Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR UNIVERSITY – ANNABA

UNIVERSITE BADJI MOKHTAR – ANNABA

جامعة باجي مختار – عنابة

Année : 2018

Faculté des sciences de l'ingéniorat
Département d'informatique

T H È S E

Présentée en vue de l'obtention du diplôme de

Doctorat en Sciences

Reconfiguration dynamique des architectures logicielles

Option
Génie logiciel

Préparée par

Rida MEZGHACHE

Jury :

Président	Djamel MESLATI	Professeur	Université Badji Mokhtar - Annaba
Directrice	Fadila ATIL	Professeur	Université Badji Mokhtar - Annaba
Examinateurs	Abdelkrim AMIRAT	Professeur	Université Mohamed Chérif Messaidia - Souk Ahras
	Allaoua CHAOUI	Professeur	Université Abdelhamid Mehri - Constantine

ملخص

استجابة للحاجة المتزايدة إلى قابلية النظم للتوسع وبالتوازي للحاجة إلى الموثوقية ، فإن العمل البحثي المقدم في هذه الأطروحة هو جزء من مجال تطور هندسة البرمجيات. وهدفها الرئيسي هو إدخال نهج لضمان موثوقية هندسة البرمجيات القائمة على المكونات في جميع مراحل تطورها. إن لغة الوصف Acme هي في صميم عملنا، وهي لغة إعلانية تقوم على المنطق الأصلي من الدرجة الأولى، والذي يدعم نموذج المكونات والموصلات بأنواعها، وكذلك الثوابت، و الأساليب المعمارية. كما أنها تدعم القيود.

يمكن تلخيص نهجنا على مرحلتين. فالخطوة الأولى هي تحديد اتساق النظم وهياكلها (أو تكوينها). أما الخطوة الثانية فهي تحديد اتساق تطورها (أو إعادة تشكيلها)، وذلك لأجل تنفيذ آليات موثوقة لإعادة التشكيل. لقد عبرنا أولا عن هذه القيود باستعمال المنطق الأصلي الأول، ثم، في خطوة ثانية، ترجمناها إلى الشكلية Acme/Armani، وذلك لضمان التوافق النحوي والتركيبي، وجعل التكوين موثوق به ومتسق مثل التصميم. أما الهدف النهائي فهو توفير سلسلة كاملة انطلاقا من التكوين الثابت إلى غاية إعادة تشكيل الديناميكية، وذلك من خلال تصميم إطار برامج مخصص لمنصة التنفيذ Acme، مع مجموعة غنية ومرنة من واجهات التحكم.

كلمات دالة : البنى القائمة على المكونات، المكون البرمجي، إعادة التشكيل الديناميكي، الموثوقية، بنيان البرمجيات، قيود التكامل، لغة وصف البنيان، Acme،Armani.

Résumé

En réponse au besoin croissant d'évolutivité des systèmes et en parallèle au besoin de fiabilité, le travail de recherche effectué dans cette thèse s'inscrit dans le cadre du domaine de l'évolution d'architecture logicielle. Il a comme objectif principal l'introduction d'une approche permettant d'assurer la fiabilité d'une architecture à base de composants tout au long de son évolution. Le langage de description d'architectures Acme est au cœur de notre travail. C'est un langage déclaratif basé sur la logique de prédicat de premier ordre, qui prend en charge le paradigme des composants et des connecteurs avec les types, ainsi que les invariants et les styles architecturaux, et il supporte également les contraintes.

Notre démarche peut se résumer en deux étapes. La première étape consiste en la définition de la cohérence des systèmes et de leur architecture (ou configuration). La deuxième étape est la définition de la cohérence de leurs évolutions (ou reconfigurations), pour la mise en œuvre des mécanismes de reconfigurations fiables. Nous avons d'abord exprimé ces contraintes dans la logique de prédicat de premier ordre, puis, dans un second temps, nous les avons traduites dans le formalisme Acme/Armani, pour assurer la conformité syntaxique et de composition, et afin de rendre la configuration aussi fiable et cohérente que la conception. L'objectif final est de fournir un continuum complet de la configuration statique à la reconfiguration dynamique, grâce à la conception d'un canevas logiciel dédié à une plateforme d'exécution propre à Acme, muni d'un ensemble d'objets de contrôle suffisamment riches et flexibles.

Mots-clés : Acme, Architecture logicielle, Architectures basées composants, Armani, Composant logiciel, Contraintes d'intégrité, Fiabilité, Langage de description d'architecture, Reconfiguration dynamique.

Abstract

In response to the growing need for systems scalability and the need for reliability in parallel, the research work done in this thesis is a part of the field of software architecture evolution. Its main objective is to introduce an approach to ensure the reliability of a component-based architecture throughout its evolution. The Acme Architectural Description Language is at the heart of our work, it is a declarative language based on first-order predicate logic, which supports the paradigm of components and connectors with types, as well as invariants, architectural styles, and also supports constraints.

Our approach can be summarized in two stages. The first step was therefore to define the coherence of the systems and their architecture (or configuration). The second step is the definition of the coherence of their evolutions (or reconfigurations), for the implementation of reliable reconfiguration mechanisms. We first expressed these constraints in the first-order predicate logic, then, in a second time, we translated them into the Acme/Armani formalism, to ensure syntactic and compositional conformance, in order to make the configuration as reliable and consistent as design. The end goal is to provide a complete continuum from static configuration to dynamic reconfiguration, through the design of a software framework dedicated to Acme execution platform, with a set of sufficiently rich and flexible control objects.

Keywords : Acme, Architecture description language, Armani, Component-based architectures, Dynamic reconfiguration, Integrity constraints, Reliability, Software architecture, Software component.

Remerciements

Je tiens à remercier tous les membres de mon jury. En premier lieu, je tiens à remercier ma directrice de thèse, Pr. Fadila ATIL, pour la confiance qu'elle m'a accordé en acceptant d'encadrer ce travail de doctoral, pour ses multiples conseils et pour toutes les heures qu'elle a consacré à diriger cette thèse de recherche.

Je remercie Pr. Djamel Meslati de m'avoir accordé l'honneur d'être le président de mon jury. Merci à Pr. Abdelkrim Amirat et à Pr. Allaoua Chaoui d'avoir accepté d'examiner et d'évaluer mon travail.

Je tiens à adresser ma gratitude à ma famille et mes proches, et en particulier mes parents qui m'ont soutenue avec patience durant mon parcours académique et durant la réalisation de ce travail.

Table des matières

Table des figures

Liste des tableaux

Liste des listings

Chapitre 1

Introduction

1.1 Contexte et problématique

Le travail que nous présentons dans cette thèse a été élaboré au niveau de l'équipe SAFA du laboratoire d'ingénierie des systèmes complexes (LISCO) de l'université de Badji-Mokhtar d'Annaba. L'équipe SAFA (Software Architecture and Formal Approaches) s'active autour des thèmes liés à l'ingénierie du logiciel. Son activité de recherche se concentre actuellement sur les nouveaux paradigmes de génie logiciel, et plus particulièrement les paradigmes d'architectures logicielles et les approches formelles.

L'architecture logicielle comme nouvelle discipline du génie logicielle, décrit de manière abstraite les systèmes à l'aide de composants logiciels, les relations entre ces composants, leurs propriétés fonctionnelles et non fonctionnelles ainsi que les principes qui régissent leur conception et leur évolution [IEE 00]. Afin de maitriser le processus de développement de logiciels basés architectures, un langage de spécification est nécessaire. Les langages de description d'architectures (ADLs) répondent à ce besoin, et fournissent une représentation abstraite des éléments de base constituant un système complexe, en réduisant le coût de détection et de correction d'erreurs [Med 02]. Ces langages sont formels et peuvent décrire l'architecture d'un système assez complexe [Cle 96]. Un ADL garanti un niveau élevé d'abstraction, ignorant ainsi les détails d'implémentation [Med 00]. Grâce à cette description abstraite, les ADLs aident les développeurs, en facilitant le contrôle du processus de développement à un stade avancé [Hus 13]. Les ADLs sont donc cruciaux dans le développement logiciel particulièrement ceux basés architecture. Chaque ADL a ses caractéristiques, certains ADLs ont été proposé pour la modélisation d'architectures dans un domaine bien précis, d'autres sont plus spécialisés. Voici une liste non exhaustive d'ADLs présents dans la littérature : LIEANNA [Tra 93], Artek [Ter 95], MetaH [Bin 96], Aesop [Sha 96], SADL [Mor 97] ; [Mor 95), Weaves [Gor 94] ; [Gor91], Wright [All 97], AESOP [Gar 95], Darwin [Mag 95], Acme [Gar 00], Rapide [Luc 95], UNICON [Sha 95] and C2 [Tay 96].

Les applications des ADLs sont très diverses, c'est pourquoi beaucoup d'ADLs sont développés. La liste est si importante qu'il est fastidieux de discuter en détail chaque ADL avec son évaluation. Dans [Hus 13], Sajjad organise ces détailles, en regroupant les secteurs industriel et universitaire.

Dans ce travail, nous nous intéressons à la manière d'assurer la fiabilité d'une architecture à base de composants. Une composition est dites fiable lorsque ses instances sont correctes dans le sens où elles vérifient les besoins du concepteur. De nos jours les produits logiciels sont complexes, rendant les méthodes de développement obsolètes, et créant ainsi le besoin de nouvelles tendances dans le développement logiciel, en introduisant de nouvelles pratiques de génie logiciel. Les architectures logicielles permettent au développeur de se concentrer sur tout le système, tout en faisant abstraction de certains détails. En plus de faciliter la tâche au développeur face à des applications complexes, les architectures logicielles réduisent le coût et facilitent l'évolution [Med 02].

1.2 Présentation des contributions

Bien que certains ADLs supportent les reconfigurations dynamiques à des niveaux variables, le problème de la fiabilité de ces reconfigurations n'est pas réellement abordé comme un objectif à part entière. En effet cet aspect est peu détaillé, et il manque clairement un mécanisme robuste pour l'évaluation des systèmes.

La programmation par composants, associée à des outils et des langages dédiés, comme les langages de description d'architecture, est un bon support pour les reconfigurations dynamiques d'architecture. Grâce à la réflexion, les modèles de composants permettent l'introspection et les modifications dans les systèmes à l'exécution, tout en maintenant une connexion causale entre architecture et système concret. Modifier dynamiquement un système à travers son architecture n'est pas sans poser des problèmes de fiabilité, et qui demeure une préoccupation importante du génie logiciel. En général, la plupart des modèles de composants ne définissent pas les besoins et les règles à respecter pour que les applications basées sur ces modèles puissent évoluer et s'adapter dynamiquement au cours de leur exécution [Ket 04].

La fiabilité des reconfigurations dynamiques est un problème global qui requiert une démarche rigoureuse, dont le point de départ serait de définir le concept de cohérence dans une architecture, essentiellement d'un point de vue structurel.

La cohérence d'une architecture est ensuite conditionnée par la satisfaction d'un ensemble de contraintes exprimées dans une certaine logique. Les contraintes doivent être extensibles, afin de permettre d'exprimer les invariants aux niveaux des configurations, et de couvrir le coté sémantique des opérations de reconfiguration. La cohérence doit être maintenue même dans le système reconfiguré. Il est aussi important de vérifier les contraintes avant de valider les modifications des reconfigurations.

Ce travail [Mez 18], permet de formaliser une configuration d'architecture (figure 1.1). Dans cet objectif, nous décrivons les contraintes en logique des prédicats du premier ordre, que nous traduisons ensuite en utilisant un style architectural Acme/Armani détaillé dans [Mon 01] afin de valider la configuration statique. En ce qui concerne l'aspect dynamique, il est validé par un autre processus, où les contraintes sont exprimées en Plastik et le système validé dans le modèle OpenCom.

Etant donné l'influence de l'architecture d'un système sur l'ensemble des attributs de qualité d'un système logiciel, il devient pertinent de développer un support pour la qualité logicielle dés le niveau architectural. En effet, ce travail représente le niveau de base qui inclut : le style architectural, les composants, les connecteurs et les contraintes de configuration. En préservant la conformité de la couche architecturale à la couche application, et en assurant la consistance, une spécification formelle d'un modèle Acme peut servir à plusieurs fins :

— Permettre une vérification formelle d'une conception Acme ;
— Fournir la base d'un langage de description d'architecture formelle pour Acme ;
— Fournir une spécification plus abstraite et indépendante du langage de programmation pour le modèle Acme ;
— Permettre une comparaison rigoureuse avec les autres modèles de composants ;
— Permettre une spécification formelle et une vérification des outils Acme ;

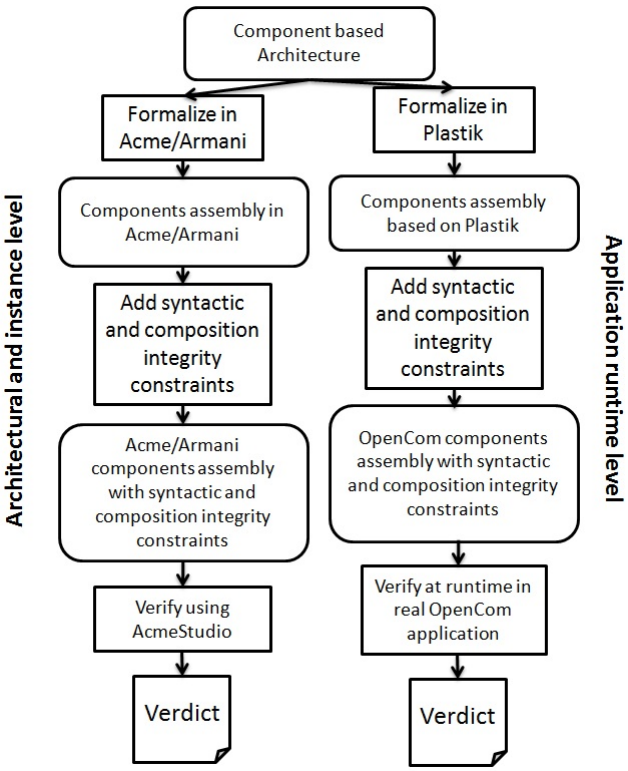

FIGURE 1.1 – Procédure de vérification des contraintes

1.3 Organisation du document

Ce document de thèse se décompose en six chapitres au total, dont un chapitre d'introduction, deux chapitres d'états de l'art, deux chapitres de présentation et d'évaluation des contributions, et un dernier chapitre de conclusion.

Le premier chapitre permet d'introduire, en premier lieu, le contexte et les motivations du travail, puis, la problématique traitée, et enfin nos contributions en réponse à la problématique. L'état de l'art est partagé en deux chapitres suivant les deux concepts clé du sujet : les reconfigurations dynamiques dans les modèles de composants (chapitre 2), et le support de la fiabilité dans les systèmes en se basant sur ces mêmes modèles (chapitre 3). Dans chacun des chapitres, nous choisissons des travaux connexes représentatifs du problème posé dans le sujet après avoir expliqué les principaux concepts et technologies référencés et utilisés.

Nos contributions concernent la modélisation des contraintes des configurations, présentées dans le chapitre 4 et des reconfigurations dans le modèle Acme, présentées dans le chapitre 5. Enfin, le chapitre 6 vient conclure cette thèse par une synthèse des contributions et perspectives de recherches associées.

Première partie

Etat de l'art

Chapitre 2

Concepts et technologies associés au sujet

Dans ce chapitre, nous présentons les principaux concepts et technologies liés à nos travaux. Dans la section 2.1, nous introduisons la programmation par composants qui est au cœur des mécanismes de reconfigurations dynamiques. La section 2.3 présente les langages de description d'architectures qui permettent d'élaborer et d'explorer les modèles à un stage avancé de conception, et mettent l'accent sur la structure du système logiciel. Dans la section 2.3, nous étalons le principe des reconfigurations qui reposent en effet par hypothèse sur les architectures à composants des systèmes considérés et sur leur dynamicité. Avant de conclure ce premier chapitre, nous présentons les propriétés de base des systèmes réflexifs.

2.1 Programmation par composants

La programmation par composant est un paradigme de développement pour la conception et l'implémentation de systèmes logiciels et qui ressort de l'incapacité de l'approche orienté objet à soutenir une réutilisation efficace [Jai 16]. Les composants sont plus abstraits que les classes d'objets et peuvent être considérés comme des fournisseurs de services autonomes. La composition doit être conforme aux règles de construction des architectures, et se fait grâce aux langages de description des architectures. En ce qui concerne les contraintes, elles sont gérées par des styles architecturaux.

Avec la croissance de la taille et de la complexité du logiciel, l'approche traditionnelle qui correspond à la construction à partir de zéro, devient de plus en plus inefficace en termes de productivité et de coût.

Afin de répondre aux exigences de qualité, et des logiciels modernes à grande échelle, différents paradigmes de développement ont été introduits dans le but de faciliter la création de systèmes évolutifs, flexibles, fiables et réutilisables.

La programmation par objets a émergé depuis une vingtaine d'année [Coi 06] et s'est ainsi imposée comme évolution majeure par rapport à la programmation procédurale. Le paradigme objet apporte en effet de bonnes propriétés, telles qu'un niveau d'abstraction plus élevé (concepts de classes et d'instances), etc. Cependant, l'objet n'est pas suffisamment abstrait et reste à un niveau de granularité trop fin pour maitriser aisément la structuration des systèmes complexes, structuration indispensable à leur maintenance et à leur évolution. Les lacunes de l'approche objet ont mené à la création d'un nouveau paradigme de programmation : *le composant.*

Le paradigme composant est une approche appropriée et méthodique, qui implique la construction d'un système en utilisant des briques de bases qui ont été développés à différents moments, par différentes personnes, et probablement en ayant à l'esprit différents concepts et utilisations [Sha 12].

Afin de pouvoir découvrir les différents composants, leurs manières de communiquer et de construire dynamiquement l'assemblage, un environnement de composants fournit toujours un mécanisme d'introspection et un mécanisme d'invocation dynamique (le référentiel des interfaces et le DII (Dynamic Invocation Interface) de CORBA [Gei 00], l'API java.lang.reflect du langage Java [Bel 99][Bel 98], les interfaces IUnknown et IDispatch dans COM). L'introspection permet de découvrir à l'exécution les interfaces d'un composant.

En Java, l'introspection permet aussi de découvrir la structure interne des composants (les attributs). Depuis JDK version 1.1, la sérialisation des objets Java met en œuvre ce mécanisme d'introspection de manière automatique afin de permettre de rendre un objet ou un graphe d'objets de la JVM persistant pour stockage ou échange. L'invocation dynamique permet de construire dynamiquement des requêtes sur les composants. L'association de ces deux mécanismes permet de construire des outils d'interaction sur les composants pour configurer/reconfigurer un assemblage de composants.

2.1.1 Modèles de composants

Le concept de composant logiciel est relativement ancien [Mcl 68]. Il est décrit comme une brique logicielle composable et facilement réutilisable, d'où l'appellation de composant sur l'étagère (Composant On The Shell ou COTS). Cependant, il n'existe pas de définition universelle d'une description d'architecture à composant.

Selon Szyperski [Szy 98], un composant « est une unité de composition dont les interfaces et les dépendances contextuelles sont spécifiées sous forme de contrats explicites ». Il peut être « déployé indépendamment et est composable par des tiers".

Selon Heieneman et Council [Cou 01]. Un composant logiciel est un élément conforme à un modèle de composition, pouvant être indépendamment déployé et composé.

Le modèle de composition définit les standards de spécification et d'interaction. L'im-

plémentation du modèle représente un ensemble d'outils logiciels supportant l'exécution des composants conformes au modèle.

En comparaison aux objets, et de part la nature des encapsulations fortes dans le composant (boite noire), les composants offrent une plus grande modularité et facilité de réutilisation, ainsi qu'une meilleure indépendance entre les briques logicielles, représentant clairement les dépendances requises et fournies.

La séparation des préoccupations (separation of concerns) est une approche de conception qui consiste à distinguer les préoccupations fonctionnelles et extra-fonctionnelles dans un système afin d'isoler des aspects indépendants ou faiblement couplés. Elle représente un point fort de la programmation par composants [Dij 82].

Le cycle de vie du logiciel est décomposé en plusieurs phases : De la conception de l'architecture, au développement du code métier des composants, suivi de leur déploiement et finalement de leur exécution.

L'architecture d'une application, quant à elle, est composée de la définition des différentes instances des différents composants utilisés et de la spécification de leurs interconnexions, c'est-à-dire les dépendances entre les services requis et fournis, ainsi que les schémas de communication utilisés lors de l'exécution [Riv 00].

Cette démarche repose sur des techniques d'injection de dépendances dans le code d'implémentation. Elle offre certains avantages, entre autre, le développeur peut se concentrer sur un problème à la fois. Les besoins et les contraintes de chaque aspect évoluent de manière autonome, grâce à la suppression des interactions entre aspects orthogonaux.

En fonction du milieu, la définition d'un composant peut varier. En effet, dans le milieu académique, un composant est vu comme étant une entité bien définie, souvent petite avec des propriétés fonctionnelles et non-fonctionnelles. Dans l'industrie par contre, un composant est considéré comme une partie réutilisable du système, mais que celle-là n'est pas nécessairement bien définie dans ses interfaces et sa conformité à un modèle de composant.

Il existe de nombreux modèles de composants selon les domaines d'application. Des modèles industriels (EJB [Ejb], CCM [Wan 01], COM [Com] et .NET [.Ne]) mettent en avant la fourniture de services extra-fonctionnels dans des « conteneurs », tels que la sécurité ou le support des transactions applicatives, tout en restant assez limités en termes de concepts d'architectures. Ce sont en effet des modèles proches de l'implémentation.

Certains modèles sont uniquement utilisés pour la conception d'architectures sans support d'exécution (tels que les composants UML). Enfin, d'autres modèles, tels SCA [Sca] ou encore OSGi [Osg] servent à implémenter des architectures orientées services (Service Oriented Architectures ou SOA).

Les modèles plus académiques ou de recherche (Fractal [Fra] et SOFA [Sof]) s'intéressent à des concepts plus avancés, notamment concernant l'analyse des architectures en termes structurels ou de comportement. Ces modèles sont généralement associés à un lan-

gage dédié pour la description des architectures des systèmes : les langages de description d'architectures (Architecture Description Languages ou ADLs).

2.1.2 Eléments d'architecture

S'il y a bien un point commun entre les différents modèles de composants académiques, c'est l'aspect structurel qui comporte généralement des concepts assez proches : *les composants*, *les connecteurs* et *les configurations* (figure 2.1). Le langage Acme [Gar 00] a été développé dans l'optique de fournir un format d'échange standard commun entre les ADLs existants. Les éléments d'architectures sont définis par ce langage d'une manière générique qui correspond à la définition utilisée dans la plupart des modèles de composant.

Un composant est une unité de calcul ou de stockage à laquelle est associée une unité d'implémentation. Il peut être simple ou composite. La spécification d'un composant repose sur trois éléments.

