Fiabilité des reconfigurations dynamiques dans les architectures à composants

Marc Léger

ED n°82 : Sciences et Technologies de l'Information et de la Communication

N°attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

T H E S E

pour obtenir le grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS

Spécialité "Informatique temps réel, robotique et automatique"

présentée et soutenue publiquement par
Marc LEGER

le 19 mai 2009

Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants

Directeur de thèse : Pierre COINTE

Jury

Isabelle DEMEURE	Institut TELECOM (Telecom ParisTech)	Rapporteur
Jean-Bernard STEFANI	INRIA Grenoble - Rhône-Alpes	Rapporteur
Pierre COINTE	Ecole des Mines de Nantes	Examinateur
Thierry COUPAYE	France Telecom Orange Labs	Examinateur
Jean-Marc JEZEQUEL	Université de Rennes 1	Examinateur
Thomas LEDOUX	INRIA Rennes-Bretagne Atlantique	Examinateur
Patrick VALDURIEZ	INRIA Rennes-Bretagne Atlantique	Examinateur

Remerciements

Résumé

L'ingénierie logicielle doit faire face à un besoin toujours croissant en évolutivité des systèmes informatiques pour faciliter leur maintenance et de manière générale leur administration. Cependant, l'évolution d'un système, et plus spécifiquement l'évolution dynamique, ne doit pas se faire au dépend de sa fiabilité, c'est à dire de sa capacité à délivrer correctement les fonctionnalités attendues au cours de sa vie. En effet, des modifications dans un système peuvent laisser ce dernier dans un état incohérent et donc remettre en cause son caractère fiable. L'objectif de cette thèse est de garantir la fiabilité des reconfigurations dynamiques utilisées pour faire évoluer les systèmes pendant leur exécution tout en préservant leur disponibilité, c'est à dire leur continuité de service. Nous nous intéressons plus particulièrement aux systèmes à base de composants logiciels, potentiellement distribués, dont l'architecture peut être le support de reconfigurations dynamiques non anticipées et concurrentes. Nous proposons une définition de la cohérence des configurations et des reconfigurations dans le modèle de composants Fractal à travers une modélisation basée sur des contraintes d'intégrité tels que des invariants structurels. La fiabilité des reconfigurations est garantie par une approche transactionnelle permettant à la fois de réaliser du recouvrement d'erreurs et de gérer la concurrence des reconfigurations dans les applications. Nous proposons enfin une architecture à composants modulaire pour mettre en oeuvre nos mécanismes transactionnels adaptés aux reconfigurations dynamiques dans des applications à base de composants Fractal.

Mots-clés : architectures à composants, reconfigurations dynamiques, gestion de concurrence, fiabilité, tolérance aux fautes, transactions, contraintes d'intégrité, administration autonome, modèle de composants Fractal

Abstract

Software engineering must cope with a more and more increasing need for evolutivity of software systems in order to make their maintenance and more generally their administration easier. However, evolution and especially dynamic evolution in a system must not be done at the expense of its reliability, that is to say its ability to deliver the expected functionalities. Actually modifications in a given system may let it in an inconsistent state and so it can have an impact on its reliability. The aim of this thesis is to guarantee reliability of dynamic reconfigurations used to make systems evolve at runtime while preserving their availibility, i.e. their continuity of service. We are especially interested in component based and distributed systems. The system architecture can be used as a support for dynamic, non-anticipated (also called ad-hoc) and concurrent reconfigurations. We propose a definition of consistency for configurations and reconfigurations in the Fractal component model with a model based on integrity constraints like for example structural invariants. Reliability of reconfigurations is ensured thanks to a transactional approach which allows both to deal with error recovery and to manage reconfiguration concurrency in systems. Finally we propose a modular component-based architecture so as to implement transactional mechanisms adapted to dynamic reconfigurations in Fractal applications.

Keywords : component-based architectures, dynamic reconfigurations, concurrrency management, reliability, fault tolerance, transactions, integrity constraints, autonomic computing, Fractal component model

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Sommaire

Ce chapitre d'introduction propose une analyse du sujet et de la problématique de cette thèse (section 1.1). La section 1.2 recense les principales contributions de nos travaux dans le cadre de cette problématique et nous terminons dans la section 1.3 par une brève présentation de l'organisation de ce document.

1.1 Problématique

Nous nous attachons dans cette section à présenter le contexte et les motivations du problème de fiabilité des reconfigurations dynamiques dans les architectures à composants et poursuivons par la description de la problématique de la thèse.

1.1.1 Contexte et motivation

Les logiciels et les systèmes informatiques en général croissent en nombre et en complexité dans des domaines d'application aussi divers que les serveurs d'entreprise ou les systèmes embarquées et contraints. Cette complexité croissante est caractérisée par l'augmentation de la taille des logiciels (un serveur d'application comme JOnAS [JOn] comporte plusieurs millions de lignes de codes), l'hétérogénéité des plateformes à tout niveau (systèmes d'exploitation, intergiciels et applications) et la multiplication des normes et des standards : Java EE [Jav] dans le domaine des serveurs d'applications regroupe ainsi de nombreuses technologies Java (JMS, JCA, etc.). L'administration de ces systèmes est également de plus en plus complexe et nécessite de leur part des capacités d'évolution pour des opérations de maintenance, des besoins de mise à jour ou d'adaptation en fonction de leur contexte d'exécution.

L'évolutivité [LP76] est une propriété essentielle qui caractérise la capacité d'un système à être modifié au cours de son cycle de vie. Une évolution (ou modification) peut concerner la correction d'erreurs de conception (« bugs ») dans le système, l'ajout de nouvelles fonctionnalités, ou encore l'optimisation de fonctionnalités existantes. La maintenance corrective [Lap92b] est ainsi un bon exemple d'évolution particulière d'un système consistant à le remettre dans un état spécifié. Deux types d'évolutions sont à distinguer suivant l'état du cycle de vie du système considéré : les évolutions statiques qui interviennent quand le système est arrêté sous forme par exemple de « patchs » de code qui nécessitent la recompilation et le redémarrage du système, et les évolutions dynamiques réalisées dans un système en train de s'exécuter, nous parlerons plus spécifiquement de reconfigurations dynamiques dans ce dernier cas.

Les reconfigurations dynamiques [KM90, MK96] sont un moyen de faire évoluer les systèmes tout en préservant leur disponibilité. Ce sont en effet des modifications effectuées pendant l'exécution d'un

système qui ne nécessitent pas d'arrêt complet ou de recompilation du système modifié et permettent donc de préserver sa continuité de services. La capacité d'évolution est au coeur de la problématique des systèmes adaptatifs et de l'informatique autonomique. Les systèmes adaptatifs [KBC02] évoluent automatiquement en fonction du contexte d'exécution : le changement de contexte entraîne typiquement une reconfiguration dynamique du système pour réaliser l'adaptation. L'informatique autonomique [KC03] est une réponse à la complexité croissante de l'administration des systèmes logiciels en les dotant de la capacité de s'auto-administrer par l'intermédiaire d'une boucle de régulation similaire aux boucles de contrôle dans le domaine de l'automatique. La réaction à l'intérieur de la boucle repose sur la reconfiguration dynamique du système auto-administré.

Les reconfigurations dynamiques peuvent être basées sur des modifications d'architecture [ADG98, OMT98] des systèmes. Le composant [Szy02] est un paradigme de programmation en génie logiciel au même titre que l'objet pour construire des systèmes logiciels. Un modèle de composant définit des standards de construction, de composition et d'interaction entre des briques logicielles appelées composants. Les systèmes à base de composants offrent de bonnes propriétés pour être reconfigurés dynamiquement comme l'encapsulation forte (boîte noires ou grises), la modularité, un couplage lâche entre composants, et la séparation des préoccupations [HL95]. En effet, une séparation claire entre le code fonctionnel et le code non-fonctionnel ainsi que la représentation explicite des dépendances entre composants permettent d'utiliser le composant comme granularité des reconfigurations en isolant leurs effets sur le système reconfiguré.

Parallèlement au besoin d'évolutivité des systèmes se pose le problème de leur fiabilité, plus particulièrement suite à leurs évolutions. L'administration au sens de l'observation et de la modification des systèmes est en effet, de part sa complexité, sujette à des erreurs qui peuvent en les modifiant rendre les systèmes non fiables [Ore98]. La fiabilité [lap92a] est un attribut de la sûreté de fonctionnement qui est la mesure de la délivrance d'un service correct pour un système donné. Autrement dit, un système n'est fiable que s'il est dans un état correct (ou cohérent) par rapport à une spécification. Dans le cas des architectures à composants, une spécification à considérer est celle du modèle de composants sous-jacent. Une condition nécessaire à la fiabilité de ce type de système est donc sa conformité au modèle de composants. Une reconfiguration dynamique doit être une transformation valide de l'état d'un système pour qu'il reste cohérent. Une reconfiguration sera ainsi invalide (i.e. mènera à un système non fiable) si elle viole la spécification du modèle de composants considéré.

Plus généralement, toute faute est susceptible d'entrainer une défaillance qui va venir perturber le fonctionnement d'un système et donc réduire sa fiabilité [Lap92b]. Les fautes considérées sont plus particulièrement les fautes logicielles internes aux applications (bugs, exceptions, etc.) et les fautes dues au système ou au matériel sous-jacent. Les reconfigurations dynamiques peuvent ainsi être exécutées en présence de fautes : une panne franche de machine sur laquelle le système est localisé, entièrement dans un contexte centralisé ou en partie dans un contexte réparti, peut interrompre l'exécution d'une reconfiguration. D'autre part, une reconfiguration, si elle est une transformation invalide, génère une faute à l'origine de l'incohérence de l'état du système reconfiguré comme c'est le cas par exemple avec la violation du modèle de composants. Une reconfiguration doit donc être exécutée malgré des fautes et ne pas créer de nouvelles fautes. Pour faire face aux différentes fautes que peut rencontrer un système au cours de son existence, il existe essentiellement deux méthodes différentes : la prévention de fautes et la tolérance au fautes. La prévention de fautes consiste à éliminer les fautes au plus tôt avant leur occurrence pour éviter qu'elles se produisent pendant l'exécution du système, elle repose essentiellement sur des techniques d'analyse statique. La tolérance aux fautes a pour objectif de laisser les fautes se produire pour les réparer de manière transparente.

1.1.2 Enoncé et analyse de la problématique

Les systèmes informatiques se doivent d'être évolutifs pour répondre à des besoins de maintenance voire d'adaptation tout en restant fiables pour assurer le bon fonctionnement des services délivrés. Les reconfigurations dynamiques sont utilisées pour faire évoluer dynamiquement les systèmes tout en préservant leur disponibilité. Ces reconfigurations peuvent reposer sur l'architecture des systèmes à base de composants. La problématique posée par cette thèse est donc la suivante : comment fiabiliser les reconfigurations dynamiques dans les systèmes à base de composants logiciels ? Nous spécifions un certain nombre d'exigences sur les reconfigurations dynamiques et les systèmes reconfigurés.

Cohérence. Un système est fiable s'il reste conforme à une spécification, nous parlerons alors de système cohérent. Le problème de fiabilité revient donc au problème de maintien de la cohérence d'un système. Une reconfiguration pour être valide doit être une transformation d'un système d'un état cohérent vers un autre état cohérent. Notre objectif est de donner une définition de la cohérence pour des systèmes à base de composants afin de faciliter la vérification de cette cohérence et de pouvoir déterminer la validité des reconfigurations dynamiques. Assurer la fiabilité d'un système et de ses évolutions nécessite donc la garantie de la cohérence de ce système et de la validité des reconfigurations.

Transparence. La propriété de fiabilité doit être prise en compte de manière la plus transparente possible dans les systèmes reconfigurés à la fois du point de vue du développeur et de l'utilisateur du système. Pour le développeur, les mécanismes mis en oeuvre doivent être intégrés de manière simple et non intrusive dans le code des applications et spécifiés suivant le principe de séparation des préoccupations notamment par rapport au code fonctionnel. D'autre part, la gestion des erreurs de reconfigurations lié au maintien de la cohérence du système doit être transparente pour l'utilisateur.

Disponibilité. Les mécanismes de fiabilisation des reconfigurations dynamiques doivent minimiser la perturbation de l'exécution du système reconfiguré et contribuer à augmenter sa disponibilité. Une reconfiguration dynamique ne nécessite pas l'arrêt complet d'un système pour le modifier. De même, les mécanismes pour rendre fiables ces reconfigurations doivent impacter un minimum cette disponibilité en cas de reconfiguration valide et contribuer à l'améliorer en cas de reconfiguration invalide en maintenant un état cohérent du système. Le facteur de diminution de la disponibilité pour les reconfigurations valides est donc lié à la problématique de performance et plus précisément au surcoût en efficacité (vitesse d'exécution) des mécanismes de fiabilisation des reconfigurations.

Ouverture. Un système ouvert se caractérise par ses capacités d'évolution et d'extension qui ne sont pas nécessairement prévues explicitement par le concepteur du système. Il existe une tension [CND+04] entre la forme pour garantir la cohérence d'un système et donc sa fiabilité et son ouverture pour supporter l'évolutivité. Les évolutions sont donc plus ou moins fortement contraintes, c'est le cas des reconfigurations dynamiques dans les architectures à composants qui doivent respecter les règles du modèle de composants utilisées pour développer l'application. Pour faciliter la vérification de la fiabilité des évolutions dans un système donné, il est par exemple possible de limiter ces évolutions à un nombre limité d'alternatives déterminées explicitement. Nous nous intéresserons plus particulièrement aux systèmes ouverts et aux reconfigurations non anticipées, i.e. des reconfigurations non prévues au moment de la conception du système en dehors des contraintes du modèle de composants.

Réflexivité. Un système réflexif [Mae87, SF96] offre à l'exécution une représentation de lui-même qui est accessible (par introspection) et modifiable (par intercession). Cette représentation est causalement connecté avec le système lui-même. Un système réflexif est d'autant plus ouvert et évolutif si il offre des capacités d'intercession. Les modèle de composants réflexifs supportent la réflexion architectural [CSST98] et facilitent ainsi les reconfigurations dynamiques des applications en exposant leur architecture à travers des APIs. Ils permettent également de vérifier de bonnes propriétés sur la représentation réifiée des systèmes qui seront aussi valables sur le système de base.

Distribution. Les systèmes informatiques sont de moins en moins centralisés sur un unique hôte déterminé mais tendent à être distribués sur plusieurs machines reliées par différents types de réseaux. Les raisons de cette distribution sont multiples : augmentation de performance par la répartition de charge dans les clusters de serveurs, augmentation de la disponibilité dans ces mêmes clusters, croissance de la mobilité et multiplication des appareils et logiciels embarqués, etc. Pour les systèmes à base de composants, cette tendance se traduit par la répartition des composants du système sur des machines différentes et doit être prise en compte par les mécanismes de fiabilisation des reconfigurations dynamiques. La distribution ajoute en effet des points vulnérabilité en terme de fiabilité à travers notamment l'utilisation du réseau et des communications réparties. Par ailleurs, non seulement les systèmes sont distribués mais aussi l'administration. Un système doit donc être reconfigurable depuis n'importe quelle machine sur laquelle il est distribué.

Concurrence. Une reconfiguration peut être soit le fait d'un administrateur humain soit le fait de l'application elle-même dans le cas par exemple de l'informatique autonomique. La concurrence peut concerner des reconfigurations initiées par des acteurs différents ou une même reconfiguration séparée en plusieurs sous reconfigurations parallèles pour des raisons d'optimisation. Des reconfigurations exécutées concurremment sont susceptibles d'entrer en conflits si leurs effets concernent la même partie du système. Ces conflits entre reconfigurations, plus ou moins nombreux suivant le niveau de concurrence, doivent être évités pour ne pas perturber le bon fonctionnement du système. De même, les reconfigurations dynamiques sont réalisées pendant l'exécution du système et il est donc nécessaire de les synchroniser avec l'exécution fonctionnelle du système pour éviter les états incohérents au cours des reconfigurations.

La problématique revue et reformulée concerne donc **la fiabilisation des reconfigurations dynamiques, concurrentes et non anticipées dans des systèmes distribués à base de composants logiciels**.

1.2 Présentation des contributions

La démarche suivie pour répondre à la problématique de fiabilisation des reconfigurations a consisté dans un premier temps à définir précisément les concept de configuration architecturale et de reconfiguration dynamique dans un modèle de composants. Nous nous sommes focalisés dans le cadre de nos travaux plus particulièrement sur le modèle Fractal [BCL+06] pour ses bonnes propriétés en terme de reconfigurabilité : le modèle est réflexif et particulièrement dynamique. La suite de la démarche a donné lieu à une réflexion sur le problème de fiabilité dans le contexte des architectures à composants. Il s'est agi pour cela de définir ce qu'est la cohérence d'une configuration et d'une reconfiguration, cohérence qui doit être maintenu dans le système pour garantir sa fiabilité. La question suivante qui s'est posée a été de déterminer quels sont les moyens possibles de garantir cette cohérence, nous nous sommes pour cela orientés vers l'utilisation de transactions en définissant le concept de reconfiguration transactionnelle. Les mécanismes de reconfigurations transactionnelles ont ensuite été mis en oeuvre sous forme d'une architecture modulaire à base de composants Fractal. Ils ont été expérimentés dans différents scénarios applicatifs, notamment autour de l'informatique autonomique.

Nos travaux ont donné lieu à trois contributions principales en réponse à la problématique de fiabilisation des reconfigurations dynamiques : la définition d'un modèle de cohérence pour les configurations de composants Fractal à l'aide de contraintes d'intégrité ainsi qu'une extension de ce modèle pour intégrer les reconfigurations dynamiques, un modèle de transactions pour le support de la fiabilité des reconfigurations dynamiques, concurrentes, réparties et non anticipées, et enfin une architecture modulaire de gestion des reconfigurations transactionnelles dans les applications à base de composants Fractal.

Un modèle pour définir la cohérence des configurations et des reconfigurations dynamiques à base de contraintes d'intégrité. La garantie de fiabilité des reconfigurations dynamiques dans un système revient à maintenir la cohérence de ce système au cours de ses évolutions. Fiabiliser les reconfigurations dans les systèmes à base de composants nécessite donc dans un premier temps de donner une définition de la cohérence dans ces systèmes. Nous proposons un modèle extensible de configurations architecturales de composants Fractal basé le concept de contraintes d'intégrité par analogie avec les bases de données. Une configuration est ainsi vue comme un ensemble d'éléments du modèle de composants (composants, interfaces, etc.), de relations entre ces éléments (liaisons, relation de hiérarchie, etc.) et de contraintes sous forme d'invariants essentiellement structurels, et marginalement comportementaux (par exemple liés au cycle de vie des composants). La cohérence d'un système est ensuite définie comme la satisfaction d'un ensemble de contraintes d'intégrité. La cohérence de l'ensemble des contraintes recensées sur le modèle Fractal est vérifiée à l'aide d'un langage de spécification et de son analyseur (Alloy [Jac02]). Les contraintes sont implémentées dans un langage dédié de navigation et de requêtes sur les configurations Fractal (FPath [DLLC09]) afin de pouvoir les vérifier sur les applications Fractal pendant leur exécution. Une extension du modèle de configurations Fractal est également proposée en utilisant les mêmes formalisme et méthodologie pour prendre en compte la dynamicité des architectures et les reconfigurations dynamiques dans le

modèle de composants. Une sémantique extensible des opérations de reconfiguration est spécifiée et implémentée sous forme de préconditions et postconditions en Alloy et en FPath.

Un modèle de transactions pour le support de la fiabilité des reconfigurations dynamiques, concurrentes, réparties et non anticipées. Pour garantir le maintien de la cohérence dans les systèmes à base de composants au cours de leurs évolutions, nous proposons un modèle de transactions (structure, propagation, etc.) adapté aux reconfigurations dynamiques dans le modèle de composants Fractal. Les transactions [GR92] sont un moyen bien connu de rendre les systèmes tolérants à différents types de fautes logicielles et matérielles et donc d'améliorer leur fiabilité. Nous déclinons les différentes propriétés classiques des transactions, les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité), dans le contexte particulier des reconfigurations définies comme des séquences d'opérations primitives d'introspection et d'intercession dans les systèmes à base de composants. Notre approche à base de reconfigurations transactionnelles permet de réaliser du recouvrement de fautes, dont les violations de contraintes d'intégrité et les pannes franches, par des techniques de reprises sur défaillance. Le support de la concurrence entre reconfigurations est également assuré par les mécanismes transactionnels. Nous contribuons également à un mécanisme d'isolation entre l'exécution fonctionnelle des applications et leur reconfiguration.

Une architecture modulaire pour gérer les reconfigurations dynamiques et transactionnelles dans les applications à base de composants Fractal. Le modèle de reconfiguration transactionnelle est mise en oeuvre dans le modèle Fractal et son implémentation de référence en Java, Julia [BCL+04]. Les mécanismes de reconfigurations transactionnelles se présentent principalement sous la forme d'un ensemble de composants constituant un gestionnaire de transactions. L'approche choisie est une approche sous forme de canevas logiciel (« à la framework ») qui a déjà fait ses preuves avec le modèle Fractal par exemple dans le domaine des intergiciels orientés messages [Que05]. L'architecture du gestionnaire est modulaire et extensible de façon à pouvoir intégrer ou non les différentes fonctionnalités des transactions que sont par exemple le support de la concurrence ou encore la persistance de l'état des composants. Nous proposons également une extension du modèle Fractal à travers la fourniture de plusieurs nouveaux contrôleurs dédiés à la gestion des reconfigurations transactionnelles ainsi que des extensions du langage de définition d'architectures Fractal ADL pour la spécification des différentes propriétés des transactions dont par exemple la spécification des contraintes d'intégrité au niveau de l'architecture des composants.

1.3 Organisation du document

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

Le présent chapitre (chapitre 1) sert d'introduction au document en présentant le contexte et motivations du sujet de thèse, la problématique et son analyse, 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 reposant 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 trois principales contributions sont regroupées dans quatre chapitres différents. La première contribution concerne la modélisation des configurations et des reconfigurations dans le modèle Fractal qui constituent respectivement les chapitres 4 et 5 du document. Notre deuxième contribution qui propose un modèle de transactions pour les reconfigurations dynamiques est détaillée dans le chapitre 6. La dernière contribution axée sur la proposition d'une architecture modulaire des mécanismes de reconfiguration transactionnelle est présentée dans le chapitre 7. Ces contributions sont évaluées sous forme d'expérimentations dans le chapitre 8 de ce document.

Enfin, le chapitre 9 vient conclure cette thèse par une synthèse des contributions et une mise en perspective de nos travaux par rapport à la problématique initiale.

Première partie

Etat de l'art

Chapitre 2

Technologies et concepts associés au sujet

Sommaire

Nous présentons brièvement dans ce chapitre un certain nombre de concepts et de technologies liés à nos travaux. La programmation par composants (section 2.1) est au coeur des mécanismes de reconfigurations dynamiques, les reconfigurations reposent en effet par hypothèse sur les architectures à composants des systèmes considérés et sur leur dynamicité (section 2.2). Enfin, la fiabilité en tant qu'attribut de la sûreté de fonctionnement (section refsection :sûreté) est une propriété essentielle des systèmes logiciels qui est potentiellement remise en cause lors de la mise en oeuvre de reconfigurations dynamiques dans un système donné.

2.1 Programmation par Composants

Le composant est au même titre que l'objet un paradigme de programmation pour la conception et l'implémentation de systèmes logiciels. Un composant est conforme à un modèle (section 2.1.1) qui spécifie des règles de construction des architectures logicielles. Les langages de description d'architecture (section 2.1.2) sont utilisés pour la spécification de l'assemblage des composants dans les architectures logicielles tandis que les styles architecturaux permettent de contraindre ces architectures.

2.1.1 Modèles de composant

De l'objet vers le composant. Le génie logiciel vise à la définition et l'application de méthodes et l'utilisation d'outils pour la production du logiciel et de son suivi. Les techniques de programmation évoluent pour tenir compte de la complexité et de la taille croissante des systèmes logiciels développés et des nouvelles exigences en termes de qualité et de fiabilité. La programmation par objets qui a émergé depuis une vingtaine d'année [Coi06] s'est ainsi imposée comme une évolution majeure par rapport à la programmation procédurale pour répondre à ces exigences avec des langages

comme Smalltalk [GR83], C++ [Str00] puis Java [GJSB05]. Le paradigme objet apporte en effet de bonnes propriétés telles qu'un niveau d'abstraction plus élevé (concepts de classes et d'instances), une meilleure réutilisabilité du code et flexibilité de conception (notion de canevas logiciels avec l'héritage et le polymorphisme), l'encapsulation des données et du comportement avec la possibilité de séparer interfaces et implémentation. 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 notamment nécessaire à leur maintenance. Le concept d'architecture logicielle est ainsi défini comme :

> *« L'organisation fondamentale d'un système par ses composants, leurs relations avec l'environnement, et les principes qui gouvernent sa conception et son évolution »* [MEH04]

La prise en compte des limitations de l'approche objet pour la conception et l'implémentation des architectures logicielles et le besoin d'une structuration à granularité variable, éventuellement à grosse granularité suivant le principe du « programming in the large »[DK75], ont mené à la création d'un nouveau paradigme de programmation : le composant.

Modèles de composant. Le concept de *composant logiciel* est relativement ancien [McI68]. Il est décrit comme une brique logicielle composable et facilement réutilisable, d'où l'appellation de composant sur l'étagère (*Composant Off The Shell* ou COTS). Il n'existe pas de réel consensus pour définir ce qu'est la description d'une architecture à composants. Pour Szyperski [Szy02], un composant est ainsi « 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 » [1]. Pour Heineman et Councill [HC01], un composant est un élément logiciel conforme à un modèle appelé modèle de composants qui définit des standards de composition et d'interactions. Un modèle de composant est habituellement indépendant de tout langage d'implémentation. L'implémentation d'un modèle fournit un environnement dédié au support de l'exécution des composants conformes au modèle. L'apport des composants par rapport à l'objet est notamment une plus grande modularité et facilité de réutilisation grâce à une encapsulation plus forte (notion de boîte noire) et un couplage plus faible entre entités logicielles avec une représentation explicite des dépendances requises et fournies. La séparation des préoccupations [Dij82] entre architecture logicielle et implémentation est un point fort de la programmation par composants pour mieux structurer les logiciels sous forme d'assemblage de briques logicielles. Le cycle de vie du logiciel peut ainsi être plus clairement décomposé en des phases de conception de l'architecture, de développement du code métier des composants, du déploiement de ces composants et enfin de leur exécution.

Il existe de nombreux modèles de composants pour différents domaines d'application. Des modèles industriels (EJB [EJB], CCM [WSO01], 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'architecture. Ce sont en effet des modèles structurellement plats et proches de l'implémentation. La séparation entre les préoccupations fonctionnelles et extra-fonctionnelles dans les conteneurs repose sur le patron d'architecture d'inversion de contrôle (*Inversion of Control* ou IoC) et sur des techniques d'injection de dépendances dans le code d'implémentation. Le développeur peut ainsi se consacrer à la logique « métier » plutôt qu'au code « technique ». Certains modèles sont uniquement utilisés pour la conception d'architecture sans support d'exécution (composants UML 2.0). Enfin, d'autres modèles tels SCA [SCA] ou encore OSGi [OSG] et les composants « declarative services » servent à implémenter des architectures orientées services (*Service Oriented Architectures* ou SOA). Les modèles plus académiques ou de recherche s'intéressent à des concepts plus avancés notamment concernant l'analyse des architectures en termes structurels ou de comportement. Ces modèles sont généralement associé à un langage dédié (*Domain Specific Language* ou DSL [vDKV00]) pour la description des architectures des systèmes : les langages de description d'architecture (*Architecture Description Languages* ou ADLs). D'autres modèles reposent sur l'extension d'un langage généraliste de programmation pour y introduire le concept de composant comme entité de première classe (e.g. ArchJava [ACN02] avec Java), la description de l'architecture et l'implémentation des composants sont alors mêlées et spécifiées dans le même langage.

1. « A component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »

Eléments d'architecture. La majorité des modèles de composants académiques comportent, au moins en ce qui concerne l'aspect structurel, des concepts communs ou proches : les composants, les connecteurs et les configurations. Le langage Acme [GMW00] a été développé pour standardiser un format commun d'échange entre les ADLs existants. Les éléments d'architectures sont définis par ce langage d'une manière génériques qui correspond à la définition reprise 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 composé et se décompose en deux parties. L'extérieur qui correspond à son interface comprend la description des interfaces fournies et requises par le composant et définit les interactions du composant avec son environnement. La seconde partie correspondant à son implémentation permet la description du fonctionnement interne du composant.

Un *connecteur* correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions. Par exemple, un connecteur 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, ou encore la diffusion d'événements asynchrones. Un connecteur comprend également deux parties. La première correspond à son interface, qui permet la description des rôles des participants à une interaction. La seconde partie correspond à la description de son implémentation. Il s'agit là de la définition du protocole associé à l'interaction. Le connecteur n'est pas nécessairement considéré comme une entité de première classe dans tous les modèles où il peut être un simple composant particulier dédié à l'implémentation d'un protocole de communication.

