22, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [Oli05] Goaer Olivier. De l'adaptation des composants logiciels vers leur évolution. Masterthesis, Laboratoire d'Informatique de Nantes Atlantique, 7 septembre 2005.
- [OMG03a] OMG. MDA guide version 1.0.1, document number : omg/2003-06-01. OMG document, June 2003.
- [OMG03b] OMG. UML 2.0 OCL specification. OMG document, url : <http://www.omg.org>, Octobre 2003.
- [OMG03c] OMG. UML 2.0 superstructure specification, final adopted specification. Omg document, August 2003.
- [OMG03d] OMG. The unified modelling language 2.0 - object constraint language 2.0 proposal. OMG document, url : <http://www.omg.org>, 2003.
- [OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.
- [PC06] Jacques Printz and Yves Caseau. *Architecture logicielle : Concevoir des applications simples, sûres et adaptables*. InfoPro. DUNOD-Paris, 1ère édition edition, 2006.
- [PM03] Jorge Enrique Pérez-Martinez. Heavyweight extensions to the UML meta-model to describe the C3 architectural style. *SIGSOFT Software Engineering Notes*, 28(3) :5–5, 2003.
- [PMJ06] Helge Parzyjegla, Gero Muhl, and Michael A. Jaeger. Reconfiguring publish/-subscribe overlay topologies. In *ICDCS Workshops*, page 29. IEEE Computer Society, 2006.
- [PMSA04] 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 1st 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.
- [RL00] Songsakdi Rongviriyapanish and Nicole Lévy. Variations sur le style architectural Pipe and Filter. In *Approches formelles dans l'assistance au développement de logiciels - AFADI'2000*, page 17 p, Grenoble, France, 2000. none. Colloque avec actes et comité de lecture. nationale.

- [RL04] Issam Rebaï and Jean-Marc Labat. Des métadonnées pour la description des composants logiciels pédagogiques. In *Technologies de l'Information et de la Connaissance dans l'Enseignement Supérieur et l'Industrie TICE 2004*, pages 80–87, Compiègne France, 10 2004. Université de Technologie de Compiègne.
- [Saa99] Mark Saaltink. The Z/EVES 2.0 user's guide. Technical report, ORA Canada, october 1999.
- [SDK⁺95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 21 :314–335, 1995.
- [Som92] Sommerville. *Le génie logiciel*. IEEE standards collection, 1992.
- [SX03] Petri Selonen and Jianli Xu. Validating UML models against architectural profiles. *SIGSOFT Software Engineering Notes*, 28(5) :58–67, 2003.
- [TMA⁺96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., E. James, and Jason E. Robbins. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22 :390–406, 1996.
- [Uma97] Amjad Umar. *Object-oriented client/server Internet environments*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1997.
- [wCGS⁺02] Shang wen Cheng, David Garlan, Bradley Schmerl, Joao Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. In *Proc of the International Conf. on Architecture of Computing Systems : Trends in Network and Pervasive Computing*, pages 67–82, 2002.
- [WD96] Jim Woodcock and Jim Davies. *Using Z : Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [XSL] W3c.extensible stylesheet language transformatio (xslt) version 1.0. november 1999. <http://www.w3.org/TR/xslt>.
- [YYX05] Qun Yang, Xian-Chun Yang, and Man-Wu Xu. A framework for dynamic software architecture-based self-healing. *SIGSOFT Software Engineering Notes*, 30(4) :1–4, 2005.
- [ZS06] Bogumil Zieba and Marten van Sinderen. Preservation of correctness during system reconfiguration in data distribution service for real-time systems (dds). In *ICDCSW '06 : Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 30, Washington, DC, USA, 2006. IEEE Computer Society.

Annexe A

Preuve du théorème de consistance

Cette annexe présente la preuve du théorème de consistance de l'étude de cas "Follow me".

theorem *consistencyStructural*

$\exists \text{Structural_Style}' \cdot \text{NewArchitecture2}$

$\exists \text{Structural_Style}' \cdot \text{NewArchitecture2}$

use *axiom\$12*

$m2 . \text{load} = 8 \Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use *axiom\$8*

$\text{maxLatency} = 5 \wedge m2 . \text{load} = 8 \Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use *axiom\$11*

$m1 . \text{load} = 2 \wedge \text{maxLatency} = 5 \wedge m2 . \text{load} = 8$
 $\Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use *axiom\$7*

$\text{maxLoad} = 5 \wedge m1 . \text{load} = 2 \wedge \text{maxLatency} = 5 \wedge m2 . \text{load} = 8$
 $\Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use *axiom\$4*