Le premier élément est la définition abstraite du composant qui représente son *type*, ce dernier est caractérisé par des *interfaces* et des *propriétés*. Une interface définit l'interaction du composant avec son environnement via des points d'accès. Les interfaces fournies définissent les services offerts par le composant, tandis que les interfaces requises spécifient les services requis. Les propriétés peuvent être fonctionnelles ou non-fonctionnelles. Les propriétés fonctionnelles sont celles qui décrivent la structure et le comportement du composant. En revanche, les propriétés non-fonctionnelles concernent les services utilisés par le composant qui ne font pas partie des services applicatifs (sécurité, performance, etc.).

Le second élément correspondant à son implémentation et permet la description du fonctionnement interne du composant, grâce à la mise en œuvre des aspects fonctionnels et non-fonctionnels de son type.

Quant au troisième élément, il représente l'instance du composant, et qui représente l'entité exécutable d'un composant au sein d'un système.

Les connecteurs représentent des entités architecturales abstraites au sein des modèles d'architectures, qui intègrent l'interaction entre composants. Avec la complexité croissante des applications, les connecteurs deviennent un facteur clé dans l'ensemble du processus de développement. Ils modélisent de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions. Leur cycle de vie commence après la spécification du déploiement des composants.

Selon les modèles, le connecteur n'est pas nécessairement considéré comme une entité de première classe où il peut être un composant particulier dédié à l'implémentation d'un protocole de communication.

La complexité de la spécification d'un connecteur varie. Elle peut décrire des interactions simples de type appel de procédure, mais aussi des interactions complexes, telles

que des protocoles d'accès à des bases de données. Un connecteur comprend également deux parties. Une partie correspondant à son interface, et qui permet la description des rôles dans une interaction donnée. L'autre partie correspond à la description de son implémentation. Il s'agit là de la définition du protocole associé à l'interaction.

Les composants et les connecteurs sont composés d'une manière spécifique dans l'architecture d'un système donné pour atteindre l'objectif de ce système. Cette composition représente la configuration du système, également appelée *topologie*.

Une configuration architecturale, ou une topologie, est un ensemble d'associations spécifiques entre les composants et les connecteurs de l'architecture d'un système logiciel. Elle définit la structure et le comportement d'une application. Une configuration structurelle peut être représentée sous la forme d'un graphe dans lequel les nœuds représentent des composants et des connecteurs, et dont les arcs représentent leurs associations. Une configuration comportementale, quant à elle, modélise le comportement en décrivant l'interaction et l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles.

L'accent est mis sur la propriété de *réutilisabilité* des composants logiciels. Il suffit de connaître le principe de fonctionnement de chaque composant et la manière d'interagir avec l'environnement. Chaque composant peut être considéré comme une boîte noire pour laquelle il suffit de connaître les services rendus et quelques règles d'interconnexion. La programmation par composition permet de construire une application par assemblage de composants.

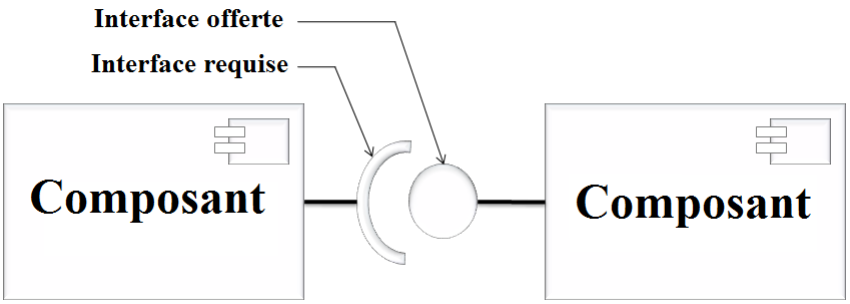

FIGURE 2.1 – Modèle générique de composants

2.1.3 Étude de quelques modèles de composants

2.1.3.1 Le modèle JavaBeans

JavaBeans [Jbe] est un modèle de composants introduit par Sun Microsystems en 1996. L'implémentation du Bean n'a rien de différent par rapport à la programmation Java standard [Jav] et permet de créer des applications en utilisant des outils d'assemblage graphiques

Le modèle JavaBeans utilise l'introspection de Java afin de déterminer les caractéristiques des composants et pouvoir les personnaliser et les combiner dans une architecture.

Construction des composants

La granularité des Beans est variable, d'un simple bouton à des composants plus complexes. L'interface du Bean est constituée des éléments suivants (Figure 2.2) :

— Les événements qu'il peut émettre ou qu'il peut recevoir.

— Les méthodes publiques qu'il implémente, et qui peuvent être appelées par les autres Beans.

— Les propriétés qui permettent de personnaliser le Bean pour l'utiliser dans un contexte particulier.

Le développement d'un Bean ne nécessite aucun concept ou élément syntaxique nouveau, et ne requiert aucune bibliothèque ni extension particulière. Toute classe Java peut alors être considérée comme un Bean, la seule contrainte à respecter est d'avoir un constructeur publique sans arguments. Bien que leur utilisation n'est pas obligatoire, deux packages ont été ajoutés à Java, fournissant des services pour simplifier le développement et l'utilisation des Beans (*java.beans et java.beans.beancontext*).

FIGURE 2.2 – Vision logique d'un JavaBean

Assemblage des composants

Un Bean est assemblé avec un autre Bean si ce dernier est déclaré comme auditeur d'un ou de plusieurs événements dont l'autre Bean est la source (Voir figure 2.3). La communication entre les Beans est assurée par le mécanisme des événements de Java.

Dans la pratique, les deux types de Beans doivent implémenter certaines méthodes. Le Bean auditeur d'événements implémente les méthodes qui servent à notifier ce type d'événements. Le Bean censé notifier l'événement doit implémenter deux méthodes pour ajouter et supprimer les auditeurs potentiels. Quant à la notification de l'événement, elle consiste à créer un objet représentant l'événement, et ensuite à appeler une méthode correspondante de l'auditeur avec l'objet événement comme argument.

En plus de la programmation, la composition d'applications JavaBeans peut aussi se faire grâce à des outils d'assemblage graphiques, afin de manipuler graphiquement les Beans. NetBeans [Net] et BeanBox[Bea] en sont des exemples.

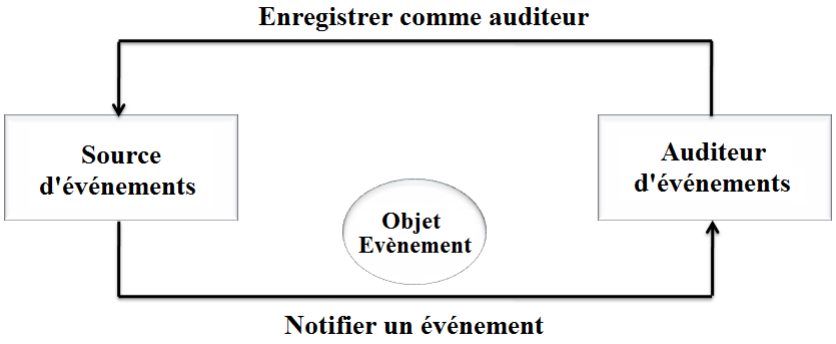

FIGURE 2.3 – Communication événementielle de java

Déploiement, exécution et aspects dynamiques

Les outils de développement manipulent les Beans sous forme de fichiers Jar. Un Jar contient principalement les classes d'un ou de plusieurs Beans et les ressources nécessaires. Le manifeste du Jar contient des méta-données décrivant les propriétés des Beans, comme leurs noms et leurs dépendances.

Etant donné le contexte d'évolution dynamique, et en comparaison avec une application java standard, deux différences importantes sont à souligner :

— Les Beans sont développés en théorie indépendamment les uns des autres, et sont composés par un élément tiers pour former des applications. Ce tiers utilise les méthodes d'assemblage définies par les Beans. Il est donc possible qu'un tiers agisse en cours d'exécution sur un assemblage de Beans et change ainsi son architecture.

— un Bean possède un état explicite, consultable et modifiable de l'extérieur en cours d'exécution. Grâce à un ensemble de propriétés personnalisables via un couple de méthodes publiques.

2.1.3.2 Le modèle Entreprise JavaBeans

Basés sur une architecture client-serveur, les EJB (Entreprise JavaBeans) [Ejb] sont un élément clé de la plate-forme J2EE (Java 2 Entreprise Edition) [J2e].

EJB est un composant logiciel écrit en Java, et qui s'exécute du côté serveur. Comme le montre la figure 2.4, la norme EJB est essentiellement orientée vers le développement d'applications offrant des services complémentaires au dessus d'un système d'information d'entreprise.

Construction des composants

Un EJB se distingue par deux interfaces spécifiques et une classe Java qui représente son implémentation effective. La première interface, de type "*javax.ejb.EJBObject*", définit l'ensemble des fonctionnalités qui peuvent être appelées par les clients de l'EJB. L'autre interface, de type "*javax.ejb.EJBHome*", définit un ensemble de méthodes permettant de gérer le cycle de vie des instances de l'EJB (création, destruction, recherche...).

Un client d'un EJB peut être n'importe quel programme pouvant interagir avec le protocole de communication supporté par le serveur EJB, et peut notamment être un autre EJB.

Il existe trois types de composants : EJB session, entité et orienté messages. [Ket 04]

Les instances d'EJB sont exécutées dans des conteneurs qui gèrent les états des instances. Il est aussi possible qu'une instance gère elle même son état.

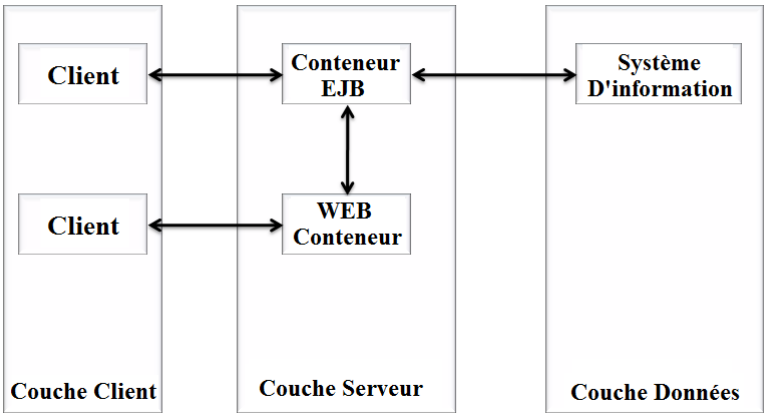

FIGURE 2.4 – Architecture multi-niveaux

Assemblage des composants

Tout comme les Beans, les composants EJB sont fournis par le développeur sous forme de fichiers Jar contenant typiquement l'ensemble des interfaces et des classes d'un ou de plusieurs composants et un descripteur de déploiement. L'assemblage se résume au regroupement de plusieurs composants EJB de base. Le résultat de l'assemblage est un module EJB qui représente une unité de déploiement de plus gros grain.

Le développeur du composant n'a pas à programmer explicitement les actions associées au début ou à la terminaison d'une transaction, ni à insérer dans le code de l'application la vérification du contrôle d'accès. Un module fourni par l'assembleur contient un ou plusieurs composants avec les instructions qui décrivent comment ces composants sont combinés dans une même unité de déploiement, comment ils doivent être déployés et les informations système non fournies par le développeur.

Déploiement, exécution et aspects dynamiques

Un EJB doit être déployé sous forme d'une archive jar contenant un fichier qui est le descripteur de déploiement et toutes les classes qui composent chaque EJB. Une archive ne doit contenir qu'un seul descripteur de déploiement pour tous les EJB de l'archive. Ce fichier au format XML doit obligatoirement être nommé ejb-jar.xml. L'archive doit contenir un répertoire META-INF qui contiendra lui-même le descripteur de déploiement. Le reste de l'archive doit contenir les fichiers .class avec toute l'arborescence des répertoires des packages. Le jar des EJB peut être inclus dans un fichier de type EAR.

Le déploiement d'un composant EJB consiste à injecter le composant effectif, contenu dans l'unité de déploiement, dans un environnement d'exécution spécifique.

Tout comme les Beans, les instances de composants EJB s'exécutent dans des conteneurs. La différence est que ces conteneurs sont eux mêmes logés dans un serveur. Ceci est illustré par la figure 2.5. Un conteneur a aussi la responsabilité de gérer le cycle de vie des instances de composants (création, destruction, activation...), et de leur fournir les services systèmes dont elles ont besoin.

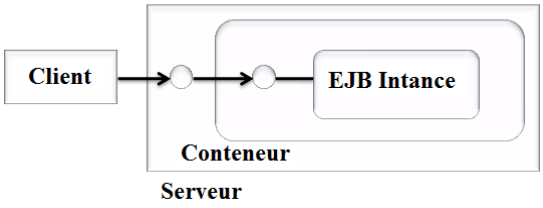

FIGURE 2.5 – Exécution des composants EJB

2.1.3.3 Le modèle COM

COM (Component Object Model) [Com] est un modèle de composants développé par Microsoft en 1995. Son objectif est de permettre le développement d'applications en assemblant des composants pouvant être écrits en différents langages, et communiquant en COM. Ceci signifie que les compilateurs des différents langages de programmation génèrent les composants suivant le même format binaire.

D'autres technologies de plus haut niveau ont été bâties par Microsoft au dessus de COM. Parmi ces technologies, nous pouvons citer COM+ [Kir 97] qui étend les capacités de COM avec des services système comme les transactions et la sécurité pour mieux supporter les systèmes d'information des entreprises.

Construction des composants

COM est indépendant d'un langage de programmation particulier. Le développement des composants peut se faire en n'importe quel langage capable de générer du code binaire en format standard de COM, comme C, C++, Pascal, Ada, Smalltalk, Visual Basic et autres. Concrètement, les clients ne peuvent utiliser une instance d'un composant qu'à travers ses interfaces (les détails d'implémentation ne sont pas visibles). La Figure 2.6 illustre une vue typique d'un composant COM.

Une interface d'un composant possède un identificateur unique (GUID) avec lequel elle est connue dans le système. D'autres interfaces permettent de gérer le cycle de vie, et offrent au clients d'un composant la possibilité de naviguer entre les différentes interfaces que ce composant supporte. Elle permet aussi de déterminer dynamiquement le comportement supporté par le composant.

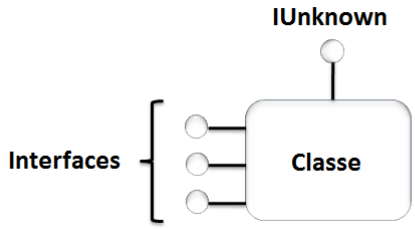

FIGURE 2.6 – Vue d'un composant COM

Assemblage des composants

Un composant COM peut être une entité indépendante ou être intégré dans des applications et déployés par les programmes d'installation de ces applications. Dans ce deuxième cas, l'application peut être vue comme le résultat d'assemblage d'un ensemble de composants (et d'autres objets qui ne sont pas forcément des composants). Comme le montre la figure 2.7, tout composant de l'application doit être utilisé à travers ses interfaces, que ce soit par les objets de l'application ou par les autres composants.

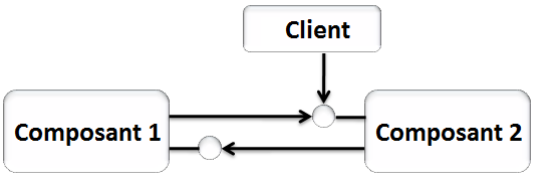

FIGURE 2.7 – Assemblage de composants COM

Déploiement, exécution et aspects dynamiques

Les composants COM se présentent sous deux formes possibles, un fichier exécutable (.EXE) ou une librairie dynamique (.DLL).

Le déploiement correspond à leur installation dans le registre Windows, et leur exécution nécessite ce système. Le système COM gère les instanciations, il répond aux requêtes d'instanciation en s'adressant au composant cible et en lui demandant éventuellement la création d'une instance, et retourne l'instance au demandeur.

L'évolution dynamique des composants COM tire avantage du mécanisme de liaisons dynamiques supportées par les DLLs. Ce mécanisme permet d'effectuer des opérations, telles que l'ajout, la suppression ou le remplacement dynamique de composants.

2.2 Langages de description d'architectures et styles architecturaux

L'architecture logicielle a émergée comme un domaine important du génie logiciel. Un langage de description d'architectures (ADL, Architecture Description Language) fournit

une syntaxe concrète et une structure générique conceptuelle pour caractériser les architectures, et permettre la construction de systèmes complexes. L'intérêt de l'architecture logicielle réside dans le fait qu'il est possible d'adapter une application en reconfigurant dynamiquement son architecture.

2.2.1 Les langages de description d'architectures

Les langages de description d'architectures permettent de déclarer des configurations architecturales sous forme d'assemblage de composants et éventuellement de connecteurs. En général le couplage entre spécification architecturale et implémentation est faible. Ils permettent, donc, d'élaborer et d'explorer les modèles à un stage avancé de conception, qui met l'accent sur la structure du système logiciel. [Sha 06]

Voici quelques-unes des définitions données par différents chercheurs.

"Les langages de description d'architecture (ADL) sont des langages formels et peuvent être utilisés pour représenter l'architecture d'un système logiciel intensif " [Cle 97].

"Un ADL pour les applications logicielles se concentre sur la structure de haut niveau de l'application globale plutôt que sur les détails d'implémentation d'un module source spécifique" [Med 00]

Un ADL est un langage décrivant une architecture logicielle [Juk 04]. Un ADL stimule la construction d'une architecture sans pour autant implémenter les composants, teste et analyse l'architecture, produit des classes et supporte également la conception descendante [Gra 02]. Les ADLs nous permettent de soutenir et de contrôler le processus de développement dès son début [Juk 04]. Les ADLs permettent de laisser de coté les ligne de code et de se concentrer sur les composants logiciels et leur structure d'interconnexion [Juk 04]. Cependant, il n'est pas tout à fait clair quel niveau de soutien un ADL devrait fournir aux développeurs mais un minimum serait d'aider à la compréhension et à la communication sur un système logiciel [Juk 04]. C'est pourquoi les ADL doivent permettre aux développeurs d'avoir une vision de l'ensemble du système, en fournissant une description simple et compréhensible.

Jukka Harkki [Juk 04] déclare que : "Un ADL n'est pas un langage de programmation, ni une notation de conception de haut niveau, ni un langage d'interconnexion de modules, ni une notation de modélisation orientée objet et non plus un langage de spécification formelle".

D'autre part. En considérant ce qui pourrait être modifié, les chercheurs proposent de fournir aux ADLs des liens vers des technologies courantes (UML et Java), des outils d'analyse puissants, une représentation graphique, des vérificateurs de modèles, des analyseurs, des compilateurs, des outils de synthèse de code, un support d'exécution, etc. [Woo 05]. Par dessus tout, pour qu'un ADL soit utile et facilement utilisable, il doit fournir un support d'outils pour le développement et l'évolution des architectures [Juk 04].

2.2.2 Contraintes et styles architecturaux

Une Contraintes est une propriété ou une affirmation à propos d'un système ou de l'un de ses éléments, dont la violation rendra le système obsolète (ou moins souhaitable) [Cle 97]. Il est impératif de spécifier des contraintes afin d'assurer l'adhésion aux utilisations prévues des composants, et d'établir des dépendances entre les différents éléments. Il existe plusieurs types d'assertions :

- Les Invariants spécifient une condition qui doit être toujours vraie.
- La construction heuristique fournissent aux architectes et aux concepteurs un moyen de capturer des règles de conception moins strictes que les invariants.
- les prés conditions spécifient des conditions qui doivent être garanties avant un traitement quelconque.
- Les post conditions spécifient des conditions qui doivent être garanties après un traitement quelconque.

Un style architectural définit une famille de systèmes connexes, généralement en fournissant un vocabulaire de conception architecturale spécifique au domaine, ainsi que des contraintes sur la façon dont les pièces peuvent s'interconnecter.

Les styles varient de génériques, tels que *client-server* et *pipe-filter* [Sha 96- Sch 13], à spécialisés, tels que *Mission Data Systems (MDS)* de NASA [Dvo 00, Dvo 02] et *l'IEEE Distributed Simulation Standard* [Kuh 99, Sym 00].

L'utilisation de styles architecturaux est l'un des piliers les plus importants de l'architecture logicielle moderne [Kim 10].

2.3 Dynamicité des architectures logicielles

Les architectures dynamiques ont la propriété de pouvoir être modifiée après que le système ait été conçu et pendant son exécution. La réflexion (section 2.2.1) permet d'exposer les architectures des systèmes afin de pouvoir les observer et les modifier par l'intermédiaire de reconfigurations dynamiques (section 2.2.2). Les systèmes adaptatifs utilisent les mécanismes de reconfigurations dynamiques de leur architecture pour s'adapter à des changements de contexte.

2.3.1 Réflexion et architectures réflexives

2.3.1.1 Systèmes réflexifs

Un système réflexif est un système logiciel capable d'utiliser sa propre représentation pour raisonner et agir sur lui-même, afin d'étendre et d'adapter son comportement [Oli 98] ainsi que sa structure interne.

Il possède plus précisément deux capacités complémentaires [Lég 09] :

— La capacité d'*introspection*, qui représente l'opération de consultation des informations décrivant le système, résultant de la phase de réification ; en particulier de sa structure et de son comportement.

— La capacité d'*intercession*, qui signifie la modification des structures contenant la représentation abstraite d'un système réflexif. Cette modification est naturellement reflétée sur le système lui-même.

L'intercession permet donc de modifier les entités réifiées du système. La réflexion est le processus par lequel un système peut raisonner et agir sur lui-même. Un système réflexif se compose d'un niveau de base et d'un niveau méta. Le niveau de base est le raisonnement du système, et le méta niveau a accès aux représentations du niveau de base. La réification est le processus par lequel les représentations abstraites du niveau de base sont générées. Les deux niveaux sont causalement connectés. Cela signifie que les changements au niveau de base provoquent des modifications au niveau méta et réciproquement.

Trois aspects complémentaires de la réflexion sont à distinguer : [Ket 04]

— **La réflexion structurelle**. Elle concerne les structures de données comme des entités de première classe [Mal 92]. Elle permet d'observer et de manipuler la façon dont le logiciel est organisé.

Dans le cas particulier des langages à objets, l'introspection correspond à la découverte dynamique de la classe d'un objet, des variables d'instances et des méthodes dont il dispose. L'intercession permet d'étendre la structure des objets mais pas de leur comportement.

L'utilisation des méta-classes représente une des approches les plus courantes de la réflexion structurelle. Les méta-classes permettent de concrétiser les classes, et de les rendre manipulables en exécution, au même titre que les objets. En intervenant sur la structure d'une classe, on peut ainsi manipuler la structure des objets.

La figure 2.8 montre une illustration d'un système réflexif basé sur les méta-classes.

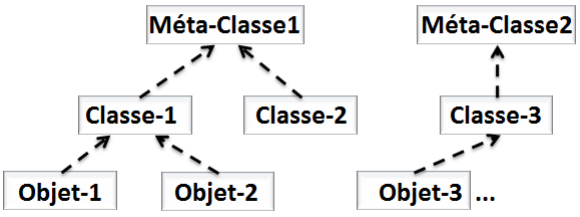

FIGURE 2.8 – Utilisation des méta-classes

La classe "java.lang.Class" de Java permet, par exemple, de retourner des informations décrivant les classes (attributs, méthodes, super-classes, ...), d'instancier les classes, etc. Toute autre classe est alors une instance de cette méta-classe.