Une *configuration* définit la structure et le comportement d'une application formée de composants et de connecteurs. Une composition de composants est une configuration. La configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs qui constituent l'application. La 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.

2.1.2 Langages de description d'architecture et styles architecturaux

Les langages de description d'architecture. La prise en compte des concepts architecturaux au niveau des langages de programmation permet une traçabilité tout au long de la phase de développement et du déploiement des composants. Les langages de description d'architecture sont des langages généralement déclaratifs dans lesquels le couplage entre spécification architecturale et implémentation est faible. Ces langages dédiés permettent de spécifier des configurations architecturales comme des assemblages de composants et éventuellement de connecteurs. Ils peuvent être classés en deux grandes familles [MT00]. La famille des langages de description à laquelle appartiennent Wright [All97] et Rapide [LKA+95] privilégient la description des éléments de l'architecture et leur assemblage structurel. Ils ont pour but la formalisation, la vérification et la validation d'architecture. Elle est accompagnée d'outils de modélisation et d'analyseur syntaxique. Certains ADL comme Wright (cf. Figure 2.1) permettent de spécifier du comportement dans les définitions de composants : à chaque port est associée une description formelle dans l'algèbre de processus CSP (*Communicating Sequential Processes*) spécifiant ses interactions avec l'environnement tandis que la partie calcul décrit le comportement interne du composant. La famille de langages de configuration à laquelle appartient Darwin [MK96] se focalisent sur la description de la configuration d'une architecture et sur la dynamique du système. Ils sont accompagnés d'outils de modélisation et de génération de code mais aussi d'une plate-forme d'exécution ou de simulation d'un système, voire de modification dynamique pendant l'exécution.

```
Component Double
    Port In = read?x →In □ close→§
    Port Out = write!x →Out ⊓ close→§
    Computation = (In.read?x →Out.write!(2*x) →Computation)
                  □ (In.close→Out.close→§)
```

FIGURE 2.1 – Exemple de composant spécifié en Wright

Contraintes et styles architecturaux. Un contrainte est une propriété ou une assertion sur un système ou une de ses parties, sa violation met le système dans un état inacceptable ou dégradé par rapport à sa spécification et à ses fonctionnalités attendues. Les contraintes, exprimées dans une logique quelconque, permettent de définir une sémantique d'un programme. Cette définition est à rapprocher de la programmation par contrat [Mey92] qui vise à spécifier des règles, des assertions formant un contrat qui précise les responsabilités entre un client et le fournisseur du code logiciel. Les trois types d'assertions possibles sont :

- Les invariants caractérisent une condition qui doit être toujours vraie.
- Les préconditions spécifient des conditions qui doivent être garanties par le fournisseur avant un traitement quelconque (e.g. invocation d'opération)
- Les postconditions spécifient des conditions qui doivent être garanties par le fournisseur après l'exécution d'un traitement quelconque.

Les styles architecturaux sont un moyen de contraindre les architectures logicielles structurellement ou dans leur comportement. Un style architectural [AAG93, AAG95] caractérise une famille de systèmes partageant des propriétés structurelles et des éléments de sémantique. Plus précisément, un style définit un ensemble de types de composants et de connecteurs, des contraintes qui déterminent les compositions autorisées des éléments qui sont des instances des types, un ensemble de propriétés fournissant des informations de sémantique et de comportement, et un ensemble d'analyses réalisables (performance, ordonnançabilité, etc.). Les styles architecturaux permettent de décrire des patrons d'architecture pour la structuration des systèmes par analogie avec les patrons de conception [GHJV94] pour la programmation orientée objet. Ainsi le patron *Pipe and Filter* est adapté au traitement de flot de données. Le patron *Layer* est utilisé pour l'implémentation de protocoles de communication qui se prêtent bien à une architecture en couches. De nombreux autres patrons d'architecture existent (*Brocker*, *BlackBoard*, etc.) et sont spécialisés pour un domaine applicatif donné. Certains ADL intègrent cette notion de style, c'est par exemple le cas du langage Acme [GMW00] avec son concept de famille d'architecture qui regroupe des ensembles de type d'éléments architecturaux :

```
Family PipesAndFiltersFam = {
   Component Type FilterT = {};
   Connector Type PipeT = {};
};

System APFSystem : PipesAndFiltersFam = {
   Component filter1 : FilterT = new FilterT;
   Component filter2: FilterT = new FilterT;
   Connector pipe : PipeT = new PipeT;

   ...
};
```

Les styles dans Acme sont réutilisables et extensibles grâce au mécanisme de sous-typage.

2.2 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 et plus particulièrement les systèmes autonomiques (section 2.2.3) utilisent les mécanismes de reconfigurations dynamiques de leur architecture pour s'adapter à des changements de contexte.

2.2.1 Réflexion et architectures réflexives

Systèmes réflexifs. Un système réflexif [Smi84] possède la capacité de raisonner et d'agir sur lui-même [KdRB91] par le biais d'une auto-représentation. Il possède plus précisément deux capacités complémentaires :

- La capacité d'introspection, qui correspond à la possibilité de pouvoir s'observer soi-même. La représentation exposée par le système correspond à la réification de certaines de ses parties, en particulier de sa structure et de son comportement.

– La capacité d'intercession, qui correspond à la possibilité de pouvoir se modifier soi-même. L'intercession permet donc de modifier les entités réifiées du système.

Un système réflexif comporte deux niveaux, un niveau de base pour désigner le code implémentant le sujet principal du système et un niveau méta (ou méta-niveau) qui représente le code réflexif qui manipule les réifications du système. Les deux niveaux sont causalement connectés [Mae87] : une modification dans le niveau de base entraîne automatiquement une modification au niveau méta et réciproquement. Deux aspects complémentaires de la réflexion sont à distinguer :

– La réflexion structurelle 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 de modifier ces éléments de structures (par exemple changer la valeur d'une variable d'instance).
– La réflexion comportementale s'intéresse à l'exécution du programme. L'introspection permet par exemple de savoir quelles sont les méthodes ou fonctions en cours d'exécution. L'intercession autorise à modifier la sémantique de certaines constructions du langage (par exemple intercepter une invocation de méthode et surcharger son exécution).

Langages réflexifs. Les langages à objets sont particulièrement adaptés à la mise en oeuvre d'architectures réflexives [Mae87] grâce à leur structure (encapsulation des données) et leur organisation (héritage et composition). Les protocoles à méta-objets *Meta Object Protocol* ou MOP) sont des implémentations à base d'objets de la réflexion [KdRB91]. Le méta-niveau est constitué par des méta-objets qui sont liés au niveau de base. Le protocole définit les interfaces et les interactions entre objets et méta-objets. Différentes catégories de MOPs peuvent être distingués en fonction du moment où sont utilisés les méta-objets : à la compilation, au chargement des classes ou à l'exécution. Le MOP de CLOS (Common Lisp Object System) est un exemple représentatif de cette approche [KdRB91].

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 de 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 nécessaire instrumentation des langages pour pouvoir être modifiés, il peut de même se poser des problèmes en terme de sécurité à partir du moment ou le comportement d'une application peut être détourné de ses fonctionnalités originales.

Architectures réflexives. L'ouverture et la flexibilité apportées par la réflexion est applicable aux architectures logicielles et plus particulièrement aux modèles de composant [CSST98]. La réflexion est notamment un bon support pour la dynamicité dans les architectures logicielles [CDP+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 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 tel 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.

2.2.2 Reconfigurations dynamiques

Reconfigurations dynamiques basées sur l'architecture. Deux catégories de systèmes peuvent particulièrement bénéficier des reconfigurations dynamiques pour leur évolution : les systèmes à haute disponibilité et les systèmes adaptatifs. En effet, une reconfiguration dynamique ne nécessite pas l'arrêt ou le redéploiement complet du système pour le modifier, l'interruption de service est minimisée

et la disponibilité du système est donc préservée. De plus, un système est capable d'évoluer par lui-même et automatiquement grâce aux reconfigurations dynamiques. L'approche architecturale pour les reconfigurations dynamiques repose sur les architectures à base de composants [OMT98] pour modifier le systèmes informatiques à l'exécution. Cette approche présente l'avantage de bénéficier de la modularité et de l'encapsulation forte des composants qui permet d'isoler précisément les effets des reconfigurations dans le système. Le choix d'une décomposition particulière en composants de l'architecture d'un système et de la granularité des composants sont par conséquent importants car ils déterminent la granularité des reconfigurations dynamiques possibles.

Deux types de reconfigurations dynamiques existent : les reconfigurations programmées, et les reconfigurations ad-hoc. Les reconfigurations programmées désignent des reconfigurations qui sont prévues au moment de la conception du système. Il est par exemple possible de prévoir plusieurs implémentations d'un même composant qui joueront le rôle d'alternatives possibles pour adapter le système au cours de son exécution. Au contraire, les reconfigurations ad-hoc ou non-anticipées ne sont pas connues lors de la conception, elles consistent en des modifications arbitraires, peuvent intervenir à un moment quelconque, et sont initiées depuis l'extérieur du système. Ces dernières reconfigurations sont par conséquent plus susceptibles de mettre le système dans un état invalide. Les deux types de reconfigurations peuvent éventuellement être supportées en même temps dans un modèle de composant.

Opérations de reconfiguration. Les opérations de reconfiguration dynamique 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

Les modèles hiérarchiques comme Fractal [BCL+06] distinguent l'opération de création ou d'instanciation de composant de l'opération purement structurelle d'ajout dans une configuration architecturale. L'opération de retrait définitif ou de destruction n'est supportée que lorsqu'il existe un mécanisme permettant de décharger le code du composant de la mémoire. L'opération de remplacement de composant est généralement considérée comme la composition d'une opération de retrait suivie d'une opération d'ajout de composant. Certains modèles exposent également des opérations pour changer la valeur des propriétés des éléments d'architecture, ainsi que des opérations liées au comportement du système comme la modification du cycle de vie des composants.

Problèmes posés par les reconfigurations dynamiques. Malgré les avantages de l'approche architecturale, plusieurs problèmes se posent néanmoins concernant les reconfigurations dynamiques dans les architectures à composants [Ore98]. 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 modèle 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 la cohérence de sa structure et la validité de son comportement, de même que certaines propriétés non-fonctionnelles (e.g. sécurité, qualité de service, etc.), et ce en dépit de possibles défaillances. L'ajout et la vérification de contraintes ou de styles architecturaux permettent par exemple de restreindre les transformations possibles des architectures.
- L'exécution des reconfigurations dynamiques doit être synchronisée avec l'exécution fonctionnelle du système. Il ne doit par exemple pas être possible pour un composant de continuer à invoquer des méthodes de l'interface fonctionnelle d'un autre composant alors qu'il sont en train d'être déconnectés. Il existe des algorithmes de synchronisation [KM90] qui visent à figer l'état des composants avant une reconfiguration.
- Un problème complémentaire à la synchronisation est la gestion de l'état des composants reconfigurés et plus précisément le transfert d'état 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'occurrence possible de plusieurs reconfigurations simultanément dans un système à l'exécution requiert une synchronisation pour éviter les conflits entre reconfigurations sous peine de

mettre potentiellement le système dans un état incohérent.

2.2.3 Systèmes adaptatifs et informatique autonomique

Systèmes adaptatifs. Une adaptation est une modification d'un système en réponse à un changement dans son contexte. L'objectif est que le système résultant soit plus apte fonctionnellement dans son nouveau contexte qu'avant son adaptation. Par conséquent, une adaptation est toujours une modification d'un système mais une modification quelconque n'est pas une adaptation si le système modifié ne peut être considéré comme « meilleur »que l'ancien selon les critères considérés (e.g. performance, fiabilité, qualité de service, etc.) dans ses nouvelles conditions d'exécution. Dès lors, un système sera dit adaptable si une entité externe (logiciel ou humaine) peut le faire évoluer. Il sera dit adaptatif s'il est capable tout seul de s'adapter automatiquement. Un système adaptatif est ainsi à la fois le sujet et l'acteur de l'adaptation [DC01] : il se modifie de façon autonome en fonction de son contexte. Cette modification se fait donc dynamiquement pendant son exécution. Par conséquent, un système adaptatif peut reposer sur des reconfigurations dynamiques de son architecture pour réaliser son adaptation [OGT+99].

Un système adaptatif est caractérisé dans [Dav05] par :
- un ensemble d'opérations pour modifier, adapter le système ;
- un contexte qui regroupe tous les éléments externes au système et qui influencent son fonctionnement ;
- une fonction d'adéquation qui permet à tout moment de savoir dans quelle mesure le système réalise ses objectifs relativement à son contexte ;
- une stratégie d'adaptation en charge de la mise en oeuvre du processus d'adaptation dans le système.

Informatique autonomique. Parmi les systèmes adaptatifs, une initiative d'IBM distingue les systèmes dits autonomiques [KC03]. Avec la complexité croissante des systèmes et des applications, le coût d'administration devient de plus en plus considérable et les difficultés commencent à dépasser les compétences des administrateurs. Par conséquent, une nouvelle tendance est apparue qui consiste à automatiser (au moins en partie) les fonctions d'administration. C'est l'un des objectifs de ce qui est appelé l'informatique autonomique (*Autonomic Computing*). L'informatique autonomique peut reposer sur l'architecture des systèmes pour la mise en oeuvre des algorithmes de contrôle [WHW+04] et plus particulièrement sur les reconfigurations dynamiques dans les architectures à composants [KM07].

Le but de l'*Autonomic Computing* est d'offrir aux systèmes des capacités d'auto-administration, à travers des capacités d'auto-adaptation face aux changements de contexte. Les systèmes autonomiques sont classés en quatre catégories selon les objectifs de l'adaptation :
- L'auto-configuration consiste à déployer et configurer automatiquement le système suivant des règles prédéfinies.
- L'auto-optimisation pour le contrôle et l'adaptation du système suivant des changements au niveau fonctionnel ou extérieur au système pour assurer un certain niveau de performance.
- L'auto-réparation pour la détection et le diagnostique des pannes et leur correction de manière automatique.
- L'auto-protection consiste à prendre des mesures nécessaires pour se protéger des attaques malveillantes et savoir se défendre contre ces attaques.

Boucle de contrôle. L'architecture d'un système autonome a été initialement proposée par IBM [KC03] sous forme d'une boucle de contrôle, concept issu de la théorie du contrôle en automatique. La figure 2.2 présente l'organisation des éléments d'une telle architecture : un élément administré est en relation avec un gestionnaire autonomique responsable de l'observation et de l'adaptation du système à l'intérieur de la boucle rétroactive.

Le gestionnaire autonomique remplit différentes fonctions :
- L'observation du système ou monitoring pour la collecte des données qui caractérisent le comportement du système.
- L'analyse des informations observées pour interpréter ces données en se basant sur une connaissance du système administré.
- La décision en utilisant des algorithmes dont le rôle est de déterminer un plan d'action.

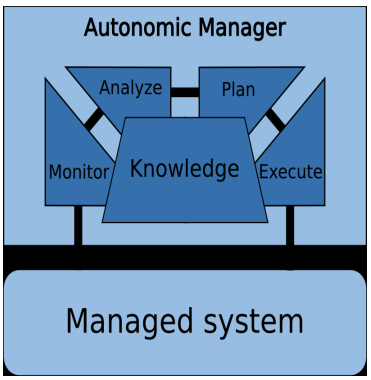

FIGURE 2.2 – Autonomic Computing (IBM)

– L'exécution pour la mise en oeuvre du plan en envoyant des commandes aux actionneurs de l'élément administré.

Le gestionnaire communique avec l'élément administré grâce aux capteurs et aux actionneurs. Les capteurs et les actionneurs offrent des interfaces principales implantées par le système administré pour observer son état et pour le modifier à des niveaux variables de granularité.

Cas de l'Auto-réparation. Un système est sujet à une défaillance quand il dévie de son comportement attendu. Ceci peut résulter d'une erreur humaine au niveau de l'utilisation de l'application (une valeur d'entrée incorrecte) ou une erreur matérielle (la panne franche). La disponibilité d'un système s'exprime généralement sous forme de taux et est la fraction de temps pendant laquelle le système est à même de délivrer un service correct. Soit A_i la disponibilité d'un système, on a plus formellement : $A_i = \frac{MTTF}{MTTF+MTTR} = \frac{MTTF}{MTBF}$ où le MTTF (Mean Time To Failure) est le temps moyen de fonctionnement entre défaillances, le MTTR (Mean Time To Repair) est le temps moyen de réparation et le MTBF (Mean Time Between Failures) est le temps moyen entre défaillances. Le but de l'auto-réparation est de garantir un certain taux de disponibilité en minimisant le MTTR grâce à la réparation automatique du système sans intervention humaine. La boucle de contrôle dans le cas de l'auto-réparation contient un gestionnaire autonomique capable de détecter l'apparition d'une erreur au niveau fonctionnel ou une panne matériel du système. L'analyse ou diagnostique consiste à interpréter la faute pour ensuite décider du plan de réparation à réaliser. Enfin, le plan de réparation est exécuté sur le système pour qu'il retrouve un état correct.

2.3 Sûreté de fonctionnement

La fiabilité est un attribut de la sûreté de fonctionnement (section 2.3.1) dont la garantie suppose la mise en oeuvre de méthodes telles que la tolérance aux fautes. Les transactions 2.3.2 sont un moyen de rendre les systèmes tolérants aux fautes par des mécanismes de reprise sur défaillance.

2.3.1 Attributs et méthodes de la sûreté de fonctionnement

La sûreté de fonctionnement pour un système peut se définir comme la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Car82]. Le service délivré correspond au comportement du système perçu par un utilisateur. L'une des préoccupations premières en génie logiciel est la qualité des systèmes produits. Un système doit délivrer les fonctionnalités spécifiées lors de la conception et attendues par son utilisateur, ce qui correspond à la propriété de fiabilité. La fiabilité est l'un des attributs de la sûreté de fonctionnement, ces attributs permettent d'exprimer les propriétés attendues du système et d'apprécier la qualité de service de ce dernier. Les différents attributs complémentaires sont les suivants [Lap92b] :

– La disponibilité : fait d'être prêt à l'utilisation
– La fiabilité : mesure de délivrance continue d'un service correct.
– La sécurité innocuité : non-occurrence de conséquences catastrophiques pour l'environnement.
– La confidentialité : non-occurrence de divulgations non autorisées de l'information.

– L'intégrité : non-occurrence d'altérations inappropriées de l'information
– La maintenabilité : aptitude aux réparations et aux évolutions.

Il convient de relier la propriété de fiabilité au concept de défaillance et plus généralement d'entrave qui regroupe les défaillances, les fautes et les erreurs. Garantir la fiabilité dans un système consiste en effet à éviter des défaillances du service. Il existe un lien de causalité entre fautes, défaillances et erreurs [Lap92b] :

– une défaillance caractérise le fait qu'un service délivré dévie de l'accomplissement de la fonction du système,
– Une erreur caractérise la partie de l'état du système susceptible d'entraîner une défaillance,
– Une faute est la cause adjugée ou supposée d'une erreur.

Les fautes peuvent être classée selon de nombreux critères parmi lesquels nous retiendrons essentiellement leur dimension qui caractérise si une faute est d'origine logicielle ou matérielle, leur cause phénoménologique qui distingue les fautes physiques des fautes humaines. Les bogues de programmation rentrent dans la catégorie des fautes humaines et logicielles tandis que les pannes machines sont ainsi des fautes physiques et matérielles. Dans le contexte des reconfigurations dynamiques, les fautes considérées sont les fautes logicielles générées par l'exécution d'une reconfiguration dans un système donné et les fautes matérielles externes aux reconfigurations.

Méthodes de la sûreté de fonctionnement. Pour pouvoir assurer les attributs de la sûreté de fonctionnement dans un système donné, il existe différentes méthodes à appliquer :

– La prévention des fautes pour empêcher l'occurrence ou l'introduction de fautes.
– La tolérance aux fautes pour fournir un service à même de remplir la fonction d'un système en dépit des fautes.
– L'élimination des fautes pour réduire la présence des fautes.
– La prévision des fautes pour estimer la présence, la création et les conséquences des fautes.

Malgré des techniques de prévention de fautes dans les phases de conception du logiciel, des erreurs peuvent toujours survenir dans un système pendant son exécution. Nous nous intéressons plus particulièrement aux systèmes dit « tolérants aux fautes »qui désignent des systèmes capable de continuer à fonctionner normalement en cas de l'occurrence de certaines fautes, la gestion des erreurs est alors réalisée de manière transparente pour l'utilisateur. Le traitement d'erreur consiste soit en du recouvrement d'erreur, soit en de la compensation d'erreur. La compensation repose sur des techniques de redondance pour continuer la délivrance du service. Le recouvrement est une technique qui permet soit de trouver un nouvel état du système pour qu'il continue de fonctionner éventuellement en mode dégradé (poursuite), soit de retrouver un état antérieur exempt d'erreur (reprise). Pour mettre en oeuvre le recouvrement d'erreur, il est nécessaire afin de pouvoir les traiter les erreurs d'identifier l'état erroné (détection) et d'évaluer les dommages créés par l'erreur et les erreurs propagés avant sa détection (diagnostique).

L'ensemble des attributs et des moyens de la sûreté de fonctionnement ainsi que les entraves décrits dans [Lap92b] sont résumés dans le schéma de la figure 2.3.

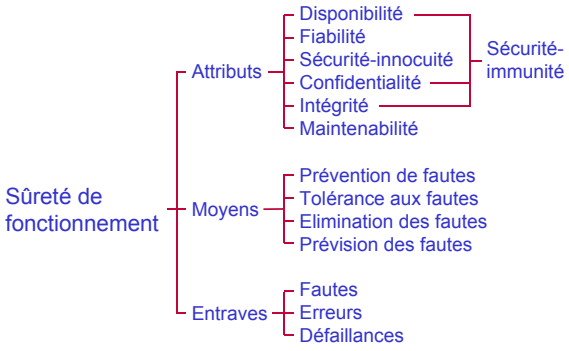

FIGURE 2.3 – Arbre de la sûreté de fonctionnement

2.3.2 Systèmes transactionnels

Les transactions ont été originellement utilisées dans les systèmes de gestion de base de données [TGGL82]. Leur utilisation s'est par la suite généralisée dans tous les systèmes informatiques ou il existe le besoin de maintenir la cohérence des informations en dépit d'accès concurrents et de l'occurrence de défaillances. Les transactions sont donc un moyen de rendre des systèmes tolérants aux fautes.

Propriétés ACID. Une transaction consiste à effectuer une opération informatique cohérente composée de plusieurs tâches unitaires. L'opération ne sera valide que si toutes les tâches unitaires sont effectuées correctement, on parle alors de validation (*commit*). Dans le cas contraire, l'ensemble des données traitées lors de l'opération reviennent à leur état initial, la transaction est abandonnée et on parle alors d'annulation (*rollback*). Le concept de transaction s'appuie sur la notion de point de synchronisation qui représente un état stable du système informatique considéré, en particulier de ses données.

Les quatre propriétés fondamentales des transactions, les propriétés « propriétés ACID »[GR92], sont définies de la manière suivante :

- *Atomicité.* Soit la transaction se termine et est validée, soit elle est abandonnée (sémantique du « tout ou rien »).
- *Cohérence.* Une transaction préserve la cohérence des objets modifiés.
- *Isolation.* Les effets d'une transaction non validée ne sont pas visibles des autres transactions en cours d'exécution.
- *Durabilité.* Les effets d'une transaction validée sont permanents.

Contrôle de concurrence. Le contrôle de concurrence vise à permettre l'exécution concurrente de transactions en garantissant la propriété d'isolation entre elles. Il repose sur le principe de sérialisabilité des transactions : une exécution concurrente de transactions validées est sérialisable si et seulement si il existe une exécution en série équivalente. De ce fait, les conflits entre transactions sont évités ainsi que la production d'un état incohérent résultant dans le système. Deux grandes méthodes de gestion de concurrence existent [BHG87] :

- Les méthodes pessimistes (contrôle continu) gèrent les dépendances entre transactions et détectent les conflits au fur et à mesure de leur apparition. C'est le cas des méthodes de verrouillage.
- Les méthodes optimistes (contrôle par certification) laissent les dépendances entre transactions se créer et repoussent le contrôle de la sérialisabilité à la fin des transactions.

Reprise après défaillance. Les transactions constituent la plus petite unité de reprise en cas de défaillance. Les défaillances auxquelles les transactions apportent une solution de recouvrement sont les suivantes :

1. Abandon de la transaction : une transaction ne se termine pas suite à une erreur de programmation, à la violation de la cohérence du système ou à un problème de sérialisabilité (rejet du contrôle de concurrence). La reprise se fait en annulant la transaction à partir de la mémoire principale.
2. Défaillance de site : arrêt du système qui nécessite un redémarrage ultérieur. La défaillance peut être due à une panne matérielle ou des couches logicielles basses (système d'exploitation). Seule la mémoire principale (volatile) est perdue, la mémoire secondaire (stable), généralement sur disque, n'est pas effacée. La reprise se fait en annulant la transaction à partir de la mémoire secondaire.
3. Défaillance de la mémoire secondaire : le contenu du support de la mémoire secondaire est perdu, par exemple suite à une panne de disque. Le problème de la perte de mémoire secondaire est résolu par des techniques classiques de redondance (réplication de disques) ou d'archivage des données.
4. Défaillance des communications dans un contexte réparti : certains sites sont isolés des autres suite à une panne du réseau. La technique de reprise dépend de la mise en oeuvre d'un protocole

de validation atomique réparti.

Validation atomique répartie. Une application est dite répartie si elle accède à des « objets »situés sur des sites différents. La répartition augmente les possibilités d'incohérence dans un système. En effet, une défaillance de site dans le cas centralisé rend tout le système non fonctionnel. Des mécanismes de journalisation permettent alors de refaire ou défaire les effets des transactions. Par contre, dans le cas réparti, la défaillance d'un site est une défaillance partielle qui peut aboutir à une incohérence de l'état global du système, une partie de l'état étant bien modifiée mais l'état du système sur le site défaillant ne l'étant pas. Se pose alors le problème de l'atomicité globale dans le système réparti ou validation atomique répartie [BCF+97]. Un site accédé par une transaction est appelé un participant, chaque participant exprime par un vote sa décision de validation de la branche dont il est le responsable. Une branche est un sous ensemble de la transaction exécutée localement par un participant. Dans le cas où il existe plusieurs sites participant à la transaction, la décision de validation doit être la même pour tous les sites participants (propriété d'unanimité).

Le protocole de validation à deux phases (Two-Phase-Commit ou 2PC [TGGL82] représenté dans la Figure 2.4) est largement utilisé pour répondre au problème de validation atomique répartie. Un des participants dans la transaction, souvent l'initiateur (qui démarre l'exécution de la transaction), joue le rôle de coordinateur en dirigeant l'ensemble des participants. Le protocole se déroule alors de la manière suivante :

- Préparation : demande de vote du coordinateur à tous les participants (message *prepare*)
- Vote : chaque participant retourne son vote au coordinateur, s'il ne reçoit pas de message *prepare*, il vote *non*.
- Décision du coordinateur : le coordinateur décide en fonction des votes reçus. La transaction n'est validée qu'en cas d'unanimité du vote oui par tous les participants.
- Acquittement des participants : chaque participant acquitte la décision reçue du coordinateur
- Terminaison : le coordinateur libère les ressources associées à la transaction une fois tous les acquittements reçus.

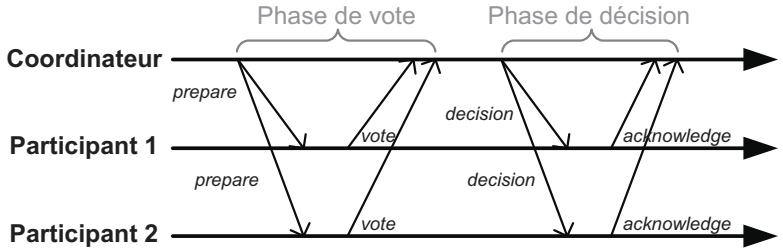

FIGURE 2.4 – Protocole de validation atomique en deux phases (2PC)

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. Ces reconfigurations sont à la base des systèmes adaptatifs et autonomiques qui sont capable de se faire évoluer eux-même à l'exécution en fonction de leur contexte d'exécution. Modifier dynamiquement un système à travers son architecture n'est pas sans poser des problèmes de fiabilité, attribut majeure de la sûreté de fonctionnement et préoccupation importante du génie logiciel. Les transactions sont une réponse à ce problème de fiabilité de part leurs bonnes propriétés de maintien de la cohérence dans les systèmes en dépit d'éventuelles défaillances.