$m1 \neq m2$
 $\wedge m1 \neq m1'$
 $\wedge m2 \neq m1'$
 $\wedge \text{maxLoad} = 5$
 $\wedge m1 . \text{load} = 2$
 $\wedge \text{maxLatency} = 5$
 $\wedge m2 . \text{load} = 8$
 $\Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use *axiom\$19*

$\text{latency}(m2, m1) = 2$
 $\wedge m1 \neq m2$
 $\wedge m1 \neq m1'$
 $\wedge m2 \neq m1'$
 $\wedge \text{maxLoad} = 5$
 $\wedge m1 . \text{load} = 2$
 $\wedge \text{maxLatency} = 5$
 $\wedge m2 . \text{load} = 8$
 $\Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2})$

use axiom\$14

$$\begin{aligned} & \text{replicate } m2 = m1' \\ & \wedge \text{latency } (m2, m1) = 2 \\ & \wedge m1 \neq m2 \\ & \wedge m1 \neq m1' \\ & \wedge m2 \neq m1' \\ & \wedge \text{maxLoad} = 5 \\ & \wedge m1 . \text{load} = 2 \\ & \wedge \text{maxLatency} = 5 \\ & \wedge m2 . \text{load} = 8 \\ & \Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2}) \end{aligned}$$

use axiom\$16

$$\begin{aligned} & \text{latency } (m1, m2) = 7 \\ & \wedge \text{replicate } m2 = m1' \\ & \wedge \text{latency } (m2, m1) = 2 \\ & \wedge m1 \neq m2 \\ & \wedge m1 \neq m1' \\ & \wedge m2 \neq m1' \\ & \wedge \text{maxLoad} = 5 \\ & \wedge m1 . \text{load} = 2 \\ & \wedge \text{maxLatency} = 5 \\ & \wedge m2 . \text{load} = 8 \\ & \Rightarrow (\exists \text{Structural_Style}' \cdot \text{NewArchitecture2}) \end{aligned}$$

prove by reduce

$$\begin{aligned} & \text{latency } (m1, m2) = 7 \\ & \wedge \text{replicate } m2 = m1' \\ & \wedge \text{latency } (m2, m1) = 2 \\ & \wedge \neg m1 = m1' \\ & \wedge \neg m2 = m1' \\ & \wedge \text{maxLoad} = 5 \\ & \wedge m1 . \text{load} = 2 \\ & \wedge \text{maxLatency} = 5 \\ & \wedge m2 . \text{load} = 8 \\ & \Rightarrow \{m1\} \cup (\{m1\} \cup (\{m1'\} \cup \{m2\})) = \{m1\} \cup (\{m1'\} \cup \{m2\}) \\ & \wedge (\neg x_1 = y_1 \\ & \wedge (x_1 = m1 \vee x_1 = m1' \vee x_1 = m2) \\ & \wedge (y_1 = m1 \vee y_1 = m1' \vee y_1 = m2) \\ & \Rightarrow (\exists T: \text{seq Manager} \\ & \cdot (\forall i: \mathbb{N} \mid 1 \leq i \wedge i \leq -1 + \# T \wedge \# T \geq 0 \end{aligned}$$