— **La réflexion comportementale.** Elle s'intéresse à l'exécution du programme. L'introspection permet par exemple de savoir quelles sont les méthodes ou fonc-

tions en cours d'exécution. L'intercession permet d'intervenir sur le déroulement de l'exécution du système, par exemple intercepter une invocation de méthode et surcharger son exécution : l'appel est intercepté, réifié, et remplacé par un appel à une méthode du méta-objet [Bru 00]. Le méta-objet peut alors faire le traitement souhaité, ou le déléguer tel quel à l'objet de base. Il peut aussi réaliser des pré et post traitements, etc. La figure 2.9 montre le lien entre un objet et son méta-objet.

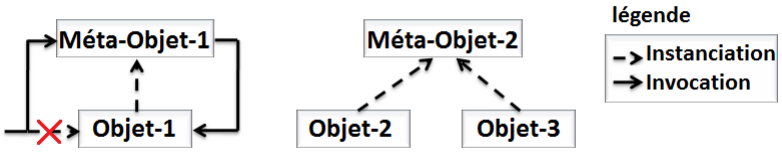

FIGURE 2.9 – Utilisation des méta-objets

— **La réflexion architecturale.** Elle s'intéresse plutôt aux interconnexions entre éléments architecturaux. Dans [Caz 99] par exemple, les auteurs ont introduit la notion des tours architecturales réflexives qui peuvent être vues comme une extension des tours réflexives introduites par Maes [Mae 87]. La figure 2.10 montre une simple illustration d'un système architecturalement réflexif. Les changements opérés à une entité d'une certaine couche sont reflétés sur l'entité qu'elle réifie, et vice-versa.

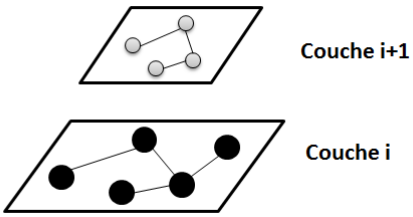

FIGURE 2.10 – Système architecturalement réflexif

Un concept clé de la réflexion est celui de la transparence : dans un système réflexif, les objets de base ne sont pas conscients de l'existence du niveau méta. Ainsi, le développement des objets est indépendant de celui des méta objets, et la connexion des méta objets aux objets est effectuée sans modifier aucun d'entres eux [Tra 99]. La réflexion améliore l'évolution en permettant la personnalisation des objets au niveau méta et en encapsulant à ce niveau la partie d'une application plus susceptible de changer lorsque de nouvelles exigences sont prises en compte [Sch 96].

2.3.1.2 Langages réflexifs

Les langages à objets sont particulièrement adaptés à la mise en œuvre d'architectures réflexives [Mae 87] grâce à leur structure (encapsulation des données) et leur organisation (héritage et composition).

La réflexion permet ainsi d'ouvrir les implémentations des langages aux programmeurs en leur fournissant un outil pour étendre et faire évoluer ces langages en fonction des besoins spécifiques d'un domaine ou d'une application. Le coût en performances de la réflexion n'est par contre pas négligeable du fait de la nécessité de l'instrumentation des langages pour pouvoir être modifiés. Il peut de même se poser des problèmes en terme de sécurité à partir du moment où le comportement d'une application peut être détourné de ses fonctionnalités originales.

2.3.1.3 Architectures réflexives

L'ouverture et la flexibilité apportées par la réflexion est applicable aux architectures logicielles et plus particulièrement aux modèles de composant [Caz 98]. La réflexion est notamment un bon support pour la dynamicité dans les architectures logicielles [Cos 99]. Dans un modèle de composant réflexif, le niveau de base correspond aux composants applicatifs dotés d'interfaces de réification au méta-niveau. Ces interfaces peuvent être invoquées à l'exécution pour observer (introspection) et modifier (intercession) la structure et le comportement des composants. Nous parlerons, dans ce dernier cas, de reconfigurations dynamiques. La réflexion structurelle permet d'exposer la structure des composants et des connecteurs en tant qu'assemblage eux-mêmes de composants et de connecteurs si le modèle est hiérarchique, et l'ensemble de leurs interfaces sont requises et fournies. En plus d'exposer la structure des systèmes à base de composants, la réflexion autorise sa modification par l'intermédiaire d'opérations d'intercession. La réflexion comportementale se traduit dans les modèles de composant par l'observation et la modification du comportement du système, tels que les interactions entre composants ou le comportement interne d'un composant. L'utilisation d'intercepteurs au niveau des interfaces des composants est, à ce titre, un moyen de réaliser ce type de réflexion. Dans un système implémenté avec un modèle de composant réflexif, une connexion causale existe entre la représentation de l'architecture exposée par le système et l'assemblage concret des composants de ce système.

2.3.2 Reconfiguration dynamique

Reconfiguration dynamique basée sur l'architecture

Certains types de systèmes nécessitent les reconfigurations dynamiques pour leurs évolutions, notamment les système adaptatifs et ceux à haute disponibilité. L'intérêt, ici, est que le système doit rester disponible et ses services interrompus au minimum.

La reconfiguration dynamique permet à un système d'évoluer de manière automatique. L'architecture de celui ci se base sur les composants pour modifier le systèmes informatiques à l'exécution [Ore 98a]. La nature de l'architecture présente l'avantage de bénéficier de la modularité et de l'encapsulation forte des composants, ce qui permet de cibler précisément les effets des reconfigurations dans le système.

Il existe deux types de reconfiguration dynamique : programmée et ad-hoc. La reconfiguration programmée est spécifiée au moment de la conception, tandis que la reconfiguration ad-hoc ou non-anticipées est imprévisible au moment de la conception. Cette dernière consiste en des modifications arbitraires, et peut intervenir à un moment quelconque de l'exécution. Les deux types de reconfigurations peuvent éventuellement être supportées en même temps dans un modèle de composant.

Opérations de reconfiguration

La modification de l'architecture peut se faire en ajoutant de nouvelles instances de composants, en supprimant des instances déjà existantes, ou en changeant les interconnexions entre les instances. Plusieurs opérations sont possibles. Celles communément supportées par les modèles de composants réflexifs sont :

— L'ajout de nouveaux composants (le composant est créé)
— Le retrait de composants existants (temporairement ou durablement)
— La connexion et la déconnexion de composants et de connecteurs

La cohérence de l'application reconfigurée

Compte tenu de la priorité du système de reconfiguration, responsable de lancer les opérations de reconfiguration par rapport à l'application qu'il reconfigure, il est nécessaire de restreindre ses privilèges pour préserver la cohérence de l'application.

Malgré les avantages de l'approche architecturale, plusieurs problèmes se posent, néanmoins concernant les reconfigurations dynamiques dans les architectures à composants [Ore 98b]. Les considérations suivantes sont ainsi à prendre en compte :

— Le modèle d'architecture d'un système doit rester cohérent avec son implémentation au cours de l'exécution et des reconfigurations. Les modèles de composants réflexifs résolvent ce problème en maintenant une connexion causale entre la représentation architecturale et le système à l'exécution.
— Les reconfigurations dynamiques dans un système doivent préserver certaines propriétés non-fonctionnelles (e.g. performances, sécurité, qualité de service, etc.), et ce en dépit de possibles défaillances. L'ajout de contraintes ou de styles architecturaux permettent par exemple de restreindre les transformations des architectures.
— La reconfiguration doit être lancée à des moments adéquats. En effet, le système de reconfiguration ne doit pas forcer l'application à se reconfigurer brusquement. L'application doit être dans un état stable, à partir duquel, elle peut reprendre normalement son exécution.
— Un problème complémentaire à la synchronisation est la gestion de l'état des com-

posants reconfigurés, et plus précisément le transfert d'états entre composants. Il peut ainsi être nécessaire de transférer l'état d'un ancien composant vers le nouveau composant qui le remplace dans l'architecture.

— L'exécution des reconfigurations dynamiques doit non seulement être synchronisée avec l'exécution fonctionnelle du système, mais aussi avec les autres occurrences de reconfigurations simultanées, afin d'éviter les conflits et les états incohérents du système.

2.4 Conclusion

Plusieurs concepts généraux et technologies autour de la fiabilité des reconfigurations dans les architectures à composants ont été présentés au cours de ce chapitre. La programmation par composants, associée à des outils et des langages dédiés, comme les langages de description d'architecture, est un bon support pour les reconfigurations dynamiques d'architecture. Grâce à la réflexion, les modèles de composants permettent l'introspection et les modifications dans les systèmes à l'exécution, tout en maintenant une connexion causale entre architecture et système concret.

Modifier dynamiquement un système à travers son architecture n'est pas sans poser des problèmes de fiabilité, préoccupation importante du génie logiciel. En général, la plupart des modèles de composants ne définissent pas les besoins et les règles à respecter pour que les applications basées sur ces modèles puissent évoluer et s'adapter dynamiquement au cours de leurs exécutions. [Ket 04].

Chapitre 3

Fiabilité des architectures logicielles reconfigurables

L'objectif de ce chapitre est de recenser et d'analyser plusieurs travaux liés aux problèmes des reconfigurations dynamiques des systèmes basés sur des descriptions architecturales. La section 3.1 introduit les concepts sous-jacents à la reconfiguration dynamique. La Section 3.2 regroupe des travaux autour de plateformes supportant la reconfiguration dynamique de systèmes à base d'architectures. La section 3.3 décrit une synthèse de quatre ADLs supportant les reconfigurations dynamiques, et décrite dans le tableau 3.1. Finalement, la section 3.4 vient conclure ce chapitre en soulignant le fait que certains problèmes restent ouverts.

L'introduction d'ADLs dans les architectures logicielles a complètement modifié les méthodes et les pratiques de l'ingénierie logicielle. L'accent est dorénavant mis sur les éléments architecturaux plutôt que le code et les sous-routines. Les langages de description d'architecture (ADLs) ont été proposés pour ce type de développement de logiciels basés sur l'architecture. Il existe un certain nombre d'ADLs différents dans les milieux universitaires et dans l'industrie ; Ils ne sont pas totalement adoptés par la communauté de l'ingénierie logicielle, mais ils ont leur importance [Hus 13].

Une reconfiguration dynamique est un ensemble d'opérations pour modifier une configuration existante au moment de l'exécution sans interruption de service. C'est principalement le cas des systèmes critiques s'ils doivent évoluer, car toute perturbation du service peut avoir des conséquences importantes.

Plusieurs travaux utilisent une spécification architecturale comme base de la reconfiguration dynamique. En effet les ADLs servent à décrire les éléments qui peuvent être reconfigurés ainsi que les contraintes auxquelles le système doit se conformer durant la reconfiguration.

Cependant, parmi plusieurs ADLs existants, seuls quelques-uns permettent la reconfiguration dynamique de la modélisation [Min 12]. π-ADL [Oqu 04], Acme/Plastik [Bat 08, Joo 05], C2SADL [Med 96, Ore 98c], DAOP-ADL [Pin 03], Darwin [Mag 96], Dynamic

Wright [All 97b, All 98], Rapide [Ver 99], Weaves [Ore 99], et xADL [Das 02] sont des exemples typiques. Néanmoins, il n'y a pas de consensus actuel sur la façon dont les ADLs doivent aborder la reconfiguration

3.1 Critères de sélection des travaux

L'objectif de ce chapitre est de recenser et d'analyser plusieurs travaux liés aux problèmes des reconfigurations dynamiques des systèmes basés sur des descriptions architecturales. [Min 12] propose une comparaison de quatre ADLs bien connus π-ADL, ACME, C2SADL et Dynamic Wright. Ces langages supportent la reconfiguration dynamique, en dépit du fait qu'ils s'appuient sur différents paradigmes, respectivement π-calcul [Oqu 04], la logique de prédicat de première ordre [Gar 00], composant et événement [Tay 96, Med 96], et les graphes de grammaires et les processus séquentiels de communication (CSP) [All 98, All 97b]. Ces ADLs se complètent les uns les autres ; π-ADL modélise les comportements des architectures, Acme s'intéresse à la structure, C2SADL Attire l'attention sur les composants et leurs événements simultanés, et Dynamic Wright supporte la définition des structures et des comportements.

Rappelons que les architectures logicielles décrivent les systèmes en spécifiant leurs éléments et les interactions entre eux [Bas 03, Sha 96]. Les concepts de base des architectures sont définis dans [Gar 10, Med 00, Sha 96] : un composant est une unité de calcul et de stockage ; un connecteur est une unité de communication ; et une configuration est une spécification d'une architecture logicielle en termes de composants, de connecteurs et de relations entre elles.

Les ports, les rôles, les comportements, les contraintes et les propriétés non fonctionnelles sont d'autres concepts utilisés pour décrire ces éléments architecturaux. Les ports et les rôles sont utilisés pour définir le mode d'interaction, les ports pour les configurations et les composants, et les rôles pour les connecteurs. Les comportements sont le calcul interne des éléments architecturaux. Les deux derniers concepts, contraintes et propriétés non fonctionnelles, sont utilisés pour désigner les assertions, les invariants, la qualité de service que les éléments architecturaux devraient satisfaire.

En outre, il est nécessaire de considérer les types et les instances de configurations, de composants et de connecteurs pour la modélisation d'une architecture logicielle [Med 00, Sha 96]. Les types sont des abstractions comme les classes dans le paradigme orienté objet qui encapsulent à la fois la structure et les comportements qui peuvent être réalisés sur sa structure. Les types seront utilisés pour créer les instances et les instances seront utilisées pour décrire les architectures logicielles.

D'un point de vue implémentation, les modèles de composants servant à la configuration des systèmes varient selon les ADLs et les plateformes utilisées. Il existe une multitude de langages utilisés pour la spécification des configurations ainsi que des reconfigurations

architecturales. Certains sont extensibles et permettent de créer de nouveaux types ainsi que de nouvelles contraintes, afin de maintenir au mieux la cohérence entre le système à l'exécution et sa représentation architecturale.

La plupart des travaux mettent en avant la possibilité de contraindre les architectures, afin de maintenir la cohérence et garantir la fiabilité des systèmes. Une reconfiguration dynamique est un ensemble d'opérations pour modifier une configuration existante au moment de l'exécution. Au niveau de l'architecture du logiciel, l'opération est définie en fonction des éléments architecturaux au niveau type ou instance. Selon les ADLs, ces opérations sont spécifiées dans des langages différents dont certains sont dédiés. Les opérations primitives sont validées à chaque étape via des contraintes (invariants pré/post conditions, de nature structurelle ou comportementale). En cas de violation de la cohérence de l'architecture, la nature de ces opérations permet un retour à un état stable.

En outre, la reconfiguration dynamique peut être *fermée* (programmée), ou *ouverte* (non-anticipée) [Ket 04]. La reconfiguration fermée est une reconfiguration dynamique programmée spécifiée au moment de la conception. Alors que celle dites ouverte peut intervenir arbitrairement pendant l'exécution du système. Avoir la possibilité d'étendre les opérations de reconfiguration en y ajoutant de nouvelles opérations, ainsi que le support des reconfigurations concurrentes représente un plus en matière de reconfiguration dynamique.

3.2 Plateformes construisant un modèle d'architecture

Cette section regroupe des travaux autour de plateformes supportant la reconfiguration dynamique de systèmes à base d'architectures.

3.2.1 π-ADL

Objectif de l'ADL. π-ADL est basé sur π-calcul et il est conçu pour décrire les systèmes concurrents et mobiles [Oqu 04]. La description des éléments architecturaux est principalement représentée par les ports et les comportements.

Support de la reconfiguration dynamique. Cet ADL permet la reconfiguration dynamique prévue et imprévue. Les changements prévus peuvent être décrits dans le comportement de n'importe quel élément architectural. Par contre ceux imprévus ont besoin d'outils de support (e.g la machine virtuelle dans archware).

Récapitulatif de π-ADL. Certains problèmes sont identifiés concernant le support de la reconfiguration dynamique par π-ADL :

1. Modifier un type nécessite une aide externe, par exemple, jeu d'outils, hyper-code et π-ARL. Le jeu d'outils et l'hyper-code sont utilisés pour la capture, la spécification

des modifications et l'application de ces modifications. Alors que π-ARL est un langage pour spécifier des rajustements dans le système ;

2. Les modifications des instances peuvent être spécifiées dans des éléments architecturaux ou aussi avec des outils et l'aide d'hyper-code. Au début, généralement, la reconfiguration dynamique est emmêlée avec un comportement nominal. Cependant, d'autres approches sont utilisées avec un composant spécifique pour spécifier une reconfiguration dynamique ;
3. Concernant les modifications de types, π-ADL ne fournit pas de mécanisme pour concevoir les états intermédiaires pour mettre à jour les instances avec l'amélioration des comportements. π-ARL représente une meilleure approche dans ce cas.
4. Avec π-ARL, il est possible de définir des contraintes et des propriétés non-fonctionnelles pour les éléments architecturaux. Cependant, il n'est pas utilisé pour évaluer la reconfiguration dynamique.

3.2.2 C2SADL

Objectif de l'ADL. L'objectif initial de C2 était de décrire l'architecture d'un logiciel en se basant sur une interface utilisateur graphique (IUG) [Tay 96].

Par conséquent, l'ADL repose sur des composants concurrents hiérarchiquement et permet également l'utilisation de messages pour notifier les éléments architecturaux. C2 est subdivisé en 2 langages : C2 IDL (langage de description d'interface) pour décrire les types de composants et C2 ADL pour spécifier les configurations. Pour les composants, il est possible de spécifier les ports supérieur et inférieur, de déclarer les méthodes internes et de spécifier le comportement de ses ports. La mise en œuvre des méthodes internes est effectuée par le développeur dans le code source généré par un ensemble d'outils. Avec C2 ADL, il est possible de définir des instances de composants et de connecteurs, ainsi que des liens entre eux.

Support de la reconfiguration dynamique. Le langage de modification d'architecturale (AML) a été créé pour étendre C2 afin de supporter la reconfiguration dynamique [Med 96, Ore 98c]. AML est un langage déclaratif qui utilise la vue de structure d'une configuration existante et des messages pour notifier cette configuration à propos d'une reconfiguration dynamique. Ce langage fournit des instructions pour créer les scripts qui spécifient la reconfiguration dynamique ; cependant, dans ces scripts, il n'est pas possible de déterminer le moment pour appliquer une reconfiguration dynamique. Ce qui rend l'intervention humaine nécessaire via un ensemble d'outils pour déclencher une reconfiguration dynamique.

Récapitulatif de C2SADL. Les lacunes suivantes ont été identifiées :

1. Des services pour supporter la reconfiguration dynamique, tels que, pour commencer et arrêter les éléments architecturaux, ne sont pas disponibles dans AML. Ainsi,

la spécification de reconfiguration dynamique est limitée.

2. La reconfiguration dynamique prévue nécessite une intervention humaine car C2SADL ne fournit pas de déclarations pour que le système applique une reconfiguration dynamique.
3. Il ne supporte pas la spécification de la reconfiguration dynamique à l'intérieur des éléments architecturaux. Par conséquent, la reconfiguration dynamique prévue est mise en œuvre uniquement au niveau implémentation ;
4. Il ne fournit pas de mécanisme pour contrôler et surveiller les états intermédiaires d'une reconfiguration dynamique.

3.2.3 Dynamic Wright

Objectif de l'ADL. Dynamic Wright se concentre sur la structure et le comportement d'une architecture [All 97b]. Il permet la description des types, des structures et des comportements des éléments architecturaux, ainsi que la configuration initiale. La structure, la configuration initiale, les instances et les liens sont décrits sous une forme déclarative, tandis que les comportements sont spécifiés dans un graphique et une variante du processus séquentiel de communication (CSP).

Support de la reconfiguration dynamique. Dynamic Wright ne supporte que la reconfiguration dynamique prévue qui est construite comme un comportement spécial de la configuration initiale. L'extension décrite dans [All 97b] propose des déclarations supplémentaires et des événements de contrôle. Les déclarations sont crées, supprimées, attachées et détachées. Les événements de contrôle permettent de définir le moment spécifique pour effectuer une reconfiguration dynamique. Dynamic Wright ne supporte pas les reconfigurations imprévues.

Récapitulatif de Wright. Les lacunes suivantes ont été identifiées :

1. Il ne fournit pas d'opérations pour définir une reconfiguration dynamique pour les types d'éléments architecturaux ;
2. Cet ADL fournit un mécanisme de contrôle des événements pour déterminer le moment pour effectuer une reconfiguration dynamique. Bien qu'il ne fournit pas un mécanisme pour contrôler et surveiller les états intermédiaires. par exemple. Dans un système bancaire, l'état doit être copié de l'instance du composant remplacé vers le nouveau composant. Dynamic Wright ne fournit pas d'opérations à cet effet.
3. En dépit de la spécification des comportements des composants et des connecteurs, il n'est pas possible de définir une reconfiguration dynamique pour ces éléments architecturaux ;
4. Tout comme les autres ADLs, Dynamic Wright ne fournit pas un mécanisme pour l'évaluation de la reconfiguration dynamique.

3.2.4 Acme/Plastik

Objectif de l'ADL. ACME/Armani est un langage déclaratif basé sur la logique de prédicat de premier ordre [Gar 00, Mon 01]. Son but initial était de définir un langage d'échange commun pour les outils de conception d'architecture. En outre, Il est conçu pour décrire les structures architecturales au niveau instance et type. Les extensions Acme/-Plastik [Bat 08, Joo 05] permettent de soutenir la reconfiguration dynamique et préservent son objectif initial. En plus, les configurations et reconfigurations dynamiques spécifiées à l'aide de cette extension (Acme / Plastik) sont basées sur la structure d'une architecture.

Le langage Acme est utilisé pour la description des architectures. Acme est un ADL générique, focalisé sur les aspects structurels. Il joue le rôle de langage pivot car suffisamment riche d'un point de vue des concepts d'architecture manipulés (Composant, port, connecteur, rôle et système), et permet de décrire des architectures spécifiées dans d'autres langages. Les représentations permettent de décrire des systèmes à un niveau plus ou moins élevé d'abstraction. Les représentations autorisent ainsi la spécification d'implémentations multiples et alternatives pour un même composant. Par ailleurs, le langage inclut la notion de style architectural à travers les types et les familles d'architectures. Le système de typage permet d'étendre la spécification des éléments. Les familles regroupent des ensembles de définitions de types et décrivent les moyens autorisés pour les composer.

Support de la reconfiguration dynamique. Acme/Plastik permet de décrire les reconfigurations dynamiques prévues et imprévues. Concernant les évolutions imprévues, il est possible de les décrire comme comportements spécifiques d'une configuration grâce à :

On (expression conditionnelle) *do* {opérations}

Ces changements peuvent être exprimés dans un script, qui sera appliqué en cours d'exécution via les outils fournis par la plateforme Plastik. Les opérations et le script sont composés d'instructions Acme et de son extension /Plastik. L'expression conditionnelle est composée dans le langage Armani [Mon 01]

Plastik [Bat 05] est à la fois un canevas logiciel pour la spécification et le développement de système à base de composants, et une plateforme d'exécution pour gérer les reconfigurations dynamiques programmées et non-anticipées de ces systèmes, tout en préservant leur intégrité au cours de leurs modifications. L'approche intègre et étend un langage de description d'architecture (Acme/Armani) et un modèle de composant réflexif avec son environnement d'exécution (OpenCOM [Cou 04]).