Chapitre 3

Etat de l'art sur la fiabilité des reconfigurations dynamiques

Sommaire

L'objectif de ce chapitre est de recenser et d'analyser plusieurs travaux liés au problème de la fiabilité des reconfigurations dynamiques dans les architectures à composants. Nous donnons dans la section 3.1 nos critères d'évaluation des différents travaux et recensons ensuite ces travaux que nous classons dans trois catégories : les modèles de composants réflexifs dans la section 3.2, les plateformes qui construisent un modèle d'architecture au-dessus des systèmes à l'exécution dans la section 3.3 et enfin les environnements d'évolution des architectures logicielles dans la section 3.4. Nous terminons dans la section 3.5 par une synthèse de nos évaluations en comparant les propriétés et fonctionnalités des travaux analysés.

3.1 Critères de sélection et d'évaluation des travaux

Nous nous focalisons dans notre état de l'art sur les travaux autour des reconfigurations dynamiques de systèmes reposant sur une architecture à base de composants. Leur point commun est de mettre en avant une forme ou une autre de fiabilité, de maintien de la cohérence des systèmes soumis aux reconfigurations. Nous nous intéressons plus particulièrement au maintien de la cohérence structurelle des architectures.

Pour classer et comparer les travaux, nous nous basons sur différents critères liés aux modèles de composants utilisés, aux reconfigurations dynamiques supportées, et aux propriétés garanties liées à la fiabilité.

Modèles de composants. Tous les travaux étudiés considèrent les architectures à composants comme base de l'évolution et des reconfigurations dynamiques des systèmes logiciels. Pour caractériser

les reconfigurations et leurs propriétés, nous étudions dans un premier temps la complexité des modèles de composants utilisés et certaines de leurs propriétés :

- **Modèle de composants de la plateforme.** Quel est le modèle de composants utilisé pour l'implémentation et la reconfiguration des systèmes dans la plateforme ?
- **Langage de configuration.** Quel langage est employé pour la spécification des configurations architecturales des systèmes ? Existe-il un langage dédié de description des architectures ?
- **Eléments d'architecture du modèle.** Quels sont les éléments d'architectures manipulés par le modèle de composants et/ou l'ADL (composants, connecteurs, etc.) ?
- **Cohérence entre modèle et système reconfiguré.** Comment est maintenue la cohérence entre le système à l'exécution et sa représentation architecturale ? Une connexion causale vérifiée par les architectures réflexives implique par exemple qu'une modification dans le système sera reflétée dans le modèle d'architecture et réciproquement.
- **Hiérarchie du modèle.** Le modèle supporte-il les composants composites pour la construction d'architecture complexes ?
- **Ouverture du modèle.** Le modèle est-il ouvert extensible (ajout de nouveaux éléments d'architectures, implémentation multiples, etc.) ?
- **Support des systèmes distribués.** Le modèle gère-t-il l'architecture de systèmes distribués en terme de configuration, d'introspection et d'intercession ?

Gestion de la cohérence dans les architectures. La fiabilité des reconfigurations repose sur le maintien de la cohérence des architectures pour les systèmes reconfigurés. La plupart des travaux mettent en avant la possibilité de contraindre les architectures, essentiellement structurellement. Nous évaluons donc les critères suivants :

- **Type de contraintes.** Quelles sont les contraintes prise en compte pour définir la cohérence des architecture (invariants, pré/post-conditions, contraintes structurelles ou comportementales, etc.) ?
- **Langage de contraintes.** Existe-il un langage dédié pour la spécification des contraintes et avec quelle formalisme ?
- **Système de typage des éléments de l'architecture.** Quelle sont les particularités du système de typage des éléments de l'architecture ?
- **Mécanisme de vérification de la cohérence des architectures et réaction en cas de violation de la cohérence.** Comment et quand sont vérifiées les contraintes dans le système et quelle est la réaction adoptée en cas de violation de la cohérence des architectures ?

Support des reconfigurations et propriétés garanties. Les dernières évaluations concernent le support des reconfigurations. Nous distinguons les évolutions statiques des architectures décorrélées des systèmes à l'exécution des reconfigurations dynamiques.

- **Langage de reconfigurations.** Comment sont spécifiées les reconfigurations, avec quel langage éventuellement dédié ?
- **Type de reconfigurations supportée.** Quel est le type des opérations de reconfiguration dynamiques supporté par les plateformes ? Nous distinguons les reconfigurations programmées qui nécessite d'être connues au moment de la conception des systèmes et les reconfiguration ad-hoc (ou non-anticipées) qui peuvent intervenir arbitrairement pendant l'exécution des systèmes.
- **Opérations primitives supportées.** Quelles sont les opérations primitives de reconfigurations dynamiques fournie par les implémentations ?
- **Extensibilité des opérations de reconfiguration.** Les opérations primitives sont-elles extensibles ? Peut-on définir facilement de nouvelles opérations ?
- **Atomicité des reconfigurations.** Une reconfiguration invalide en cas de violation de contrainte est-elle annulée et le système remis dans un état cohérent ?
- **Support des reconfigurations concurrentes.** Plusieurs reconfigurations peuvent-elles être exécutées en même temps dans le système en étant synchronisées entre-elles pour éviter les conflits ?
- **Durabilité des effets des reconfigurations.** Une fois une reconfiguration exécutée, les effets sont-ils persistant même en cas de panne du système ?

Nous répartissons les travaux étudiés dans trois catégories différentes. La première catégorie concerne les modèles de composants réflexifs (section 3.2). Les systèmes implémentés avec ces mo-

dèles offrent une représentation de leur architecture instrospectable et modifiable dynamiquement. La connexion causale entre le système reconfiguré et sa représentation architecturale est ainsi directement supporté par ces modèles. La deuxième catégorie de travaux qui est évaluée consiste à maintenir un modèle d'architecture au-dessus des systèmes administrés (section 3.3). Ces derniers peuvent être implémentés avec un modèle de composants, imposé ou non par la plateforme, qui peut éventuellement et déjà être un modèle réflexif. La troisième et dernière catégorie (section 3.4) regroupe des plateformes qui ne supporte pas directement la reconfiguration de systèmes à l'exécution mais se concentrent sur l'évolution statique de leur architecture et la garantie de certaines propriétés au cours de ces évolutions. Ces propriétés pourraient cependant être appliquées aux reconfigurations dynamiques si un « mapping » était réalisé entre les évolutions d'architecture et les reconfigurations des systèmes à l'exécution.

3.2 Modèles de composants réflexifs

Nous nous intéressons dans cette section aux modèles de composants réflexifs qui répondent à certains de nos critères d'évaluation, notamment en termes de bonnes propriétés des reconfigurations dynamiques.

3.2.1 FORMAware

Description

Introduction. FORMAware [MBC02, MBC03, MBC04] est un canevas logiciel en Java qui permet de développer et d'adapter dynamiquement des systèmes distribués à base de composants. L'adaptation dynamique est réalisée au moyen d'opérations de reconfiguration et est contrainte par des ensembles de règles de maintien de l'intégrité architecturale appelées styles architecturaux [AAG93]. Les objectifs de FORMAware se résument ainsi :

- Proposer un modèle de composants pour le développement de systèmes modulaires.
- Pouvoir concevoir des applications flexibles à travers des mécanismes d'introspection et d'adaptation.
- Fournir une représentation explicite à l'exécution des architectures associées à des contraintes de styles.
- Garantir la sûreté des modifications architecturales en vérifiant que l'ensemble modifiable de règles de style est toujours respecté dans l'architecture.
- Garantir l'atomicité des modifications architecturales grâce à une adaptation transactionnelle.

Modèle de composants. Le modèle de composants proposé est un modèle hiérarchique dans lequel des composants de base implémentent la logique métier et des composants composites sont des conteneurs de composants de base. Les composants et les interfaces sont des entités de première classe tandis que les connecteurs sont des composants spécialisés. Le modèle est réflexif sur le plan structurel et deux méta-modèles différents coexistent pour décrire les mécanismes d'introspection. Dans le méta-modèle d'interfaces, chaque composant est capable d'exposer ses interfaces requises et fournies. Le méta-modèle d'architecture spécifie qu'un composant fournit une représentation de son contenu. Ce contenu consiste en sa topologie interne (graphe de sous-composants et de connecteurs) et un ensemble de contraintes de style. La distribution des composants est gérée grâce à des composants qui joue le rôle de proxy pour les composants distants et leur délègue l'exécution des opérations via Java RMI. Cependant, la sémantique de communications pour les connecteurs entre composants répartis reste ouverte pour implémenter par exemple des communications asynchrones de type publication-souscription.

Styles architecturaux. Un style architectural spécifie un ensemble de concepts qui peuvent être utilisés pour décrire les éléments dans les architectures. Il définit également un ensemble de règles ou contraintes qui restreignent les combinaisons possibles entre éléments. FORMAware permet d'associer un style architectural aux composants composites du système. Le développeur peut soit utiliser et étendre le style par défaut soit définir son propre style. Une règle de style est une classe qui fournit

obligatoirement une méthode *VerifyValidity* de vérification des contraintes pour cette règle. Elle est caractérisée par son type et par les types des opérations de reconfiguration auxquelles elle s'applique.

Reconfigurations dynamiques. Les reconfigurations sont implémentées dans des méthodes Java par l'intermédiaire de méta-objet adaptateurs. Les composites sont responsables de la configuration et de la reconfiguration de leurs sous-composants. Les opérations peuvent concerner la manipulation des graphes de composants (ajout, retrait, connexion, etc.) mais ne sont pas fixées par le modèle et sont extensibles. Une opération de reconfiguration invoquée sur un adaptateur est dans un premier temps transférée à un gestionnaire de style (*StyleManager*) chargé de la vérification du style (cf. figure 3.1). Cette vérification se traduit concrètement par l'exécution d'une opération de style correspondant à l'opération de reconfiguration initiale. Par exemple, à une opération *plug* de connexion entre interfaces de composants correspond une opération *stylishPlug* qui va notamment s'assurer de la compatibilité entre interfaces. Si les règles de style sont vérifiées, l'opération est finalement invoquée sur le graphe d'architecture pour refléter la reconfiguration dans le système.

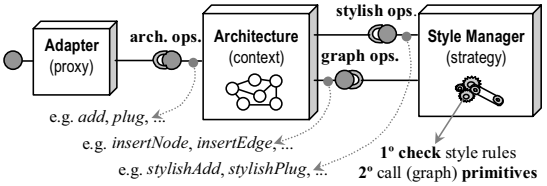

FIGURE 3.1 – Invocation et vérification d'une opération de reconfiguration

Garanties et propriétés des reconfigurations. Pour éviter qu'un système soit mis dans un état incohérent, l'ensemble des contraintes architecturales (règles de style) est vérifié avant l'exécution d'une opération de reconfiguration. Par ailleurs, un gestionnaire de transactions permet d'initier, de valider ou d'annuler les opérations de reconfiguration sans perturber l'exécution du système. L'adaptation est ainsi exécutée de manière atomique. Le service de transactions (cf. figure 3.2) créé des transactions à la demande et les transmet à un gestionnaire de verrouillage pour le support de la concurrence des reconfigurations. Un thread est associé à chaque transaction qui ordonnance l'exécution des opérations en fonction de la politique d'adaptation choisie. Une politique d'adaptation sert à synchroniser l'adaptation avec l'exécution du système et à figer l'état des composants reconfigurés. Elle peut par exemple soit ignorer l'état du composant lors de la reconfiguration (politique *SKIP*), attendre l'état stable (*WAIT*) ou bien forcer l'état stable (*FORCE*). Enfin, un unique thread exécute les différentes opérations suivant l'algorithme du tourniquet (*round robin*).

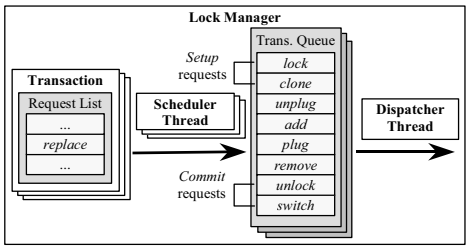

FIGURE 3.2 – Utilisation du service de transactions pour une opération de remplacement de composant

Evaluation

FORMAware propose ainsi un modèle de composants hiérarchique, réflexif, et ouvert tout en garantissant la cohérence des architectures reconfigurées. L'adaptation dynamique des applications est réalisée par des opérations de reconfiguration extensibles sous forme de scripts en Java. La réflexivité et l'ouverture du modèle permettent l'exécution de reconfigurations dynamiques non-anticipées. L'approche suivie à base de différents méta-modèles de haut niveau facilite l'extension du modèle de

composants, des opérations de reconfigurations et des styles architecturaux. Le framework spécifie un processus générique de reconfiguration et de vérification de la cohérence des architectures par rapport à des règles de style.

Le formalisme sous-jacent aux opérations de reconfiguration consiste en des transformations de graphe (ajout et retrait de noeuds et d'arcs). Si la sémantique des opérations de reconfiguration est laissée libre, une sémantique par défaut est tout de même proposée. Cependant cette sémantique est directement spécifiée en Java, langage de programmation généraliste et ne permet donc pas ou difficilement d'effectuer d'analyses de l'effet des opérations sur le système reconfiguré. Il n'existe pas non plus de formalisme pour l'expression des contraintes. Ces dernières semblent être limitées à des préconditions des opérations de reconfiguration sans la possibilité d'exprimer des invariants globaux. Elles ne permettent donc pas de passer par des états intermédiaires incohérents avant la validation.

La solution inclus un gestionnaire de transactions pour garantir les propriétés d'atomicité et de cohérence des reconfigurations. Avant l'exécution de chaque opération dans le système, le gestionnaire de style vérifie les préconditions de l'opération par rapport au style défini. Si les préconditions sont vérifiées, l'opération est effectivement exécutée dans le système. Le gestionnaire de transaction peut annuler des reconfigurations sans toutefois que le processus d'annulation soit spécifié, le système semble en effet être modifié à chaque exécution d'opération, les opérations déjà exécutée en cas d'échec d'une transaction doivent donc normalement être défaites. Le contrôle de concurrence entre reconfiguration est réalisé par un gestionnaire de verrouillage, cependant la granularité du verrouillage n'est pas explicitée. La synchronisation entre reconfigurations et exécution du système est faite par l'intermédiaire des trois politiques d'adaptation proposées pour ignorer, attendre, ou forcer un état stable avant de reconfigurer.

3.2.2 K-Component

Description

Introduction. K-Component [DCC01, DC01] est un modèle de composants supportant les reconfigurations dynamiques grâce à la réflexion architecturale pour construire des systèmes adaptatifs. Le modèle garantit l'intégrité et la sûreté des évolutions des systèmes à l'exécution en modélisant les reconfigurations comme des opérations de transformation de graphes sur les architectures. De plus, des programmes spécifiques appelés contrats d'adaptation permettent de séparer les préoccupations entre le code d'adaptation et le code fonctionnel des composants.

Modèle de composants. L'architecture des systèmes est réifiée sous forme de graphe connecté et typé : les noeuds du graphe représentent les interfaces et sont annotés par les composants qui leur appartiennent et les implémentent, les arcs représentent les connecteurs entre interfaces de composants et sont annotés par des propriétés reconfigurables. Un exemple d'une telle propriété est le protocole de communication mis en oeuvre par un connecteur. Un composant de gestion de configurations (« configuration manager ») est en charge de la réification de l'architecture des systèmes et de l'exécution des reconfigurations. Le code des connecteurs est généré à partir des définitions IDL des interfaces et expose des opérations de connexion et de déconnexion de composants. L'implémentation de composants et leur assemblage sont réalisés en C++ sans passer par l'utilisation d'un ADL. Le méta-modèle d'architecture du système est alors extrait et généré sous forme d'un graphe de configuration décrit en XML. Ce descripteur est complété par le programmeur pour y introduire les composants développés. L'architecture peut finalement être instanciée par le gestionnaire de configuration.

Reconfigurations dynamiques. Les reconfigurations dynamiques sont modélisées comme des transformations conditionnelles des graphes de configuration (cf. figure 3.3). Une transformation de graphe est une règle qui définit quand et comment le graphe est modifié. Le gestionnaire de reconfiguration exécute les reconfigurations transactionnellement. Cependant, les seules reconfigurations autorisées dans le modèle sont le remplacement de composants et le changement des propriétés des connecteurs. Un protocole de reconfiguration est proposé pour l'atteinte d'un état stable au cours des reconfigurations des composants. Ce protocole permet de figer l'état des composants et des connecteurs impactés par les reconfiguration.

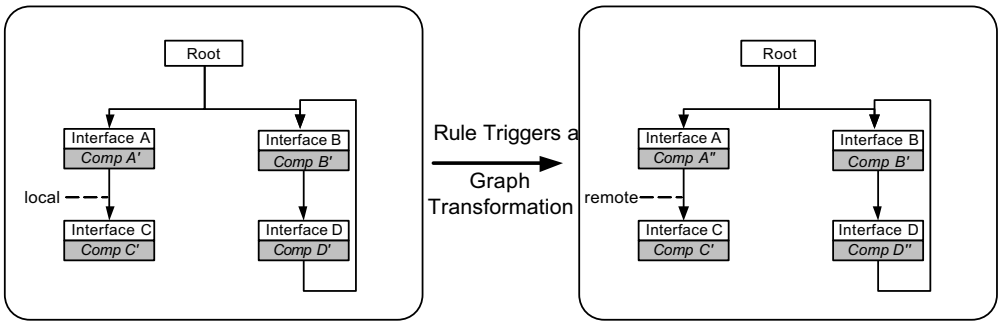

FIGURE 3.3 – Les reconfigurations dynamiques comme des transformations de graphes

Contrats d'adaptation. Les contrats d'adaptation contiennent des règles conditionnelles pour la transformation des architectures. Ils jouent ainsi le rôle de contraintes architecturales en restreignant les reconfigurations possibles sur une architecture donnée. Les contraintes architecturales sont en effet des propriétés ou des assertions sur les configurations de composants et de connecteurs. Les reconfigurations spécifiées dans les contrats sont déclenchées par des événements d'adaptation issue du système reconfiguré. Ces contrats sont spécifiés dans un langage dédié et associe des événements d'adaptation avec des opérations de reconfiguration. Le gestionnaire de configurations a la charge de l'ordonnancement et de l'exécution des contrats d'adaptation en testant l'occurrence des événements d'adaptation.

Evaluation

K-Component propose un modèle réflexif de composants et de reconfiguration dynamique reposant sur des transformations de graphes. Les garanties apportées par le modèle sont de plusieurs ordres. Un protocole de synchronisation entre reconfigurations et exécution des systèmes appelé protocole de reconfiguration est présenté. Les reconfigurations sont contraintes par l'occurrence d'événements d'adaptation pour vérifier la satisfaction d'assertion dans les architectures. Enfin les reconfigurations sont des transformations de graphes simples réalisées de manière transactionnelle.

L'approche comporte cependant plusieurs limitations et suppose des hypothèses contraignantes sur les reconfigurations. D'une part, les reconfigurations possibles sont limitées à deux types de transformation de graphe. D'autre part, les mécanismes de contraintes par des contrats d'adaptation nécessite obligatoirement l'émission d'événements d'adaptation, une reconfiguration ne pouvant être exécutée spontanément de l'extérieur du système. Enfin, si un mécanisme transactionnel est mentionné pour l'exécution des reconfigurations sur les architectures, aucune précision n'est donné sur d'éventuelles garanties d'atomicité, de concurrence et de durabilité.

3.2.3 OpenRec

Description

Introduction. OpenRec [HW04] est un canevas logiciel ouvert pour la reconfiguration dynamique de systèmes à composants. Il s'intéresse tout particulièrement au problème de la synchronisation entre reconfigurations et exécution fonctionnelle des systèmes et à la gestion de l'état des composants reconfigurés. Alors que beaucoup de plateformes ne supportent qu'un seul algorithme fixé de synchronisation, OpenRec permet de choisir parmi un ensemble extensible un algorithme adapté au style architectural d'un système donné. Ces algorithmes ont pour objectif d'identifier parmi les composants du système reconfiguré lesquels doivent être mis dans un état stable avant d'opérer les modification architecturales. De plus, le framework permet d'observer l'impact des algorithmes de synchronisation en terme de temps d'exécution et de perturbation du fonctionnement des systèmes.

Architecture du framework. Le modèle de composants dans OpenRec est un modèle réflexif qui distingue essentiellement les concepts de composant et de connecteur. Le framework est construit

avec des composants OpenRec, il est par conséquent lui-même reconfigurable. Les trois couches qui le constituent (cf. figure 3.4) sont les suivantes :

- Le *ChangeDriver* auquel sont soumises en cas de reconfigurations non-anticipées, les scripts de reconfiguration spécifiés en OpenRecML, langage basé sur XML. Le composant est également capable, en fonction d'événements issus de l'observation des changements de l'architecture, de déclencher des reconfigurations programmées.
- Le *ReconfigurationManager* est en charge de l'exécution d'un algorithme de reconfiguration et de sa synchronisation.
- La couche *Application* consiste en la configuration des composants et des connecteurs de l'application.

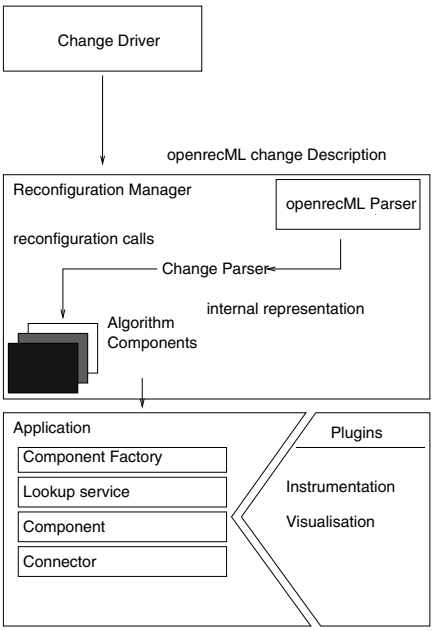

FIGURE 3.4 – Architecture du framework OpenRec

Processus de synchronisation. OpenRec permet de choisir son propre algorithme de synchronisation éventuellement choisi parmi des algorithmes existants, d'en observer les effets en terme de nombre de composants impactés par l'algorithme et d'en mesurer le temps d'exécution. L'intérêt du choix de l'algorithme est de minimiser les coûts en disponibilité en fonction du style architectural de l'application considérée. Certains algorithmes spécialisés sont en effet conçu spécialement pour des styles particuliers (e.g. [Mit00] pour le style « pipe and filter »). Les algorithmes statiques (e.g. [KM90]) repose ainsi sur la connaissance de l'architecture pour identifier les composants dont l'état est à synchroniser. Les algorithmes dynamiques (e.g. [Che02]), généralement plus efficaces pour minimiser la perturbation de l'exécution des système, permettent de bloquer les interactions entre composants par exemple au niveau des connecteurs. Les différents algorithmes sont implémentés à l'aide du patron de conception *Strategy* dans le gestionnaire de reconfigurations. L'algorithme a accès aux composants du système par réflexion, il peut ainsi vérifier certaines contraintes avant son exécution comme exemple la présence d'une interface nécessaire à son bon déroulement. Après d'éventuelles optimisations, l'algorithme procède à la synchronisation du système avec la reconfiguration à exécuter, et enfin réalise les modifications structurelles.

Evaluation

L'objectif d'OpenRec est focalisé sur le problème de la synchronisation entre reconfigurations et l'exécution des systèmes reconfigurés qui implique notamment une gestion fine de l'état des composants impactés. L'originalité de l'approche est d'ouvrir la plateforme à différents algorithmes de synchronisation pour pouvoir choisir le plus adapté pour minimiser l'indisponibilité des systèmes

pendant les reconfigurations. Les algorithmes génériques qui gère l'état des composants sont par exemple plus coûteux que des algorithmes spécialisés pour certaines architectures où les composants n'ont pas d'état interne.

Les modèle de composants étant réflexif, les algorithmes peuvent vérifier des contraintes sur l'architecture des applications, le framework n'apporte cependant pas de facilité de spécification ou de vérification de ces contraintes. De plus, aucun mécanisme ne permet d'annuler l'exécution d'un algorithme en cas d'erreur. La concurrence entre reconfigurations est géré simplement par leur exécution en série.

3.3 Plateformes construisant un modèle d'architecture

Cette section regroupe des travaux autour de plateformes supportant la reconfiguration dynamique de systèmes à base de composants. Ces plateformes sont plus ou moins adhérentes à un modèle de composants particulier.

3.3.1 Plastik

Description

Introduction. Plastik [BJC05] 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 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 [GMW00]/Aramani [Mon01]) et un modèle de composant réflexif avec son environnement d'exécution (OpenCOM [CBG+04]).

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 interfaces fournies sont appelées réceptacles et peuvent être constituées de 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. Le modèle met également en avant le concept de « Component Framework » qui sont des composants composite regroupant des composants focalisés sur un domaine ou une fonctionnalité spécifique telle que l'ordonnancement des threads par exemple.