$\bullet x_{-1} \in \text{ran } T$
 $\wedge y_{-1} \in \text{ran } T$
 $\wedge \text{ran } T \in \mathbb{P}(\{m1\} \cup (\{m1'\} \cup \{m2\}))$
 $\wedge (Ti = m1 \wedge T(1+i) = m1'$
 $\vee Ti = m1' \wedge T(1+i) = m1$
 $\vee Ti = m1 \wedge T(1+i) = m2$
 $\vee Ti = m2 \wedge T(1+i) = m1))$
 $\wedge (\neg x = y \wedge (x = m1 \vee x = m2) \wedge (y = m1 \vee y = m2))$
 $\Rightarrow (\exists T_{-0}: \text{seq Manager}$
 $\bullet (\forall i_{-0}: \mathbb{N} / 1 \leq i_{-0} \wedge i_{-0} \leq -1 + \#T_{-0} \wedge \#T_{-0} \geq 0$
 $\bullet x \in \text{ran } T_{-0}$
 $\wedge y \in \text{ran } T_{-0}$
 $\wedge \text{ran } T_{-0} \in \mathbb{P}(\{m1\} \cup \{m2\})$
 $\wedge (T_{-0} i_{-0} = m1 \wedge T_{-0}(1+i_{-0}) = m2$
 $\vee T_{-0} i_{-0} = m2 \wedge T_{-0}(1+i_{-0}) = m1))$
 $\wedge (\neg x_{-0} = y_{-0}$
 $\wedge (x_{-0} = m1 \vee x_{-0} = m1' \vee x_{-0} = m2)$
 $\wedge (y_{-0} = m1 \vee y_{-0} = m1' \vee y_{-0} = m2))$
 $\Rightarrow (\exists T_{-1}: \text{seq Manager}$
 $\bullet (\forall i_{-1}: \mathbb{N} / 1 \leq i_{-1} \wedge i_{-1} \leq -1 + \#T_{-1} \wedge \#T_{-1} \geq 0$
 $\bullet x_{-0} \in \text{ran } T_{-1}$
 $\wedge y_{-0} \in \text{ran } T_{-1}$
 $\wedge \text{ran } T_{-1} \in \mathbb{P}(\{m1\} \cup (\{m1'\} \cup \{m2\}))$
 $\wedge (T_{-1} i_{-1} = m1 \wedge T_{-1}(1+i_{-1}) = m1'$
 $\vee T_{-1} i_{-1} = m1' \wedge T_{-1}(1+i_{-1}) = m1$
 $\vee T_{-1} i_{-1} = m1 \wedge T_{-1}(1+i_{-1}) = m2$
 $\vee T_{-1} i_{-1} = m2 \wedge T_{-1}(1+i_{-1}) = m1))$

cases

$\text{latency}(m1, m2) = 7$
 $\wedge \text{replicate } m2 = m1'$
 $\wedge \text{latency}(m2, m1) = 2$
 $\wedge \neg m1 = m1'$
 $\wedge \neg m2 = m1'$
 $\wedge \text{maxLoad} = 5$
 $\wedge m1 . \text{load} = 2$
 $\wedge \text{maxLatency} = 5$
 $\wedge m2 . \text{load} = 8$
 $\Rightarrow \{m1\} \cup (\{m1\} \cup (\{m1'\} \cup \{m2\})) = \{m1\} \cup (\{m1'\} \cup \{m2\})$

apply *extensionality2* to predicate $\{m1\} \cup (\{m1\} \cup (\{m1'\} \cup \{m2\}))$
 $= \{m1\} \cup (\{m1'\} \cup \{m2\})$

$$\begin{aligned}
& \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency } (m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \Rightarrow \{m1\} \cup (\{m1\} \cup (\{m1'\} \cup \{m2\})) \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \wedge \{m1\} \cup (\{m1'\} \cup \{m2\}) \in \mathbb{P} (\{m1\} \cup (\{m1\} \cup (\{m1'\} \cup \{m2\})))
\end{aligned}$$

prove by reduce

true

next

$$\begin{aligned}
& \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency } (m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x_1 = y_1 \\
& \wedge (x_1 = m1 \vee x_1 = m1' \vee x_1 = m2) \\
& \wedge (y_1 = m1 \vee y_1 = m1' \vee y_1 = m2) \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \bullet (\forall i: \mathbb{N} / 1 \leq i \wedge i \leq -1 + \# T \wedge \# T \geq 0 \\
& \bullet x_1 \in \text{ran } T \\
& \wedge y_1 \in \text{ran } T \\
& \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \wedge (T i = m1 \wedge T (1 + i) = m1' \\
& \vee T i = m1' \wedge T (1 + i) = m1 \\
& \vee T i = m1 \wedge T (1 + i) = m2 \\
& \vee T i = m2 \wedge T (1 + i) = m1)))
\end{aligned}$$

instantiate $T == \langle m1', m1, m2 \rangle$

$$\begin{aligned}
& \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1'
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{latency}(m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge (x = m1 \vee x = m1' \vee x = m2) \\
& \wedge (y = m1 \vee y = m1' \vee y = m2) \\
& \wedge \neg (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \text{seq Manager} \\
& \wedge (\forall i: \mathbb{N} \\
& \quad | 1 \leq i \\
& \quad \wedge i \leq -1 + \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \geq 0 \\
& \quad \bullet (x \in \text{ran} (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge y \in \text{ran} (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \text{ran} (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \quad \wedge ((\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1 \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1' \\
& \quad \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1' \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1 \\
& \quad \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1 \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m2 \\
& \quad \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m2 \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1)) \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \quad \bullet (\forall i_0: \mathbb{N} \mid 1 \leq i_0 \wedge i_0 \leq -1 + \# T \wedge \# T \geq 0 \\
& \quad \bullet x \in \text{ran } T \\
& \quad \wedge y \in \text{ran } T \\
& \quad \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \quad \wedge (T i_0 = m1 \wedge T (1 + i_0) = m1' \\
& \quad \vee T i_0 = m1' \wedge T (1 + i_0) = m1 \\
& \quad \vee T i_0 = m1 \wedge T (1 + i_0) = m2 \\
& \quad \vee T i_0 = m2 \wedge T (1 + i_0) = m1))
\end{aligned}$$