Pour gérer les reconfigurations programmées, le langage Acme est étendu via une primitive qui spécifie une condition de déclenchement des opérations de reconfiguration. La primitive est un prédicat Armani du type : *On* (<predicate>) *do* <action>

L'exécution des actions de reconfiguration est réalisée séquentiellement de manière transactionnelle, où les invariants Armani (figure 3.1) sont utilisés pour garantir que des

contraintes architecturales sont bien préservées malgré les reconfigurations. Il n'est pas possible d'exprimer des préconditions ou postconditions sur les opérations de reconfiguration.

Pour spécifier les reconfigurations, Plastik introduit des extensions dans Acme, notamment pour déconnecter et retirer des éléments de l'architecture ; les opérations d'ajout et de connexion étant déjà présentes dans le langage initial. La plateforme d'exécution de Plastik (figure 3.2) définit deux niveaux de spécifications ADL, le niveau style pour spécifier des familles d'architectures et des invariants génériques, et le niveau instance qui doit être conforme au style choisi. Le niveau d'exécution consiste en l'environnement d'exécution d'OpenCOM dans lequel sont instanciés les composants du système. Plastik réalise un mapping entre les niveaux ADL et exécution [Joo 05].

```
Style PlastikMF {
          Port Type ProvidedPort, RequiredPort;
          Role Type ProvidedRole, RequiredRole;
          ….
};

Component Type OSIComp: PlastikMF {
    ProvidedPort Type upTo, downTo;
    RequiredPort Type downFrom, upFrom;

    Property Type layer =
       enum {application, transport, network, link};
};

Connector Type conn2Layers: PlastikMF {
    ProvidedRole Type source;
    RequiredRole Type sink;
};

Invariant
    Forall c:OSIComp in sys.Components
          cardinality(c.layer = application) = 1 and
          cardinality(c.layer = transport) = 1 and
          cardinality(c.layer = network) = 1 and
          cardinality(c.layer = link) = 1 and

Property Type applicationProtocol;
Property Type transportProtocol;
Property Type networkProtocol;
Property Type linkProtocol;
```

FIGURE 3.1 – Exemple de spécification en Acme avec des contraintes en Armani

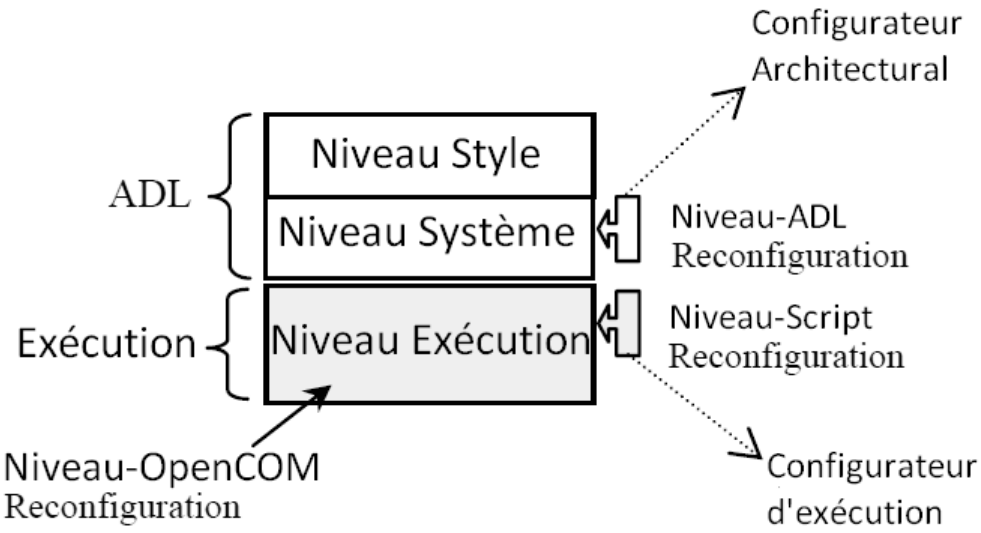

FIGURE 3.2 – Architecture Plastik

Modèle de composants. Le modèle de composants dans Plastik est OpenCOM. Il s'agit d'un modèle hiérarchique et réflexif qui réifie les composants comme unité d'exécution et de déploiement, et les interfaces pour l'interaction entre composants. Les composants sont déployés dans des capsules dotées de capacité de (re)configuration (e.g., connexion entre interfaces et réceptacles, chargement/déchargement de composants). Les composants sont implémentés par défaut en C++ et le langage de définition d'interface utilisé est l'OMG IDL. L'environnement d'exécution d'OpenCOM repose sur plusieurs méta-modèles réflexifs pour l'introspection et la reconfiguration : un méta-modèle d'architecture qui expose un graphe causalement connecté des composants du système, un méta-modèle d'interception pour l'introduction d'intercepteurs au niveau des liaisons entre composants, et un méta-modèle d'interface pour la découverte dynamique des interfaces de composants.

OpenCOM a largement été utilisé au cours des dernières années pour créer des systèmes reconfigurables, tels que les environnements middleware et les environnements de réseaux programmables [Cou 03].

Une vue de haut niveau du modèle de programmation est donnée dans la figure 3.3. Les composants (les rectangles remplis) sont des unités encapsulées de fonctionnalité et de déploiement qui interagissent avec leur environnement (c'est-à-dire d'autres composants) exclusivement à travers des interfaces (les petits cercles) et des réceptacles (le petits demi-cercles). Un composant peut supporter plusieurs interfaces et réceptacles et peut être un composite interne (c'est-à-dire composé d'autres composants).

Les composants sont déployés au moment de l'exécution dans des environnements appelés capsules (la boîte en pointillé) qui prennent en charge un «API» d'exécution contenant des opérations pouvant charger/décharger des composants (et aussi lier/déconnecter

les interfaces et les réceptacles).

Une interface représente un élément qui fournit des services, tandis qu'un réceptacle représente une unité qui requiert des services.

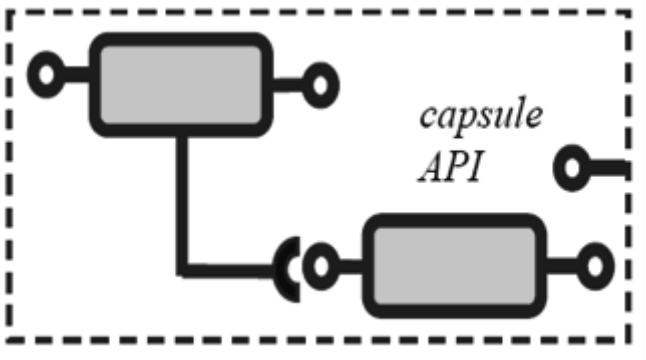

FIGURE 3.3 – Le modèle de composants OpenCOM

OpenCOM est un modèle de composants qui cible les systèmes devant être reconfigurés dynamiquement, et ceci grâce à ses capacités d'introspection, en particulier les systèmes d'exploitation, les intergiciels, les PDA et les systèmes embarqués.

Depuis la version 2, OpenCOM fournit les quatre notions suivantes [Cou 07] : capsule, caplet, loader et binder. Une capsule est l'entité qui enveloppe et administre les composants applicatifs. Un caplet est une partie d'une capsule qui contient un sous-système de l'application. Les binders et les loaders sont des entités de première classe qui permettent différentes opérations de chargement et de liaison pour les composants. Les caplets, les loaders et les binders sont eux-mêmes des composants. Les composants OpenCOM fournissent des interfaces et requièrent des réceptacles.

OpenCOM se base sur un noyau simple, efficace, minimaliste et extensible, et ceci afin de soutenir un large éventail de domaines et d'environnements de déploiement et dans le but de maximiser le potentiel générique des modèles à base de composants.

La conception d'OpenCOM tente de répondre aux exigences suivantes :

— Une technologie de construction de systèmes à usage général ne devrait fournir que des fonctionnalités génériques et abstraite, tout en étant extensible afin qu'elle puisse être spécialisée de manière naturelle si nécessaire. Il en est de même pour la reconfiguration qui doit être supportée, sans pour autant dicter les politiques qui la contrôlent et la gèrent.

— La technologie devrait être déployable directement dans un large éventail d'environnements, des PC, des superordinateurs, des PDAs et des systèmes embarqués.

— En plus d'avoir des besoins en mémoire minimes, la technologie devrait limiter ses besoins en termes de ressources de traitement.

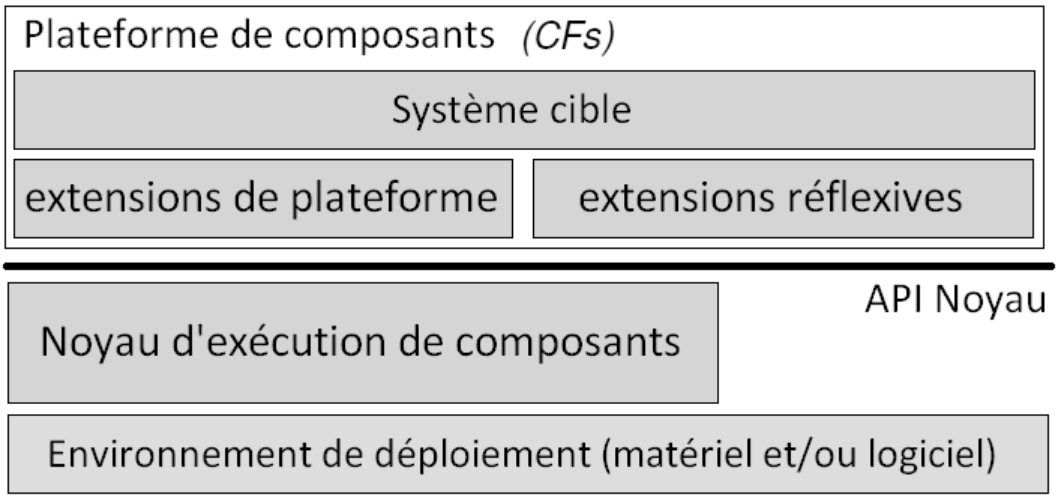

FIGURE 3.4 – Architecture globale OpenCOM

Récapitulatif d'Acme/Plastik. Les problèmes de reconfiguration dynamique peuvent être résumés comme suit :

1. En termes de structure, Acme/Plastik permet la reconfiguration dynamique. En termes de comportement, les évolutions sont décrites par les propriétés des éléments par n'importe quel langage externe, notamment pour les évolutions prévues. la limite de cette approche réside dans le fait que les éléments architecturaux (composant et connecteurs) nécessitent généralement la reconfiguration dynamique de leurs comportements, mais Acme/Plastik est dépourvu d'éléments de spécification de comportements. L'architecte doit utiliser des langages externes (en général un langage formel, tel que CSP).
2. Acme/Plastik ne possède pas de mécanismes qui contrôlent les états intermédiaires d'une reconfiguration.
3. Acme/Plastik se base sur des scripts externes pour gérer les évolutions imprévues. Cette approche requière un interpréteur externe associé à l'ADL afin d'interpréter le script externe et d'opérer la reconfiguration du système.

3.3 Synthèse des ADL étudiés

Une synthèse des quatre ADL supportant les reconfigurations dynamiques est décrite dans le tableau 3.1. π-ADL et Acme/Plastik sont les seules ADL qui supportent les reconfigurations dynamiques prévues et non prévues. C2SADL, avec le langage AML, ne fournit pas un moyen de spécifier une initiation interne de la reconfiguration dynamique. La reconfiguration dynamique prévue ne peut pas être déclenchée automatiquement. Dynamic Wright ne supporte que la reconfiguration prévue en définissant le comportement spécial d'une configuration.

Les quatre ADLs ne traitent pas le problème de la reconfiguration cohérente (reconfiguration des types et des instances), comme le montre le tableau 3.1. π-ADL et Acme fournissent des opérations pour les deux niveaux, mais les deux ADL utilisent une aide externe pour spécifier une reconfiguration au niveau du type. C2SADL AML fournit des opérations pour modifier les instances. Dynamic Wright ne fournit pas d'opérations pour spécifier une reconfiguration dynamique pour le type d'éléments architecturaux.

	Reconfigurations dynamiques de types	Reconfigurations dynamiques d'instances	Reconfigurations dynamiques programmées	Reconfigurations dynamiques non-anticipées
π-ADL	+	+	+	(Outil Archware)
Acme /Plastik	+	+	(comportement spécial dans la configuration)	(Plateforme Plastik)
C2SADL	-	(Opérations définies dans AML)	+	(Script externe en AML)
Dynamic Wright	-	(Opérations définies dans une extension de l'ADL	(comportement spécial dans la configuration)	-

TABLE 3.1 – Support des reconfigurations dynamiques

Bien que ces ADLs supportent les reconfigurations dynamiques à des niveaux variables, le problème de la fiabilité de ces reconfigurations n'est pas réellement abordé comme un objectif à part entière. En effet cet aspect est peu détaillé, et il manque clairement un mécanisme robuste pour l'évaluation du système. π-ADL peut compter sur π-AAL pour définir et analyser les contraintes architecturales et les propriétés non fonctionnelles. Alors que Acme/Plastik possède Armani, même s'il n'a pas de sémantique formelle, et donc, le système n'est pas soumis à une analyse rigoureuse [Wae 04]. C2SADL fournit une vérification de type pour la communication d'événements. Dynamic Wright fournit un mécanisme de contrôle des événements pour déterminer le moment adéquat pour effectuer une reconfiguration dynamique.

3.4 Conclusion

Ce chapitre aborde certaines questions importantes pour le support de la reconfiguration dynamique au niveau architectural : Les changements prévus et imprévus ; Modifications de niveau d'instance et type ; Structure et comportement pour tous les éléments architecturaux ; Définition du comportement de reconfiguration nominale et dynamique pour tous les éléments architecturaux ; Et, l'analyse de la reconfiguration dynamique.

Nous pouvons conclure que certains problèmes restent ouverts car aucun ADL ne prend en charge tous les aspects en même temps : comment appliquer les modifications apportées au niveau type à leurs instances respectives ? Comment contrôler l'ensemble des états intermédiaires d'un système logiciel pendant une reconfiguration dynamique ? Comment évaluer la reconfiguration dynamique ?

La fiabilité des reconfigurations dynamiques est un problème global qui requiert une démarche rigoureuse, dont le point de départ serait de définir le concept de cohérence dans une architecture, essentiellement d'un point de vue structurel.

La cohérence d'une architecture est ensuite conditionnée par la satisfaction d'un ensemble de contraintes exprimées dans une certaine logique. Les contraintes doivent être extensibles, permettre d'exprimer les invariants aux niveaux des configurations, et couvrir le coté sémantique des opérations de reconfiguration et ce grâce aux pré et post conditions. La cohérence doit être maintenue dans le système reconfiguré. Des mécanismes doivent permettre de synchroniser les opérations de reconfigurations entre elles, et d'isoler l'exécution fonctionnelle du système. Il est aussi important de vérifier les contraintes avant de valider les modifications des reconfigurations.

Deuxième partie

Contribution

Chapitre 4

Modélisation des configurations Acme

Ce chapitre aborde le problème de la spécification des configurations de composants dans le modèle Acme et la définition de contraintes d'intégrité sur ces configurations dans le but de garantir la cohérence des architectures des systèmes [Mez 18]. Nous commençons, dans la section 4.1, par présenter la démarche globale suivie pour spécifier er vérifier la cohérence des configurations Acme. La section 4.2 est dédiée à la présentation du modèle Acme, qui a été choisi comme base de notre étude en raison de sa relative simplicité et de sa nature de format commun pour les outils de conception d'architectures. Acme fournit une forme intermédiaire pour l'échange d'informations entre différents outils ADLs [Gar 97], et propose en plus des capacités d'introspection ainsi qu'un environnement de reconfiguration [Bat 05]. Dans la section 4.3, nous proposons un méta-modèle simple du modèle de composants, ainsi que la possibilité de l'étendre par des ensembles de contraintes. Nous verrons enfin, dans la section 4.4, comment vérifier la cohérence entre les contraintes du modèle et comment déclarer et valider ces contraintes.

4.1 L'ADL Acme

Acme repose sur la prémisse qu'il existe suffisamment d'aspects communs entres les ADLs, ce qui permettrait de partager un ensemble d'informations significatives indépendamment de l'ADL cible. Pour aborder le problème de la fiabilisation des configurations ainsi que des reconfigurations dynamiques dans les architectures à composants, nous sommes partis de l'étude d'un modèle de composants particulièrement générique, fort de ses extensions Armani/Plastik, permettant respectivement de contraindre les configurations, ainsi que d'exprimer les conditions déclenchant des reconfigurations programmées, tel que détruire un élément, et attaché ou détacher certains éléments architecturaux.

4.1.1 Cœur du modèle Acme

Dans [Gar 97], Garlan décrit Acme en mettant en avant ses caractéristiques fondamentales comme suit :

— Une ontologie architecturale constituée de sept éléments basiques de description d'architectures ;
— Un mécanisme d'annotation flexible supportant l'association d'informations non structurelles grâce à des définitions externes de sous-langages ;
— Un mécanisme de typage pour définir de manière abstraite des éléments réutilisables, tel que les styles ;
— Une plateforme sémantique ouverte pour le raisonnement sur les descriptions architecturales.

Acme est construit sur l'ontologie de base de sept types d'entités pour la représentation architecturale : composants, connecteurs, systèmes, ports, rôles, représentations, et carte de représentations. Ceux-ci sont illustrés dans la Figure 4.1.

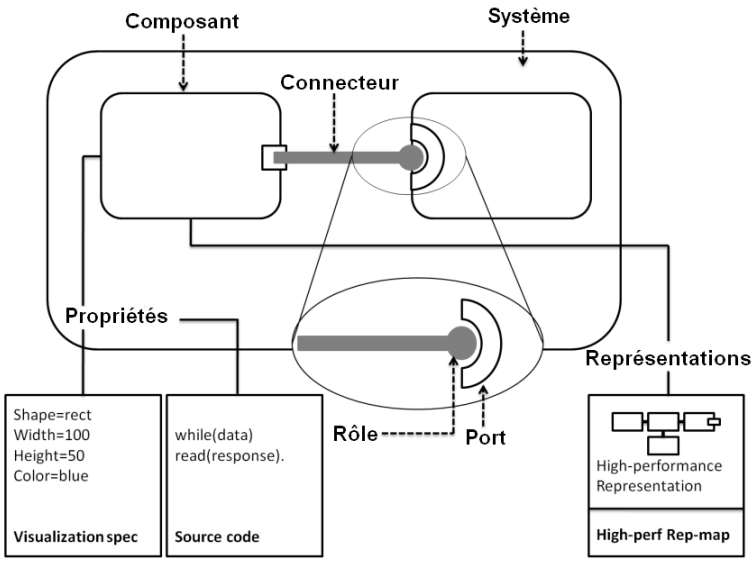

FIGURE 4.1 – Elements d'une description Acme [Acm a]

Les éléments décrits ci-dessus sont les éléments clés pour décrire une architecture comme un graphe hiérarchique de composants et de connecteurs. Cependant, il y a plus d'intérêt dans une description architecturale que l'aspect structurel. Certaines informations auxiliaires doivent être ajoutées pour compléter la description avec certains aspects, tels que le type de données communiquées entre les composants, les protocoles d'interaction et plus encore. Pour ce faire, Acme soutient l'annotation de la structure architecturale avec des propriétés. Une propriété Armani est constituée d'un nom, un type et une valeur. Les annotations peuvent être ajoutées à chacun des sept éléments.

4.1.2 Implémentations et outils associés au modèle

L'écosystème du modèle Acme offre un certain nombre d'outils et de langages pour l'ingénierie des systèmes. Parmi ces outils, nous utilisons principalement les éléments suivants :

— AcmeStudio [Stu], qui est un environnement d'édition personnalisable et un outil de visualisation pour les conceptions architecturales de logiciels basées sur le langage de description architectural Acme (ADL).

— Armani le contrôleur de contraintes intégré pour vérifier les règles de conception architecturale. Le langage de prédicat Armani est la partie du langage de conception Armani utilisée pour spécifier les invariants des prédicats et les heuristiques de conception. Le langage de prédicat est basé sur la logique du premier ordre, avec des termes, des fonctions et des quantificateurs.

— Plastik [Bat 05] est une plateforme d'exécution pour gérer les reconfigurations dynamiques de systèmes à bases de composants. L'approche étend le langage de description d'architecture Acme.

4.2 Processus de vérification de la cohérence

Acme est un langage de description d'architectures, considéré comme un pivot unifiant les concepts utilisés par les autres ADLs, tout en étant assez simple pour faciliter son analyse. Le projet Acme a débuté en 1995 dans le but de fournir un langage commun, dédié à l'échange de descriptions architecturales entre différents outils architecturaux. Depuis le projet a évolué et plusieurs outils ont vu le jour, au point où Acme et sa bibliothèque (Acme lib) fournit une infrastructure générique et extensible pour la description, la génération et l'analyse de descriptions d'architectures logicielles. Si nous voulons déterminer si une configuration d'une application décrite sous Acme est valide, il nous faut vérifier qu'elle est conforme au modèle de composants, et donc, il est nécessaire de préciser cette spécification. A la base, Acme permet de modéliser les configurations (ou architectures) statiques des systèmes ; mais si nous voulons également pouvoir représenter les modifications (reconfigurations) de ces architectures, il est impératif d'utiliser les extensions mise à disposition par la communauté [Bat 05]. Il est important de pouvoir à tout moment de son cycle de vie valider une application Acme, par exemple au niveau instance, et ce en minimisant l'impacte sur les performances, et en se reposant sur les propriétés d'introspection et d'intercession d'Acme.

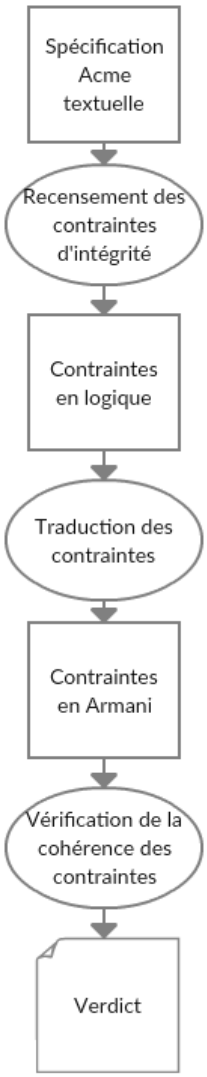

FIGURE 4.2 – Processus de vérification des contraintes

La démarche suivie (cf. Figure 4.2) consiste, à partir de la spécification textuelle, à extraire un ensemble de contraintes sur le modèle Acme que nous appelons contraintes d'intégrité. Cet ensemble de contraintes doit être nécessairement satisfait par une configuration pour qu'elle puisse être cohérente. Les contraintes sont dans un premier temps spécifiées dans un formalisme logique indépendant de tout langage d'implémentation. Elles sont ensuite traduites dans un langage de spécification, Armani, qui grâce à son analyseur permet de vérifier la non contradiction entre les contraintes.