ADL et contraintes. Le langage ACME est utilisé pour la description des architectures. ACME est un ADL générique, focalisé sur les aspects structurels, et suffisamment riche d'un point de vue des concepts d'architecture manipulés pour pouvoir décrire des architectures spécifiées dans d'autres langages et servir ainsi de langage pivot (en Anglais : « architecture description interchange language »). Outre les concepts classiques de composant, de port, de connecteur, de rôle et de système (assemblage de composants et de connecteurs), 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'architecture. Le système de typage permet de mutualiser et d'étendre la spécification des éléments d'architecture et peuvent être instancié dans des systèmes. Les familles regroupent des ensembles de définitions de types et décrivent les moyens autorisés pour les composer. Pour compléter ACME, Armani est utilisé pour l'expression de contraintes sur la composition des architectures sous la forme d'invariants en logique du premier ordre (cf. Figure 3.5).

```
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 3.5 – Exemple de spécification en ACME avec des contraintes en Armani

Reconfigurations dynamiques. La combinaison de l'ADL ACME et du langage de contraintes Armani ne supporte pas de base l'expression de la reconfiguration dynamique de système à la différence du modèle de composants OpenCOM. Les invariants Armani sont utilisés pour garantir que des contraintes architecturales sont bien préservées malgré les reconfigurations. 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 pour la description des configurations statiques. La plateforme d'exécution de Plastik (cf. figure 3.6) définit deux niveaux de spécification ADL, le niveau style pour spécifier des motifs 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 [JBCG05].

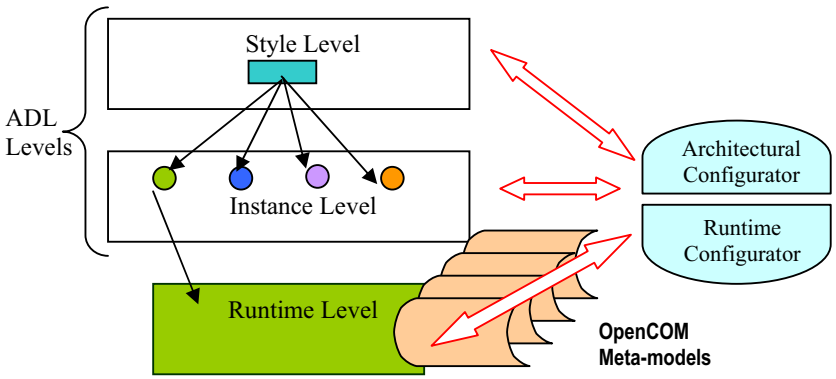

FIGURE 3.6 – Architecture de la plateforme Plastik

Le framework supporte à la fois les *reconfigurations programmées*, prévues au moment de la conception du système, et les *reconfigurations ad-hoc* qui ne sont au contraire pas anticipées. Pour gérer les reconfigurations programmées, une nouvelle primitive est ajoutée au langage ACME pour spécifier une condition de déclenchement de la reconfiguration sous forme de prédicats Armani du type :

```
On (<predicate>) do <action>
```

Les reconfigurations ad-hoc peuvent être exécutées soit par des scripts de modification d'architec-

ture spécifiés en ACME étendu, soit directement par l'invocations d'opérations en OpenCOM. Les scripts sont appliqués à la spécification du système cible qui est ensuite recompilée pour produire un script de différences (« diff script ») directement exécutable dans le système en OpenCOM. Pour toutes les reconfigurations, l'exécution des actions de reconfiguration est réalisée séquentiellement de manière transactionnelle, la plateforme vérifie qu'aucune contrainte architecturale n'est violée par les modifications du système.

Evaluation

Plastik propose une approche langage reposant sur l'utilisation d'un ADL étendu pour la spécification de reconfigurations et d'un langage de contrainte en logique du premier ordre. La solution permet ainsi l'exécution de reconfigurations programmées et non-anticipées de système à base de composants OpenCOM.

Le modèle de composants sous-jacent est réflexif et extensible. En revanche, l'extension du langage pour prendre en compte de nouvelles opérations de reconfiguration ne semble pas aussi simple. Par ailleurs, si les changements au niveau de l'ADL sont bien reflétés dans le système, la connexion causale avec la représentation ADL n'est pas maintenue si les changements sont effectués au niveau de l'exécution.

Les contraintes sont limitées à des invariants, il n'est pas possible d'exprimer des préconditions ou postconditions sur les opérations de reconfiguration. Les invariants architecturaux sont dans un premier temps vérifiés sur la représentation sous forme d'ADL avant toute exécution d'une opération de reconfiguration. De plus, les opérations de reconfiguration sont exécutées en séquence et de manière transactionnelle pour garantir leur atomicité. Aucun détail n'est cependant donné sur d'autre propriétés éventuellement garanties par le modèle de transaction, notamment en terme d'isolation et de durabilité.

3.3.2 Rainbow

Description

Introduction. Les canevas logiciel Rainbow [GCH+04] propose de réaliser l'adaptation dynamique de l'architecture des systèmes en permettant de spécifier des stratégies d'adaptation en fonction de différentes préoccupations. Une boucle de contrôle est mise en place avec un modèle d'architecture pour observer un système à l'exécution, évaluer la violation de contraintes dans ce système et réagir en conséquence en adaptant le système.

Modèle d'architecture. Rainbow utilise un modèle d'architecture réflexif en représentant les systèmes à l'exécution sous forme d'un graphe d'éléments. Les noeuds du graphe sont des composants, potentiellement hiérarchiques et dotés d'interfaces, et les arcs sont les connecteurs, chemin d'interaction et de communication entre les composants. Les éléments architecturaux peuvent être annotés par des propriétés précisant leur comportement. Les décisions d'adaptation reposent dès lors sur l'observation de la topologie des systèmes exposée par le modèle. Il est également possible d'exprimer des contraintes sur les architectures. Une modification dans un système n'est alors valide que si l'architecture résultante du système satisfait bien les contraintes. L'implémentation du modèle pour la spécification d'architecture est réalisée par le langage de description d'architecture ACME [GMW00].

Architecture du framework. Le framework (cf. figure 3.7) est divisé d'une part en une infrastructure d'adaptation réutilisable pour différents systèmes et d'autre part en les informations spécifiques à un système et nécessaire à l'adaptation. L'infrastructure d'adaptation est composée de trois couches. La couche système définit les interfaces d'observation des propriétés du système à l'aide de sondes, des mécanismes de découverte des ressources et d'action pour modifier les systèmes. La couche d'architecture expose le modèle d'architecture du système adapté et contient un évaluateur de contraintes qui évalue périodiquement les contraintes d'architecture et déclenche l'adaptation en cas de violation. Cette adaptation est effectuée par un moteur d'adaptation. La couche de traduction réalise un mapping entre le système et son modèle architectural et réciproquement.

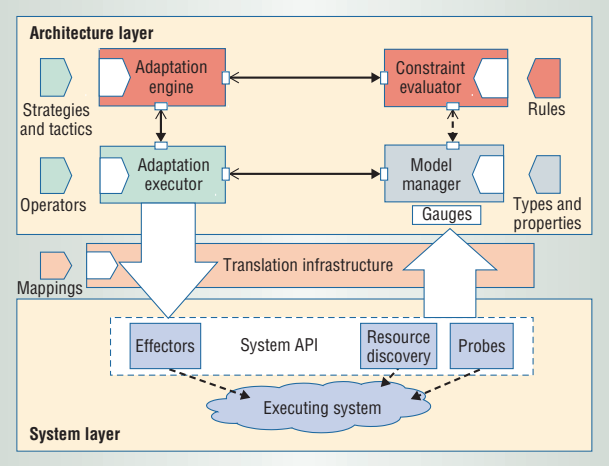

FIGURE 3.7 – Architecture du framework Rainbow

Reconfigurations dynamiques et garanties. Rainbow met en avant le concept de style architectural pour contraindre l'adaptation des architectures des systèmes. Le framework étend cette notion statique pour supporter l'adaptation dynamique avec des styles d'adaptations. Un style d'adaptation spécifie les opérations de reconfiguration (ou opérateurs d'adaptation) qui peuvent s'exécuter sur un système donné et comment les composer suivant une stratégie pour obtenir l'effet voulu dans le système. Les opérateurs d'adaptation définissent ainsi un ensemble d'actions spécifiques à un style pour reconfigurer les architectures comme par exemple des opérations d'ajout et de retrait de composants. Les stratégies d'adaptation spécifie les opérations à exécuter en cas de violation de contraintes (cf. figure 3.8) sous forme d'invariants sur les valeurs des propriétés et la topologie des systèmes. Les invariants sont exprimés en logique du premier ordre avec le langage de contraintes Armani [Mon01] qui étend l'ADL ACME.

```
invariant (bandwidthToHHP (self)
          > minHHBandwidth)
  !→ HHBandwidthStrategy(self);
```

(a) Contrainte déclenchant l'adaptation