prove by reduce

true

next

$$\begin{aligned}
& \text{latency}(m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency}(m2, m1) = 2
\end{aligned}$$

$$\begin{aligned}
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge (x = m1 \vee x = m2) \\
& \wedge (y = m1 \vee y = m2) \\
& \Rightarrow (\exists T_0: \text{seq Manager} \\
& \quad \bullet (\forall i_0: \mathbb{N} / 1 \leq i_0 \wedge i_0 \leq -1 + \# T_0 \wedge \# T_0 \geq 0 \\
& \quad \bullet x \in \text{ran } T_0 \\
& \quad \wedge y \in \text{ran } T_0 \\
& \quad \wedge \text{ran } T_0 \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \quad \wedge (T_0 i_0 = m1 \wedge T_0 (1 + i_0) = m2 \\
& \quad \vee T_0 i_0 = m2 \wedge T_0 (1 + i_0) = m1)))
\end{aligned}$$

instantiate $T_0 == \langle m1', m1, m2 \rangle$

$$\begin{aligned}
& \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency } (m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge (x = m1 \vee x = m2) \\
& \wedge (y = m1 \vee y = m2) \\
& \wedge \neg (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \text{seq Manager} \\
& \wedge (\forall i: \mathbb{N} \\
& \quad / 1 \leq i \\
& \quad \wedge i \leq -1 + \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \geq 0 \\
& \quad \bullet (x \in \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge y \in \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \quad \wedge ((\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1
\end{aligned}$$

$$\begin{aligned}
& \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m2 \\
& \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m2 \\
& \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1 \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \bullet (\forall i_0: \mathbb{N} / 1 \leq i_0 \wedge i_0 \leq -1 + \# T \wedge \# T \geq 0 \\
& \bullet x \in \text{ran } T \\
& \wedge y \in \text{ran } T \\
& \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \wedge (T i_0 = m1 \wedge T(1 + i_0) = m2 \\
& \vee T i_0 = m2 \wedge T(1 + i_0) = m1))
\end{aligned}$$

prove by reduce

$$\begin{aligned}
& i \in \mathbb{Z} \\
& \wedge \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency } (m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge i \geq 0 \\
& \wedge 1 \leq i \\
& \wedge i \leq 2 \\
& \wedge (x = m1 \vee x = m2) \\
& \wedge (y = m1 \vee y = m2) \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \bullet (\forall i_0: \mathbb{N} / 1 \leq i_0 \wedge i_0 \leq -1 + \# T \wedge \# T \geq 0 \\
& \bullet x \in \text{ran } T \\
& \wedge y \in \text{ran } T \\
& \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \wedge (T i_0 = m1 \wedge T(1 + i_0) = m2 \\
& \vee T i_0 = m2 \wedge T(1 + i_0) = m1))
\end{aligned}$$

instantiate $T == \langle m2, m1 \rangle$

$$\begin{aligned}
& i \in \mathbb{Z} \\
& \wedge \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1'
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{latency}(m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge i \geq 0 \\
& \wedge 1 \leq i \\
& \wedge i \leq 2 \\
& \wedge (x = m1 \vee x = m2) \\
& \wedge (y = m1 \vee y = m2) \\
& \wedge \neg (\langle m2 \rangle \hat{\ } \langle m1 \rangle \in \text{seq Manager}) \\
& \wedge (\forall i_0: \mathbb{N} \\
& \quad / 1 \leq i_0 \wedge i_0 \leq -1 + \# (\langle m2 \rangle \hat{\ } \langle m1 \rangle) \wedge \# (\langle m2 \rangle \hat{\ } \langle m1 \rangle) \geq 0 \\
& \quad \bullet (x \in \text{ran} (\langle m2 \rangle \hat{\ } \langle m1 \rangle)) \\
& \quad \wedge y \in \text{ran} (\langle m2 \rangle \hat{\ } \langle m1 \rangle) \\
& \quad \wedge \text{ran} (\langle m2 \rangle \hat{\ } \langle m1 \rangle) \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \quad \wedge ((\langle m2 \rangle \hat{\ } \langle m1 \rangle) i_0 = m1 \wedge (\langle m2 \rangle \hat{\ } \langle m1 \rangle) (1 + i_0) = m2 \\
& \quad \vee (\langle m2 \rangle \hat{\ } \langle m1 \rangle) i_0 = m2 \wedge (\langle m2 \rangle \hat{\ } \langle m1 \rangle) (1 + i_0) = m1))) \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \quad \bullet (\forall i_1: \mathbb{N} / 1 \leq i_1 \wedge i_1 \leq -1 + \# T \wedge \# T \geq 0 \\
& \quad \bullet x \in \text{ran } T \\
& \quad \wedge y \in \text{ran } T \\
& \quad \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup \{m2\}) \\
& \quad \wedge (T i_1 = m1 \wedge T (1 + i_1) = m2 \\
& \quad \vee T i_1 = m2 \wedge T (1 + i_1) = m1)))
\end{aligned}$$