Les principes suivants ont guidé la conception du langage Armani : [Mon 01]

— Armani est construit comme une extension d'Acme pour favoriser une traduction facile vers et depuis Acme. Cette interchangeabilité devrait permettre aux utilisateurs d'Armani de profiter des outils existants ou futurs basés sur Acme.

— Armani se concentre sur la description de la structure du système et sur la façon dont cette structure peut évoluer au fil du temps, plutôt que le comportement dynamique de l'exécution des systèmes.

— Armani est un langage déclaratif. Les spécifications d'Armani décrivent la structure architecturale et les contraintes sur cette structure ; Elles ne décrivent pas les opérations pour modifier ou supprimer des structures architecturales. De façon informelle, une spécification Armani décrit la structure d'un système et les contraintes qui doivent maintenir ce système à mesure qu'il évolue. Ce sont les outils de vérification du langage qui vérifient la validité des contraintes (analyseur).
— Le langage de contrainte d'Armani utilise un langage basé sur la logique de prédicat de premier ordre pour exprimer les contraintes de conception. La vérification de la satisfaction des contraintes d'Armani est possible en quantifiant les variables uniquement sur des ensembles finis.

4.3 Spécification du modèle Acme

4.3.1 Un méta-modèle pour Acme

Pour clarifier la spécification du modèle Acme, nous présentons d'abord un méta-modèle (Figure 4.3). Cette représentation permet de mettre en évidence les concepts de base du modèle : les éléments architecturaux (composants, connecteurs, systèmes, ports, rôles, représentations et rep-maps), les relations entre ces éléments (lien, hiérarchie, etc.) et le système de type.

Pour simplifier ce méta-modèle et afin d'en faciliter la manipulation, nous fournissons une abstraction (Figure 4.4). Cette représentation est essentiellement un sous-ensemble du méta-modèle de la figure 4.3, mais qui est plus homogène car elle ne considère que les éléments architecturaux et les relations entre ces éléments.

Le but de ce graphe est de simplifier au maximum la représentation du modèle pour faciliter la validation des configurations, tout en conservant autant d'informations que possible. Le graphique comprend les principaux concepts définis dans le premier méta-modèle, tout en simplifiant les éléments du méta-modèle. C'est le cas des liens (bindings et attachments) qui apparaissent comme hiérarchie de classes dans le diagramme UML et ont été réduits à une simple relation (arc) dans le graphe ; ce qui respecte la philosophie du modèle Acme.

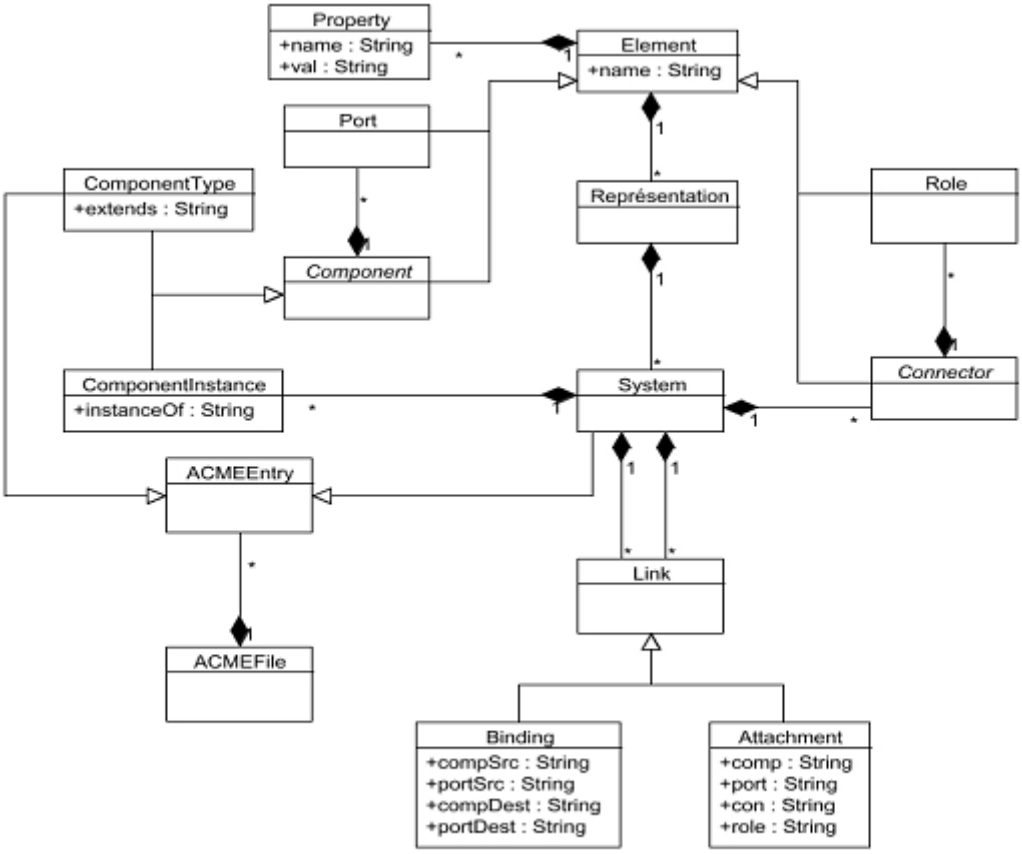

FIGURE 4.3 – Un méta-modèle Acme [Bau 06]

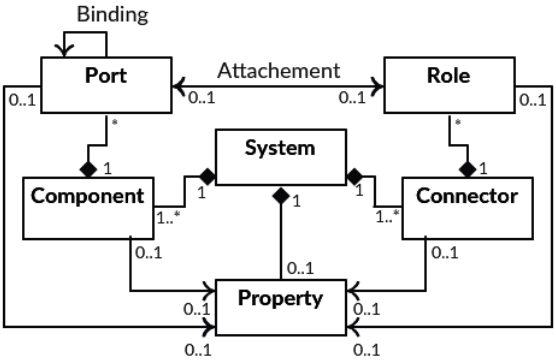

FIGURE 4.4 – Méta-modèle Acme simplifié

Représentation par graphe du modèle Acme. À un moment donné, une application Acme est modélisée par une configuration d'éléments architecturaux. Cette approche a été utilisée dans [Lég 09] Nous définissons donc une configuration Acme comme un graphe orienté dont les sommets (ou nœuds) et les arêtes sont typés. Une configuration est dites valide (ou cohérente) si son graphe est cohérent avec le modèle Acme (la sémantique de la relation de conformité est définie par la suite). Ce graphe est défini comme

$S = (E.R)$, où l'ensemble des sommets E représente les éléments architecturaux du système tel que défini dans le modèle et l'ensemble des arcs R les relations entre ces éléments. Les éléments architecturaux manipulés dans les configurations sont des données du modèle de composants qui peuvent être représentées par le triplet $M = (M_E, M_R, M_I)$ où :

— M_E est l'ensemble des éléments architecturaux qui peuvent apparaître dans les instances du modèle.

— M_R est l'ensemble des relations possible entre éléments architecturaux.

— M_I est l'ensemble des contraintes d'intégrité qui doivent être vérifiées par les instances du modèle pour être considéré comme valide.

Une contrainte d'intégrité est un prédicat binaire $\pi(E, R)$.

Nous définissons alors qu'une configuration (ou un système) S est conforme au modèle M (M modélise S ou vice versa) et notons $M| = S$ si et seulement si :

$E \in M_E$

$R \in M_R$

$\forall \pi \in M_I, \pi(E, R)$

Nous nous concentrons initialement sur la modélisation du noyau du modèle Acme défini par ses trois composantes : Acme = (AcmeE , AcmeR, AcmeI).

Nous utilisons la logique du premier ordre comme formalisme de spécification du modèle, c'est-à-dire pour spécifier les relations architecturales et les contraintes d'intégrité sur les configurations. Nous considérons l'utilisation des opérateurs logiques classiques (Booléens) $\vee, \wedge, \neg$ et l'implication $\Longrightarrow$, le quantificateur universel $\forall$ et l'existentiel $\exists$. L'ensemble des opérateurs utilisés sont : l'union $\cup$, l'intersection $\cap$, la différence \et l'égalité =.

Le graphe du modèle de configuration est défini dans ce qui suit par ce formalisme :

Types de données : Nous manipulons les types de données primitifs (Integer, String, Float, Boolean, Sequence, Set, Enum, Record). Nous définissons les types suivants qui sont des extensions des types primitifs :

— Nom qui étend String : nom d'un élément.

— Signature qui étend String : Signature des interfaces.

Opérateurs arithmétiques : Les opérateurs arithmétiques habituels (+, -, x, /) sont utilisés pour les types Numériques et également pour les comparaisons. Les tests d'égalité (opérateur =) sont définis pour tous les types de données.

A. Les éléments architecturaux et leurs propriétés : Les éléments architecturaux du modèle Acme sont les sommets du graphe de la configuration et sont quatre : composants, connecteurs, ports et rôles. Chaque sommet du graphe peut être associé à un certain nombre de propriétés. Ces données primitives caractérisent les éléments architecturaux représentés :

— Les composants : Composant = Name

— Les connecteurs : Connecteur = Name

— Les rôles : Rôle = Name x Type (client/Server)

— Les ports : Port = Name x Signature x Type (Requis/fourni)

Nous avons tous les éléments architecturaux du modèle Acme :

$\text{Acme}_E = Composant \cup Connecteur \cup Rôle \cup Port$

Chaque instance d'élément architectural à un nom unique qui la distingue au niveau système.

B. Les relations architecturales :

Les relations entre les éléments architecturaux du modèle Acme sont des relations binaires xRy où (x,y)$\in E^2$et correspondent aux arcs dans le graphe de configuration. Figure 4.5. Nous considérons les relations primitives suivantes dans Acme :

— La relation hasPort définie sur Composant x Port : détermine si un composant possède un port donné (interne ou externe).

— La relation hasRole définie sur Connecteur x Rôle : détermine si un connecteur a un rôle donné.

— La relation hasChild définie sur Elément x Elément : détermine si un élément est sous-composant direct d'un autre élément. hasPort et hasRole sont des spécialisations de cette relation.

— La relation hasBinding définie sur Composant x Composant : détermine si un Composant est lié à un autre Composant. Cette relation n'est pas réflexive parce qu'un port ne peut être lié à lui-même.

— La relation hasAttachement définie sur Port x Rôle : détermine si un port est directement lié au rôle d'un connecteur. Il n'est pas transitif parce qu'il ne considère que les connexions directes.

Nous avons toutes les relations primitives définies dans le modèle :

$\text{Acme}_R = hasRole \cup hasPort \cup hasChild \cup hasBinding \cup hasAttachement$

4.4 Spécification du modèle par des contraintes d'intégrité

En plus de la définition des éléments architecturaux, de leurs propriétés et de leurs relations architecturales, nous utilisons des contraintes d'intégrité pour compléter la spécification du modèle. Les contraintes utilisées dans cette spécification sont des invariants structurels et comportementaux. Nous pouvons distinguer les contraintes architecturales sur les relations et les contraintes sur les valeurs des propriétés des éléments architecturaux.

Nous nous concentrons d'abord sur les contraintes du modèle Acme. Ces contraintes d'intégrité sont des prédicats binaires $\Pi_i(E, R)$qui garantissent la conformité syntaxique et compositionnelle d'une configuration. Ce sont des invariants génériques qui spécifient

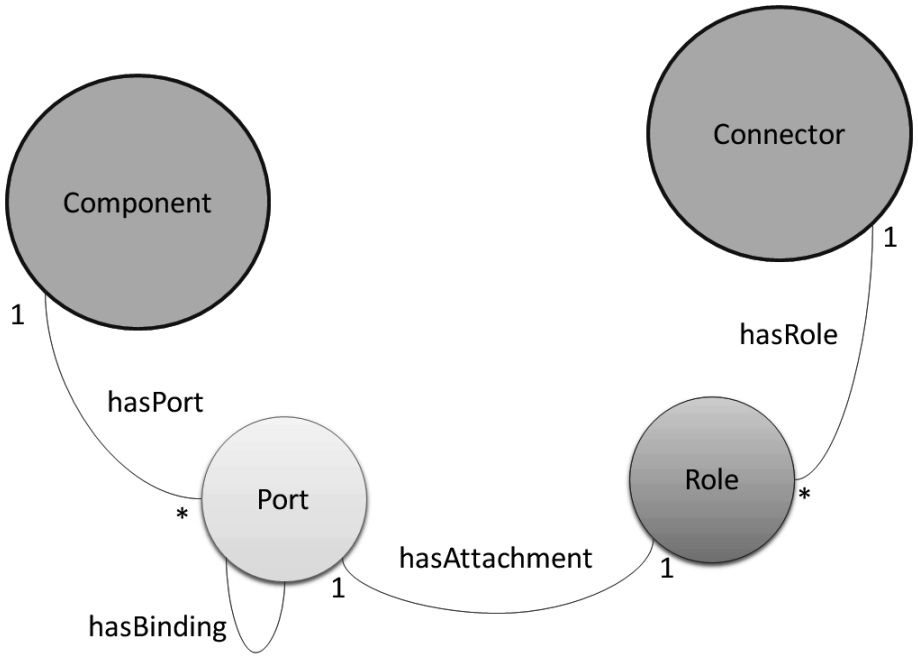

FIGURE 4.5 – Représentation des éléments et des relations architecturales dans Acme

la modélisation d'un système Acme. La difficulté est de ne pas sous-contraindre le modèle pour éviter des configurations incohérentes (faux positifs), ou sur-contraindre, empêchant les configurations cohérentes d'être validées (faux négatifs). En outre, les contraintes ne doivent pas être contradictoires entre elles et, dans la mesure du possible, ne pas être redondantes.

Un premier ensemble de contraintes concerne la cardinalité de la relation entre les composants et les ports et entre les composants et les propriétés.

Unicité d'appartenance d'un port : Un port appartient à un seul composant.

$$uniquePortOwner(E, R) = \forall P \in E, \exists! C \in E, hasPort(C, P)$$

Unicité d'appartenance d'un rôle : Un rôle appartient à un seul connecteur.

$$uniqueRoleOwner(E, R) = \forall R \in E, \exists! Cn \in E, hasRole(Cn, R)$$

Contraintes concernant l'unicité des noms d'éléments architecturaux

Unicité des noms de ports avec la même visibilité : Un composant peut avoir à plus d'un port avec un nom donné.

$$uniquePortName(E, R) = \forall (C, P_1, P_2) \in$$
$$E^3, (hasPort, (C, P_1) \wedge hasPort, (C, P_2) \wedge name(P_1) = name(P_2)) \Rightarrow P_1 = P_2$$

Unicité des noms de rôles : Un connecteur peut avoir au plus un rôle avec un nom donné.

$$uniqueRoleName(E,R) = \forall(Cn,R_1,R_2) \in$$
$$E^3, (hasRole,(Cn,R_1) \wedge hasRole,(Cn,R) \wedge name(R_1) = name(R_2)) \Rightarrow R_1 = R_2$$

Unicité des noms de propriété : un élément (composant, connecteur, rôle et port) peut avoir au plus une propriété avec un nom donné.

$$uniquePropName(E,R) = \forall(e,p_1,p_2) \in$$
$$E^3, (hasProperty(e,p_1) \wedge hasProperty(e,p_2) \wedge name(p_1) = name(p_2)) \Rightarrow p_1 = p_2$$

Compatibilité des ports associés : Un lien entre deux ports est valide si et seulement si les deux ports sont du même type. Soit $\leq_i$, la relation de sous-typage des ports :

$$bindingType(E,R) = \forall(p_1,p_2) \in E^2, hasBinding(p_1,p_2)$$
$$\Longrightarrow signature(p_1) \leq signature(p_2)$$

Tous les ports requis doivent être liés :

$$mandatoryBindings(E,R) = \forall(C,P) \in E^2, hasPort(C,P_1) \wedge client(P_1)$$
$$\Longrightarrow \exists P_2 \in E, hasBinding(P_1,P_2)$$

Tous les rôles doivent être liés : Tous les rôles sont attachés.

$$mandatoryAttachements(E,R) = \forall(Cn,R) \in E^2, hasRole(Cn,R)$$
$$\Longrightarrow \exists P \in E, hasAttachement(R,P)$$

Les liens primitifs locaux : Les connexions primitives ne peuvent pas traverser les limites des composites pour respecter le principe de l'encapsulation. Il n'existe que trois catégories possibles de liens primitifs :

- Connexions normales entre des ports dont les composants ont au moins un parent commun direct **(Attachment),** en passant par un connecteur.
- Les connexions d'exportation entre un port client interne d'un composite et un port serveur externe de l'un de ses sous-composants directs. **(Binding).**
- Les connexions d'importation entre un port client externe d'un composant et un port serveur interne de ses parents **(Binding)**.

$$normalAttachment(P_c,P_s) = client(P_c) \wedge external(P_c) \wedge server(P_s) \wedge$$
$$external(P_s) \wedge \exists C_c, C_s, C_p \in E, hasINterface(C_c,P_c) \wedge hasINterface(C_s,P_s)$$
$$\wedge hasINterface(C_p,P_p) \wedge hasChild(C_p,C_c) \wedge hasChild(C_p,Cs_c)$$

$$exportBindings(P_c,P_s) = client(P_c) \wedge internal(P_c) \wedge server(P_s) \wedge external(P_s)$$
$$\wedge \exists C_c, C_s \in E, hasInterface(C_c,P_c) \wedge hasINterface(C_s,P_s) \wedge hasChild(C_c,C_s)$$

$$importBindings(P_c,P_s) = client(P_c) \wedge external(P_c) \wedge server(P_s) \wedge internal(P_s)$$
$$\wedge \exists C_c, C_s \in E, hasInterface(C_c,P_c) \wedge hasINterface(C_s,P_s) \wedge hasChild(C_s,Cs_c)$$

$$localBinding(P_c, P_s) = normalAttachment(P_c, P_s) \vee exportBindings(P_c, P_s) \vee importBindings(P_c, P_s)$$

Le prédicat de localité est donc le suivant :

$localBindings(E, R) = \forall (P_c, P_s) \in E^2, hasBindings(P_c, P_s) \Longrightarrow localBindings(P_c, P_s)$

Selon les définitions précédentes des contraintes d'intégrité, nous pouvons finalement définir l'ensemble des contraintes du modèle Acme_I.

$$Acme_I = \left\{ \begin{array}{l} uniqueName, uniquePortOwner, uniqueRoleOwner, uniquePropOwner, \\ bindingType, mandatoryBindings, mandatoryAttachment, localBinding \end{array} \right\}$$

4.5 Traduction de formalisme Armani

4.5.1 Elements architecturaux

Nous proposons le style architectural suivant appelé WFCA (Well Formed Component Based Architecture) Listing 4.1.

Listing 4.1 – Style architecturale WFCA

```
Family WFCA= {
Property Type
basic_types=enum{Boolean,Real,Integer,String, Void}
Property Type
parameter=Record(parameter_type:basic_types);
Property Type
multi_parameter=Sequence<parameter>;
Property Type
signaturesRecord[name:String;p:multi_parameters;result:
   basic_types]
Property Type
sub_program=Set{signature}; Port Type ProvidedInterface
{       Property Provided_Service:sub_program; }
Port Type RequiredInterface
{Property Required Service:sub_program; }
Component Type WFCA_Comp
{
rule atleastInterface=invariant size( self PORTS)>=1;
```

```
rule ProvidedRequiredlnterface=invariant forall p Port in
   self.PORTSIdelaresType(p,Providedlnterface) OR (
   delaresType(p,Requiredlnterface) }
Connector Type WFCA_Conn
{       Role servers=
{
rule provided_port=invariant
size(self.AttachedPorts)>=1;
rule Provided_lnterface=invariant forall p:Port in self.
   AttachedPortsI delaresType(p,Providedlnterface);
Role clients=
{
rule required_port=invariant size(self.AttachedPorts)>=1;
rule Required_lnterface=invariant forall p:Port in self.
   AttachedPorts| delaresType(p,Requiredlnterface);
})
}
```

4.5.2 Contraintes

Pour les besoins de la formalisation, il est nécessaire de faire une correspondance totale des fonctions d'Armani, indiquées dans [Mon 01] :

Fonctions de connectivité du graphe

attached(conn : Connector, comp : Component) : boolean Renvoie Vrai si le connecteur est attaché au composant comp, Faux sinon.

attached(r : Role, p : Port) : boolean Renvoie Vrai si le role r est attaché au port p, Faux sinon.

connected(c1, c2 : Component) : boolean Renvoie Vrai si le composant c1 est directement connecté au composant c2 par au moins un connecteur ; et Faux sinon.

reachable(c1, c2 : Component) : boolean Reachable(. . .) est la fermeture transitive de Connected(. . .). Elle renvoie Vrai si le composant c2 est joignable à partir du composant c1 ; Faux sinon. Cette fonction examine uniquement les connectivités de graphe non dirigées.

Fonctions parents enfants

parent(c : Component) : System Renvoie le système dans lequel le composant c est instancié ; ou nul si c n'a pas de parent.

parent(c : Connector) : System Renvoie le système dans lequel le connecteur c est instancié ; ou nul si c n'a pas de parent.

parent(p : Port) : Component Renvoie le composant dans lequel le port p est instancié ; ou nul si p n'a pas de parent.

parent(r : Role) : Connector Renvoie le connecteur dans lequel le rôle r est instancié ; ou nul si r n'a pas de parent.

parent(s : System) : Representation Renvoie la représentation dans laquelle le system s est déclaré ; ou nil si s est un système haut niveau et n'est donc pas déclaré dans une représentation.

parent(p : Property) : Element Renvoie l'élément dont p est une propriété ; ou nul si p n'a pas de parent.

parent(r : Representation) : Element Renvoie l'élément dont r est une représentation ; ou nul si r ne représente pas de système.

Un premier ensemble de contraintes concerne la cardinalité de la relation entre les composants et les ports et entre les composants et les propriétés.

Listing 4.2 – Cardinalité des composants

```
Uniqueness belonging of port:
Rule uniquePortOwner(sys:System) : boolean = Forall cl :
   Component in sys.Components | Exists c2 : Component in sys
   .Components | Forall P:Port in sys.Ports | parent(pl)=cl
   and parent(Pl)=c2 =>cl = c2
Uniqueness belonging of a role:
Rule uniqueRoleOwner(sys:System) : boolean = Forall cl :
   Connector in sys.Connectors | Exists c2 : Connector in sys
   .Connectors | Forall RrRole in sys.RoiesI parent(R)=cl and
    parent(R)=c2 =>cl = c2
```

Contraintes concernant l'unicité des identifiants et des noms d'éléments architecturaux.