```
strategy HHBandwidthStrategy
           (HandheldT HH) {
  let HHP1 = findBestHHP(HH);
  HH.move(HHP1);
  return true;
}
```

(b) Stratégie d'adaptation

FIGURE 3.8 – Exemple de stratégie d'adaptation déclenchée par la violation d'une contrainte

Application à l'auto-réparation. Une application possible de Rainbow est l'adaptation des systèmes pour l'auto-réparation et l'auto-optimisation. Le framework est ainsi étendu dans [GS02] pour la conception d'une boucle autonomique pour la détection et la réparation de fautes (cf. Figure 3.9). La description architecturale du système est utilisée comme base de diagnostic et de réparation des fautes car elle fournit une vue abstraite et globale du système en cours d'exécution et joue le rôle de support pour raisonner sur les fautes et leurs réparations. Pour un système à l'exécution (1), l'approche consiste à observer son comportement (2) en rapport avec des propriétés du modèle architectural (3). Les modifications des propriétés déclenchent l'évaluation des contraintes (4) sur le système pour déterminer si les valeurs des propriétés restent dans un domaine acceptable. En cas de violation de contraintes, un mécanisme de réparation (5) adapte l'architecture. Les modifications sont finalement propagé dans le système à l'exécution (6). Cette approche est par exemple applicable pour le maintien d'une latence au dessous d'un seuil acceptable dans une application client-serveur.

Evaluation

Rainbow adopte une approche basée sur les styles architecturaux pour contraindre les adaptations dynamiques de systèmes. Il étend le concept de styles avec le concept dynamique de style

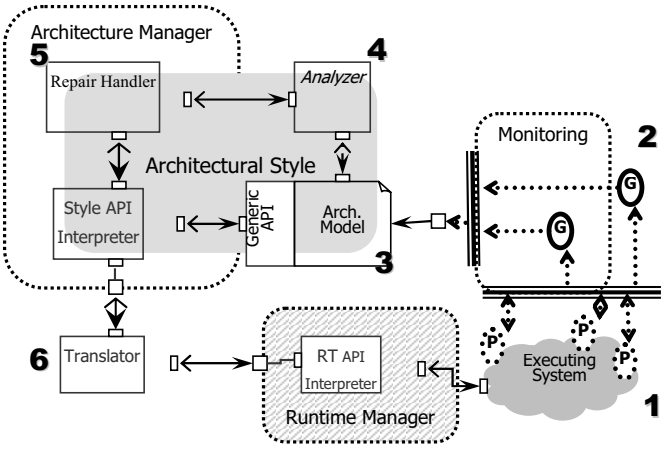

FIGURE 3.9 – Architecture du framework Rainbow

d'adaptation consistant en un ensemble d'opérations de reconfigurations spécifiques et des stratégies d'adaptation qui spécifie des contraintes au niveau du modèle architectural des systèmes et le code de l'adaptation en cas de violation de contraintes.

Le modèle d'architecture est ouvert car il est possible de définir ses propres opérations primitives de reconfiguration pour un style donné. En revanche, aucune facilité n'est donnée pour la spécification de la sémantique de ces opérations, seule leur composition est explicite à l'intérieur du code des stratégies d'adaptation. Le mapping effectué entre le système et le modèle d'architecture n'étant pas générique, la difficulté de sa réalisation n'est pas abordée.

La garantie de la cohérence des adaptations dynamiques repose sur l'association des reconfigurations à des stratégies. Une reconfigurations ne peut se produire que si les conditions de l'adaptation sont remplies, i.e. si certaines contraintes sont satisfaites dans le modèle d'architecture. Cependant, la solution choisie limite les reconfigurations à des reconfigurations programmées, il ne semble en effet pas possible de définir de nouvelles adaptations dynamiquement dans le système et donc d'exécuter des reconfigurations non-anticipées. De plus, aucun mécanisme ne permet de garantir l'atomicité des reconfigurations, il n'apparaît pas possible de définir de stratégie générique pour ce type de propriété. Enfin, le modèle d'exécution pour évaluation des contraintes n'est pas précisé (mode push ou pull).

L'infrastructure de Rainbow est centralisée mais peut être déployée dans des systèmes distribués avec des limitations en terme de passage à l'échelle et de tolérance aux pannes. De plus, la concurrence des adaptations doit être gérée par le programmeur des stratégies d'adaptation pour éviter les conflits avec les autres stratégies.

3.3.3 ArchStudio

Description

Introduction. L'approche présentée dans [DvdHT02] propose de construire des systèmes capables de s'auto-réparer à l'aide d'un framework conçu pour l'adaptation d'architectures à composants dont les communications sont basées sur des événements en suivant le style C2. La solution est implémentée et intégrée dans un environnement de développement d'architecture logicielles appelé ArchStudio. Les objectifs sont les suivants :

- décrire l'architecture des systèmes,
- exprimer des modifications arbitraires d'architecture pour servir de plan de réparation,
- analyser le résultat de la réparation et garantir sa validité,
- exécuter le plan de réparation sur un système à l'exécution sans avoir à le redémarrer.

Spécification des architectures et de leur modifications. Les architectures des systèmes considérés sont décrite en xADL 2.0 [DvdHT01], un ADL extensible avec une syntaxe basé sur XML. Un système est ainsi instancié à partir de la description de son architecture grâce à un mapping existant entre le langage xADL 2.0 et des classes Java. Un composant de gestion des évolutions

(Architecture Evolution Manager) maintient la correspondance entre le modèle d'architecture et l'architecture du système (cf. figure 3.10). Une reconfiguration architecturale est exprimée sous forme d'une différence architecturale (*diff*) qui décrit les différences entre des spécifications d'architecture en ADL. Dans le contexte de l'auto-réparation, le *diff* décrit la différence entre l'architecture du système avant et après la réparation. Un *diff* est exprimé dans une extension du langage xADL 2.0 et permet simplement de spécifier l'ajout ou le retrait d'éléments de l'architecture (composants, connecteurs, types). Un changement à l'intérieur d'un élément est donc traité comme un remplacement. L'outil ArchDiff prend ainsi deux définitions d'architecture en paramètre pour retourner le *diff* correspondant. Un outil complémentaire, ArchMerge, permet d'appliquer un *diff* à une architecture de base. Il est alors possible d'appliquer un « patch architectural »à une architecture pour la faire évoluer. Cependant, l'application d'un patch ne conduit pas toujours à une architecture valide, des connecteurs peuvent par exemple à l'issue de la modification se trouver déconnectés d'un côté.

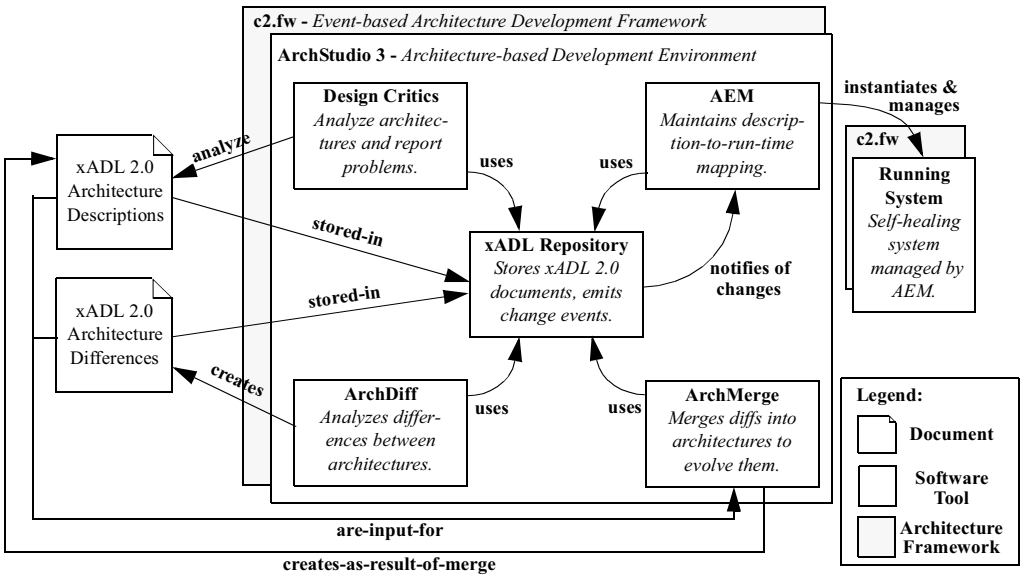

FIGURE 3.10 – Gestion des reconfigurations dans ArchStudio 3

Validité des reconfigurations. La vérification de la cohérence des architectures après modification est du ressort de composants appelés « design critics ». Chaque « design critic » observe les changements dans l'architecture et lorsqu'un changement est détecté, il vérifie que d'éventuelles contraintes sont bien satisfaites pour une préoccupation déterminée. Ces analyses sont de plus composables. Les composants sont implémentés en Java suivant le style architectural événementiel de C2 [TMA+95]. Le choix de la communication par événement permet de simplifier la synchronisation entre reconfigurations et exécution du système. Une politique de synchronisation est définie pour suspendre les communications avec les composants sur le point d'être retirés de l'architecture.

Exécution des réparations. Lors de la détection d'un faute dans le système instancié (cf. figure 3.11), un plan de réparation constitué de modifications de l'architecture du système est conçu. Une copie de la description de l'architecture du système est faite avant l'exécution de la réparation. Une différence d'architecture réalisant la réparation est alors appliquée sur la copie de l'architecture. Cette opération déclenche la vérification de la validité de l'architecture résultante. Si les contraintes sont satisfaites, la différence architecturale est alors appliquée sur l'architecture du système à l'exécution. En cas d'architecture invalide, un nouveau plan de réparation peut être construit.

Evaluation

Différentes techniques et outils on été intégrés dans l'environnement ArchStudio pour construire et gérer des systèmes auto-réparables. L'originalité de l'approche tient à un mécanisme de création de

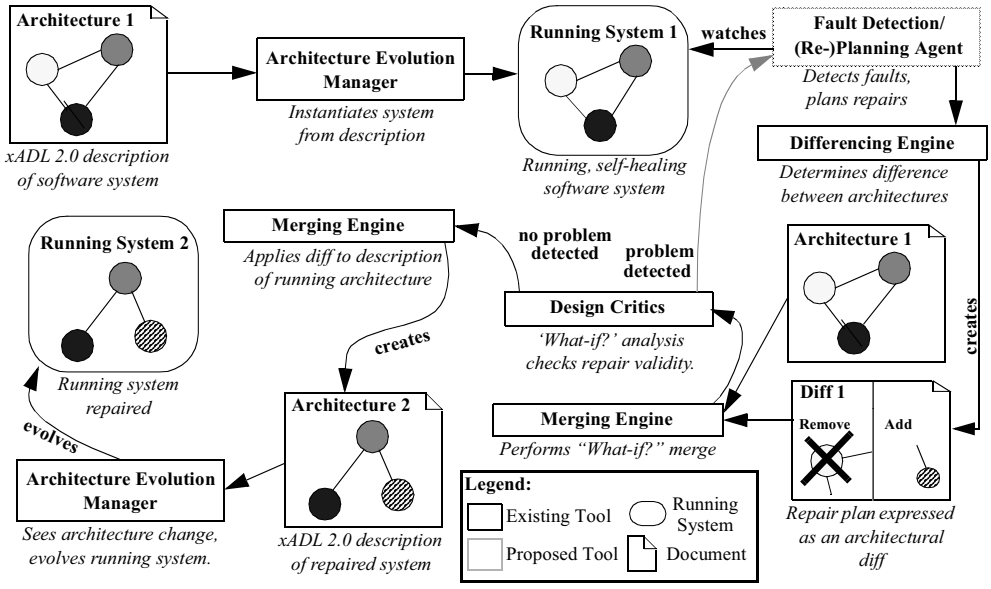

FIGURE 3.11 – Processus d'auto-réparation

patchs architecturaux pour spécifier les reconfigurations des systèmes à réparer. Par ailleurs, l'adaptation est spécifiée en terme de but à atteindre et non en termes d'opérations à exécuter. En effet, l'état initial représente l'architecture du système à réparer et l'architecture finale est l'architecture objectif à atteindre pour réparer le système. Les transitions pour atteindre l'état final sont automatiquement déduite lors de l'opération de différence architecturale. L'application de ces patchs est réalisée de manière transactionnelle.

L'approche comporte cependant plusieurs limitations. Les systèmes considérés sont centralisés, la réparation de systèmes distribués pose de nouveaux problèmes notamment pour ce qui est de la tolérance aux fautes. Le style architectural utilisé est figé (style C2) et nécessite des systèmes avec uniquement des communications événementielles. Enfin, les seules opérations possibles sont les retraits et les ajouts d'éléments dans les architectures.

3.3.4 JADE

Description

Introduction. JADE [BHQS05] est un canevas logiciel dédié à la construction de systèmes distribués auto-administrables. Les éléments dans JADE forment une architecture de boucle de contrôle. Ces boucles sont en charge de la réparation et de l'optimisation du comportement du système administré. JADE a été notamment utilisé dans le contexte de l'auto-réparation dans un cluster de serveurs d'application J2EE.

Modèle de composants et reconfigurations. Le système administré est construit à l'aide de composants Fractal [BCL+06] et utilise l'ADL Fractal pour décrire son architecture. Fractal est un modèle de composants qui permet l'implémentation, le déploiement et l'observation de systèmes incluant en particulier les intergiciels. Le modèle est hiérarchique, réflexif et ouvert. JADE repose sur Fractal pour l'introspection de l'architecture des systèmes à réparer et sur son support des reconfigurations dynamiques pour exécuter les réparations. Toutes les ressources du système aussi bien matérielles (noeuds dans un cluster) que logicielles (serveurs, applications, etc.) sont des composants et leurs dépendances sont exprimées à l'aide de propriétés d'encapsulation et de liaisons.

Boucle de réparation. L'architecture des système cibles est divisée en domaines d'administration. Un domaine de réparation est ainsi défini par une ensemble de composants, appelés noeuds, qui

sont des abstractions de machines physiques. Les opérations sur les noeuds dans le cadre d'une réparation consistent à l'introduction d'un nouveau noeud et à récupérer des informations sur un noeud défaillant. L'introduction d'un nouveau noeud est assimilé à l'instanciation d'un nouveau composant dans son état initial. Les sous-composants à l'intérieur d'un noeud peuvent être également ajoutés ou retirés. La boucle d'auto-réparation (cf. figure 3.12) comporte un mécanisme d'observation du système administré qui permet notamment de détecter des défaillance de noeuds en cas de panne franche de machine. Le gestionnaire autonomique implémente l'algorithme d'analyse et de décision suite à la remonté d'événements particuliers, le gestionnaire de réparation implémente ainsi la politique de réparation choisie. Dans le contexte d'un cluster J2EE, cette politique peut ainsi traiter des défaillances de composants J2EE à l'intérieur de noeuds en les redéployant sur le même noeud. En cas de panne machine, le gestionnaire demande l'allocation d'un nouveau noeud dans le cluster et installe une configuration identique à celle de l'ancien noeud sur le nouveau noeud.

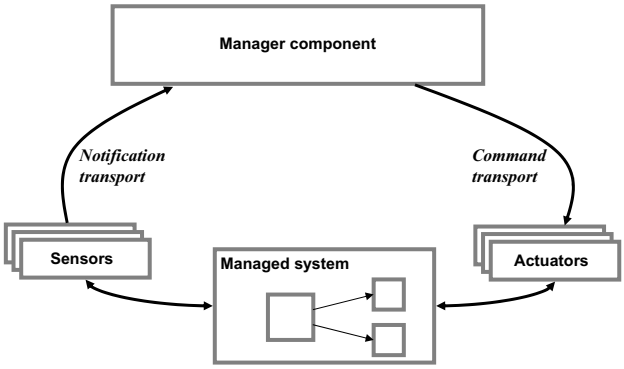

FIGURE 3.12 – Boucle de contrôle autonomique de Jade

Représentation du système. L'élément central du mécanisme d'auto-réparation est la représentation du système administré appelée « représentation système » (*system representation*). La représentation système maintient une vue abstraite, introspectable et modifiable du système à l'exécution correspondant à la description de son architecture courante. Elle est construite automatiquement à partir de la description ADL du système administré et consiste en une configuration de composants isomorphique à la configuration du système. Une connexion causale est maintenue entre le système administré et sa représentation. Par conséquent, toute modification de l'état du système est reflétée dans sa représentation et réciproquement, toute modification de la représentation système est répercutée dans l'architecture du système à l'exécution. Les actions de réparation (ajout et retrait de noeuds) sont ainsi exécutées par le gestionnaire de réparation sur la représentation système pour modifier le système lui-même. Cette représentation sert également à remonter les changements d'état du système administré au gestionnaire autonomique et permet en cas de panne franche de noeud de retrouver par introspection de la représentation la configuration du noeud défaillant pour le redéployer sur une autre machine.

Evaluation

JADE[1] ne vise pas directement à garantir la fiabilité des reconfigurations dynamiques dans les systèmes à composants mais à rendre ces systèmes tolérant aux fautes logicielles et matérielles par l'intermédiaire d'une boucle d'auto-réparation. Cependant les actions de réparation à l'intérieur de cette boucle sont vues comme des reconfigurations de l'architecture des systèmes administrés. L'originalité de JADE est le maintien d'une représentation isomorphique et causalement connectée de l'architecture d'un système pour observer les changements d'état de ce système et pour modifier dynamiquement son architecture.

La représentation système est une réplication de la configuration du système administré, elle permet ainsi de garantir la durabilité des reconfigurations dynamiques exécutées dans un système

1. JADE a été étendu dans le cadre du projet collaboratif ANR Selfware (`http://sardes.inrialpes.fr/selfware/`), contexte dans lequel s'est effectuée cette thèse, et intégré dans la plateforme d'administration Jasmine (`http://wiki.jasmine.ow2.org/`).

distribué en cas de panne franche de machine. La représentation est en effet toujours accessible par un mécanisme de réplication active, elle peut donc être introspectée pour retrouver l'architecture du système instancié sur les noeuds défaillants.

3.4 Environnements pour la gestion de l'évolution des architectures

Nous considérons dans cette section des environnements pour spécifier et vérifier l'évolution des architectures logicielles. Même si les reconfigurations de systèmes à l'exécution ne sont pas prises en charge directement par ces travaux, des propriétés intéressantes y sont garanties sur les évolutions possibles des architectures.

3.4.1 Mae

Description

Introduction. Mae [RHMRM04] est un environnement de gestion de l'évolution des architectures logicielles qui repose sur des concepts des systèmes de gestion de configuration logicielle. Il permet ainsi de spécifier des architecture sous forme d'un langage de description d'architecture, de garder un historique des modifications effectuées grâce à un mécanisme de versionnement et de « check-out / check-in », de sélectionner une version particulière d'une configuration et d'en faire une analyse de la cohérence.

Modèle d'architecture. Le modèle d'architecture dans Mae est un modèle théorique qui reprend les éléments couramment contenus dans les ALDs existants [MT00] : composants, interfaces et connecteurs. Il supporte les architectures hiérarchiques et propose une séparation forte entre types et instances. Les types de composants sont composés de l'ensemble des types de ses interfaces (services fournis et requis) et du contenu architectural des composants constitué d'une hiérarchie de composants et de connecteurs. De plus, des contraintes et des comportements peuvent être ajoutés pour restreindre les façon d'accéder aux interfaces d'un composant. Les contraintes portent sur les valeurs des paramètres lors de l'invocation de services, les comportements spécifient des protocoles d'interaction entre les composants pour invoquer les services. Le modèle est implémenté sous forme d'extension de l'ADL xADL 2.0 [Das02] dont la syntaxe est basée sur XML et est organisée en un ensemble de schémas modulaires pour la description des différents éléments d'architecture.

Spécification de l'évolution des configurations. Pour supporter l'évolution des architectures, Mae introduit dans son modèle d'architecture des notions pour modéliser l'optionalité, la variabilité et l'évolution. L'optionalité désigne la présence nécessaire ou non d'un élément dans une architecture. Pour spécifier la condition de présence, une garde booléenne est associée aux éléments du modèle. Cette garde est évaluée au cours d'un processus de sélection et en fonction de propriétés choisies. La variabilité traduit la coexistence de plusieurs versions d'un même élément architectural. Cette propriété regroupe deux types de variabilité : la divergence et la convergence. La convergence concerne les éléments dont l'évolution se sépare en plusieurs branches différentes. Une nouvelle branche est formée en créant un nouveau type qui suit dès lors sa propre évolution. La convergence regroupe des branches différente de l'évolution d'un élément à des points particuliers de variation. Pour capturer l'évolution des architectures, Mae utilise le versionnement et le sous-typage. Tous les types des éléments sont versionnés par un numéro de révision. Il est également indiqué si la propriété de sous-typage entre les versions des types est conservée de manière pour informer sur la substituabilité des éléments entre versions.

Processus d'évolution. L'environnement d'évolution des architectures définit le processus de création et d'évolution des architectures. Il est composé d'une bibliothèque pour la manipulation de documents XML décrivant des architectures en xADL 2.0 (cf. figure 3.13) ainsi que de trois modules pour la conception, la sélection et l'analyse des éléments d'architecture. Le module de conception

permet de représenter et d'éditer graphiquement des architectures avec des fonctionnalités de versionnement. Une fois enregistrée, une version d'un élément est immutable pour garantir l'intégrité des autres éléments qui en dépendent. Le module de sélection permet de sélectionner des configurations suivant une version spécifiée. L'outil peut également choisir automatiquement une version d'élément adaptée en fonction des propriétés attendues spécifiée sous forme de paires attribut-valeur. Le module d'analyse permet de détecter des incohérences dans les configurations architecturales en fonctions des contraintes et des comportements spécifiés. Les analyses peuvent inclure la vérification de contraintes lié à la topologie de l'architecture par l'ajout un plugin de vérification de style. L'analyse sémantique utilise des techniques de vérification de type en vérifiant que les services fournis et requis sont compatibles.

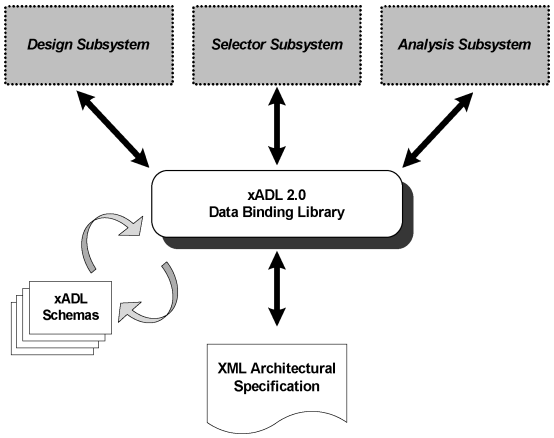

FIGURE 3.13 – Architecture de Mae

Evaluation

Mae adopte une approche originale pour la gestion de l'évolution des architectures logicielles en reprenant des concepts présents dans les systèmes de gestion de configuration pour capturer les modifications du code source des applications. Mae introduit ainsi un mécanisme de versionnement des éléments d'architecture ainsi qu'un modèle qui capture l'évolution de ces éléments. L'évolution suit un processus défini qui permet notamment de vérifier la cohérence des configurations construites à partir de différentes versions d'éléments architecturaux grâce à la vérification de contraintes et de la relation de sous-typage.

La solution proposée se concentre sur l'évolution des architectures à un niveau d'abstraction indépendant de l'implémentation concrète des systèmes reconfigurés à travers une représentation en ADL. Par conséquent, les problèmes de l'exécution concrète des reconfigurations dans le système ainsi que le maintien de la cohérence entre le système à l'exécution et son modèle architectural ne sont pas abordés.

L'environnement de gestion de l'évolution d'architecture possède des bonnes propriétés issues des systèmes de gestion de configuration. Les éléments versionnés sont immutables afin d'éviter de créer des incohérences des éléments qui en dépendent. Des modifications concurrentes de la même version d'un élément aboutiront à deux branches différentes. Le maintien d'un historique des versions de manière durable permet éventuellement de revenir à une configuration précédente pour par exemple défaire des modifications indésirables. La cohérence des reconfigurations repose sur le principe de substituabilité grâce au sous-typage, à la vérification de la satisfaction de contraintes au niveau des interfaces des composants, et à des gardes booléennes pour gérer l'optionalité des éléments.

3.4.2 C2SADEL

Description

Introduction. C2SADEL [Med99] (Software Architecture Description and Evolution Language) est un langage dédié au support de l'évolution des architectures logicielles à travers des mécanismes de sous-typage des éléments architecturaux. Le typage dans les architectures permet de restreindre les évolutions possibles des éléments grâce à la relation de sous-typage et la propriété de substituabilité d'un élément par un autre repose sur la vérification de types.

Sous-typage et vérification des types. Les types d'architecture se distinguent des types de données classiques car leurs instances ne peuvent être échangées ni par valeur ni par copie entre les éléments d'une configuration architecturale. Un type de composant comprend un nom, un ensemble d'interfaces requises ou fournies, un comportement associé et une éventuelle implémentation. Le comportement d'un composant consiste en des invariants et un ensemble d'opérations. Les invariants spécifient des propriétés qui doivent rester vraies quelque soit l'état du composant. Chaque opération possède des préconditions et des postconditions et éventuellement un résultat pour exprimer leur sémantique attendue. La relation de sous-typage pour les composants peut alors être définie suivant plusieurs niveaux de conformité :

- Conformité des noms : un sous-type partage avec son super-type les interfaces et les noms des paramètres
- Conformité du comportement : les invariants du super-type sont vérifiés dans le sous-type et chaque opération fournie dans le super-type doit correspondre avec une opération fournie dans le sous-type avec les mêmes ou plus faibles préconditions ou moindre, les mêmes ou plus fortes postconditions tout en préservant la covariance du résultat. La relation est inversé pour les interface requises. Ce niveau de conformité garantie la substituabilité des composants entre un super-type et un sous-type de composant dans une architecture.
- Conformité de l'implémentation : les opérations dans le sous-type ont la même implémentation que dans le super-type, la syntaxe des interfaces pouvant être différente.

La spécification des architectures suit le style de C2 [TMA+95] : les connecteurs sont modélisés explicitement, un composant ne peut être connecté qu'à un seul connecteur « en haut » (*top*) et « en bas » (*bottom*) et un connecteur peut être attaché à plusieurs composants et connecteurs. Les communications se font uniquement par messages. Les messages de notification de changement d'état descendent dans l'architecture tandis que les messages de requête remontent dans l'architecture.

Environnement de développement. Une architecture en C2SADEL est définie par des types de composants, des types de connecteurs, et une topologie qui définit des instances de composants et de connecteurs et leurs interconnexions. Un environnement appelé DRADEL (cf. figure 3.14) a été conçu pour la modélisation, l'analyse, l'évolution et l'implémentation d'architectures spécifiées en C2SADEL. Plusieurs vérifications sont effectuées sur les architectures par différents composants :

- L'*InternalConsistencyChecker* effectue un certain nombre de vérifications de base sur l'architecture, par exemple que les connecteurs sont bien connectés dans l'architecture.
- Le *TopologicalConstraintChecker* s'assure que les règles définies dans le style C2 sont bien respectées.
- Le *TypeChecker* exécute deux analyses : la vérification des relations de typage entre les composants et les connecteurs connectés dans l'architecture et des relations de sous-typage entre les ensemble de types de composants.

L'environnement possède des capacités de génération des squelettes de code en Java à partir de l'architecture pour implémenter les applications.

Reconfigurations dynamiques. Des travaux on été menés pour introduire des éléments de spécification de reconfiguration dynamique des architectures en C2SADEL sous forme d'un langage de modification d'architectures (Architecture Modification Langage ou AML [OMT98]). Une extension de l'ADL est proposée dans [Med96] pour l'ajout, le retrait et le remplacement de composants tout en respectant les relations de sous-typage.

Evaluation

C2SADEL est un langage de description d'architecture qui suit le style C2 associé à un modèle de type pour les éléments d'architecture. L'environnement DRADEL permet de spécifier les architectures et de réaliser un certain nombre de vérification notamment en terme de typage. La vérification

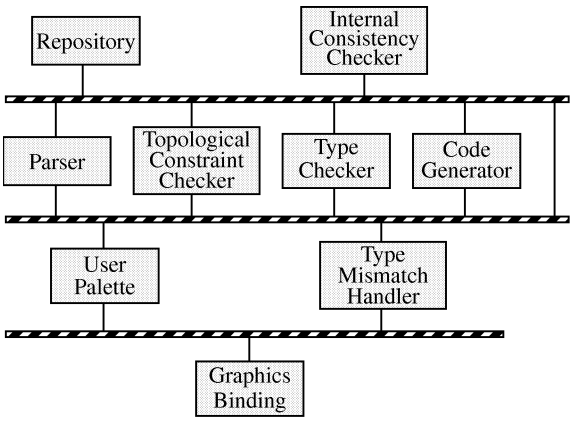

FIGURE 3.14 – Architecture de l'environnement DRADEL en style C2

de sous-typage entre les évolutions des éléments d'une architecture est ainsi un moyen d'en maintenir la cohérence. Les différents niveaux de conformité dans la relation de sous-typage autorise une flexibilité en fonction des propriétés attendues.

L'approche a pour limitation principale de ne pas gérer directement la reconfiguration des systèmes à l'exécution (hormis les travaux mentionnés de [Med96]) même si l'environnement comporte des capacités de génération de code pour l'implémentation des composants dans un langage de programmation généraliste. D'autre part, les architectures sont limitées au style C2 sans pouvoir suivre d'autres styles architecturaux.

3.5 Synthèse

Nous synthétisons nos évaluations des travaux dans quatre tableaux différents. Un premier tableau concerne l'évaluation des modèles de composants et de leur propriétés dans les différentes plateformes considérées. Un deuxième tableau présente la gestion de la cohérence dans les architectures des systèmes reconfigurés. Un troisième tableau recense les reconfigurations dynamiques supportées. Enfin, un quatrième et dernier tableau analyse les propriétés garanties par les reconfigurations dynamiques ou les évolutions d'architecture.

La légende utilisée pour l'ensemble des tableaux est la suivante :

- Les cases marquées « n/a » indiquent que le critère correspondant n'est pas applicable.
- Un symbole « ? » signifie que nous n'avons pas assez d'informations pour en faire une évaluation.
- Une case vide signifie que la fonctionnalité ou la propriété n'est pas prise en compte dans la plateforme évaluée.
- Une case marquée d'un symbole « x » indique que la fonctionnalité ou la propriété est prise en compte dans la plateforme évaluée. Des commentaires accompagnent éventuellement le symbole pour préciser l'information.
- Les symboles « - » et « + » permettent de nuancer certains critères d'évaluation. Les symboles peuvent être répétés pour renforcer la nuance.

Modèles de composants. Le tableau 3.1 présente les différents modèles de composants utilisés dans les travaux évalués. Les trois premières plateformes (FORMAware, K-Component et OpenRec) reposent directement sur un modèle de composant réflexif pour l'introspection et la modification des architectures à l'exécution avec connexion causale. Pour les quatre travaux suivants (Plastik, Rainbow, ArchStudio et JADE), le modèle d'architecture est plus ou moins dépendant d'un modèle de composants particulier pour l'implémentation des systèmes reconfigurés. Un ADL sert en effet de point d'entrée pour l'introspection et les modifications des architectures, ces modifications étant reflétées dans le système à l'exécution, le reflet des modifications directe dans le système ne sont par contre pas toujours prises en compte. Les deux derniers travaux (Mae et C2SADEL) ne gèrent pas la correspondance entre évolution d'architecture et reconfiguration d'un système réel.

	Objectif et domaine d'application	Modèle de composants pour l'implémentation des systèmes reconfigurés	Langage de configuration	Eléments d'architecture	Cohérence entre modèle architectural et système à l'exécution		Modèle hiérar-chique	Modèle ouvert	Support de systèmes distribués
					modèle -> runtime	runtime -> modèle			
FORMAware	Automatisation du développement de l'adaptation de systèmes à composants	FORMAware	Java	Graphe implicite de composants	x	x	x	++	+
K-Component	Programmation sûre de systèmes adaptatifs	K-Component	C++	Graphe explicite de composants et de connecteurs	x	x		--	+
OpenRec	Utilisation de mécanismes génériques et ouverts de reconfigurations dynamiques	OpenRec	OpenRecML	Graphe implicite de composants et de connecteurs	x	x	?	+	+
Plastik	Construction et administration de systèmes dynamiquement reconfigurables	OpenCOM	ACME	Graphe implicite de composants et de connecteurs	x		x	+	+
Rainbow	Programmation de mécanismes externes d'adaptation dans des systèmes	Implémentation ouverte	ACME	Graphe implicite de composants et de connecteurs	x	Unique-ment les propriétés (sondes)	x	++	+
ArchStudio	Auto-réparation de systèmes avec architectures à composants événementielles	C2 (implémentation c2.fw en Java)	xADL 2.0	Graphe implicite de composants et de connecteurs	x	x		-	-
JADE	Auto-réparation avec application aux clusters J2EE	Fractal (implémentation Julia en Java)	Fractal ADL	Graphe implicite de composants	x	x	x	++	+
Mae	Gestion contrôlée de l'évolution des architectures	n/a	xADL 2.0	Graphe implicite de composants et de connecteurs	n/a	n/a	x	+	n/a
C2SADEL	Gestion de l'évolution des architectures basée sur le typage	n/a[1]	C2SADEL	Graphe implicite de composants et de connecteurs	n/a	n/a		-	n/a

1. Génération de squelette de code java suivant le style C2

TABLE 3.1 – Modèles de composants et propriétés

Cohérence des architectures. La deuxième partie de l'évaluation concerne les moyens de spécifier la cohérence des architectures des systèmes reconfigurés et de vérifier cette cohérence. La plupart des travaux utilise des styles architecturaux pour contraindre les architectures qui associent système de typage plus ou moins complexe des éléments d'architecture et contraintes structurelles voire comportementales. Les contraintes servent à définir la cohérence d'une architecture donnée et donc à garantir la validité des reconfigurations en vérifiant la satisfaction de ces contraintes dans les architectures reconfigurées. Nous résumons les différents types de contraintes disponibles, les moyens de les spécifier et de les vérifier dans le tableau 3.2.

	Type de contraintes	Langage de contraintes	Typage et restrictions de l'évolution des éléments d'architecture	Mécanismes de vérification et réaction en cas de violation de la cohérence de l'architecture du système
FORMAware	- Préconditions quelconques sur les opérations de reconfiguration	Java	- Typage lié au style défini	Exécution transactionnelle des opérations de reconfigurations avec vérification de préconditions avant chaque exécution d'opération
K-Component	- Propriétés et assertions sur les configurations dans des contrats d'adaptation	ACDL	- Typage des composants et des connecteurs	La violation de contraintes déclenche l'adaptation du système décrite dans les contrats d'adaptation
OpenRec	- Invariants structurels	?	?	?
Plastik	- Invariants structurels en FOL dans les configurations	Armani	- Typage des composants et des connecteurs dans des styles architecturaux	Vérification des invariants avant l'exécution des reconfigurations dans le système à l'exécution et annulation en cas de violation (exécution transactionnelle)
Rainbow	- Invariant structurels et sur les valeurs des propriétés en FOL dans les configurations	Armani	- Typage des composants et des connecteurs dans des styles architecturaux	La violation de contraintes déclenche l'adaptation du système suivant une stratégie définie
ArchStudio	- Contraintes de style et invariants structurels - Autre ?	?	- Typage des composants et des connecteurs	Vérification des invariants après reconfiguration d'une copie du modèle d'architecture avant de appliquer les modifications dans le système en cas de succès.
JADE			- Typage des composants à travers les types de leurs interfaces	
Mae	- Préconditions et postconditions sur les valeurs des paramètres des opérations dans les interfaces des composants - Invariants de comportement liés à des protocoles d'interaction entre composants	Mae	- Typage des composants et des connecteurs - Notions d'optionalité et de variabilité - Versionnement des éléments	Notification de l'incohérence à l'utilisateur
C2SADEL	- Contraintes du style C2 - Invariants structurels	C2SADEL	- Système de typage flexible pour les composants et les connecteurs avec différents niveaux de conformité	Notification de l'incohérence à l'utilisateur

TABLE 3.2 – Gestion de la cohérence des architectures

Support des reconfigurations et propriétés garanties. Le tableau 3.3 présente le support des reconfigurations dynamiques dans les plateformes. Les deux travaux Mae et C2SADEL n'apparaissent pas dans ce tableau car ils ne gèrent que l'évolution des architectures et pas l'exécution des reconfigurations dans les systèmes. Cette évolution se fait par l'intermédiaire d'un environnement dans lequel il possible de configurer et de modifier graphiquement les représentations des architectures. A part le modèle K-Component qui restreint les configurations à des remplacement de composants, les autres modèles proposent tous plus ou moins les mêmes opérations d'ajout/retrait d'éléments (composant ou connecteur), de connexion /déconnexion d'élément et éventuellement de modification des propriétés des éléments. Certains modèles proposent des opérations pour gérer explicitement le cycle de vie des éléments.

Propriétés des reconfigurations et des évolutions. Le dernier tableau (3.4) recense les différentes propriétés des reconfigurations dynamiques ou des évolutions d'architecture qui permettent d'en assurer la fiabilité. Outre le maintien de la cohérence des architectures via la satisfaction de contraintes présenté précédemment dans le tableau 3.2, nous retrouvons ici les propriétés classiques ACID garanties par les transactions. Tous les travaux ne mentionnent pas l'utilisation de transactions pour l'exécution des reconfigurations et lorsqu'ils le font, les mécanismes et les propriétés garanties sont peu détaillés. La question de la garantie de l'atomicité des reconfigurations peut notamment se poser dans le cas de reconfigurations distribuées. Mae est particulièrement intéressant à ce titre car même si les reconfigurations dynamiques ne sont pas gérées par l'environnement, son approche qui combine évolution architecturale et mécanisme de gestion de configuration (utilisé habituellement

	Langage de reconfiguration	Type de reconfiguration dynamique supporté		Opérations de reconfiguration dynamiques fournies	Extensibilité des opérations
		Programmée	Ad-hoc		
FORMAware	Java	x	x	- ajout / retrait de composants et d'interfaces - remplacement de composants - ajout / retrait de propriétés	+
K-Component	ACDL (transformations conditionnelles de graphes)	x		- remplacement de composants - modification de la valeur des propriétés des connecteurs	-
OpenRec	OpenRecML		x	- ajout / retrait / remplacement de composants - connexion /déconnexion de composants et de connecteurs	-
Plastik	extension d'ACME	x	x	- ajout / retrait de composants - connexion/déconnexion de composants et de connecteurs - instanciation de composants	-
Rainbow	extension d'ACME	x		?	+
ArchStudio	diff xADL 2.0 (application de patchs)	x	x	- ajout / retrait de composants et de connecteurs - connexion / déconnexion des composants et des connecteurs - suspension / reprise de l'activité des composants et des connecteurs	-
JADE	Java	x		- ajout / retrait de composants - connexion / déconnexion d'interfaces - modification de valeurs d'attributs - modification du cycle de vie	+

TABLE 3.3 – Reconfigurations et propriétés garanties

pour le code source) est intéressante. Elle est notamment la seule à gérer explicitement une forme de concurrence par verrouillage des éléments.

	Coordination reconfiguration / exécution du système	Atomicité	Gestion de la concurrence	Durabilité des effets des modifications
FORMAware	x (trois politiques d'adaptation possibles)	x (exécution transactionnelle)		
K-Component	x (protocole de reconfiguration fixé dans le modèle)			
OpenRec	x (support d'algorithmes multiples)			
Plastik	x (repose sur l'implémentation d'OpenCOM)	x (exécution transactionnelle avec mise à jour différée)	?	x (sous forme de description ADL)
Rainbow		x (exécution transactionnelle)	?	?
ArchStudio	x (gérée dans c2.fw, implémentation Java de C2)	x (vérification de la cohérence avant reconfigucation)	?	x (sous forme de description ADL)
JADE	x (gérée dans Julia, implémentation Java de Fractal)			x (maintien d'une représentation système répliquée activement)
Mae	n/a	x (annulation possible d'une évolution)	x (verrouillage des éléments modifiés)	x (conservation de l'historique des modifications)
C2SADEL	n/a	x (annulation possible d'une évolution)	?	?

TABLE 3.4 – Propriétés garanties pour les reconfigurations et les évolutions

Conclusion. Etant donné l'évaluation de nos critères dans différents travaux, nous pouvons remarquer que le problème de la fiabilité des reconfigurations dynamiques des architectures à composants n'est pas réellement abordé comme un objectif à part entière. Si certains travaux mentionnent l'utilisation de transactions pour l'exécution des reconfigurations, le modèle et les propriétés garanties soit sont peu nombreuses, soit ne sont pas détaillées. Il s'agit en effet d'un problème global qui demande dans un premier temps de définir le concept de cohérence dans une architecture, plus spécifiquement d'un point de vue structurel. Cette cohérence peut ainsi être définie comme la satisfaction d'un ensemble de contraintes extensibles exprimées dans une logique avec à la fois la possibilité de spécifier

des invariants dans les configurations architecturales mais aussi de spécifier la sémantique des opérations de reconfigurations à base de préconditions et de postconditions. Il s'agit alors de maintenir cette cohérence dans les systèmes reconfigurés en garantissant de bonnes propriétés pour les reconfigurations telles que les propriétés ACID. Ces propriétés permettent le support du recouvrement en cas de défaillances logicielles (par exemple les violations de contraintes) et matérielles (par exemple les pannes machines) et le support de la concurrence pour les reconfigurations. Il faut également, concernant la propriété d'isolation, assurer la synchronisation entre reconfigurations dynamiques et exécution fonctionnelles du système reconfiguré. L'ensemble de ces conclusions constituera le point de départ de nos travaux.

Deuxième partie

Contributions

Chapitre 4

Modélisation des configurations Fractal

Sommaire

Ce chapitre aborde le problème de la spécification des configurations de composants dans le modèle Fractal et de la définition de contraintes d'intégrité sur ces configurations en vue d'assurer la cohérence des architectures de sytèmes. Nous présentons en premier lieu la démarche globale qui a été suivie pour spécifier et vérifier la cohérence des configurations Fractal dans la section 4.1. La section 4.2 est dédiée à la présentation du modèle Fractal, modèle qui a été choisi comme base de notre étude et de nos expérimentations en raison de sa relative simplicité et de ses bonnes propriétés en terme de reconfigurabilité. Nous proposons ensuite (section 4.3) un méta-modèle simple du modèle de composants ainsi que la possibilité de l'étendre par des ensembles de contraintes. Nous verrons enfin (section 4.4) comment vérifier la cohérence entre les contraintes du modèle et comment déclarer et valider ces contraintes dans des applications concrètes.

4.1 Démarche de vérification de la cohérence des configurations

Le modèle Fractal est défini par une spécification textuelle informelle et par une API de haut niveau [BCS03] écrite dans un pseudo IDL (Interface Definition Language), sous-ensemble modifié du langage Java. L'API est également traduite en plusieurs langages d'implémentation dont le Java et le C. Afin de pouvoir plus facilement déterminer si une configuration d'une application à base de composants Fractal est *valide*, i.e. conforme au modèle de composants, il est nécessaire de préciser cette spécification. Un des objectifs est de pouvoir à tout moment de son cycle de vie valider une application Fractal, par exemple au cours de son exécution, et ce de façon simple, non intrusive et la plus légère possible du point de vue des performances. Nous voulons non seulement modéliser les configurations (ou architectures) statiques des systèmes mais également pouvoir représenter les modifications (reconfigurations) de ces architectures. Une des caractéristiques de la spécification est

donc qu'elle soit exécutable et pourra pour cela se reposer sur les propriétés réflexives du modèle Fractal. Par ailleurs, le modèle étant aisément extensible, cette spécification doit l'être également.

La démarche suivie (cf. Figure 4.1) consiste à partir de la spécification textuelle à extraire un ensemble de contraintes sur le modèle Fractal que nous appelons *contraintes d'intégrité*. Cette 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, Alloy [Jac02], qui grâce à son analyseur permet de vérifier la non contradiction entre les contraintes. Après cette vérification préalable, les contraintes sont finalement traduites dans le langage FPath [DLLC09, DL06b] pour être intégrées dans des applications concrètes.

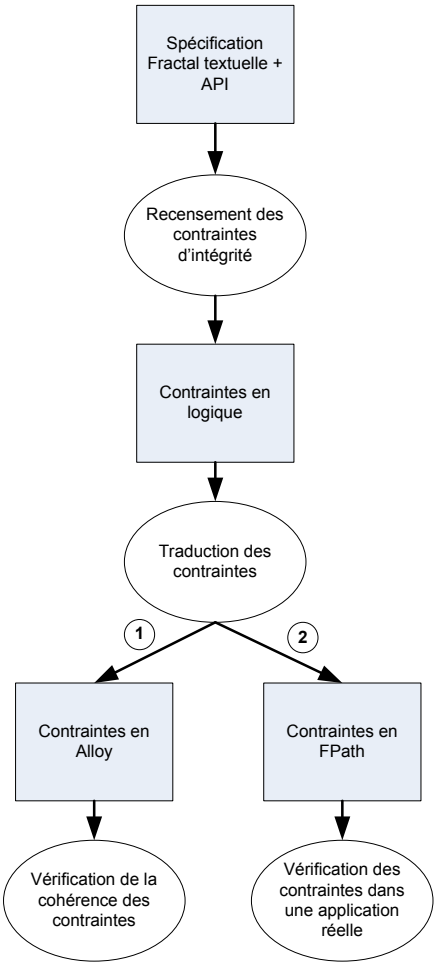

FIGURE 4.1 – Processus de vérification des contraintes

4.2 Le modèle de composants Fractal

Pour aborder le problème de la fiabilisation des reconfigurations dynamiques dans les architectures à composants, nous sommes partis de l'étude d'un modèle de composants particulièrement adapté à la dynamicité des architectures tout en étant suffisamment simple pour en faciliter son analyse. Le modèle de composants Fractal [BCL+06] propose des concepts riches d'un point de vue architectural (hiérarchie, partage) tout en étant fortement dynamique. Par ailleurs, plusieurs implémentations et outils lui sont associés pour faciliter le développement et l'administration d'applications à base de composants Fractal.

4.2.1 Coeur du modèle

Le modèle de composants Fractal [BCL+06] est développé conjointement par France Telecom R&D et l'INRIA et est distribué en open source par le consortium OW2 [1]. Il s'agit d'un modèle ouvert donc facilement extensible, réflexif et récursif (hiérarchique sur plusieurs niveaux) avec notion de partage. Fractal vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants ainsi qu'une séparation des préoccupations fonctionnelles et non fonctionnelles.

Motivations. L'objectif principal du modèle est ainsi la construction de systèmes répartis et hautement adaptables en proposant des techniques d'ingénierie des systèmes par assemblage de composants logiciels. Un des points forts du modèle est sa vision homogène et intégré du logiciel : tout est composant à une granularité arbitraire. En effet, grâce à son aspect hiérarchique, on peut aussi bien représenter des ressources de bas niveau voire des ressources matérielles par des composants à fine granularité que des briques logicielles à grosse granularité telles que des serveurs complets (ex : wrapping du serveur J2EE Jonas en Fractal dans Jade [BHQS05]). De plus, le partage de composants permet de modéliser le partage de ressources aussi bien que le concept de domaine pour diverses préoccupations (administration, sécurité, isolation de fautes, etc.). De cette façon, Fractal peut être utilisé pour des besoins logiciels très variés : applications, services, intergiciel, système d'exploitation, etc. L'architecture réifiée à l'exécution et des capacités d'introspection et d'intercession permettent notamment de construire des systèmes adaptatifs (Safran [DL06a]). Un autre avantage du modèle est qu'il couvre complètement le cycle de vie du logiciel : depuis la conception de son architecture, son développement et jusqu'à son administration. L'architecture du système conçue à base de composants Fractal correspondra ainsi à celle du système développée et celle qui sera administrée.

Spécification. La spécification est relativement simple et volontairement peu détaillée sur certains aspects de manière à laisser la possibilité de fournir plusieurs interprétations du modèle. L'ouverture de modèle se retrouve par exemple dans la sémantique non figée pour les liaisons entre composants, la composition et la réflexivité. Il est également possible d'intégrer dans le modèle de nouvelles préoccupations extra-fonctionnelles (eg., la sécurité avec Cracker [TJ06], les aspects avec FAC [PSDC08]). Le modèle de base peut être utilisé de manière classique pour spécifier l'implémentation de frameworks, pour construire des applications à base de composants, ou bien le modèle peut être vu comme un méta-modèle et son implémentation comme un framework de framework de composants (cf. Frascati [2], implémentation du modèle SCA reposant sur une implémentation de Fractal).

Modèle. Le modèle est abstrait et indépendant des langages d'implémentation. L'implémentation de référence de la spécification est un framework Java, Julia [BCL+04]. Il existe d'autre implémentations aussi bien en Java (AOKell [3], ProActive [4]), que dans d'autre langages : en C (Think [5] [FSLM02], Cecilia [6]), .NET (FractNet), SmallTalk (FracTalk). Les implémentations peuvent reposer sur des besoins différents, Julia est plutôt utilisé pour le développement d'intergiciels, ProActive pour le développement de composants actifs sur les grilles de calcul, tandis que Cecilia ou Think vise à l'implémentation de systèmes embarqués et de systèmes d'exploitation.

Les concepts clés du modèle sont les composants, les interfaces et les liaisons (cf. Figure 4.2) :

- **Les composants.** Les composants sont des unités de compilation et d'exécution du système. Ils peuvent être imbriqués, d'où les notions de sous-composants, de composants composites et de composants partagés. Un composant partagé est un composant possédant plus d'un composant parent. Un composant Fractal est formé d'une membrane et d'un contenu, la membrane représente la partie de contrôle du composant, et son contenu correspond à son implémentation fonctionnelle.

- **Les interfaces.** Les interfaces sont les points d'accès entre les composants pour l'émission ou la réception d'invocation d'opérations, et plus généralement de signaux et de flux. Elles sont

1. `http://fractal.objectweb.org/`
2. développé dans le cadre du projet SCOrWare : `http://www.scorware.org/`
3. `http://fractal.objectweb.org/aokell/index.html`
4. `http://proactive.inria.fr/`
5. `http://think.objectweb.org/`
6. `http://fractal.objectweb.org/cecilia-site/current/index.html`

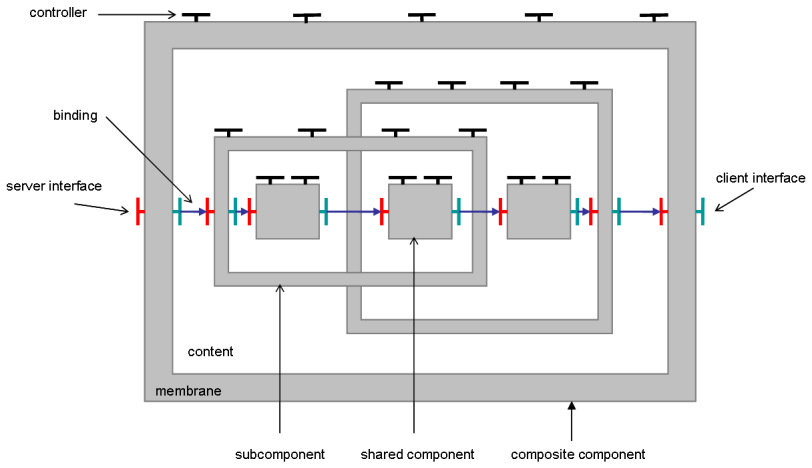

FIGURE 4.2 – Modèle Fractal

divisées en deux catégories : les interfaces métier et les interfaces de contrôle (ou contrôleurs). Les interfaces de contrôle permettent de gérer les propriétés non fonctionnelles des composants. Les interfaces sont typées et l'ensemble des types des interfaces métier d'un composant définit le type du composant. Une interface est de rôle client ou serveur : une interface serveur reçoit des opérations d'invocation alors qu'une interface cliente peut en émettre. Deux catégories d'interfaces existent au niveau des composants composites, les interfaces internes permet de relier un composite à ses sous-composants, les interfaces externes servent à communiquer avec les composants en dehors du composite.

– **Les liaisons.** Les liaisons en Fractal sont des canaux de communication entre interfaces de composants dynamiquement manipulables et dont la sémantique est arbitraire et extensible. Une liaison primitive relie deux interfaces de composants dans un même espace d'adressage et la sémantique de communication par défaut est l'invocation synchrone de méthode. Contrairement à d'autres modèles de composants (OpenCOM [CBG+04], C2 [TMA+95], Wright [All97]), il n'existe pas de concept de connecteur en tant qu'entité de première classe dans le modèle. Une liaison composite est un assemblage de composants de liaisons et de liaisons primitives. Ce dernier type de liaison permet d'implémenter des communications complexes : communications réparties, asynchrones, unicast ou multicast (cf. Proactive), etc., et peut prendre en compte des préoccupations non fonctionnelles telle que le contrôle de la qualité de service ou bien la sécurité. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée.

Instanciation des composants. L'instanciation de composants dans le modèle passe par l'utilisation du patron de conception Factory [GHJV94]. Des composants particuliers, les usines de composants permettent de créer de nouveaux composants. Le composant de bootstrap est une usine de composants qui ne nécessite pas d'être créé et qui permet d'instancier d'autres composants. Un nouveau composant est instancié à partir de son type, de la description de son implémentation et de celle de sa membrane. Fractal propose également un mécanisme de composants template plus efficace en cas d'instanciation répétées du même type de composants. Les templates sont des usines de composants qui ne produisent que des composants isomorphe à eux-même.

Des contrôleurs pour les reconfigurations dynamiques. Fractal en tant que modèle réflexif propose une séparation des préoccupations entre un niveau méta qui correspond au contrôle dans les membranes des composants et le niveau de base qui est l'implémentation fonctionnelle du contenu des mêmes composants (cf. Figure 4.3). Le modèle supporte aussi bien la réflexion structurelle que la réflexion comportementale. La réflexion structurelle se situe au niveau du contenu des composants qui est réifié pour pouvoir être observé (mécanisme d'introspection) et modifié (mécanisme d'intercession) par les contrôleurs à travers une représentation causalement connectée. La réflexion comportementale

passe par l'utilisation d'intercepteurs dans la membrane qui permettent d'enrichir voire de redéfinir le comportement des méthodes des interfaces des composants.

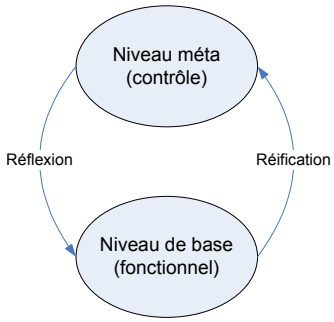

FIGURE 4.3 – Réflexion et connexion causale

Les domaines typiques du contrôle sont l'administration du composant (gestion du cycle de vie et de la configuration), la gestion de propriétés non-fonctionnelles telles que la sécurité, la persistance, la tolérances aux fautes, les contrats, etc. Un certain nombre de contrôleurs sont définis dans la spécification pour l'introspection et la reconfiguration dynamique de composants :

- Le *BindingController* gère les liaisons primitives entre composants. Il est possible de lier, délier des interfaces serveurs d'un composant et de connaître les interfaces serveur qui sont liées à ses interfaces clientes.
- Le *ContentController* gère le contenu d'un composant composite par introspection de la structure. Il propose des opérations d'ajout, de retrait de sous-composants et est utilisé pour récupérer les sous-composants d'un composite ainsi que ses interfaces internes.
- Le *SuperController* gère le contenant d'un composant en permettant de connaître l'ensemble de ses composants parents.
- Le *NameController* offre la possibilité de récupérer ou de changer le nom d'un composant.
- L'*AttributeController* sert à configurer des attributs d'un composant. Un attribut est une propriété configurable qu'il est possible de lire/écrire à l'extérieur du composant.
- Le *LifeCycleController* gère le cycle de vie d'un composant, soit le démarrage, et l'arrêt du composant. Il permet également de récupérer son état.

De nouveaux contrôleurs peuvent cependant être ajoutés aux composants pour étendre le modèle (e.g. les contrôleurs d'adaptation dans [Dav05] pour la construction de systèmes adaptatifs avec Fractal).

Niveaux de conformité. La spécification Fractal met en avant plusieurs niveaux de conformité au modèle pour une implémentation donnée. A chaque niveau correspond notamment la fourniture d'un certain nombre des interfaces de contrôle standard. Nous ne considérons par la suite le niveau de conformité maximum (niveau 3.3 de la spécification) qui offre toutes les capacités de reconfiguration du modèle.

4.2.2 Implémentations et outils associés au modèle

L'écosystème du modèle Fractal offre un certains nombres d'outils et de langages pour l'ingénierie des système et des intergiciels. Parmi ces outils, nous utilisons essentiellement les suivants : le support pour la distribution (Fractal RMI[7]), un langage modulaire et extensible de description d'architecture (Fractal ADL [LOQS07]) et son compilateur, un langage de navigation dans les architectures (FPath [DLLC09]) et un langage de reconfiguration dynamique (FScript [DLLC09]). Parmi les différentes implémentations du modèle Fractal, nous choisissons d'utiliser Julia, implémentation de référence en Java. Cette implémentation servira non seulement comme base de nos expérimentations mais aussi comme référence pour la sémantique du modèle.

Implémentation Julia. L'objectif principal du framework Julia est la programmation de contrôleurs et intercepteurs qui constituent la membrane des composants. Les contrôleurs définis dans la

7. `http://fractal.objectweb.org/fractalrmi/index.html`

spécification sont implémentés dans Julia sous la forme d'un assemblage de mixins Java optionnels. Un mixin est une classe abstraite qui spécifie un ensemble de champs et de méthodes obligatoirement présents dans la classe parente. Les classes Java résultant de la composition de mixins sont générées automatiquement par un générateur de code. L'optionalité des mixins pour la constitution des contrôleurs permet d'obtenir les différents niveaux de compatibilité de la spécification Fractal.

Julia offre la possibilité d'ajouter des intercepteurs dans les membranes pour intercepter les appels de méthodes sur les interfaces fonctionnelles des composants. L'interception peut se faire sur les interfaces serveurs (invocations entrantes dans le composant) et sur les interfaces clientes (invocations sortantes du composant). Il est aussi possible de spécifier à la manière des points de jonction de la programmation par aspects [KLM+97] les méthodes à intercepter et les actions de type « before », « after »ou « around »pour les appels de méthode suivant l'intercepteur utilisé. Les intercepteurs sont généralement associés à un contrôleur auquel ils délèguent le traitement de l'interception et qui permet de rendre accessible via les interfaces de contrôle des fonctionnalités liées à l'interception.

Fractal RMI. Fractal RMI est constitué d'un ensemble de composants pour la création de liaisons distribuées entre des composants Fractal. Un protocole de type Java RMI permet l'invocation de méthodes distantes sur des interfaces de composants. Chaque interface de composant est accessible à distance. Un composant de bootstrap peut lui-même être accessible à travers un registre pour l'instanciation de composants à distance. Fractal RMI est utilisable avec toute implémentation du modèle Fractal en Java dont Julia.

Fractal ADL. L'architecture d'une application à base de composants Fractal peut être décrite à l'aide du langage de description d'architecture Fractal ADL. Cet ADL (Architecture Description Language) permet de manière déclarative de définir des configurations avec une syntaxe abstraite indépendante basée sur XML. Le langage est modulaire : les différents éléments d'une configuration (composants, interfaces, etc.) sont définis dans des modules XML séparés conforme à une DTD. Un atout de cet ADL est son extensibilité, il est possible de définir un nouveau concept d'architecture, lui associer une syntaxe particulière et étendre l'ADL Fractal par de nouveaux modules pour la prise en compte de ce concept : un module pour le packaging du code des composants peut par exemple être ajouté pour le déploiement. Pour illustrer la syntaxe, la définition ADL pour une application simple (cf. Figure 4.4) est donnée dans le listing 4.1.

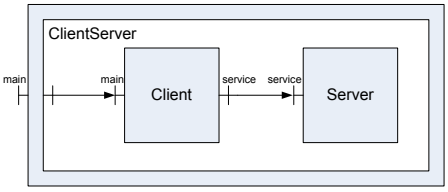

FIGURE 4.4 – Une simple application client-serveur en Fractal