prove by reduce

true

next

$$\begin{aligned}
& \text{latency}(m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency}(m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8
\end{aligned}$$

$$\begin{aligned}
& \wedge \neg x_0 = y_0 \\
& \wedge (x_0 = m1 \vee x_0 = m1' \vee x_0 = m2) \\
& \wedge (y_0 = m1 \vee y_0 = m1' \vee y_0 = m2) \\
& \Rightarrow (\exists T_1: \text{seq Manager} \\
& \quad \bullet (\forall i_1: \mathbb{N} / 1 \leq i_1 \wedge i_1 \leq -1 + \# T_1 \wedge \# T_1 \geq 0 \\
& \quad \bullet x_0 \in \text{ran } T_1 \\
& \quad \wedge y_0 \in \text{ran } T_1 \\
& \quad \wedge \text{ran } T_1 \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \quad \wedge (T_1 i_1 = m1 \wedge T_1 (1 + i_1) = m1' \\
& \quad \vee T_1 i_1 = m1' \wedge T_1 (1 + i_1) = m1 \\
& \quad \vee T_1 i_1 = m1 \wedge T_1 (1 + i_1) = m2 \\
& \quad \vee T_1 i_1 = m2 \wedge T_1 (1 + i_1) = m1))
\end{aligned}$$

instantiate $T_1 == \langle m1', m1, m2 \rangle$

$$\begin{aligned}
& \text{latency } (m1, m2) = 7 \\
& \wedge \text{replicate } m2 = m1' \\
& \wedge \text{latency } (m2, m1) = 2 \\
& \wedge \neg m1 = m1' \\
& \wedge \neg m2 = m1' \\
& \wedge \text{maxLoad} = 5 \\
& \wedge m1 . \text{load} = 2 \\
& \wedge \text{maxLatency} = 5 \\
& \wedge m2 . \text{load} = 8 \\
& \wedge \neg x = y \\
& \wedge (x = m1 \vee x = m1' \vee x = m2) \\
& \wedge (y = m1 \vee y = m1' \vee y = m2) \\
& \wedge \neg (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \text{seq Manager} \\
& \wedge (\forall i: \mathbb{N} \\
& \quad / 1 \leq i \\
& \quad \wedge i \leq -1 + \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \# (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \geq 0 \\
& \quad \bullet (x \in \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge y \in \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \\
& \quad \wedge \text{ran } (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \quad \wedge ((\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1 \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1' \\
& \quad \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1' \\
& \quad \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1 \\
& \quad \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m1
\end{aligned}$$

$$\begin{aligned}
& \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m2 \\
& \vee (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) i = m2 \\
& \wedge (\langle m1 \rangle \hat{\ } (\langle m1 \rangle \hat{\ } \langle m2 \rangle)) (1 + i) = m1))) \\
& \Rightarrow (\exists T: \text{seq Manager} \\
& \bullet (\forall i_0: \mathbb{N} / 1 \leq i_0 \wedge i_0 \leq -1 + \# T \wedge \# T \geq 0 \\
& \bullet x \in \text{ran } T \\
& \bullet y \in \text{ran } T \\
& \wedge \text{ran } T \in \mathbb{P} (\{m1\} \cup (\{m1'\} \cup \{m2\})) \\
& \wedge (T i_0 = m1 \wedge T(1 + i_0) = m1' \\
& \vee T i_0 = m1' \wedge T(1 + i_0) = m1 \\
& \vee T i_0 = m1 \wedge T(1 + i_0) = m2 \\
& \vee T i_0 = m2 \wedge T(1 + i_0) = m1)))
\end{aligned}$$

prove by reduce

true

next

true