Listing 4.3 – Unicité des noms et identifiants

```
Uniqueness of identifiers:
Analysis uniqueId(sys:System) : boolean =Forall el : Element
   in sys.Elements | Forall e2 : Element in sys.Elements | el
   .Id = e2.Id => el=e2*
Local primitive bonds:
Rule normalAttachement(sys:System) : boolean = Forall Pc:Port
    in sys.Ports | Forall Ps:Port in sys.Ports | Exists Ce :
   Component in sys.Components | Exists Cs : Component in sys
   .Components | Exists Cp : Component in sys.Components |
   Parent(Pc)=Cc and Parent(Ps)=Cs and Pc.role="client" and
```

```
   Pc.visibility="external" and ps.role="server" and Ps.
   visibility=external and Connected(Cc,Cs)
Rule exportBinding(sys:System) : boolean = Forall Pc:Port in
   sys.Ports | Forall Ps:Port in sys.Ports | Exists Cc :
   Component in sys.Components | Exists Cs : Component in sys
   .Components | Exists Cp : Component in sys.Components |
   Parent(Pc)=Cc      and     Parent(Ps)=Cs   and     Pc.role
   ="client"        and     Pc.visibility="internal"   and ps
   .role="server" and Ps.visibility=external and Connected(Cc
   ,Cs)
Rule importBinding(sys:System) : boolean = Forall Pc:Port in
   sys.Ports | Forall Ps:Port in sys.Ports | Exists Cc :
   Component in sys.Components | Exists Cs : Component in sys
   .Components | Exists Cp : Component in sys.Components |
   Parent(Pc)=Cc      and     Parent(Ps)=Cs   and     Pc.role
   ="client"        and     Pc.visibility="external"
   and ps.role="server" and Ps.visibility="internal" and
   Connected (Cc,Cs)
Rule localBinding(sys:System) : boolean = normalAttachement(
   sys:System) or exportBinding(sys:System) or importBinding(
   sys :System)
```

4.6 Test et validation

Nous avons développé notre exemple après l'étude présentée par [ATM], à partir d'une série d'exemples complets d'analyse orientée objet, de la conception à la programmation.

4.6.1 Déclaration des exigences pour l'exemple système ATM

Le système ATM "automated teller machine" a un ensemble de composants :

— Un lecteur de carte magnétique ;
— Une console (clavier et écran) pour l'interaction ;
— Un slot pour imprimer les reçus des clients ;
— Un slot pour le dépôt d'enveloppes ;
— Un distributeur de billets ;
— Un lien dédié à la communication avec la banque.

Le distributeur peut servir un client à la fois. Un client devra insérer une carte ATM et entrer un numéro d'identification personnel (PIN). La combinaison sera utilisée par la banque pour la validation de chaque transaction.

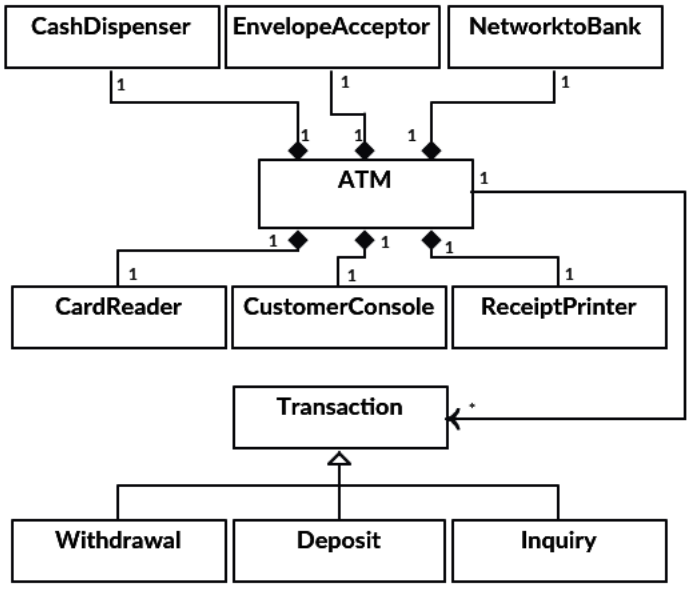

FIGURE 4.6 – Diagramme de classe ATM

Lorsque le client indique qu'il ne souhaite plus de transactions supplémentaires, la carte sera libérée par le guichet automatique.

L'ATM fournit les services suivants :

Retrait (Withdrawal) de n'importe quel compte approprié lié à la carte.

Dépôt (Deposit) sur tout compte lié à la carte, constitué d'espèces et/ou de chèques dans une enveloppe. Le client saisira le montant du dépôt dans l'ATM, sous réserve d'une vérification manuelle lorsque l'enveloppe est retirée de la machine par un opérateur.

Extrait de compte (Balance inquiry) de n'importe quel compte lié à la carte. Toutes les transactions fournies doivent être approuvées par la banque après un processus de vérification.

La banque valide le PIN du client. Si le PIN est invalide, le client devra ré-entrer le PIN. Si l'opération échoue, la carte sera conservée par l'ATM, jusqu'à ce que le client contacte la banque pour la récupérer.

Le guichet automatique fournira au client un reçu imprimé détaillé pour chaque transaction réussie.

La figure 4.6 représente le diagramme de classe ATM. Seul le nom de chaque classe est affiché sur le diagramme, afin d'en réduire la taille. Les composants sont détaillés plus loin, car cette forme de représentation est la plus proche du style architectural.

4.6.2 Diagramme de modélisation de l'ATM

4.6.2.1 Diagramme de composant

Le diagramme de composant associé au système est représenté par la figure 4.7. Le diagramme de composants a pour but principal de montrer les relations structurelles entre les composants d'un système. Les composants exposent leur fonctionnalité à travers leurs ports. Un port représente un point de contact entre le composant et son environnement. Un port peut représenter une source de requêtes, un service fourni par le composant, ou représenter une source ou une destination pour les données.

Les Composants de l'ATM :

— CardReader
— CashDispenser
— CustomerConsole
— EnvelopeAcceptor
— NetworktoBank
— ReceiptPrinter

ATM L'ATM collabore avec CashDispenser, NetworktoBank et CustomerConsole. Il démarre une nouvelle session lorsque la carte est insérée par le client et permet d'accéder aux composants pour les transactions.

CardReader Il collabore avec l'ATM et la carte. Indique l'ATM lorsque la carte est insérée, et lit les informations de la carte.

CashDispenser En partant du montant initial, il permet de suivre les transactions. Il indique si une quantité suffisante de liquidités est disponible et dispense des liquidités.

CustomerConsole Elle affiche différents messages :

Affiche un message, et accepte le PIN entré au clavier.

Affiche un message, et accepte un choix entré au clavier.

Affiche un message, et accepte un montant entré au clavier.

EnvelopeAcceptor Il accepte l'enveloppe du client ; et signale si l'opération prend trop de temps ou est annulée.

NetworktoBank Envoi un message à la banque et attend la réponse.

ReceiptPrinter Imprime les reçus.

Les composants correspondants aux différentes opérations :

— Transaction
— Withdrawal (Retrait)
— Deposit (Dépôt)
— Inquiry (Extrait de compte)

La transaction abstraite Elle permet au client de choisir un type de transaction.

Effectue un cas d'utilisation de transaction

Exécute une extension PIN non valide

Retrait(Withdrawal) Effectue des opérations propres au cas d'utilisation de retrait
Dépôt(Deposit) Effectue des opérations propres au cas d'utilisation de dépôt.
Extrait de compte(Inquiry) Effectue des opérations propres au cas d'utilisation extrait de compte

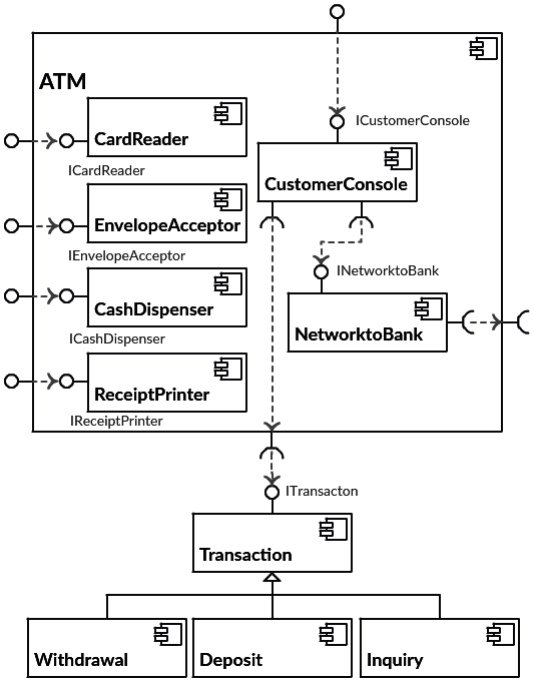

FIGURE 4.7 – Diagramme de composant de l'ATM

4.6.2.2 Interfaces de l'ATM

Dans cette section, nous présentons des signatures de services d'interfaces appartenant à des composants. Le composant ATM a les mêmes interfaces que ses composites.

Interface IEnvelopeAcceptor

Le rôle de cette interface est d'accepter l'enveloppe du client. Elle offre deux services :

— **acceptEnvelope()** Si la transaction de dépôt est approuvée, la machine accepte une enveloppe du client contenant de l'argent et / ou des chèques.

— **cancelOperation()** Après avoir demandé au client d'insérer l'enveloppe, un compte à rebours est démarré. Si le client ne parvient pas à insérer l'enveloppe dans un délai raisonnable, la transaction est automatiquement annulée.

Interface ICardReader Cette interface gère la carte du client et offre deux services :

— **readCard()** Lorsqu'un client insère une carte ATM dans la fente du lecteur de carte de la machine, l'ATM tire la carte dans la machine et la lit.

— **ejectCard()** Lorsque le client effectue des transactions, la carte est éjectée de la machine.

Interface ReceiptPrinter Cette interface imprime le reçu des différentes transactions possibles, offrant un service unique

— **printReceipt()** Imprimez le reçu correspondant.

Interface ICashDispenser L'interface fournit un service :

— **dispenseCash()** Cette opération dispense de l'argent au client.

Interface INetworktoBank

— **sendMessage()** Ce service envoie un message à la banque et attend la réponse.

Interface ITransaction

— **makeTransactions()** L'opération valide les opérations des transactions, comme la mise à jour du montant du compte.

Interface ICustomerConsole

L'interface permet une interaction avec le client, offrant les services suivants :

— **display()** Affiche un message au client, par exemple en invitant le client à entrer un code PIN ou à inviter le client à entrer le montant.

— **readPIN()** Cette opération lit le code du client.

— **readMenuChoice()** L'opération permet au client de choisir parmi un menu de types possibles de transaction.

— **readAmount()** Lorsque le client choisit un type de transaction à partir d'un menu d'options, le client sera invité à fournir les détails appropriés, y compris le montant de l'argent.

4.6.3 Formalisation d'un ATM en Acme/Armani

Dans cette section, nous proposons une formalisation Acme/Armani du modèle ATM décrit à la Section 4.6.2. Pour ce faire, nous décrivons les types de données et les signatures des services offerts par la demande en utilisant judicieusement la typographie et la construction familiale offertes par Acme. Le schéma de composants du système est formalisé dans Acme en utilisant le système de construction. Enfin, les règles de cohérence sont établies en utilisant la contrainte invariante prise en charge par Armani. L'assemblage des composants UML2 .0 de l'application est modélisé par un système appelé ATM qui dérive de la famille WFCA. Pour ce faire, nous appliquons les règles suivantes définies par [Kmi 12], et qui sont tout à fait applicables à notre exemple :

R1 : Un composant UML est traduit par un composant Acme.

R1.1 : Une interface associée à un composant UML2.0 est traduite par un port Acme.

R1.2 : Un service déclaré dans une interface UML2.0 est traduit par une propriété Acme, attachée au port qui formalise cette interface. Le type de propriété représente la signature du service.

R2 : Un connecteur d'assemblage UML2.0 reliant une interface disponible et une interface requise est modélisé par un connecteur binaire Acme avec deux rôles.

R3 : Les propriétés attachées à un rôle doivent être identiques à celles du port correspondant.

Les listings de 4.4 à 4.9 représentent les différentes parties du système ATM en Acme/Armani.

Listing 4.4 – Composant ATM

```
Component ATM = {
        Port IATMCardReader = {
                Property readCard:signaturel;
                Property ejectCard:signature2;}
        Port IATMEnvelopeAcceptor = {
                Property acceptEnvelope:signature3;
                Property cancelOperation:signature4;}
        Port IATMCashDispenser = {
                Property dispenseCash: signatures5 )
        Port IATMReceiptPrinter = { Property printReceipt:
           signatures6)
        Port IATMCustomerConsole = (
                Property display: signature7;
                Property readPIN: signature8;
                Property readMenuChoice:signature9;
                Property readAmount:signature10;}
        Port IATMTransaction = {
                Property makeTransaction:signaturel2;}
        Port IATMNetworktoBank = {
                Property sendMessage:signaturel3;}
```

Listing 4.5 – Représentation du système

```
Representation R ={
        System details={
Component CardReader = {
        Port ICardReader = {
                        Property readCard:signature1;
                        Property ejectCard:signature2;
Component EnvelopeAcceptor = {
        Port IenvelopeAcceptor = {
                        Property acceptEnvelope:signature3;
                        Property cancelOperation:signature4
                           ;}}
Component CashDispenser = {
        Port ICashDispenser = {
                        Property dispenseCash:signature5;}}
```

```
Component ReceiptPrinter = {
        Port IReceiptPrinter = {
                        Property printReceipt:signature6;}}
Component CustomerConsole = {
        Port IcustomerConsole = {
                        Property display:signature7;
                        Property readPIN:signature8;
                        Property readMenuChoice:signature9;
                        Property readAmount:signature10;}
        Port ICustomerTransaction = {
                        Property makeTransaction:signature12;
        Port ICustomerBank = {
                        Property makeTransaction:signature11
                           ;}}
Component NetworktoBank = {
        Port INetworktoBank = {
                        Property sendMessage:signature11;}
        Port INetworktoATM = {
                        Property sendMessage signature13;}}
```

Listing 4.6 – Liste des bindings

```
Bindings {
        IATMCardReader to CardReader.ICardReader;
        IATMEnvelopeAcceptor to EnvelopeAcceptor.
          IEnvelopeAcceptor;
        IATMCashDispenser to CashDispenser.ICashDispenser;
        IATMReceiptPrinter to ReceiptPrinter.IReceiptPrinter;
        IATMCustomerConsole to CustomerConsole.
          ICustomerConsole;
        IATmNetworktoBank to NetworktoBank.INetworktoATM;
```

Listing 4.7 – Liste des connecteurs

```
Connector Customer_Bank = {
        Role Customer_msg = {
                        Property Message:signature11;}
        Role Bank_msg = {
                        Property Message:signature11;}}
Connector ATm_Transaction = {
        Role ATm_execute = {
```

```
                        Property Message:signature12;}
        Role Transaction_execute = {
                        Property Message:signature12;}}
};};};
```

Listing 4.8 – Liste des attachements

```
Attachment CustomerConsole.ICustomerBank to Customer_Bank.
   Customer_msg;
Attachment NetworktoBank.INetworktoBank to Customer_Bank.
   Bank_msg;
Attachment ATM_Transaction.ATM_execute to ATM.IATMTransaction
   ;
Attachment ATM_Transaction.Transaction_execute to Transaction
   .ITransaction;
```

Listing 4.9 – Composant abstrait Transaction

```
Component Transaction ={
        Port ITransaction = {
                        Property makeTransaction:signature12
                           ;}}
Component Withdrawal = {
        Port ITransaction = {
                        Property maketransaction:signature12
                           ;}}
Component Deposit = {
        Port ITransaction = {
                        Property makeTransaction:signature12
                           ;}}
Component Inquiry = {
        Port ITransaction = {
                        Property makeTransaction:signature12
                           ;}}
```

4.6.4 Règles de cohérence et vérification du modèle

Dans cette section, nous proposons des règles de cohérence, en plus de celles définies dans le style WFCA. Ces règles sont modélisées à l'aide du concept invariant d'Acme. Ils concernent la vérification des attachements et (bindings) de la configuration ATM.

Règle : Un rôle et un port attachés ont le même nombre de propriétés (voir listing 4.10).

Listing 4.10 – Règles de cohérences d'attachements port/rôle

```
rule sizeproperty1 = invariant forall p:Port in self.
   Customer_Bank.Customer_msg.ATTACHEDPORTS | size(self.
   Customer_Bank.Customer_msg.PROPERTIES)==size(p.PROPERTIES)
   ;

rule sizeproperty2 = invariant forall p:Port in self.
   Customer_Bank.Bank_msg.ATTACHEDPORTS | size(self.
   Customer_Bank.Bank_msg.PROPERTIES)==size(p.PROPERTIES);

rule sizeproperty3 = invariant forall p:Port in self.
   ATM_Transaction.ATM_execute.PROPERTIES)==size(p.PROPERTIES
   );

rule sizeproperty4 = invariant forall p:Port in self.
   ATM_Transaction.Transaction_execute.ATTACHEDPORTS|size(
   self.ATM_Transaction.Transactoin_execute.PROPERTIES)==size
   (p.PROPERTIES);
```

4.7 Conclusion

Dans ce chapitre, nous avons proposé une modélisation d'un guichet automatique dans Acme/Armani. L'objectif de ce modèle est de valider la vérification syntaxique, réalisée par AcmeStudio. Pour ce faire, nous avons décrit les signatures des services fournis par le système, en utilisant la définition de familles fournie par Acme. En cas de détection d'erreur, l'éditeur affiche un message, tel que montré par la Figure 4.8. Au moment de la rédaction, il n'y a pas de style entièrement conçu ; c'est pour cette raison que nous avons fournit des aspects clés de la famille d'architecture.

Nous avons établit un modèle Acme bien défini pour la spécification d'architectures logicielles, travaillant sur la maîtrise de la fiabilité des architectures logicielles et l'amélioration de la cohérence entre la conception et le système. Cela se fait en maintenant la cohérence entre la conception de l'architecture à l'état initial et la configuration. Notre approche repose sur la définition d'un ensemble de contraintes d'intégrité, basées sur la logique de prédicat de premier ordre. Notre modèle garantit une conception logicielle fiable et une configuration cohérente. Cela ne peut être réalisé qu'en proposant différentes règles de configuration pour permettre la construction de systèmes incrémentaux et atomiques qui tiennent compte de l'évolutivité et des aspects de variabilité de l'architecture logicielle.

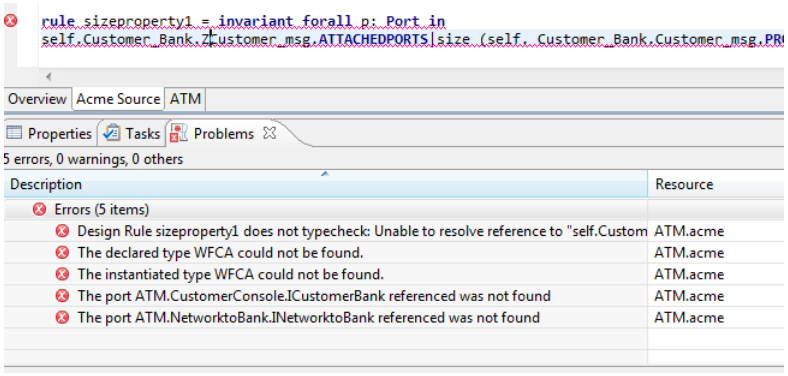

FIGURE 4.8 – Erreur dans Acmestudio

Chapitre 5

Spécification des reconfigurations dynamiques

Ce chapitre propose la prise en compte de la dynamicité des modèles d'architectures pour définir la cohérence des applications, en étudiant la sémantique des reconfigurations dynamiques.

L'étude porte plus particulièrement sur l'ADL Acme/Plastik. C'est un ADL générique (pivot), supportant les reconfigurations dynamiques d'architectures grâce à son extension Plastik.

La section 5.1 reprend le concept de contraintes d'intégrité pour spécifier les reconfigurations dynamiques dans Acme/Plastik. Nous nous intéresserons, dans la section 5.2, à la sémantique des opérations de reconfiguration comme des événements déclencheurs d'opérations de transformation de graphes de configuration à base de conditions. Enfin, et en section 5.3, nous verrons comment simuler les effets des opérations dans le modèle et comment les spécifier dans des applications concrètes à base de composants OpenCom.

5.1 Gestion des reconfigurations dynamiques par les ADLs

Certains ADLs supportent l'évolution des systèmes en proposant des capacités de reconfiguration dynamique. Ces capacités permettent de modifier les systèmes pendant leur exécution sans nécessairement les arrêter totalement (C2 [Tay 96], Darwin [Mag 96], Weaves [Gor 91], et Wright [All 95]).

Les capacités réflexives des modèles représentent un des aspects clés permettant la reconfiguration dynamique, en permettant l'inspection et l'adaptation des différents aspects du système, au moment de l'exécution.

L'aptitude de reconfiguration dynamique d'un système est très utile lorsque ces reconfigurations peuvent se produire de manière arbitraire et à partir de n'importe quel

point du système (pas seulement des membres participants à la reconfiguration). C'est ce qu'on appelle la reconfiguration dynamique de tiers [Cla 01] Cependant, tout système qui permet ce type de reconfiguration doit maintenir sa stabilité et son intégrité face à de telles opérations. Par exemple, la suppression d'un composant alors qu'il est encore utilisé entraînera un résultat imprévisible, comme le crash du système.

OpenCOM prend en charge la reconfiguration dynamique et arbitraire des composants et leurs connexions. Cela se fait grâce à la gestion des interdépendances à l'exécution et au graphe du système. OpenCOM rend la reconfiguration dynamique sécurisée en associant un verrou à chaque récipient. Pendant ce temps, toutes les opérations de reconfiguration impliquant la partie du système en cours de reconfiguration sont bloquées. Une fois que le verrou est acquis, toutes les invocations futures continuent d'être interrompues jusqu'à ce que le verrouillage soit réinitialisé.

De plus, la dynamicité des architectures doit être prise en compte dans la définition de la cohérence d'un système par des contraintes d'intégrité. En plus des invariants relatifs aux configurations statiques, ne nouveaux invariants qui portent sur les reconfigurations dynamiques sont définis.

5.1.1 Aspects clés des reconfigurations dynamiques

L'architecture Plastik. Elle comprend trois nivaux : style, system et exécution. Le niveau style contient des types génériques d'ADL et peut être utilisé dans la spécification de familles de systèmes. Plus précisément, un style spécifie un ensemble de types de composants et de connecteurs ainsi que des invariants dictant la manière de les combiner. Le niveau système contient des instances de styles spécifiques qui conforment les invariants imposés par le style. Par ailleurs, chaque système peut avoir ses propres invariants. Au final, le niveau exécution contient Les mécanismes d'exécution qui supporte l'exécution d'un système et les règles de sa reconfiguration.

Une spécification Acme standard est définie en termes de composants, ports, connecteurs, rôles, attachements, représentations et systèmes. Les composants (potentiellement composite) sont des unités de calculs munies d'interfaces appelées ports, les ports sont reliés aux ports d'autres composants en utilisant des intermédiaires appelés connecteurs ; ce qui permet de relier les rôles aux ports des composants. Les représentations sont des décompositions alternatives d'un composant donné. Les systèmes sont des graphes dans lesquels les nœuds sont des composants et les arêtes sont des connecteurs.