```
<definition name="ClientServer">
 <interface name="main" role="server" signature="java.lang.Runnable" />
 <component name="client">
  <interface name="main" role="server" signature="java.lang.Runnable" />
  <interface name="service" role="client" signature="Service" />
  <content class="ClientImpl" />
 </component>
 <component name="server">
  <interface name="service" role="server" signature="Service" />
  <content class="ServerImpl" />
 </component>
 <binding client="this.main" server="client.main" />
 <binding client="client.service" server="server.service" />
</definition>
```

Listing 4.1 – Définition Fractal ADL d'une application client-server

Un compilateur est associé à l'ADL pour générer le code des composants à déployer ou pour instancier directement les définitions de composant (utilisation en tant qu'interpréteur). Le compilateur peut générer vers plusieurs implémentations Fractal différentes. Son architecture modulaire est constituée d'un parseur de définitions ADL qui construit des arbres de syntaxe abstraite (AST). Le compilateur définit un ensemble de tâches de compilation avec leurs dépendances à partir de cet AST. Le « builder »définit un comportement concret pour ces tâches, par exemple de génération de code, et les exécute.

Le langage de navigation FPath. FPath est un langage de navigation, de sélection et de requêtes dans des architectures Fractal. La syntaxe est inspirée de XPath [Wor07], langage de requêtes sur les documents XML. FPath est sans effets de bord sur les configurations avec des uniquement des capacités d'introspection. Une expression FPath est faite d'un ensemble de pas : pas1/pas2/.../pasN, chaque pas représentant la sélection d'un ensemble d'éléments de l'architecture. La syntaxe générale est donnée dans la figure 4.5. FPath représente les applications Fractal sous forme de graphe avec comme noeuds des composants, des interfaces et des attributs, et comme arcs des axes de navigation entre ces éléments. Des axes sont prédéfinis comme l'axe *child* pour sélectionner l'ensemble des sous-composants d'un composant donné, mais de nouveaux axes peuvent être ajoutés. FPath manipule les éléments dans des variables (noms préfixés par un $) et propose des fonctions pour introspecter les éléments tels que la fonction *name*() pour retourner le nom d'un composant.

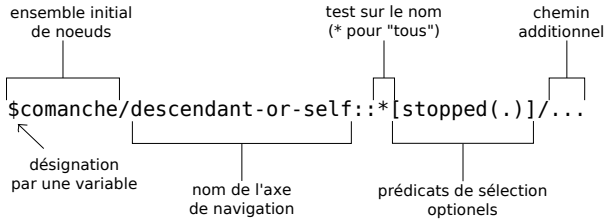

FIGURE 4.5 – Syntaxe des expressions FPath

Un exemple d'expression FPath pour sélectionner l'ensemble des composants partagés (i.e avec au plus un composant parent) dans un composant désigné par la variable $*root* est le suivant :

```
$root/descendant-or-self : :*[size(./parent : :*) > 1]
```

Le langage de reconfiguration FScript. FScript est un langage dédié pour spécifier et exécuter des reconfigurations et plus particulièrement des reconfigurations dans Fractal. Une des principales motivations de FScript est le minimalisme des API de reconfigurations en Fractal et la verbosité de leur programmation en Java. Le langage FPath est inclus dans FScript pour introspecter les configurations. C'est un langage de script impératif avec les structures de contrôle limitées (itération et branchement conditionnel), il ne permet cependant pas les définitions récursives pour éviter la récursion infinie. FScript permet la définition d'actions de reconfiguration pouvant intégrer des expressions FPath, une action peut retourner ou non une valeur et modifie l'architecture du système à la différence des fonctions FPath. Des actions primitives correspondant aux principales opérations de reconfiguration de l'API Fractal sont déjà fournies : *add*, *remove*, *bind*, etc. Ces actions invoquent directement les opérations de l'API dans l'implémentation Fractal sous-jacente. L'interpréteur FScript se charge d'exécuter les actions et permet de charger de nouvelles actions définies par l'utilisateur.

Un exemple d'action FScript prenant deux noeuds composants en paramètre est donné ci-dessous :

```
/* Copie la valeur de tous les attributs du composant $src
 * dans les attributs du même nom du composant $dest.
 */
action copy-attributes(src, dest) {
  for oldAttr : $src/attribute::* {
    newAttr = $dest/attribute::*[name(.) == name($oldAttr)];
    set-value($newAttr, value($oldAttr));
  }
}
```

Listing 4.2 – Exemple d'action de reconfiguration FScript

4.3 Spécification et extension du modèle Fractal

Un modèle de composants fournit des règles de bonne construction des systèmes en leur apportant entre autres des propriétés de structuration et de modularité [Szy02]. Le modèle Fractal définit ainsi quels sont les constituants de l'architecture d'un système que nous appellerons éléments architecturaux, et les relations entre ces éléments. La spécification du modèle est agnostique quant au langage d'implémentation et volontairement définie de façon la plus simple possible, la force du modèle résidant dans son extensibilité. Des implémentations du modèle complètent le modèle par des extensions qui leurs sont propres. AOKell, une implémentation de Fractal en Java, construit par exemple les membranes des composants à base de composants, Julia construit les contrôleurs dans les membranes avec des mixins traitant chacun de préoccupations différentes.

Nous nous proposons de spécifier les possibilités d'extension du modèle de manière plus structurée et formelle sous forme de *profils*. Un *profil* est une une extension du coeur du modèle construit à base de contraintes d'intégrité [LCL06] par analogie avec les contraintes dans les bases de données [TGGL82]. Les contraintes se présentent sous forme d'invariants de configuration. Un même profil peut servir à valider un ensemble d'application Fractal, i.e. à garantir que des configurations de systèmes lui sont conformes.

4.3.1 Un méta-modèle pour Fractal

En vue de préciser la spécification du modèle Fractal, nous définissons dans un premier temps un méta-modèle 4.6 capturant au mieux la spécification textuelle. Cette représentation permet de mettre en avant les concepts fondamentaux du modèle : les éléments architecturaux (composants, interfaces, et attributs), les relations entre ces éléments (liaison, hiérarchie, etc.), et le système de typage. Ce méta-modèle ne représente que le coeur de la spécification et ne fait notamment pas apparaître les raffinements présents dans les implémentations telles que Julia.

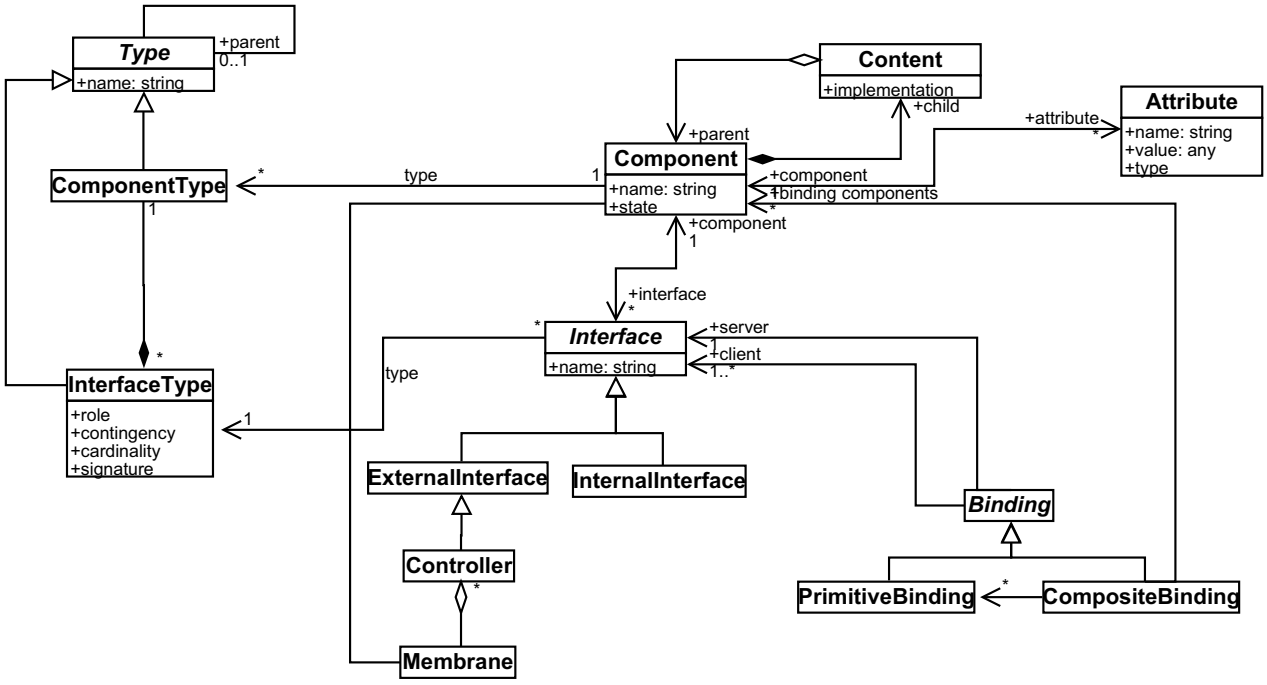

FIGURE 4.6 – Métamodèle Fractal

En vue de simplifier ce métamodèle pour en faciliter la manipulation à l'exécution, nous nous inspirons des structures de données utilisées dans l'interpréteur du langage de navigation et de requête

FPath pour en fournir une abstraction 4.7. Cette représentation est essentiellement un sous-ensemble du métamodèle de la figure 4.6 mais qui est plus homogène car elle ne considère que des éléments architecturaux et des relations entre ces éléments. Les opérations de reconfiguration correspondent dès lors soit à un parcours du graphe (introspection) soit à une modification de la topologie du graphe (intercession).

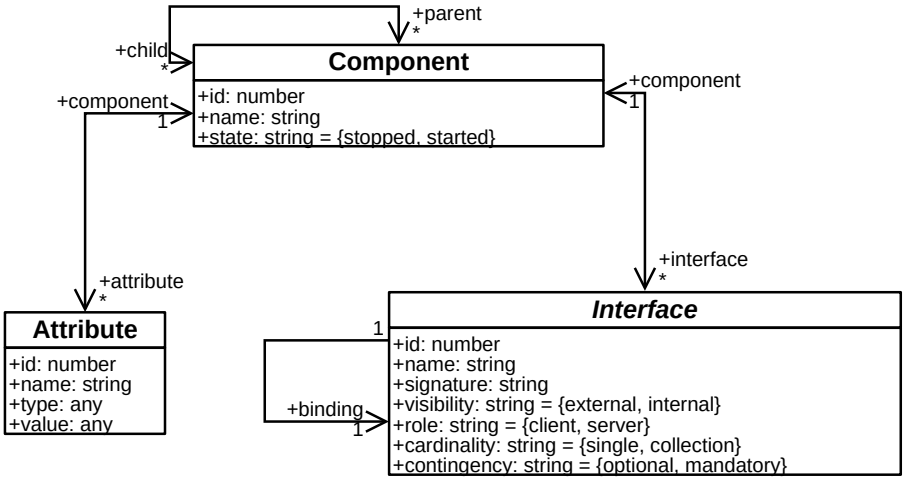

FIGURE 4.7 – Métamodèle Fractal simplifié

L'objectif de ce graphe est de simplifier au maximum la représentation du modèle pour faciliter la validation de configurations à l'exécution tout en conservant le plus d'informations possibles. Le graphe reprend les principaux concepts définis dans le premier méta-modèle. Des éléments du méta-modèle ont cependant été simplifiés dans le graphe. C'est la cas des liaisons (*bindings*) qui apparaissent en tant que hiérarchie de classes dans le diagramme UML et qui ont été réduites à une simple relation (arc) dans le graphe. Cela reste dans la philosophie du modèle Fractal qui ne réifie pas les liaisons en entité de première classe. Le typage n'est pas directement représentée, car il repose sur des propriétés du langage sous-jacent d'implémentation des interfaces mais il reste possible d'exprimer la relation de sous-typage dans le graphe à travers les signatures des interfaces.

Représentation du modèle Fractal sous forme de graphe. Une application Fractal à un instant donné est modélisée par une configuration statique d'éléments architecturaux appartenant au modèle. Nous définissons dès lors une configuration Fractal sous la forme d'un graphe orienté dont les sommets (ou noeuds) et les arcs sont typés. Une configuration est dite valide (ou cohérente) si son graphe est conforme au modèle Fractal (la sémantique de la relation de conformité est définie dans la suite). Ce graphe est défini par $\mathcal{A} = (E, R)$ où l'ensemble des sommets E représente les éléments architecturaux du système tels que définis 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 peut être assimilé à un triplet représentant le modèle $\mathcal{M} = (\mathcal{M}_E, \mathcal{M}_R, \mathcal{M}_P)$ où :

- $\mathcal{M}_E$ est l'ensemble des éléments architecturaux qui peuvent apparaître dans les instances du modèle.
- $\mathcal{M}_R$ est l'ensemble des relations possibles entre éléments architecturaux.
- $\mathcal{M}_P$ est l'ensemble des contraintes d'intégrité qui doivent être vérifiées par les instances du modèle pour pouvoir être considérées valides. Une contrainte d'intégrité est un prédicat binaire $\pi(E, R)$.

Nous définissons dès lors qu'une configuration (ou architecture) $\mathcal{A}$ est *conforme* au modèle $\mathcal{M}$ (ou réciproquement $\mathcal{M}$ *modélise* $\mathcal{A}$), et nous notons $\mathcal{M} \models \mathcal{A}$ si et seulement si :

$$\begin{array}{rcl} E & \subseteq & \mathcal{M}_E \\ R & \subseteq & \mathcal{M}_R \\ \forall \pi & \in & \mathcal{M}_P, \pi(E,R) \end{array}$$

Nous nous attachons dans un premier temps plus particulièrement à la modélisation du coeur du modèle Fractal $\mathcal{F}$ défini par ses trois constituants :

$$\mathcal{F} = (\mathcal{F}_E, \mathcal{F}_R, \mathcal{F}_P)$$

Nous utilisons la logique du premier ordre comme formalisme de spécification du modèle, i.e pour spécifier les relations architecturales et les contraintes d'intégrité sur les configurations. Ce formalisme logique présente l'avantage d'être agnostique du point de vue du langage d'implémentation. Nous considérons ainsi l'utilisation des opérateurs logiques (booléens) classiques $\wedge$, $\vee$, $\neg$, et l'implication $\Rightarrow$, les quantificateurs universel $\forall$ et existentiel $\exists$. Les opérateurs ensemblistes utilisés sont les suivant : l'union $\cup$, l'intersection $\cap$, la différence $\setminus$ et l'égalité $=$. Le modèle de graphes de configuration est défini dans la suite avec ce formalisme.

Types de données. Des propriétés sont associées aux éléments architecturaux dans le graphe et nous manipulons donc les types de données primitifs suivants sans plus de précision sur leur implémentation concrète : chaînes de caractères (type String), nombres (type Number), booléens (type Boolean), ensemble (type Set) et le type Any qui correspond à n'importe quel type dans le langage d'implémentation sous-jacent des composants. Pour représenter les propriétés des éléments, nous définissons les types suivant qui sont des extensions des types primitifs :

- Id qui étend Number : identifiant d'un élément.
- Name qui étend String : nom d'un élément.
- State qui étend String et qui a valeur dans l'ensemble $\{Started, Stopped\}$: état du cycle de vie d'un composant.
- Visibility qui étend String et qui a valeur dans l'ensemble $\{External, Internal\}$: visibilité des interfaces de composant.
- Role qui étend String et qui a valeur dans l'ensemble $\{Client, Server\}$: rôle des interfaces.
- Cardinality qui étend String et qui a valeur dans l'ensemble $\{Single, Collection\}$: cardinalité des interfaces.
- Contingence qui étend String et qui a valeur dans l'ensemble $\{Optional, Mandatory\}$: contingence des interfaces.
- Signature qui étend Any : signature des interfaces.
- Type qui étend Any : type des attributs de composants.
- Value qui étend Any : valeur des attributs.
- Definition qui étend Any : définition d'un composant nécessaire à son instanciation.

Opérateurs arithmétiques. Les opérateurs arithmétiques habituels ($+$, $-$, $\times$, $/$) sont utilisés pour les types $Number$ ainsi que les comparaisons. Les tests d'égalité (opérateur $=$) sont définis pour tous les types de données.

Fonctions. Les fonctions sont sans effets de bord et sont soit des prédicats booléens, soit des accesseurs en lecture sur les propriétés des éléments du modèle, soit des fonctions de navigation dans le graphe de configuration.

Les éléments architecturaux et leurs propriétés. Les éléments architecturaux réifiés dans le modèle Fractal sont les sommets du graphe de configuration et sont au nombre de trois : les composants, les interfaces et les attributs. A chaque sommet du graphe peut être associé un certain nombre de propriétés. Ces données primitives caractérisent l'élément architectural représenté :

- les composants : $Component = \mathsf{Id} \times \mathsf{Name} \times \mathsf{State}$
- les interfaces : $Interface = \mathsf{Id} \times \mathsf{Name} \times \mathsf{Signature} \times \mathsf{Visibility} \times \mathsf{Role} \times \mathsf{Cardinality} \times \mathsf{Contingency}$

– les attributs de configuration : $Attribute = \mathsf{Id} \times \mathsf{Name} \times \mathsf{Type} \times \mathsf{Value}$

Nous avons ainsi l'ensemble des éléments architecturaux du modèle Fractal :

$$\mathcal{F}_E = Component \cup Interface \cup Attribute$$

Toute instance d'élément architectural possède un unique identifiant id afin de pouvoir être distinguée dans une configuration à l'exécution. Cet identifiant sert notamment à garantir l'unicité de la désignation des éléments au cours des reconfigurations dynamiques. L'opérateur = utilise l'égalité entre les identifiants pour tester l'égalité entre éléments architecturaux. Les trois types d'éléments sont aussi nommés. Le nom des composants est défini comme la chaîne de caractère retourné par la méthode `getFcName()` du contrôleur `name-controller`. Si le composant ne fournit pas cette interface, ou si la méthode retourne la valeur `null`, le nom du composant est alors défini par la chaîne de caractère vide (""). Il faut noter qu'il n'y a pas de garantie de l'unicité des noms pour les composants d'un système dans la spécification Fractal, cette contrainte peut cependant être ajoutée dans une extension du modèle. Le type d'un attribut est le type de retour de l'accesseur en lecture (getter) correspondant dans le contrôleur `attribute-controller` du composant (ce type doit correspondre au type du paramètre de l'accesseur en écriture (setter) si l'accès en écriture pour cet attribut est possible).

Pour chacune des propriétés des éléments, nous définissons des fonctions accesseurs pour récupérer leur valeur dans une configuration donnée, leur nom est identique au nom de la propriété considérée (par exemple, $id : \mathcal{F}_E \rightarrow \mathsf{Id}$, $role : Interface \rightarrow \mathsf{Role}$). Nous utilisons de manière équivalente l'opérateur . pour accéder aux propriétés de éléments, pour une élément $e \in E$, $e.p_i$ désignera la valeur de la propriété p_i de l'élément e. Des prédicats sont définis sur les éléments architecturaux lorsque leurs propriétés ont valeur dans des ensembles de valeurs : *state*, *visibility*, *role*, *cardinality*, *contingency*. La propriété *state* d'un composant qui a valeur dans l'ensemble $\{Stopped, Started\}$ est ainsi associée à un prédicat correspondant $started : Component \rightarrow \mathsf{Boolean}$ qui est défini comme $started(c) \Leftrightarrow state(c) = Started$.

Les relations architecturales. Les relations entre les éléments architecturaux dans le modèle Fractal sont des relations binaires du type $x\mathcal{R}y$ où $(x, y) \in E^2$ et correspondent aux arcs dans le graphe de configuration (cf. Figure 4.8). Les relations pourraient être généralisées à des relations n-aires mais il s'avère que dans le modèle de base, seules des relations binaires sont utilisées. Nous considérons les quatre relations primitives suivantes dans Fractal :

– La relation *hasInterface* définie sur $Component \times Interface$: détermine si un composant possède une interface donnée (interne ou externe).
– La relation *hasAttribute* définie sur $Component \times Attribute$: détermine si un composant possède un attribut donné.
– La relation *hasChild* définie sur $Component \times Component$: détermine si un composant est sous-composant direct d'un autre composant. Cette relation est irréflexive car un composant ne peut être son sous-composant direct. Elle n'est pas symétrique, le premier composant doit être le composant parent, le second le composant fils. Elle n'est pas transitive car on ne considère que les sous-composants directs.
– La relation *hasBinding* définie sur $Interface \times Interface$: détermine si une interface est liée directement à une autre interface. Cette relation est irréflexive car une interface ne peut être liée à elle-même. Elle est symétrique car elle ne fait pas de distinction entre les rôles des interfaces (*server* ou *client*). Elle n'est pas transitive car on ne considère que les liaisons directes.

Nous avons ainsi l'ensemble des relations primitives définies dans le modèle :

$$\mathcal{F}_R = hasInterface \cup hasAttribute \cup hasChild \cup hasBinding$$

Nous préfèrerons utiliser une notation fonctionnelle pour représenter les relations, i.e. $hasChild(x, y)$ plutôt que la forme $x\ hasChild\ y$. Les relations peuvent ensuite être composées pour construire de nouvelles relations. Par ailleurs, les relations primitives définies sur un ensemble (ex : *hasChild*) peuvent être dérivées par transitivité, par réflexivité et par symétrie notées pour une relation $\mathcal{R}$ respectivement $\mathcal{R}^{trans}$ et $\mathcal{R}^{refl}$ et $\mathcal{R}^{sym}$. Nous pouvons ainsi définir par exemple la fermeture transitive de la relation *hasChild*, $hasChild^{trans}$, qui déterminera si un composant donné est un sous-composant d'un autre composant à quelque niveau hiérarchique que ce soit. La fermeture réflexive

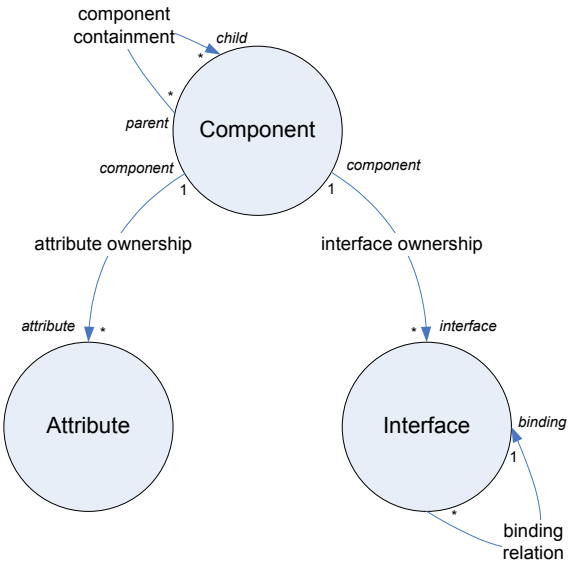

FIGURE 4.8 – Représentation des éléments et des relations architecturales dans Fractal

correspondante $hasChild^{refl}$ déterminera pour deux composants si le second est un sous-composant direct du premier ou est lui est égal. La fermeture symétrique $hasChild^{sym}$ déterminera si il existe une relation de parenté directe entre deux composants.

La relation $hasChild^{trans}$ peut se définir de manière récursive à partir de la relation $hasChild$:

$$\begin{aligned} \forall (c, c') \in E^2, hasChild^{trans}(c, c') \quad &= \quad hasChild(c, c') \\ &\quad \vee \ (\exists c'' \in E, hasChild(c, c'') \wedge hasChild^{trans}(c'', c')) \end{aligned}$$

La relation inverse d'une relation $\mathcal{R}$ notée $\mathcal{R}^{-1}$ peuvent être utilisée : la relation inverse de $hasChild$ définie sur $Component \times Component$ déterminera par exemple si un composant est parent d'un autre composant :

$$\forall (c, c') \in E^2, hasChild^{-1}(c, c') = hasChild(c', c)$$

Pour chacune des relations, nous associons des fonctions de navigation dans le graphe de configuration. L'opérateur . est utilisé de manière similaire que pour l'accès aux propriétés :

- $interfaces(Component\ c)$ (ou $c.interfaces$) retourne l'ensemble des interfaces d'un composants c.
- $attributes(Component\ c)$ (ou $c.attributes$) retourne l'ensemble des attributs d'un composant c.
- $children(Component\ c)$ (ou $c.children$) retourne l'ensemble des sous-composants du composant c.
- $bindings(Interface\ i)$ (ou $i.bindings$) retourne l'ensemble des interfaces connectées à l'interface i.

Exemple de configuration. Toute configuration Fractal peut être ainsi représentée par un graphe orienté conforme au modèle de la figure 4.8. Soit une simple application de type *client-serveur* doté de trois composants : un composite *ClientServer*, et deux primitifs *Client* et *Server*. Le composant *Server* fournit une interface *Printer* pour imprimer des messages sur la sortie standard préfixé par un attribut *header*. La configuration de cette application peut être représentée par le graphe de la figure 4.9 (les interfaces de contrôle des composants ne sont pas représentées dans le graphe pour des raisons de simplification).

4.3.2 Spécification du modèle par des contraintes d'intégrité

En plus de la définition des éléments architecturaux, de leurs propriétés et des relations architecturales, nous utilisons des contraintes d'intégrité pour compléter la spécification du modèle. Les

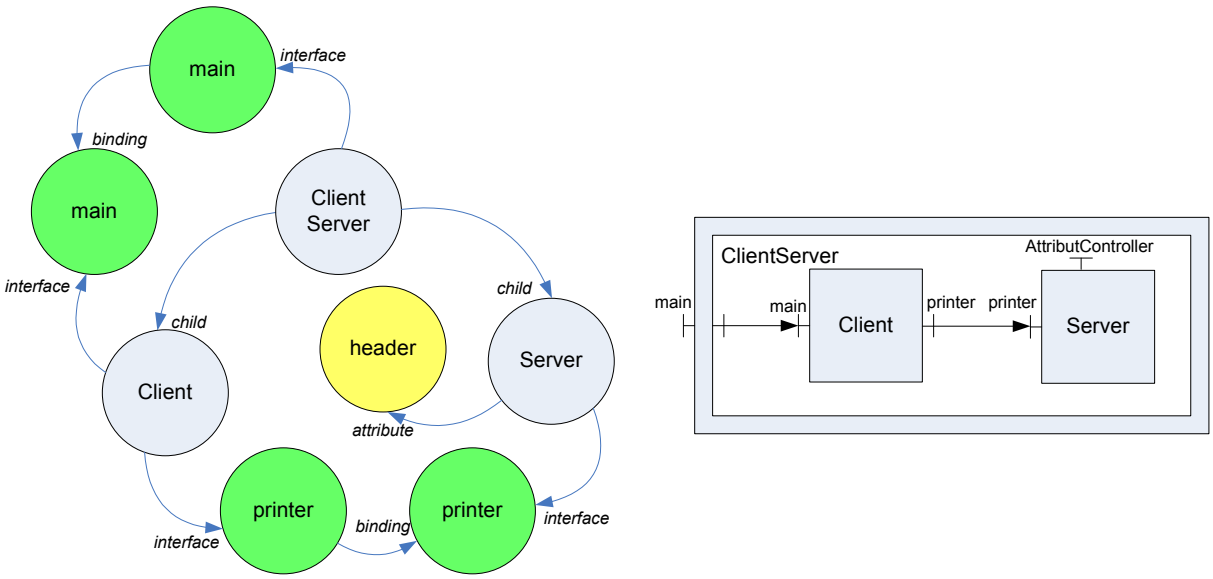

FIGURE 4.9 – Exemple du graphe de configuration d'une application ClientServer

contraintes utilisées pour cette spécification sont des invariants de structure et de comportement. Nous pouvons distinguer des contraintes sur les relations architecturales et des contraintes sur les valeurs des propriétés des éléments architecturaux.

Nous nous intéressons dans un premier temps aux contraintes au niveau du modèle Fractal. Ces contraintes d'intégrité sont des prédicats binaires notées $\pi_i(E, R)$ portant sur la validité d'une configuration. Ce sont des invariants génériques qui précisent la modélisation d'un système Fractal [8]. Une des difficultés est de ne pas sous-contraindre le modèle pour ne pas laisser passer des configurations incohérentes (faux positifs), ni de la surcontraindre en empêchant des configurations cohérentes d'être validées (faux négatifs). De plus, 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 les cardinalités des relations entre composants et interfaces et entre composants et attributs.

Unicité de l'appartenance d'une interface : Une interface appartient à un unique composant.

$$uniqueItfOwner(E, R) = \forall i \in E,\ \exists! c \in E, hasInterface(c, i)$$

Unicité de l'appartenance d'un attribut : Un attribut appartient à un unique composant.

$$uniqueAttrOwner(E, R) = \forall a \in E,\ \exists! c \in E, hasAttribute(c, a)$$

Des contraintes concernent l'unicité des identificateurs et du nommages des éléments architecturaux.

Unicité des identifiants : Les identifiants des éléments architecturaux sont globalement uniques dans une configuration donnée.

$$uniqueId(E, R) = \forall (e_1, e_2) \in E^2,\ id(e_1) = id(e_2) \Rightarrow e_1 = e_2$$

Unicité des noms des interfaces avec la même visibilité : Un composant ne peut avoir au plus qu'une seule interface externe avec un nom donné, de même pour les interfaces internes.

$$\begin{aligned} uniqueItfNameVis(E, R) \quad = \quad & \forall (c, i_1, i_2) \in E^3,\ (hasInterface(c, i_1) \\ & \wedge\ hasInterface(c, i_2) \wedge name(i_1) = name(i_2) \\ & \wedge\ (visibility(i_1) = visibility(i_2))) \Rightarrow i_1 = i_2 \end{aligned}$$

8. Nous ne prétendons pas effectuer ici un recensement exhaustif de toutes les contraintes décrivant le modèle Fractal mais seulement les plus importantes pour l'objectif visé, à savoir garantir la cohérence des reconfigurations dynamiques.

Unicité des noms des attributs : Un composant ne peut avoir au plus qu'un seul attribut avec un nom donné.

$$\begin{aligned} uniqueAttrName(E,R) &= \forall (c,a_1,a_2) \in E^3,\ (hasAttribute(c,a_1) \\ &\quad \wedge hasAttribute(c,a_2) \\ &\quad \wedge name(a_1) = name(a_2)) \Rightarrow a_1 = a_2 \end{aligned}$$

Un autre ensemble de contraintes concerne spécifiquement les propriétés et conventions de nommages des interfaces de contrôles.

Interfaces de contrôle standard. La spécification Fractal définit des interfaces de contrôle standard. Nous définissons dans un premier temps des prédicats utilitaires pour les identifier (l'exemple est uniquement donné pour l'interface *NameController*) :

$$\begin{aligned} controllerItf(i,n) &= external(i) \wedge server(i) \wedge single(i) \wedge optional(i) \\ &\quad \wedge name(i) = n \\ nameCtrlItf(i) &= controllerItf(i,'\texttt{name-controller}') \end{aligned}$$

Nous assumons enfin que chaque composant fournit au moins un ensemble de ces contrôleurs standards (d'autres choix peuvent être faits pour correspondre à un niveau de conformité différent à la spécification Fractal)

$$\begin{aligned} standardControllers(E,R) &= \forall c \in E, \\ &\quad \exists i \in E :\ hasInterface(c,i) \wedge componentItf(i) \\ &\quad \wedge \exists i \in E,\ hasInterface(c,i) \wedge nameCtrlItf(i) \\ &\quad \wedge \exists i \in E,\ hasInterface(c,i) \wedge lifecyleCtrlItf(i) \\ &\quad \wedge \exists i \in E,\ hasInterface(c,i) \wedge superCtrlItf(i) \end{aligned}$$

Seuls les composites peuvent contenir des sous-composants. Seuls les composants composites, caractérisés par la possession d'une interface `content-controller`, peuvent contenir des sous-composants. Nous définissons d'abord un prédicat pour déterminer si un composant est composite :

$$composite(c) = \exists i \in E,\ hasInterface(c,i) \wedge contentCtrlItf(i)$$

$$parentsAreComposite(E,R) = \forall c \in E,\ (\exists c' \in E, hasChild(c,c') \Rightarrow composite(c))$$

Seuls les composites ont des interfaces internes.

$$internalItfs(E,R) = \forall c \in E,\ (\exists i \in E,\ hasInterface(c,i) \wedge internal(i)) \Rightarrow composite(c)$$

Les composites ont des interfaces internes duales. Nous définissons d'abord le prédicat suivant pour tester si deux interfaces sont duales (i.e. même nom, signature, etc., mais une est interne l'autre est externe et l'une est cliente et l'autre est serveur). Le premier argument doit être une interface externe et le second une interface interne.

$$\begin{aligned} dual(i,i') &= name(i) = name(i') \\ &\quad \wedge signature(i) = signature(i') \wedge cardinality(i) = cardinality(i') \\ &\quad \wedge contingency(i) = contingency(i') \wedge external(i) \wedge internal(i') \\ &\quad \wedge (client(i) \wedge server(i') \vee server(i) \wedge client(i')) \end{aligned}$$

Le prédicat est ainsi le suivant :

$$\begin{aligned} dualItfs(E,R) &= \forall c,i \in E,\ composite(c) \wedge hasInterface(c,i) \wedge \neg controllerItf(i) \\ &\quad \Rightarrow (\exists i' \in E,\ hasInterface(c,i') \wedge (dual(i,i') \vee dual(i',i)) \end{aligned}$$

Pas de cycle dans la hiérarchie de composants. Un composant ne peut pas se contenir lui-même, directement ou indirectement pour éviter les récursions infinies.

$$noCycle(E,R) = \forall c \in E,\ \neg hasChild^{trans}(c,c)$$

Orientation des liaisons. Une liaison entre interfaces va toujours de l'interface cliente vers l'interface serveur.

$$bindingDirection(E,R) = \forall (i_1,i_2) \in E^2,\ hasBinding(i_1,i_2) \Rightarrow client(i_1) \wedge server(i_2)$$

Au plus une liaison par interface simple. Une interface cliente de cardinalité *Single* ne peut être liée au plus qu'à une seule interface serveur.

$$\begin{aligned} bindingCardinality(E,R) \quad = \quad & \forall i \in E,\ single(i) \wedge client(i) \\ & \Rightarrow (\exists (i_1,i_2) \in E^2,\ hasBinding(i,i_1) \wedge hasBinding(i,i_2) \\ & \Rightarrow i_1 = i_2) \end{aligned}$$

Compatibilité des interfaces liées. Une liaison entre deux interfaces est valide si et seulement si l'interface serveur est un sous-type de l'interface cliente. Soit $\leq_i$ la relation de sous-typage du langage d'implémentation des interfaces de composants (ex : Java), alors :

$$bindingType(E,R) = \forall (i_1,i_2) \in E^2,\ hasBinding(i_1,i_2) \Rightarrow signature(i_2) \leq_i signature(i_1)$$

Toutes les interfaces requises doivent être liées. Un composant ne peut être dans l'état *Started* seulement si toutes ses interfaces clientes sont liées.

$$\begin{aligned} mandatoryBindings(E,R) \quad = \quad & \forall (c,i) \in E^2,\ started(c) \wedge hasInterface(c,i) \wedge client(i) \\ & \wedge\ mandatory(i) \Rightarrow \exists i' \in E,\ hasBinding(i,i')) \end{aligned}$$

Localité des liaisons primitives. Les liaisons primitives ne peuvent traverser les membranes des composants pour respecter le principe d'encapsulation. Il n'y a que trois catégories possibles de liaisons primitives :

1. les *liaisons normales*, entre interfaces dont les composants ont au moins un parent direct commun
2. les *liaisons d'export*, entre une interface cliente interne d'un composite et une interface externe serveur d'un de ses sous-composants directs
3. les *liaisons d'import*, entre une interface externe cliente d'un composant et une interface interne serveur d'un de ses parents directs.

$$\begin{aligned} normalBinding(i_c,i_s) \quad = \quad & client(i_c) \wedge external(i_c) \wedge server(i_s) \wedge external(i_s) \\ & \wedge\ \exists c_c, c_s, c_p \in E,\ hasInterface(c_c,i_c) \wedge hasInterface(c_s,i_s) \\ & \wedge\ hasChild(c_p,c_c) \wedge hasChild(c_p,c_s) \\ exportBinding(i_c,i_s) \quad = \quad & client(i_c) \wedge internal(i_c) \wedge server(i_s) \wedge external(i_s) \\ & \wedge\ \exists c_c, c_s \in E,\ hasInterface(c_c,i_c) \wedge hasInterface(c_s,i_s) \\ & \wedge\ hasChild(c_c,c_s) \\ importBinding(i_c,i_s) \quad = \quad & client(i_c) \wedge external(i_c) \wedge server(i_s) \wedge internal(i_s) \\ & \wedge\ \exists c_c, c_s \in E,\ hasInterface(c_c,i_c) \wedge hasInterface(c_s,i_s) \\ & \wedge\ hasChild(c_s,c_c) \\ localBinding(i_c,i_s) \quad = \quad & normalBinding(i_c,i_s) \vee exportBinding(i_c,i_s) \\ & \vee\ importBinding(i_c,i_s) \end{aligned}$$

Le prédicat de localité est ainsi le suivant :

$$localBindings(E,R) = \forall (i_c,i_s) \in E^2,\ hasBinding(i_c,i_s) \Rightarrow localBinding(i_c,i_s)$$

Etant donné toutes les définitions de ces contraintes d'intégrité, nous pouvons finalement définir l'ensemble des contraintes du modèle $\mathcal{F}_P$ par :

$$\begin{aligned}\mathcal{F}_P = \{\ & uniqueId, uniqueItfOwner, uniqueAttrOwner, uniqueItfNameVis,\\ & uniqueAttrName, standardControllers, parentsAreComposite,\\ & internalItfs, dualItfs, noCycle, bindingDirection,\\ & bindingCardinality, bindingType, mandatoryBindings,\\ & localBindings\ \}\end{aligned}$$

Système de typage des composants. Nous définissons en complément de ces contraintes la relation de sous-typage entre composants. Le système de typage de base dans Fractal repose sur le typage du langage sous-jacent de spécification des interfaces des composants. En effet, le type d'un composant en tant que boîte noire correspond à l'union des types de ses interfaces. Le principe de ce système est de garantir la substituabilité des composants. Soit $subTypeOf$ la relation de sous-typage Fractal également noté $\leq$ et $\leq_i$ la relation de sous-typage dans le langage d'implémentation sous-jacent des interfaces de composants (ex : Java) alors la relation de sous-typage pour les interfaces de composant s'exprime de la façon suivante :

$$\begin{aligned}\forall (i_1, i_2) \in E^2,\ i_1 \leq i_2 \quad = \quad & role(i_1) = role(i_2) \wedge name(i_1) = name(i_2)\\ & \wedge\ (collection(i_2) \Rightarrow collection(i_1))\\ & \wedge\ (server(i_1) \Rightarrow signature(i_1) \leq_i signature(i_2)\\ & \wedge\ (mandatory(i_1) \Rightarrow mandatory(i_2)))\\ & \wedge\ (client(i_1) \Rightarrow signature(i_2) \leq_i signature(i_1)\\ & \wedge\ (optional(i_2) \Rightarrow optional(i_1)))\end{aligned}$$

La relation de sous-typage $\leq$ pour les composants se traduit ensuite par :

$$\begin{aligned}\forall (c_1, c_2) \in E^2,\ c_1 \leq c_2 \quad = \quad & (\forall i_1 \in E,\ hasInterface(c_1, i_1) \wedge client(i_1)\\ & \Rightarrow \exists i_2 \in E,\ hasInterface(c_2, i_2) \wedge client(i_2) \wedge i_1 \leq i_2)\\ & \wedge\ (\forall i_2 \in E,\ hasInterface(c_2, i_2) \wedge server(i_2)\\ & \Rightarrow \exists i_1 \in E,\ hasInterface(c_1, i_1) \wedge server(i_2) \wedge i_1 \leq i_2)\end{aligned}$$

4.3.3 Extensions du modèle sous forme de contraintes

Une fois le coeur de la spécification défini sous forme de graphe associé à un ensemble de contraintes d'intégrité, nous pouvons étendre et spécialiser le modèle avec de nouvelles contraintes qui ne rentrent pas dans le modèle de base. Notre modèle de contraintes est constitué de trois niveaux (cf. Figure 4.10) correspondant à trois niveaux d'abstraction différents : le coeur du modèle Fractal correspondant à la spécification de base, le profil conforme au modèle qui est raffiné par un certain nombre de contraintes d'intégrité communes à un type d'application donné (ex : applications sans partage de composants), et enfin le niveau applicatif conforme à un profil qui ajoute des contraintes spécifiques à une configuration applicative donnée. Cette séparation en plusieurs niveaux de contraintes autorise ainsi plus de flexibilité pour la définition de la cohérence d'une application à base de composants Fractal. Les contraintes au niveau modèle sont normalement immutables et partagées par toutes les applications alors que les contraintes dans les profils et les contraintes applicatives peuvent être définies au besoin.

Extension avec des profils. Un profil est constitué d'un ensemble de contraintes d'intégrité $\mathcal{P}_i$ qui viennent s'ajouter au modèle. Les contraintes d'un profil sont de la même forme que les contraintes du modèle, elles sont :

– génériques et concernent un ensemble d'éléments d'une configuration,

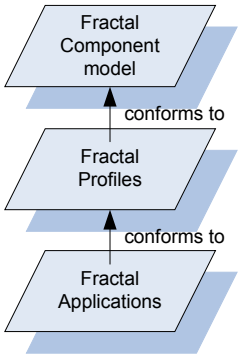

FIGURE 4.10 – Les trois niveaux de contraintes d'intégrité.

– valables pour un ensemble de configurations de systèmes.

Un profil doit rester conforme au modèle, par conséquent aucune contrainte du profil ne peut être en contradiction avec les contraintes du modèle. De même, un profil ne peut se définir que de manière additive (par des ajouts de contraintes), il n'est pas possible de retirer des contraintes à la définition du modèle. Un profil sert à valider un ensemble d'applications qui partage les mêmes propriétés architecturales génériques. Nous avons dès lors $\mathcal{M}_P = \mathcal{F}_P \cup \mathcal{P}_i$.

Un profil peut par exemple interdire le partage de composants en ajoutant la contrainte suivante stipulant qu'un composant ne peut avoir plus d'un seul composant parent :

$$\begin{aligned} noSharing(E,R) \quad = \quad & \forall (c_1,c_2,c_3) \in E^3,\ child(c_2,c_1) \\ & \Rightarrow \neg child(c_3,c_1) \end{aligned}$$

Il est aussi possible de garantir l'unicité de nommage des composants au même niveau hiérarchique de manière à avoir des chemins absolus dans l'architecture toujours différents pour les composants. Deux composants ont ainsi toujours deux chemins absolus différents en partant du composant racine du système. Nous définissons d'abord un prédicat pour déterminer si deux composants sont au même niveau hiérarchique :

$$siblings(c_1,c_2) = \exists c_3 \in E,\ hasChild(c_3,c1) \wedge hasChild(c_3,c2)$$

Le prédicat sur le nommage est alors le suivant :

$$uniqueName(E,R) = \forall (c_1,c_2) \in E^2,\ siblings(c_1,c2) \Rightarrow \neg(name(c_1) = name(c_2))$$

Contraintes applicatives. Les contraintes applicatives constituent le troisième niveau de contraintes que l'on peut exprimer sur une configuration. Elles doivent être conformes aux autres contraintes et ne pas être contradictoires entre elles. Ces contraintes sont :
– spécifiques à une configuration donnée,
– s'appliquent exclusivement à des instances particulières d'éléments dans une configuration.

Le fait que les contraintes applicatives ne s'appliquent qu'à des instances déterminées d'un système les rend peu réutilisables pour d'autre systèmes car elles ne sont généralement valables que pour ce système particulier. Ce type de contraintes est particulièrement utiles lorsque l'on veut définir des contraintes très spécifiques, peu réutilisables pour d'autres systèmes, et très ciblée dans l'architecture du système.

Les contraintes applicatives sont spécifiées de la même façon que pour les autres contraintes sinon qu'elles ne concernent que des composants et plus particulièrement des instances particulières de composants. Elles sont de la forme $\pi_i(E,R,id)$ ou id est l'identifiant de l'instance du composant sur lequel porte la contrainte. Si nous reprenons la contrainte générique de profil interdisant le partage de composants *noSharing* définie précédemment, nous pouvons la transformer en une contrainte applicative sur une instance bien précise de composant désigné par son identifiant id. La contrainte

interdira le partage uniquement pour cette instance et pas pour les autres composants de l'application. Pour une configuration $\mathcal{A} = (E, R)$, on aura donc :

$$\begin{aligned} noSharingApp(E, R, id) \quad = \quad & \exists c \in E,\ id(c) = id \\ & \wedge\ \forall (c_1, c_2) \in E^2,\ child(c_2, c) \\ & \Rightarrow \neg child(c_3, c) \end{aligned}$$

4.4 Traduction et vérification de la spécification

La spécification du modèle Fractal présentée en 4.3 nous est utile à des fins de documentations et pour raisonner de manière générale sur le modèle. Néanmoins, l'objectif principal est de pouvoir vérifier la validité d'une configuration et des reconfigurations pour un système réel. Pour cela, nous vérifions au préalable la cohérence de la spécification elle-même (cf. Figure 4.11). Le modèle spécifié en logique du premier ordre est traduit dans un langage de spécification analysable automatiquement. Le langage Alloy [Jac02] est adapté à nos besoins car il repose également sur la logique du premier ordre et est doté d'un analyseur de cohérence des spécifications. Pour la vérification dans un système concret à l'exécution, les contraintes d'intégrité de la spécification sont ensuite dans un langage exécutable sur un système Fractal développé en Julia. Nous avons choisi le langage FPath pour son intégration forte avec le modèle Fractal et son interpréteur programmé en Java.

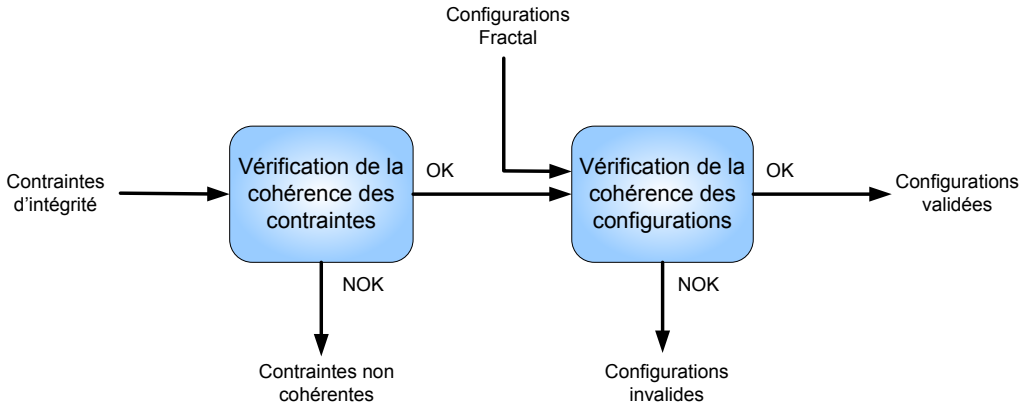

FIGURE 4.11 – Schéma globale de vérification de cohérence

Le but de cette section n'est pas de donner une traduction de toute les contraintes décrites dans les sections précédentes mais plutôt de donner les règles de traduction globale dans les langages cibles en fournissant notamment quelques exemples simples. On se référera à l'annexe A pour la traduction complète en Alloy.

4.4.1 Vérification de la cohérence de la spécification

Alloy (v4) [9] est un langage de spécification qui permet de vérifier la cohérence de notre modèle et des contraintes d'intégrité. Ce langage présente l'avantage d'être relativement simple, déclaratif et d'utiliser la logique du première ordre et la théorie des ensembles, ce qui facilite la traduction de notre modèle déjà exprimé dans ce formalisme. De plus, contrairement à d'autre langages de spécifications comme UML et OCL [OCL05], Alloy fournit un analyseur basé sur un solveur de contraintes du type SAT (« SATisfiability »en anglais) pour vérifier automatiquement la cohérence des spécifications, notamment la non contradiction entre les contraintes sur le modèle. L'analyseur de Alloy agit comme un *model checker* en générant des exemples et des contre-exemples d'instances du modèle si ils existent. Si l'analyse est finie et bornée, le modèle n'a cependant pas besoin de tenir compte de cette limitation dans sa spécification, Alloy permettant donc de simuler un modèle pour un nombre d'instances donné.

9. `http://alloy.mit.edu/alloy4/`

Alloy permet de regrouper les spécifications dans des modules séparés et de les importer dans d'autres modules. L'import d'un module est réalisé par la primitive *open* et est équivalent à inclure l'ensemble de la spécification contenu dans le module importé. Le coeur de la spécification Fractal pour la modélisation des configuration est donc défini dans un module principal (*module fractal/configuration*). Nous spécifions en Alloy notre modèle et l'ensemble des contraintes d'intégrité associées au modèle puis vérifions avec l'analyseur que le modèle est bien cohérent. Cette cohérence est traduite par le fait que l'analyseur est capable de générer un exemple d'instance du modèle. Nous donnons dans la suite les règles de traduction de notre modèle en Alloy.

Traduction des types de données primitifs. Les types de données sont traduits de manière presque immédiate en *atomes* (*atom*) dans Alloy. Les *atomes* sont des entités primitives indivisibles et immutables, ils peuvent être associés pour former des ensembles (*set*). Les atomes sont déclarés comme des signatures (*sig*). Le type Name est par exemple défini comme suivant :

```
sig Name {}
```

Le type Role ainsi que tous les types à valeur dans un ensemble nécessite de définir des sous-types. Role est ainsi défini en Alloy comme un ensemble dont Client et Server sont deux sous-ensembles disjoints :

```
abstract sig Role {}
sig Client, Server  extends Role {}
```

Traduction des opérateurs logiques. Alloy est basé sur la logique du première ordre et l'algèbre relationnel et tous les opérateurs logiques, mathématiques et ensemblistes utilisés dans notre spécification y sont définis.

Traduction des éléments architecturaux. Les éléments architecturaux sont définis comme des ensembles de propriétés. Nous définissons l'ensemble des éléments comme des entités possédant un identifiant.