En outre, Les spécifications Acme utilisent des propriétés, types et styles architecturaux. Les propriétés sont un triplet (nom, type, valeur) qui peuvent être attachées comme annotation à n'importe quel élément de l'ADL.

Les types sont utilisés pour capturer les structures et les relations récurrentes. Le système de types d'Acme permet une flexibilité additionnelle en permettant l'extension

des types. Les styles architecturaux définissent les ensembles de types de composants, connecteurs et propriétés, ainsi que les règles qui contraignent la composition des instances de ces types dans un domaine architectural réutilisable.

Finalement, les invariants ou contraintes dans une architecture Acme sont définies en utilisant l'extension Armani [Mon 98], qui permet d'exprimer des invariants structurels de composition. Les invariants sont constitués d'opérateurs logiques standards, de quantificateurs existentiels et universels, et de fonctions logiques Armani (intégrées et définies par l'utilisateur). Même si Armani semble introduire un élément de dynamique, il est important de souligner qu'Acme/Armani ne supporte pas la reconfiguration dynamique des systèmes [Wil 01, Med 00, Bat 05].

Extension du language Acme. Afin de permettre à Acme d'exprimer une reconfiguration programmée, Plastik introduit ces quatre opérations de reconfiguration primitives, et qui produisent des modifications simples d'une configuration :

— on (<condition>) do <actions> : Cette construction permet à l'architecte d'exprimer des conditions d'exécution, qui entrainent des reconfigurations programmées, ainsi qu'une spécification des modifications à opérer. Les contraintes sont des expressions Armani, et les actions de modification contiennent des déclarations Acme/Armani. La sémantique prévue est que les éléments spécifiés dans la partie actions sont instanciés lorsque la condition "est déclenchée". Le déclenchement ne se produit que sur une transition de la condition de faux vers le vrai.

— detach <role> from <port> ; et remove <elem> : Ces actions sont principalement employées dans l'action on-do pour activer les modifications partielles d'une architecture. Detach supprime un attachement entre un rôle et un port ; et remove détruit un composant, connecteur ou une représentation existante. La suppression d'éléments est possible uniquement lorsqu'ils n'ont plus aucune attache.

— dependencies <statements> : Cette construction permet l'expression de dépendances entre les éléments architecturaux en cours d'exécution. Cette clause spécifie un ensemble d'éléments Acme dont l'élément "cible" a besoin. La sémantique est telle que : chaque fois que l'élément cible est instancié (par exemple, dans la clause d'action de l'instruction on-do), les dépendances indiquées sont également instanciées ; Et chaque fois que l'élément cible est détruit (en remove/detach), les dépendances indiquées sont également détruites.

— active property <propertydefinition> : Représente un type spécial de propriété qui ne peut être attachée aux ports et aux rôles. Son objectif est de se référer au flux dynamique de données sur un port ou un rôle.

Intercession et introspection. Un modèle réflexif propose à la fois des capacités d'introspection des éléments et des capacités d'intercession. La distinction entre intercession et introspection n'est pas explicite au niveau de l'ADL Acme, et les opérations Acme/Armani contiennent aussi bien des opérations d'introspection que d'intercession. Les opérations

d'introspection sont des opérations sans effet de bord sur la configuration du système, elles permettent de connaître l'état de la configuration d'un système. Les opérations d'intercession modifient au contraire la configuration du système.

5.1.2 Contraintes d'intégrité spécifiques à la reconfiguration

Dans cette section, nous étendons le concept de contraintes d'intégrité sur les configurations pour les appliquer à des architectures dynamiques, qui peuvent être reconfigurées pendant l'exécution des systèmes. En effet, une configuration peut être valide statiquement, subir une reconfiguration pour aboutir à une nouvelle configuration également valide ; alors que la reconfiguration elle-même est invalide. Il est donc important de contraindre l'ensemble des transformations possibles sur une configuration pour s'assurer de la conformité au modèle. Nous pouvons ainsi valider non seulement les états du système qui sont les configurations, mais aussi les transitions entre ces états qui sont les reconfigurations.

Une opération de reconfiguration primitive transforme une configuration $A = (E, R)$ en une autre configuration $A' = (E', R')$ où A et A' sont respectivement les états initiaux et finaux du système et op l'opération associée à la transition entre ces deux états : $A \xrightarrow{op} A'$

Aet A' doivent être valides toutes les deux, conformes au même modèle M. De plus, pour que A' soit une évolution valide de A, des invariants de configuration doivent être respectées par toutes les opérations primitives de reconfiguration. Nous appellerons ces contraintes d'intégrité des contraintes dynamiques, par opposition aux contraintes statiques définies dans le chapitre 4 et nous les notons $\pi_i = (E, E', R, R')$

Les contraintes dynamiques recensées au niveau modèle sont les suivantes :

Changement au niveau des éléments architecturaux. De nouveaux éléments peuvent être créés ou supprimés. L'architecture résultant de la reconfiguration contient donc un ensemble d'éléments de taille variable indépendamment de l'ensemble initial.

$$EltSet(E, E^{'}, R, R^{'}) = \forall e \in E, \exists e^{'} \in E^{'}, id(e) = id(e^{'}) \vee id(e) \neq id(e^{'})$$

Conservation des interfaces. L'ensemble des interfaces possédées par un élément est fixé lors de sa création :

Conservation des ports.

$$PortNoConserv(E, E^{'}, R, R^{'}) = \forall (c, p) \in E^2, hasport(c, p) \Rightarrow \exists (c^{'}, p^{'}) \in E^{'}2, eq(c, c^{'}) \wedge eq(p, p^{'}) \wedge hasport(c^{'}, p^{'})$$

Conservation des rôles.

$$RoleNoConserv(E, E^{'}, R, R^{'}) = \forall (cn, r) \in E^2, hasrole(cn, r) \Rightarrow \exists (cn^{'}, r^{'}) \in E^{'}2, eq(cn, cn^{'}) \wedge eq(r, r) \wedge hasrole(cn^{'}, r^{'})$$

Conservation des attributs. L'ensemble des attributs appartenant à un composant est fixé au moment de la création du composant ou connecteur.

$$attConserv(E, E^{'}, R, R^{'}) = \forall (c, a) \in E^2, hasattribute(c, a) \Rightarrow \exists (c^{'}, a^{'}) \in E^2, eq(c, c^{'}) \wedge eq(a, a^{'}) \wedge hasattribute(c^{'}, a^{'})$$

$$attConserv(E, E^{'}, R, R^{'}) = \forall (cn, a) \in E^2, hasattribute(cn, a) \Rightarrow \exists (cn^{'}, a^{'}) \in E^2, eq(cn, cn^{'}) \wedge eq(a, a^{'}) \wedge hasattribute(cn^{'}, a^{'})$$

5.2 Analyse des opérations de reconfiguration dans Acme /Plastik

5.2.1 Spécification des opérations primitives

OpenCOM [Cou 02] est une technologie de programmation basée sur les composants, légère, efficace, indépendante du langage et destinée à soutenir le développement de systèmes de bas niveau, ainsi que les applications et de soutenir leur reconfiguration dynamique ; par exemple, des plates-formes de middleware adaptatives et des environnements de réseau actifs [Bla 04].

En plus du niveau de granularité habituel (composants individuels), un aspect clé utilisé dans OpenCOM consiste à construire des applications ou systèmes en termes de plate-fromes de composants (CFs « Component frameworks »). Les CFs sont des ensembles étroitement couplés qui fonctionnent ensemble pour aborder un domaine de fonctionnalité spécifique.

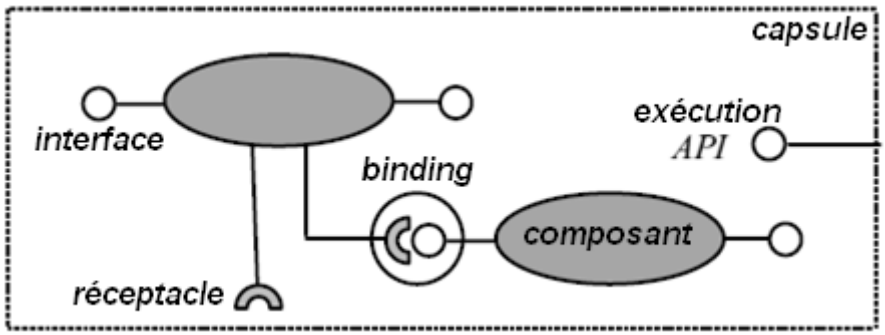

FIGURE 5.1 – Modèle de composant OpenCOM

La figure 5.1 donne une vue d'ensemble des éléments du modèle de programmation basé sur les composants d'OpenCOM et le listing 5.1 montre (une version simplifiée de) l'API du noyau d'exécution des composants (CRTK « Component Run Time Kernel ») qui prend en charge ce modèle. L'API intègre également un service de transaction intégré qui permet d'exécuter atomiquement une séquence d'opérations liées à la reconfiguration

(par exemple, load (), bind (), destroy () etc.), avec renversement en cas d'échec. Pour cela, CRTK ajoute comme argument un ID de transaction à chaque appel d'API.

Les capsules contiennent des entités qui offrent l'API CRTK et définissent un espace de noms pour les composants. Les composants sont des unités de fonctionnalité et de déploiement encapsulées, indépendantes du langage, qui interagissent avec d'autres composants exclusivement à travers des «interfaces» et des «réceptacles ». Les modèles de composants sont déployés dans des capsules à l'aide de load () et instanciés à l'aide de instantiate (). Les modèles sont chargés à partir d'un référentiel et sont annotés avec des paires <clé, valeur>. Les traces des modèles de composants sont collectées à l'aide de unload (), et les instances sont détruites à l'aide de destroy ().

Listing 5.1 – Cardinalité des composants

```
template load(comp_type name, predicate p);
comp_inst instantiate (template t);
status unload (template t);
status destroy (comp_inst comp);
comp_inst bind (ipnt_inst interface,ipn_inst receptacle);
status putprop (ID entity, ID key, opaque value);
opaque getprop (ID entity, ID key);
status notify(Callback *callback);
```

Les interfaces sont des ensembles de signatures d'opérations et des types de données associés. Les composants peuvent supporter de multiples interfaces. Les prises sont des interfaces "requises" utilisées pour rendre explicites les dépendances d'un composant à d'autres. Une liaison est une association entre une interface et un réceptacle. Les liaisons sont créées en utilisant bind () qui renvoie une instance de composant représentant la liaison (les liaisons sont supprimées à l'aide de destroy ()). Putprop () et getprop () donnent accès à un registre CRTK interne, qui enregistre des méta-données arbitraires, exprimées en termes de paires <key, value>, et qui peuvent être attachées à l'exécution à n'importe quel élément de modèle de composant.

Enfin, le but de notify () est d'informer les parties intéressées de l'activité sur l'API. Lorsqu'un rappel est enregistré avec notify (), chaque appel subséquent sur le CRTK (c'est-à-dire load (), bind () etc.) est signalé au rappel.

Au-dela du CRTK, OpenCOM prend en charge des services génériques supplémentaires qui facilitent la construction de systèmes et d'applications complexes. Ils sont eux-mêmes mis en œuvre en termes de composants et sont donc facultatifs dans n'importe quelle configuration. Le plus important est un ensemble de méta-modèles réflexif [Cou 02] qui facilitent la reconfiguration dynamique des systèmes en permettant que les différents aspects du système soient inspectés, adaptés et étendus par programme lors de l'exécution. Le méta-modèle d'architecture expose la topologie de composition des composants dans

une capsule en termes de structure de graphes ; Le méta-modèle d'interface permet de découvrir des informations sur les types d'interface au moment de l'exécution et d'appeler des instances d'interface découvertes dynamiquement au moment de l'exécution. Le méta-modèle d'interception permet d'interposer les intercepteurs aux liaisons entre les interfaces de composant. En outre, les aspects du méta-modèle de ressources permettent de geler les threads afin que les composants puissent être remplacés en toute sécurité au moment de l'exécution

5.2.2 Traduction des conditions et contraintes

Mapping Acme-vers-OpenCOM

Le mapping de base des concepts Acme aux concepts OpenCOM est présenté dans le tableau 5.1. Les styles spécifiés par l'ADL correspondent aux plate-fromes de composants CFs en tant qu'éléments spécifiques au domaine de fonctionnalités réutilisables et dynamiquement reconfigurables. Cela simplifie la connexion causale au moment de l'exécution entre les couches ADL et exécution. La plupart des autres correspondances sont intuitivement claires. Le mapping des connecteurs se fait vers des composants OpenCOM car cela permet de supporter facilement des connecteurs «abstraits», tels que des liens SQL ou des canaux RPC.

Acme/Armani	**OpenCOM**
Système	CF
Composant/Connecteur	Composant
Port/Rôle	Interface/réceptacle
Attachement	binding
Représentation	Composant composite
Propriété	Meta donnée du registre CRTK

TABLE 5.1 – Correspondance Acme vers OpenCOM [Joo 05]

5.2.3 L'infrastructure d'exécution

L'infrastructure requise pour supporter un CF OpenCom généré depuis une spécification système du niveau ADL se compose principalement des éléments suivants (voir figure 5.2) :

— Configurateur Architectural, qui compile les spécifications, écrites dans l'ADL étendue, dans les scripts de configuration et de reconfiguration. Il est nécessaire au moment de l'exécution car il doit traiter à la volée des demandes de reconfiguration de niveau ADL qui modifient les spécifications existantes.

— Configurateur d'exécution, qui exécute les scripts de reconfiguration. Il est également responsable de la gestion des représentations d'exécution des invariants spécifiés par ADL et des demandes de reconfiguration ad hoc basées sur l'évaluation

dynamique de ces invariants.

— Proxy CRTK, qui est une version par CF de CRTK. Chaque CF reçoit un CRTK proxy dédié pour fournir une isolation inter-CF. Les composants dans un CF ne peuvent interférer avec les composants dans d'autres. Notons cependant que les composants de différentes CF peuvent communiquer directement tant que cela ne viole pas les invariants imposés par les CF.

— Interpréteur Lua, interprète les scripts ainsi que la représentation d'exécution et les conditions, invariants et on do, qui sont implémentés en tant que programmes dans le langage Lua [Ler 96].

— Modèle réflexif OpenCOM, utilisé dans l'évaluation des invariants et des conditions on-do.

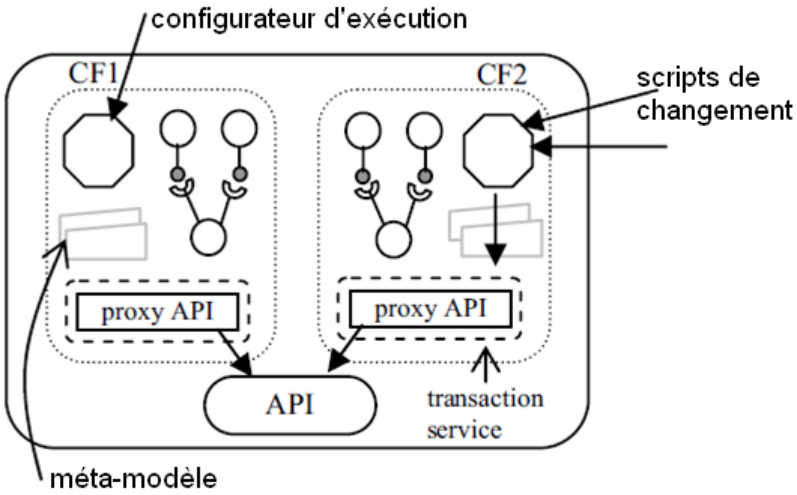

FIGURE 5.2 – Infrastructure d'exécution

Le niveau ADL de Plastik ajoute quatre nouvelles constructions à Acme/Armani qui, conjointement avec l'évaluation en temps réel des invariants Armani, prennent en charge l'expression et la gestion de la reconfiguration anticipée et non anticipée. Plastik se base sur une correspondance à partir d'une description de niveau ADL d'un système vers une plate-forme de composant OpenCOM.

OpenCOM offre des performances élevées, et permet à Plastik de fonctionner de telle sorte que ses opérations de reconfiguration fonctionnent de manière «hors bande» [Cou 04] ; c'est-à-dire qu'ils n'engagent de surcouts que lorsque le CRTK est appelé à charger / décharger / lier / séparer, etc. Dans l'exécution normale «en bande» d'une application à un état stable, ils ne comportent pas de surcouts.

L'outil AcmeLIB [Acm b] est utilisé pour implémenter la compilation et la génération de prédicats. Le langage interprété Lua sert à l'expression et l'évaluation des prédicats. Cependant, le compilateur qui génère Lua à partir d'Acme n'est pas encore entièrement opérationnel.

5.3 Plate-forme d'exécution

L'approbation émergente d'un segment de la communauté scientifique d'Acme en tant que langage d'architecture commun, est probablement ce qui se rapproche le plus d'un consensus sur les ADLs [Med 00]. Cependant, Il ne fournit pas non plus de moyens pour interpréter les caractéristiques d'une description architecturale qui sont placées dans des listes de propriétés. La où plusieurs ADLs utilisent des compilateurs architecturaux, à la base Acme est strictement utilisé comme notation de modélisation et ne fournit aucun support de génération d'implémentation.

Compte tenu des problèmes de reconfiguration dynamique dans Acme/Plastik, notamment l'absence d'éléments de spécification de comportements, les évolutions sont décrites par les propriétés des éléments dans un langage externe (combinaison OpenCom et Lua dans Plastik) ; Ce qui requière un interpréteur externe afin d'opérer la reconfiguration du système.

Ces considérations donnent lieu à une réflexion sur la conception d'un canevas logiciel dédié à une plate-forme d'exécution propre à Acme, qui permet de programmer les membranes des composants et des connecteurs, en assemblant une collection de contrôleurs.

La plate-forme doit permettre d'ajuster la complexité des configurations, selon les besoins, allant de configurations entièrement statiques et très efficaces à des configurations dynamiquement reconfigurables et moins performantes ; en proposant des éléments architecturaux génériques mais pouvant être paramétrés en choisissant l'équilibre performance/dynamicité.

5.3.1 Principales structures de données

L'objectif principal de la conception est de mettre en place une plate-forme pour programmer des objets de contrôle des éléments architecturaux, mettant ainsi à disposition du développeur une liste de contrôleurs Acme de façon à minimiser le surcoût en temps d'exécution des composants ainsi que l'impacte mémoire sur les applications.

Les contrôleurs représentent un ensemble d'interfaces qui reprennent les opérations d'introspection et d'intercession existantes au niveau de l'ADL, permettant ainsi soit d'accéder à certaines informations propres à l'élément architectural soit d'en modifier le contenu notamment les propriétés qui représentent l'aspect comportemental d'une architecture Acme. Les contrôleurs sont aussi dotés d'interfaces qui offrent différentes sémantiques de chargement et de liaison pour les composants tout comme les binders et les loaders (OpenCom) et les contrôleurs (Fractal).

Un composant/connecteur Acme est formé de plusieurs objets abstraits que l'on peut séparer en plusieurs groupes :

— Les objets qui implémentent le contenu. Ces objets peuvent être des sous-composants (dans le cas de composants composites) ou des objets primitifs (pour les compo-

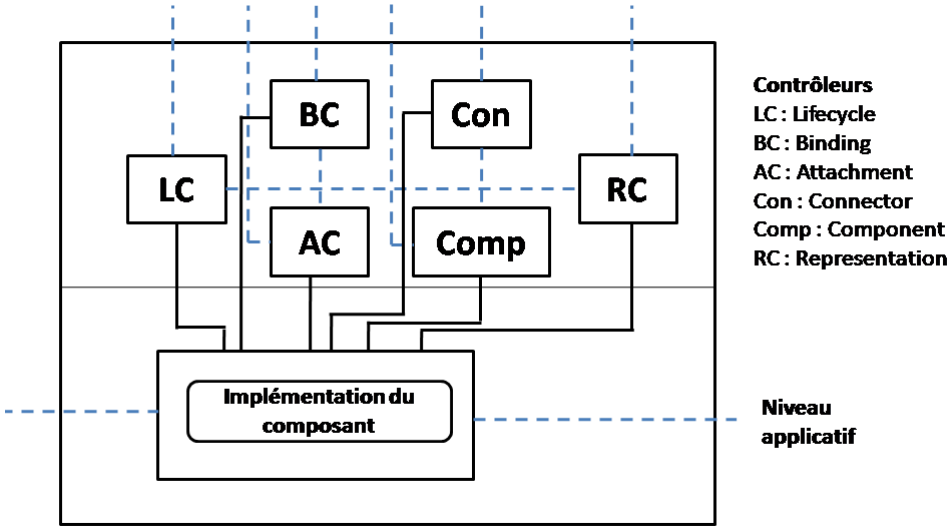

FIGURE 5.3 – Membrane de contrôle pour les composants primitifs

sants primitifs/connecteurs).

— Les objets qui implémentent la partie de contrôle. Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle et les intercepteurs optionnels qui interceptent les appels de méthodes entrants et sortants. Les fonctions de contrôle n'étant généralement pas indépendantes, les contrôleurs et les intercepteurs possèdent généralement des références les uns vers les autres.

— Les objets qui référencent les ports du composant. Ces objets sont le moyen pour un composant de posséder des références vers un autre composant (binding)/connecteur (Attachment).

— Les objets qui référencent les rôles du connecteur. Ces objets sont le moyen pour un connecteur de posséder des références vers un autre composant (attachement)

La membrane d'un composant est composée d'un ensemble de contrôleurs. Chaque contrôleur est dédié à une tâche précise : gestion des liaisons, du cycle de vie, etc. Ces contrôleurs collaborent entre eux afin de remplir la fonction qui leur est assignée. Ainsi, Le canevas logiciel utilise les concepts de composants, connecteurs, ports et rôles pour concevoir le niveau applicatif et le niveau de contrôle. Une membrane est un assemblage de contrôleurs et contenant un nombre quelconque de sous-composants ou de connecteurs. Chacun d'eux implémente une fonctionnalité de contrôle particulière.

La figure 5.3 illustre l'architecture d'une membrane de base. Cette membrane fournit six contrôleurs pour gérer le cycle de vie (LC), les liaisons (AC) (BC), les références vers la représentation du sous système (RC), les caractéristiques communes à tout composant (Comp) les caractéristiques communes à tout connecteur Acme (Con). La fonction de contrôle des composants est le résultat de la coopération des contrôleurs.

Contrôleur Lifecycle LC

Ce contrôleur gère le cycle de vie de l'élément auquel il appartient. Les deux états

possibles sont started et stopped. Les méthodes start() et stop() modifient les états.

Contrôleur Binding BC

Il gère les binding, association entre un port d'un composant et un port d'une représentation.

bind(port, port)

lookup(port) renvoi le port auquel le port de ce composant est lié.

unbind(port, port) supprime la liaison entre les deux ports.

Contrôleur Attachment AC

Ce contrôleur gère les attachements, interaction entre un port d'un composant et un rôle de connecteur.

attach(port, role)

lookupA(port) renvoi le rôle auquel le port de ce composant est lié.

unAttach(port, role) supprime l'interaction.

Contrôleur Représentation RC

Il maintient le lien entre un composant et la représentation de son sous-système.

getRepresenation(comp) renvoie la représentation qui contient le composant.

Contrôleur Component Comp

Ce contrôleur sert à l'introspection d'un composant.

getport(comp) renvoie une liste des ports du composant.

Contrôleur Connector Con

Il sert à l'introspection d'un connecteur.

getrole(con) renvoie une liste des rôles du connecteur.

5.3.2 Principe

Ces étapes expliquent comment programmer une application basée sur un composant Acme, La programmation ce fait en plusieurs étapes, de la création des types jusqu'au démarrage de l'application :

1. Création des types (composants, connecteurs)
2. Création des éléments architecturaux (composants, connecteurs)
3. Assemblage des composants (Binding, Attachment) selon les étapes :
   (a) Création des hiérarchies d'imbrication (Representation)
   (b) Création des liaisons
4. Démarrage de l'application.

5.4 Conclusion

Ce chapitre permet d'étendre le concept de contraintes d'intégrité aux architectures dynamiques. Nous avons précisé, dans un premier temps, la différence entre configuration

et reconfiguration, ainsi que la distinction entre opérations d'introspection et d'intercession dans le cadre des reconfigurations dynamiques dans l'optique de déterminer la nature des reconfigurations dynamiques dans l'ADL Acme/Armani.

Dans un second temps, nous avons spécifié la sémantique des opérations de reconfiguration comme des opérations de transformation du graphe de configuration avec un ensemble de pré conditions. Ces contraintes sont traduites dans le langage de spécification Armani et l'extension Plastik pour simuler les effets des opérations de reconfiguration sur le modèle d'une configuration ; et elles sont implémentées en OpenCOM pour pouvoir être vérifiées pendant l'exécution.

Le deuxième objectif du chapitre est de fournir un continuum complet de la configuration statique à la reconfiguration dynamique. Le but est de fournir une plate-forme d'exécution munie d'un ensemble d'objets de contrôle suffisamment riches et flexibles.

Chapitre 6

Conclusion et Perspectives

Ce chapitre vient clôturer la présentation de nos contributions. Nous rappelons brièvement dans la section 6.1 la problématique de cette thèse et présentons une synthèse de nos contributions pour la fiabilisation des reconfigurations dynamiques dans les architectures à composants. La section 6.2 propose les perspectives de nos travaux.

6.1 Synthèse et bilan des contributions

Rappel de la problématique.

En réponse au besoin croissant d'évolutivité des systèmes et en parallèle du besoin de fiabilité, le travail de recherche effectué dans cette thèse s'inscrit dans le cadre du domaine de l'évolution d'architecture logicielle. Il a comme objectif principal l'introduction d'une approche permettant d'assurer la fiabilité d'une architecture à base de composants tout au long de son évolution. En effet, Dans une application en cours d'exécution, et qui change donc d'état constamment, une simple anomalie dans une opération de reconfiguration peut par conséquent rendre le système incohérent et indisponible. Un système dont l'état est incohérent n'étant pas fiable, nous nous sommes donc attachés à garantir le maintien de la cohérence des systèmes reconfigurés. Voici une liste non exhaustive de certains points clés souvent abordés par les travaux liés à ce sujet [Ket 04] :

- Identifier les éléments de l'architecture sur lesquels les opérations de reconfiguration doivent opérer (selon le modèle de composant choisi).
- Exprimer les contraintes que doit satisfaire l'architecture pour pouvoir être reconfiguré dynamiquement.
- Définir les contrôleurs et les mécanismes nécessaires pour superviser le déroulement des opérations de reconfiguration, et intervenir en cas de problème.

Synthèse des contributions.

Pour fiabiliser les reconfigurations dynamiques, nous nous sommes plus particulièrement intéressés à l'ADL Acme, et qui semble être un ADL simple et générique. Notre démarche peut se résumer en deux étapes. Maintenir la cohérence des systèmes au cours

de leurs évolutions suppose d'avoir une définition précise de la cohérence des architectures à composants. La première étape a donc consisté en la définition de la cohérence des systèmes, et de leur architecture (ou configuration). La deuxième étape a été la définition de la cohérence de leurs évolutions (ou reconfigurations), afin de mettre en œuvre des mécanismes de reconfigurations fiables. A ces deux étapes correspondent nos principales contributions :

— Une modélisation des configurations et des reconfigurations pour définir la cohérence des systèmes. Cela ne peut être réalisé qu'en proposant différentes règles de configuration qui tiennent compte de l'évolutivité de l'architecture logicielle. Nous avons précisé la différence entre configuration et reconfiguration ainsi que la distinction entre opérations d'introspection et d'intercession afin d'étendre le concept de contraintes d'intégrité aux architectures dynamiques. Ces contraintes sont traduites dans le langage de spécification Armani et l'extension Plastik pour simuler les effets des opérations de reconfiguration sur le modèle d'une architecture.

— conception d'un canevas logiciel qui permet de fournir un continuum complet allant de la configuration statique jusqu'à la reconfiguration dynamique. Ce canevas logiciel est spécialement dédié à une plateforme d'exécution propre à Acme, et munie d'un ensemble d'objets de contrôle suffisamment riches et flexibles.

6.2 Perspectives

Le travail réalisé dans cette thèse a permis d'explorer quelques approches de reconfiguration dynamique. Beaucoup de travail reste à faire afin d'obtenir un système de reconfiguration convenable. Nos travaux pourraient bénéficier de quelques améliorations pour corriger certaines limitations ou pour élargir les perspectives d'utilisation des reconfigurations fiables.

La fiabilisation des reconfigurations dynamiques repose sur un ensemble de contraintes d'intégrité, avec un processus de vérification divisé en deux axes. Le premier axe s'intéresse à la prévention de fautes, cette technique de vérification au plus tôt repose sur l'analyse statique du code par l'outil Armani. Le deuxième axe quant à lui nécessite l'implémentation d'un modèle basé sur le canevas proposé, qui fournit une généralisation des éléments architecturaux et des relations entre ces éléments. L'implémentation doit bénéficier d'optimisations, notamment concernant la disponibilité du système, en synchronisant l'exécution fonctionnelle avec les reconfigurations ou en introduisant la notion de parallélisation entre opérations de reconfigurations.

Nous envisageons les perspectives suivantes :

— Enrichir les contraintes d'intégrité au niveau statique et dynamique.

— Renforcer la conception du canevas logiciel et résoudre les problèmes d'optimisation de l'exécution des reconfigurations.

Bibliographie

[.Ne] .NETSpecification. https://www.microsoft.com/net/

[Acm a] Acme Home page, http://acme.able.cs.cmu.edu/acmeweb/download.php

[Acm b] http://www.cs.cmu.edu/~acme/docs/language_overview.html

[All 94] Allen, Robert, and Dav id Garlan. "Formalizing architectural connection." Proceedings of the 16th international conference on Software engineering. IEEE Computer Society Press, 1994.

[All 97a] Allen, R.J. (1997). A Formal Approach to Software Architecture. Carnegie Mellon Univers ity, Pittsburgh, USA.

[All 97b] Allen, Robert, Remi Douence, a nd David Garlan. "Specifying dynamism in software architectures." (1997).

[All 98] Allen, Robert, Remi Douenc e, and David Garlan. "Specifying and analyzing dynamic software architectures." Fundamental Approaches to Software Engineering (1998): 21-37.

[ATM] http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

[Bas 03] Bass, Len, and Paul Clements. "Rick Kazman Software Architecture in Practice Chap. 4, 5 Boston." (2003).

[Bat 05] Batista, Thais, Ackbar Joolia, and Geoff Coulson. "Managing dynamic reconfiguration in component-based systems." European workshop on software architecture. Springer, Berlin, Heidelberg, 2005: 439-480.

[Bat 08] Batista, Thais, et al. "On the interplay of aspects and dynamic reconfiguration in a specification-to-deployment environment." European Conference on Software Architecture. Springer, Berlin, Heidelberg, 2008.

[Bau 06] Baudry, J. (2006) ATL Transformation METAH2ACME. INRIA, Rennes - Bretagne Atlantique
https://www.eclipse.org/atl/atlTransformations/metah2acme/metah2acme.pdf

[Bea] Sun Microsystems. The BeanBox.
http://www.oracle.com/technetwork/articles/javase/beancontext-136614.html

[Bel 98] Bellot, Patrick. Langage Jave. Ed. Techniques Ingénieur, 1998. H3088

[Bel 99] BELLOT, Patrick, and Christophe MATIACHOFF. "Applications distribuées en Java Java/RMI et IDL/CORBA." Techniques de l'ingénieur. InformatiqueH2760 (1999): H2760-1.

[Bin 96] Binns, P. Engelhart, M. Jackson, M. and Vestal, S, (1996) 'DomainSpecific Software Architectures for Guidance, Navigation, and Control', International Journal of Software Engineering and Knowledge Engineering , vol.6 No.2,

[Bla 04] Blair, Gordon S., Geoff Coulson, and Paul Grace. "Research directions in reflective middleware: the Lancaster experience." Proceedings of the 3rd workshop on Adaptive and reflective middleware. ACM, 2004.

[Bru 00] Bruneton, Eric, and Michel Riveill. JavaPod: une plate-forme à composants adaptable et extensible. Diss. INRIA, 2000.

[Caz 98] Cazzola, Walter, et al. "Architectural reflection: Bridging the gap between a running system and its architectural specification." In proceedings of 6th Reengineering Forum (REF'98. 1998.

[Caz 99] Cazzola, Walter, et al. "Architectural reflection: concepts, design and evaluation." Technical Report RI-DSI 234–99 (1999).

[Cla 01] Clarke, Michael, et al. "An efficient component model for the construction of adaptive middleware." IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2001.

[Cle 96] Clements, Paul C. "A survey of architecture description languages." Software Specification and Design, 1996., Proceedings of the 8th International Workshop on. IEEE, 1996.

[Cle] Clements, P. C. "Working Paper for the Constraints Sub-Group." EDCS Architecture and Generation Cluster (http://www. sei. cmu. edu/~ edcs/CLUSTERS/ARCH/index. html) (1997).

[Coi 06] Cointe, Pierre. "Les langages à objets. pub. vui, December 2006." Chapitre de l'Encyclopédie de l'informatique et des systèmes d'information.

[Com] Com specification. https://www.microsoft.com/com/default.mspx

[Cos 99] Costa, Fabio M., et al. "The role of reflective middleware in supporting the engineering of dynamic applications." Workshop on Reflection and Software Engineering. Springer, Berlin, Heidelberg, 1999.

[Cou 01] Councill, Bill, and George T. Heineman. "Definition of a software component and its elements." Component-based software engineering: putting the pieces together (2001): 5-19.

[Cou 02] Coulson, Geoff, et al. "The design of a configurable and reconfigurable middleware platform." Distributed Computing 15.2 (2002): 109-126.

[Cou 03] Coulson, Geoff, et al. "NETKIT: a software component-based approach to programmable networking." ACM SIGCOMM Computer Communication Review 33.5 (2003): 55-66.

[Cou 04] Coulson, Geoff, et al. "A component model for building systems software." (2004).

[Cou 07] Coupaye, T., et al. "Intergiciel et construction d'applications réparties." Chapitre Le système de composants Fractal (2007).

[Das 02] Dashofy, Eric M., André Van der Hoek, and Richard N. Taylor. "Towards architecture-based self-healing systems." Proceedings of the first workshop on Self-healing systems. ACM, 2002.

[Dij 82] Dijkstra, Edsger W. "On the role of scientific thought." Selected writings on computing: a personal perspective. Springer New York, 1982. 60-66.

[Dvo 00] Dvorak, Daniel, et al. "Software architecture themes in JPL's Mission Data System." Aerospace Conference Proceedings, 2000 IEEE. Vol. 7. IEEE, 2000.

[Dvo 02] Dvorak, Daniel. "Challenging encapsulation in the design of high-risk control systems." (2002).

[Ejb] Sun Microsystems. "Enterprise JavaBeans Technology". http://www.oracle.com/technetwork/java/index-jsp-140203.html

[Fra] Frtactal specification.http://fractal.ow2.org/specification/

[Gar 00] Garlan, David, Robert T. Monroe, and David Wile. "Acme: Architectural description of component-based systems." Foundations of component-based systems 68 (2000): 47-68.

[Gar 10] Garlan, David, Robert Monroe, and David Wile. "Acme: An architecture description interchange language." CASCON First Decade High Impact Papers. IBM Corp., 2010.

[Gar 95] Garlan, D. (1995) An introduction to the Aesop System. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps

[Gei 97] Geib, Jean-Marc, Christophe Gransart, and Philippe Merle. CORBA: des concepts à la pratique. Ed. Techniques Ingénieur, 1997.

[Gor 91] Gorlick, M,M. Razouk R,R. (1991), 'Using Weaves for Software Construction and Analysis', in ICSE13 1991. 13th International conference on Software Engineering. Austin, TX, USA ,pp. 23-34.

[Gor 94] Gorlick, M,M. Quilici, (1994), 'A Visual Programming in the Large versus Visual Programming in the Small', Proceedings of IEEE Symposium on Visual Languages. pp. 137-144.

[Gra 02] Grau, Andreas, Basem Shihada, and Mohamed Soliman. "Architectural Description Languages and their Role in Component Based Design." Project Report, Department of Computer Science, University of Waterloo, Canada, Available: http://www. cs. uwaterloo. ca/~ bshihada/adl. pdf (2002).

[Hus 13] Hussain, Sajjad. "Investigating architecture description languages (ADLs) a systematic literature review." (2013).

[Iee 00] IEEE, ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[J2e] Sun Microsystems. "Java 2 Platform, J2EE Management Specification". http://www.oracle.com/technetwork/java/javaee/management-139416.html

[Jai 16] Jain, Prateek. "Towards the adoption of modern software development approach: Component based software engineering." Indian Journal of Science and Technology 9.32 (2016).

[Jav] Sun MicroSystems. "Java Technology". http://www.oracle.com/technetwork/java/index.html

[Jbe] Sun Microsystems. JavaBeans 1.01 specification 2017. http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[Joo 05] Joolia, Ackbar, et al. "Mapping adl specifications to an efficient and reconfigurable runtime component platform." Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on. IEEE, 2005.

[Juk 04] H. Jukka "Architecture Description Languages ADL", 18.2.2004. http://www.cs.tut.fi/~kk/webstuff/ArchitectureDescriptionLanguages.pdf,

[Ket 04] Ketfi, Abdelmadjid. Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. Diss. Université Joseph-Fourier-Grenoble I, 2004.

[Kim 10] Kim, Jung Soo, and David Garlan. "Analyzing architectural styles." Journal of Systems and Software 83.7 (2010): 1216-1235.

[Kir 97] Kirtland, Mary. "The COM+ Programming Model Makes it Easy to Write Components in any Language-COM+ makes the COM programmingmodel more like the programming model of the language you use--you write classes in your." Microsoft Systems Journal-US Edition 12.12 (1997): 19-30.

[Kmi 12] Kmimech, Mourad. "Vérification d'assemblages de composants logiciels: Application aux modèles de composants UML2. 0 et Ugatze." (2012).

[Kuh 99] Kuhl, Frederick, Richard Weatherly, and Judith Dahmann. Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall PTR, 1999.

[Lég 09] Léger, M. and Cointe, P. (2009). Fiabilité des reconfigurations dynamiques dans les architectures à composants. École Nationale Supérieure des Mines de Paris, Paris, France.

[Ler 96] Lerusalimschy, Roberto, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. "Lua-an extensible extension language." Softw., Pract. Exper. 26.6 (1996): 635-652.

[Luc 95] Luckham D,C. Kenney J,J. Augustin L,M. Vera J. Bryan D. Mann W. (1995) 'Specification and Analysis of System Architecture Using Rapide', Software Engineering, IEEE Transactions, Vol. 21, No.4, p.336-355.

[Mae 87] Maes, Pattie. "Concepts and experiments in computational reflection." ACM Sigplan Notices. Vol. 22. No. 12. ACM, 1987.

[Mag 95] Magee, J. Dulay, N. Eisenbach, S. Kramer, J. (1995) ' Specifying Distributed Software Architectures', the 5th European Software Engineering Conference (ESEC'95), Spain. pp.137-153.

[Mag 96] Magee, Jeff, and Jeff Kramer. "Self organising software architectures." Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops. ACM, 1996.

[Mal 92] Malenfant, Jacques, Christophe Dony, and Pierre Cointe. "Behavioral Reflection in a prototype-based language." Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures. 1992.

[Mcl 68] McIlroy, M. D. "Software Engineering: Report on a conference sponsored by the NATO Science Committee." (1968): 138-155.

[Med 00] Medvidovic, Nenad, and Richard N. Taylor. "A classification and comparison framework for software architecture description languages." IEEE Transactions on software engineering 26.1 (2000): 70-93.

[Med 02] Medvidovic, N. Rosenblum, DS. Redmiles, DF. Robbins, JE. (2002) 'Modeling software architectures in the Unified Modeling Language.' ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 11 No.1, pp.2-57.

[Med 96] Medvidovic, Nenad. "ADLs and dynamic architecture changes." Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops. ACM, 1996.

[Mez 18] Mezghache, R. and Atil, F. (2018) 'A template for formalising reliable Acme-based software architecture', Int. J. Computer Applications in Technology, Vol. 57, No. 1, pp.14–27.

[Min 12] Minora, Leonardo, et al. "Issues of Architectural Description Languages for Handling Dynamic Reconfiguration." (2012).

[Mon 01] Monroe, R.(2001) Capturing Software Architecture Design Expertise With Armani. [online] Technical report, CMU-CS-98-163, Carnegie Mellon University School of Computer Science. http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-163R.pdf

[Mon 98] Monroe, Robert T. Capturing Software Architecture Design Expertise with Armani Version 1.0. No. CMU-CS-98-163. Carnegie-mellon univ pittsburgh pa school of computer Science, 1998.

[Mor 97] Moriconi, M and Riemenschneider,R,A. (1997) Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI International.

[Mor 95] Moriconi, M. Qian, X. and Riemenschneider, R,A.(1995) 'Correct Architecture Refinement', Software Engineering, IEEE Transactions, Vol. 21, No.4, pp. 356-372.

[Net] NetBeans. Ide. http://www.netbeans.org/

[Oli 98] Oliva, Alexandre, Islene Calciolari Garcia, and Luiz Eduardo Buzato. "The reflective architecture of guaraná." Proceeding of. 1998.

[Oqu 04] Oquendo, Flavio. "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures." ACM SIGSOFT Software Engineering Notes 29.3 (2004): 1-14.

[Ore 98a] Oreizy, Peyman, and Richard N. Taylor. "On the role of software architectures in runtime system reconfiguration." IEEE Proceedings-Software 145.5 (1998): 137-145.

[Ore 98b] Oreizy, Peyman. "Issues in modeling and analyzing dynamic software architectures." Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis. 1998.

[Ore 98c] Oreizy, Peyman, Nenad Medvidovic, and Richard N. Taylor. "Architecture-based runtime software evolution." Proceedings of the 20th international conference on Software engineering.

[Ore 99] Oreizy, Peyman, et al. "An architecture-based approach to self-adaptive software." IEEE Intelligent Systems and Their Applications 14.3 (1999): 54-62.

[Osg] OSGi Specification. https://www.osgi.org/developer/specifications/

[Pin 03] Pinto, Mónica, Lidia Fuentes, and Jose María Troya. "DAOP-ADL: an architecture description language for dynamic component and aspect-based development." International Conference on Generative Programming and Component Engineering. Springer, Berlin, Heidelberg, 2003.

[Riv 00] Riveill, Michel. Programmation par composants. Ed. Techniques Ingénieur, 2000. H2759

[Sca] SCA Specification. https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.1.0/com.ibm.cics.ts.applicationprogramming.doc/bundleinterface/sca.html

[Sch 13] Schmidt, Douglas C., et al. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Vol. 2. John Wiley & Sons, 2013.

[Sch 96] Schmidt, D., et al. "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns." (1996).

[Sha 06] Shaw, Mary, and Paul Clements. "The golden age of software architecture." IEEE software 23.2 (2006): 31-39.

[Sha 12] Shareef, Jawwad Wasat, and Rajesh Kumar Pandey. "Component-Based Software Development with Component Technologies: An Overview." Vol. 3 (1) , 2012

[Sha 95] Shaw, M. DeLine, R. Klein, D,V. Ross T,L. Young D,M. and Zelesnik G. (1995) ' Abstractions for Software Architecture and Tools to Support Them', Software Engineering, IEEE Transactions, Vol. 21, No.4, p.314-335.

[Sha 96] Shaw, Mary, and David Garlan. Software architecture: perspectives on an emerging discipline. Vol. 1. Englewood Cliffs: Prentice Hall, 1996.

[Sof] Sofa specification. http://sofa.ow2.org/

[Stu] http://www.cs.cmu.edu/~acme/AcmeStudio/index.html

[Sym 00] Symington, Susan, Katherine L. Morse, and Mikel Petty. "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Framework and Rules (IEEE Std 1516-2000)." (2000).

[Szy 98] Szyperski, Clemens. "Component Software: beyond object-oriented software." Reading, MA: ACM/Addison-Wesley (1998).

[Tay 96] Taylor, Richard N., et al. "A component-and message-based architectural style for GUI software." IEEE Transactions on Software Engineering 22.6 (1996): 390-406.

[Ter 95] Terry, A .London, R. Papanagopoulos, G and Devito,M. (1995) The ARDEC/Teknowledge Architecture Description Language (ArTek), Version 4.0,technical report, Teknowledge Federal Syst.,and U.S. Army Armament Research, Development, and Eng.

[Tra 93] Tracz, W. (1993), 'LILEANNA: A Parameterized Programming Languages,' in Proceedings of Advances in Software Reuse, Selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, pp. 24-26.

[Tra 99] Tramontana, Emiliano. "Managing evolution using cooperative designs and a reflective architecture." Workshop on Reflection and Software Engineering. Springer, Berlin, Heidelberg, 1999.

[Van 00] Van Deursen, Arie, Paul Klint, and Joost Visser. "Domain-specific languages: An annotated bibliography." ACM Sigplan Notices 35.6 (2000): 26-36.

[Ver 99] Vera, James, Louis Perrochon, and David C. Luckham. "Event-based execution architectures for dynamic software systems." Software Architecture. Springer US, 1999. 303-317.

[Wae 04] Waewsawangwong, Pakorn. "A constraint architectural description approach to self-organising component-based software systems." Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on. IEEE, 2004.

[Wan 01] Wang, Nanbor, Douglas C. Schmidt, and Carlos O'Ryan. "Overview of the CORBA component model." Component-Based Software Engineering. Addison-Wesley Longman Publishing Co., Inc., 2001.

[Wil 01] Wile, David. "Using dynamic acme." Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia. 2001.

[Woo 05] Woods, Eoin, and Rich Hilliard. "Architecture description languages in practice session report." Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on. IEEE, 2005.