```
abstract sig Element {
 id: Id
}
```

Les interfaces tout comme les composants et les attributs sont des sous-ensembles disjoints de cet ensemble :

```
sig Interface  extends Element {
 name : ItfName,
 visibility: Visibility,
 signature: Signature,
 role: Role,
 cardinality: Cardinality,
 contingency: Contingency
}
```

Traduction des relations architecturales. Tout est relation en Alloy, la traduction des quatre relations primitives de Fractal se fait donc en ajoutant le concept de configuration définie par un ensemble de chaque type d'élément architectural et de relations architecturales :

```
abstract sig Configuration {
 components : set Component,
 interfaces : set Interface,
 attributes : set Attribute,
 child: Component some -> some Component,
 interface: Component one -> some Interface,
 attribute: Component one -> some Attribute,
```

```
  binding: Interface lone -> lone Interface
  ...
}
```

La relation *hasChild* peut alors être testée par le prédicat Alloy suivant sur une configuration donnée :

```
pred hasChild [a:Configuration, parent:Component, child:Component] {
  parent->child in a.child
}
```

Le méta-modèle résultant des configurations Fractal sans les contraintes d'intégrité et généré par l'analyseur est présentée dans la figure 4.12. Les propriétés mutables y apparaissent sous forme de parallélogrammes, les propriétés immutables sous forme de rectangles. La spécification complète des configurations Fractal en Alloy est donnée en annexe A.1.1.

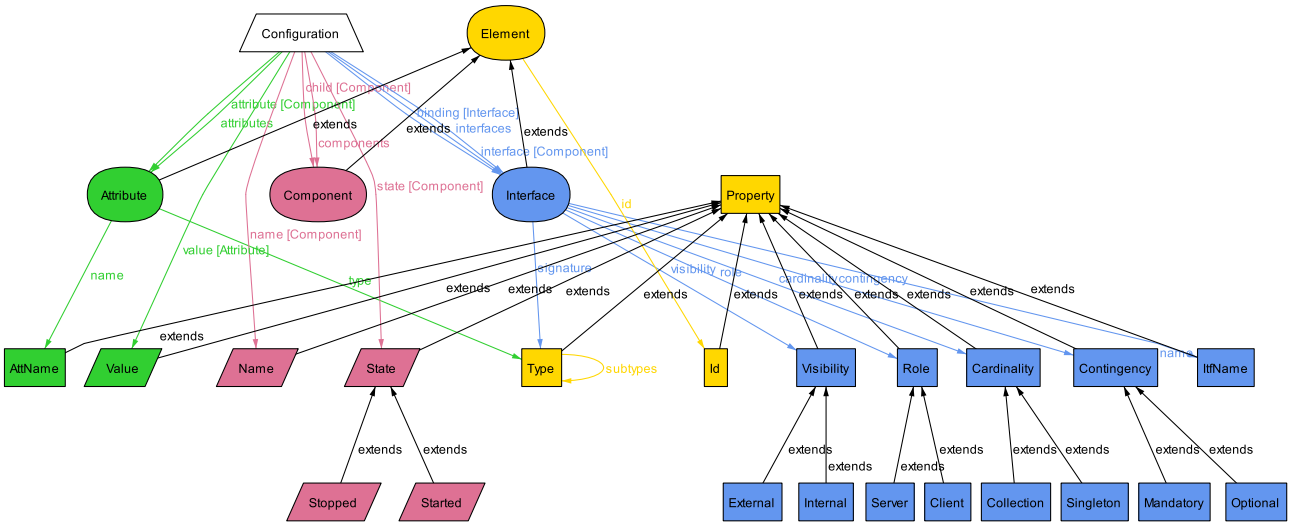

FIGURE 4.12 – Méta-modèle des configurations Fractal en Alloy

Traductions des contraintes d'intégrité sur les configurations Fractal. Alloy manipule des faits (*fact*) et des assertions (*assert*). Les faits sont des contraintes supposés toujours vrais dans la spécification. Ces invariants permettent ainsi de restreindre les instances possibles du modèle : lorsque l'analyseur cherche des exemples d'instance du modèle, il supprime tous ceux qui violent les faits. Les assertions servent à vérifier des propriétés sur le modèle, lorsque l'assertion est fausse, un contre-exemple est généré par l'analyseur. Nous représentons nos contraintes d'intégrité uniquement avec des faits car elles font partie de la spécification. Par contre, les assertions permettent un développement incrémentale de la spécification, avant d'être intégrées au modèle comme des faits, les contraintes peuvent en effet être traitées comme de simples assertions.

Un exemple de fait est la contrainte *noCycle* de non cyclicité dans la hiérarchie de composants :

```
fact noCycle {
  all a:Configuration |  no c: Component | c->c  in ^(a.child)
}
```

L'opérateur `^` désigne ici la fermeture transitive de la relation *child*. La spécification de l'ensemble des contraintes d'intégrité est donnée en annexe A.1.2.

Spécification des profils. La définition d'un profil se fait dans un nouveau module qui importe le module Fractal principal de définition des contraintes pour l'étendre avec de nouvelles contraintes sous forme de faits. Un profil doit être conforme au coeur du modèle, cette conformité est vérifiée en passant le module décrivant le profil dans l'analyseur Alloy 4.13.

Pour illustration, un profil qui interdirait le partage de composant se définirait comme tel :

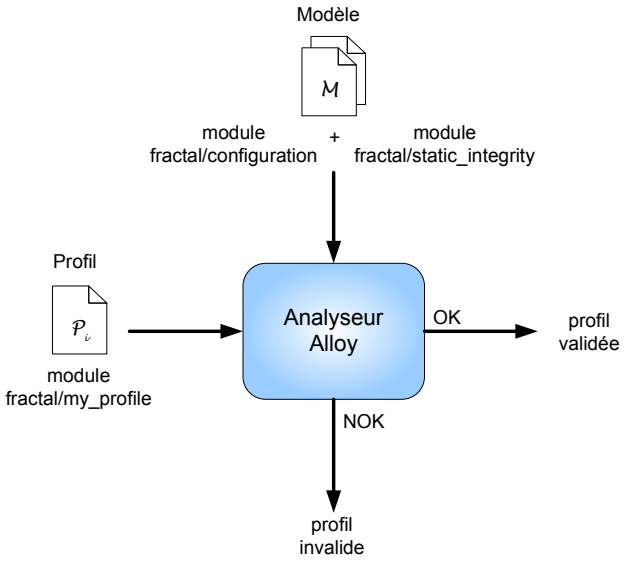

FIGURE 4.13 – Schéma de validation des contraintes des profils

```
module fractal/my_profile
open fractal/static_integrity

fact noSharing {
  all a: Configuration |  all c1, c2, c3: Component | (c1->c3  in a.child)
   => ! (c2->c3  in a.child)
}
```

Les contraintes applicatives. Les contraintes applicatives ne peuvent être intégrées dans la spécification du modèle en Alloy. En effet, ces contraintes s'appliquent à des instances d'éléments dans une configuration particulière et leur modélisation en Alloy nécessiterait une modélisation au niveau instance des configurations.

Le langage Alloy et son analyseur sont ainsi utilisés comme un outil de vérification de la cohérence de notre modélisation des configurations Fractal et de la non contradiction entre contraintes d'intégrité au niveau du modèle. Nous pouvons également vérifier de la même façon la conformité des contraintes dans des profils étendant le modèle.

4.4.2 Traduction des contraintes en langage exécutable

La spécification du modèle est traduite en langage exécutable pour pouvoir valider des configurations de systèmes réels. Le langage FPath [DLLC09] est particulièrement adapté pour spécifier les relations entre éléments et les contraintes d'intégrité sur ces éléments :

- C'est un langage de pure introspection sans effets de bord qui gère les types de données primitifs, les expressions arithmétiques et booléennes ainsi que les fonctions.
- FPath est conçu à la base pour être un langage de navigation et de sélection dans les architectures Fractal.
- Un interpréteur programmé en Java est disponible et rend le langage de contraintes directement exécutable dans une implémentation du modèle en Java comme Julia.

FPath sert ainsi à implémenter les contraintes d'intégrité du modèle et des profils et l'interpréteur FPath peut être utilisé pour valider ces contraintes pendant l'exécution du système. L'ensemble des contraintes d'intégrité associées à la spécification Fractal sont définies dans un fichier séparé (*fractal.spec*) chargé par le vérificateur de contraintes. Nous donnons dans la suite les règles de traduction des contraintes dans le langage FPath.

Traduction des types de données primitifs. FPath gère nativement les différents types de données utilisés que sont les chaînes de caractère, les nombres (en utilisant l'implémentation Java des *double*) et les booléens. FPath gère en plus les ensembles d'éléments (type $NodeSet$) retournés par les expressions.

Traduction des opérateurs logiques. Le langage gère les opérateurs de logique booléenne, $\wedge$ et $\vee$ ont leur équivalents en FPath *and* et *or*. L'implication n'existe pas en FPath mais peut être obtenue avec la relation classique $P \Rightarrow Q \Leftrightarrow \neg P \vee Q$. Des opérateurs ensemblistes existent pour l'union |, l'intersection & et la différence \. Les opérateurs mathématiques classiques et les comparaisons font aussi partie du langage. Il n'existe pas de quantificateurs en FPath, de plus toute expression se construit à partir d'une variable de départ. Le quantificateur existentiel se traduit donc par le test de non nullité du résultat retournée par les expressions FPath : $\exists \Leftrightarrow not(\$expr == null)$. Le quantificateur universel se traduira soit par l'utilisation du prédicat de sélection FPath $*$ qui permet de sélectionner tous les éléments d'un ensemble, soit par l'application sur tous les composants de manière recursive d'une fonction FPath donnée.

Traduction des éléments architecturaux. Les éléments architecturaux de la spécification sont les mêmes que dans FPath, la traduction est donc immédiate, les types de noeuds FPath correspondant étant $ComponentNode$, $InterfaceNode$ et $AttributeNode$.

Traduction des relations architecturales. Les relations architecturales correspondent à des axes de navigation, elles se traduisent donc simplement par des fonctions FPath utilisant un prédicat de sélection sur l'axe correspondant à la relation. La relation *hasChild* se traduit par exemple par la fonction FPath suivante :

```
function has_child(c1, c2) {
  return not($c1::child/c2 == null);
}
```

Traduction des contraintes d'intégrité. Les contraintes d'intégrité spécifiant le modèle Fractal sont traduites en expressions FPath. Il peut exister plusieurs traductions équivalentes pour une même contrainte, notamment quand FPath propose des fonctions utilitaires prédéfinies. De même, la traduction littérale de la contrainte exprimée en logique n'est pas toujours la plus concise.

La contrainte de non cyclicité dans les architectures *noCycle* qui fait partie de la spécification Fractal peut se traduire par l'application sur tous les composants d'une configuration de la fonction FPath suivante :

```
function no_cycle(c) {
  for component : $c::descendant/* {
   if ($c == $component) {
      return false();
  }
  return true();
}
```

La contrainte est finalement définie par l'expression FPath suivante :

```
no_cycle(.);
```

La syntaxe utilise le caractère . désignant en FPath l'élément courant, le vérificateur de contrainte l'interprétera comme l'application de la contrainte à tous les composants de l'application.

Spécification des profils. Les contraintes du profil se définissent de la même façon que les contraintes du modèle. Par exemple, pour définir la contrainte *noSharing* interdisant le partage de composant, plutôt qu'une traduction littérale de la logique du premier ordre, nous préférerons la fonction FPath suivante pour sa concision qui utilise la fonction *size* de FPath retournant la taille d'un ensemble :

```
function no_sharing(c) {
  return size($c/parent::*) <= 1;
}
```

Cette fonction s'appliquera récursivement à tous les composants d'une configuration :

```
no_sharing(.);
```

Les contraintes applicatives. Une contrainte applicative est une contrainte qui est associée à une instance particulière d'un composant dans la configuration d'un système. Les contraintes sont donc spécifiées dans la définition ADL de chaque composant. La forme de la contrainte est alors exactement la même que les contraintes d'un profil hormis le caractère . qui désigne alors l'élément auquel est associé la contrainte uniquement, i.e. le composant dont la définition ADL « encapsule »la définition de la contrainte. La contrainte *noSharing* spécifique à une instance se traduit donc comme pour un profil, seule l'interprétation du caractère . est différente :

```
<component name="server">
  ...
  <constraint value="no_sharing(.);" />
</component>
```

Les contraintes applicatives peuvent également être ajoutées ou retirées dynamiquement à l'aide d'un contrôleur dédié, elles sont alors spécifiées en FPath comme pour les contraintes spécifiées statiquement.

Validation des configurations Fractal. La configuration à un instant donné d'un système en Fractal est directement accessible par introspection grâce à la réflexivité du modèle de composant. Une configuration peut également être définie sous la forme d'une définitions en Fractal ADL. Le composant en charge de la validation des configurations (cf. figure 4.14) prend donc en entrée la définition d'une configuration (application ou définition ADL) accompagnée de ses contraintes applicatives, ainsi que les contraintes du modèle et des profils. Il détermine ensuite si la configuration en entrée satisfait ou non l'ensemble des contraintes d'intégrité.

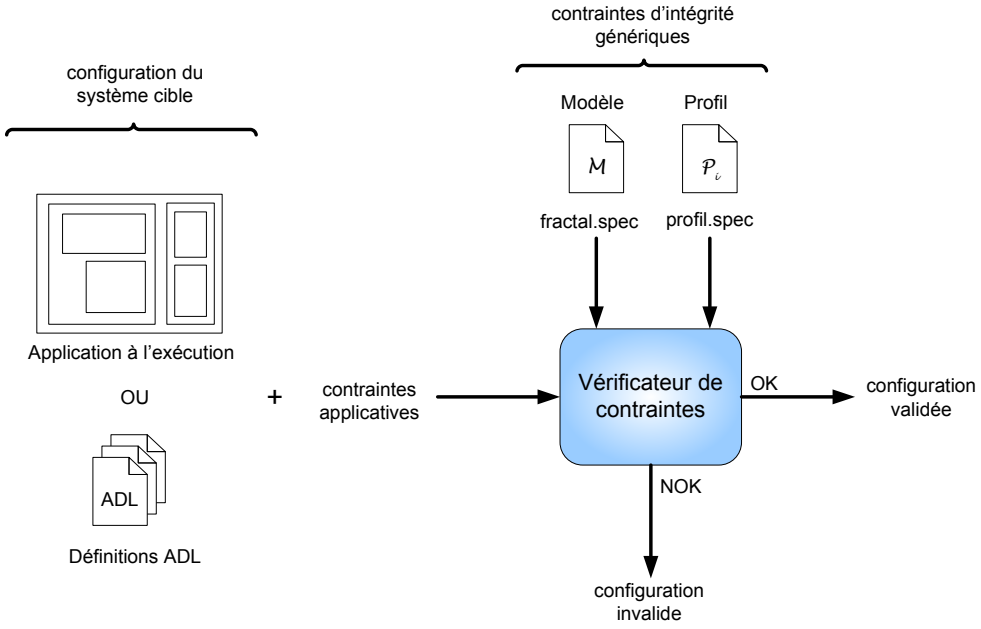

FIGURE 4.14 – Schéma de validation des configurations

4.5 Conclusion

Nous avons présenté dans ce chapitre le modèle Fractal et les différents outils et langages qui constituent son écosystème. Parmi ces outils, nous nous sommes intéressés tout particulièrement aux langages FPath et FScript, dédiés respectivement à l'introspection et à la reconfiguration des architectures des applications Fractal.

Après cette présentation du modèle Fractal, nous avons vu comment à partir de la spécification textuelle du modèle de composants, nous avons extrait et proposé un métamodèle simple de Fractal associé à un ensemble de contraintes d'intégrité. Ces contraintes d'intégrité sont des invariants de configuration qui permettent aussi bien de spécifier que d'étendre le métamodèle sous forme de profils. Elles servent à valider la cohérence des configurations dans Fractal, i.e. déterminer si la configuration est conforme au modèle. Nous avons exprimé ces contraintes dans un formalisme reposant sur la logique du première ordre indépendamment des langages d'implémentation.

Les contraintes ont finalement été traduites d'une part dans le langage de spécification Alloy puis dans le langage FPath utilisé ici comme langage de contraintes. La traduction en Alloy et de l'utilisation de l'analyseur Alloy a servi à vérifier la cohérence de la spécification et donc la non contradiction entre les contraintes (à l'image du travail fait pour la spécification Fractal fait dans [MS08]). La traduction en FPath a permis d'implémenter directement les mécanismes de support de ces contraintes dans des applications Fractal développées par exemple avec Julia.

Chapitre 5

Spécification des reconfigurations dynamiques

Sommaire

Ce chapitre propose la prise en compte de la dynamicité des modèles de composant pour définir la cohérence des applications en étudiant la sémantique des reconfigurations dynamiques. L'étude porte plus particulièrement sur le modèle Fractal, modèle réflexif supportant les reconfigurations dynamiques d'architectures. La section 5.1 reprend le concept de contraintes d'intégrité introduit dans le chapitre 4 pour spécifier les reconfigurations dynamiques dans le modèle Fractal. Nous nous intéresserons dans la section 5.2 à définir la sémantique des opérations de reconfiguration comme des opérations de transformation de graphes de configuration à base de préconditions et de postconditions et nous analysons leurs propriétés liées à la composition. Nous verrons enfin comment simuler les effets des opérations dans le modèle et comment les spécifier dans des applications concrètes à base de composants Fractal.

5.1 Dynamicité des architectures à base de composants

Pour supporter l'évolution des systèmes, certains modèles de composants ([MDEK95, BCL+06, CBG+04]) proposent des capacités de reconfiguration dynamique qui permettent de modifier ces systèmes pendant leur exécution sans nécessairement les arrêter totalement. Les reconfigurations dynamiques basées sur l'architecture [OMT98] utilisent les capacités réflexives des modèles pour manipuler les architectures des systèmes pendant leur exécution. Le composant comme unité de base de construction des systèmes est aussi la granularité de base des reconfigurations. L'étude des reconfigurations dynamiques dans les modèles de composants et plus particulièrement dans le modèle Fractal conduit à émettre un certain nombre d'hypothèses sur la nature des reconfigurations considérées et du système reconfiguré notamment en rapport à sa répartition. 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, de nouveaux invariants sont définis qui ne portent donc plus uniquement sur des configurations statiques mais sur les reconfigurations.

5.1.1 Hypothèses sur les reconfigurations dynamiques

Tout comme il n'existe pas de définition unique et universellement acceptée du concept de composant ([Szy02, HC01]), la nature des reconfigurations dynamiques peut être multiple suivant les modèles de composants, il est donc important de préciser la définition utilisée dans notre analyse. Cette définition découle du concept de reconfiguration dynamique dans le modèle Fractal et amène à émettre des hypothèses sur le système reconfiguré.

Composition des reconfigurations. En Fractal, une reconfiguration est réalisée à travers l'invocation d'opérations implémentées par les contrôleurs des composants. Ces opérations de reconfigurations sont dites primitives car elles produisent des modifications simples d'une configuration (ex : ajout de composant, modification de la valeur d'un attribut de composant, etc.). Elles sont composables pour constituer des reconfigurations plus complexes telles que le remplacement de composant qui suppose l'exécution des opérations suivantes :

- ajouter le nouveau composant dans tous les parents de l'ancien composant,
- arrêter l'ancien composant [1]
- déconnecter toutes les interfaces de l'ancien composant,
- retirer l'ancien composant de tous ses composants parents,
- connecter les interfaces du nouveau composant,
- démarrer le nouveau composant.

Intercession et introspection. Un modèle réflexif comme Fractal propose à la fois des capacités d'introspection des composants et des capacités d'intercession. La distinction entre intercession et introspection n'est pas explicite au niveau du modèle Fractal et les contrôleurs Fractal contiennent aussi bien des opérations d'introspection que d'intercession. Les opérations d'intercession sont des opérations sans effet de bord sur la configuration du système, elles permettent de connaître l'état de la configuration d'un système. L'opération d'introspection *getFcSubComponent* du *ContentController* permet ainsi de connaître l'ensemble des sous-composants d'un composant donné. Les opérations d'intercession modifient au contraire la configuration du système. L'opération d'intercession *addFcSubComponent* du *ContentController* ajoute un sous-composant dans un composant donné.

Distinction entre configuration et reconfiguration. La distinction faite entre configuration et reconfiguration repose généralement sur le moment du cycle de vie du système où est exécutée l'opération. La configuration est généralement effectuée avant l'exécution du système avant ou juste après son déploiement, les reconfigurations ont par contre lieu pendant l'exécution du système. La différence entre opérations de configuration et opérations de reconfiguration dans Fractal n'existe pas car les opérations exécutées sont les mêmes, Fractal considérant le cycle de vie complet des systèmes. En effet, les opérations utilisées pour instancier par exemple une configuration ADL sont les mêmes opérations de l'API utilisées pour les reconfigurations. Par exemple, l'ajout d'un composant lors de l'instanciation du système et pendant son exécution se traduit par le même appel à l'opération *addFcSubComponent* de l'API. Seul l'état du cycle de vie des composants impactés est alors différent.

Le cycle de vie de base d'un composant en Fractal comporte simplement deux état : *Stopped* et *Started* (cf. Figure 5.1) atteints en exécutant respectivement avec succès les opérations *stopFc* et *startFc* de l'API [2]. La sémantique de ces états est volontairement laissée imprécise dans la spécification du modèle. Ce cycle de vie peut être cependant étendu soit en précisant ou en modifiant la sémantique des états existants ou en ajoutant de nouveaux états dans l'automate, pour tenir compte par exemple de la problématique de déploiement (comme dans le modèle OSGi [OSG]). Ces états agissent sur le flot d'exécution au niveau des interfaces fonctionnelles des composants :

- Dans l'état *Started*, un composant peut émettre ou accepter des invocations d'opérations. L'invocation de *startFc* sur un composant dans l'état *Stopped* correspond donc à reprendre son activité au niveau fonctionnel.

1. Suivant la sémantique adoptée, il peut être nécessaire d'arrêter le parent du composant à retirer.
2. L'implémentation Julia ajoute un nouvel état au cycle de vie qui est un état interne non visible : *Stopping*. Ce nouvel état sert ainsi à synchroniser l'arrêt d'un composant et celui des ses sous-composants.

– Dans l'état *Stopped*, un composant ne peut émettre d'invocations d'opérations et ne peut en accepter que sur ses interfaces de contrôle. L'invocation de *stopFc* sur un composant dans l'état *Started* correspond donc à suspendre son activité au niveau fonctionnel.

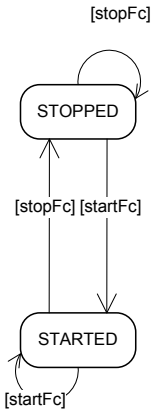

FIGURE 5.1 – Automate du cycle de vie d'un composant Fractal

Etant donné ce cycle de vie, une opération de reconfiguration peut être exécutée :
– sur une partie de système qui est dans un état arrêté (état *Stopped* des composants à reconfigurer),
– sur une partie de système qui est démarré (état *Started* des composants à reconfigurer).

Problématique du déploiement. Nous n'adressons pas dans notre analyse certains problèmes spécifiques au déploiement de logiciels. Le déploiement d'un logiciel dans son sens le plus large est un processus pouvant couvrir une part importante du cycle de vie du logiciel, les activités de déploiement peuvent en effet aller de l'installation du logiciel à sa désinstallation en passant par sa mise à jour, sa reconfiguration, l'activation et la désactivation de partie du logiciel [Dea07]. Nous nous intéressons au sens restreint qui désigne essentiellement l'installation et à la désinstallation du code des systèmes incluant la base logicielle nécessaire au fonctionnement de ces systèmes (système d'exploitation, JVM, etc.). Le déploiement n'est pas une opération spécifiée explicitement dans le modèle Fractal mais des extensions de Fractal (Jade [BHQS05]) propose ce mécanisme. Fractal ne définit ainsi pas de notion de modules en tant qu'unités de déploiement des composants. Pour les opérations d'instanciation de nouveaux composants, nous considérerons donc qu'un « bootstrap »est toujours présent sur la machine où a lieu l'instanciation et la création du « bootstrap »n'entre pas dans les opérations de reconfiguration que nous considérons. Le code utilisé pour l'instanciation peut ou non être présent sur la machine de déploiement, nous supposons dans une première approche que l'opération de déploiement est incluse dans l'opération d'instanciation qui se charge donc de récupérer si nécessaire le code du nouveau composant (mécanisme fourni de manière transparente par Fractal RMI). Cependant rien n'empêche de définir une nouvelle opération de déploiement indépendante de l'instanciation dans notre modèle.

Reconfiguration de systèmes distribués. La gestion de la distribution est supportée dans l'implémentation de référence Julia à travers l'utilisation des liaisons réparties de Fractal RMI. Elle n'apparaît pas explicitement dans la spécification sinon à travers la possibilité de réaliser des liaisons composites potentiellement répartis. Pour les opérations de reconfiguration, la distribution est gérée de manière transparente dans le modèle et l'implémentation de Fractal RMI. L'invocation d'une opération sur un composant qu'il soit local ou distant est la même. Des opérations supplémentaires sont cependant nécessaires pour la création du registre de nommage, et pour l'enregistrement et le désenregistrement des instances de composants dans le registre. Lors de l'instanciation de composant à distance, il est nécessaire d'enregistrer le composant de bootstrap distant dans le registre pour être accessible à distance.

5.1.2 Contraintes d'intégrité liées à la dynamicité des configurations

Dans cette section, nous étendons le concept de contraintes d'intégrité sur les configurations pour les appliquer à des architectures dynamiques, i.e. des architectures qui peuvent être reconfigurées pendant l'exécution des systèmes. Une configuration peut en effet 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 $\mathcal{A} = (E, R)$ en une autre configuration $\mathcal{A}' = (E', R')$ suivant le schéma suivant où $\mathcal{A}$ et $\mathcal{A}'$ sont respectivement les états initiaux et finaux du système et *op* l'opération associée à la transition entre ces deux états :

$$\mathcal{A} \xrightarrow{op} \mathcal{A}'$$

$\mathcal{A}$ et $\mathcal{A}'$ doivent être valides toutes les deux, i.e. conforme au même modèle $\mathcal{M}$. De plus, pour que $\mathcal{A}'$ soit une évolution valide de $\mathcal{A}$, 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 :

Les éléments architecturaux ne disparaissent pas. De nouveaux éléments peuvent être créés (ex : l'instanciation de composants) mais Fractal ne fournit pas (dans le modèle de base) de moyen de détruire un élément existant notamment un composant [3]. L'architecture résultant de la reconfiguration contient donc au moins l'ensemble des éléments présents dans l'architecture initiale :

$$noEltDestruction(E, E', R, R') = \forall e \in E,\ \exists e' \in E',\ id(e) = id(e')$$

Nous définissons le prédicat suivant pour comparer les éléments architecturaux entre différentes instances de l'architecture basé sur leur identificateur :

$$eq(e_1, e_2) = (id(e_1) = id(e_2))$$

Conservation des interfaces. L'ensemble des interfaces possédées par un composant est fixé lors de sa création [4].

$$\begin{aligned} itfConservationNoDestruction(E, E', R, R') \quad = \quad & \forall (c, i) \in E^2,\ hasInterface(c, i) \\ & \Rightarrow \exists (c', i') \in E'^2,\ eq(c, c') \wedge eq(i, i') \\ & \wedge\ hasInterface(c', i') \end{aligned}$$

$$\begin{aligned} itfConservationNoCreation(E, E', R, R') \quad = \quad & \forall (c', i') \in E'^2,\ hasInterface(c', i') \\ & \Rightarrow \exists (c, i) \in E^2,\ eq(c, c') \wedge eq(i, i') \\ & \wedge\ hasInterface(c, i) \end{aligned}$$

Immutabilité des interfaces. Les propriétés d'une interface ne peuvent être modifiées par une reconfiguration.

3. Cette limitation pourrait être supprimée par l'ajout d'une méthode de destruction dans l'interface *Factory* des usines de composants.

4. Strictement parlant, cette contrainte ne fait pas partie de la spécification mais elle est vérifiée dans toutes les implémentations connues du modèle en pratique.

$$
\begin{aligned}
itfImmutability(E, E', R, R') \quad = \quad & \forall i \in E,\ \exists i' \in E',\ eq(i, i') \wedge name(i) = name(i') \\
& \wedge\ signature(i) = signature(i') \\
& \wedge\ cardinality(i) = cardinality(i') \\
& \wedge\ contingency(i) = contingency(i') \\
& \wedge\ visibility(i) = visibility(i') \\
& \wedge\ role(i) = role(i')
\end{aligned}
$$

Conservation des attributs. L'ensemble des attributs appartenant à un composant est fixé au moment de la création du composant [5].

$$
\begin{aligned}
attConservationNoDestruction(E, E', R, R') \quad = \quad & \forall (c, a) \in E^2,\ hasAttribute(c, a) \\
& \Rightarrow \exists (c', a') \in E'^2,\ eq(c, c') \wedge eq(a, a') \\
& \wedge\ hasAttribute(c', a')
\end{aligned}
$$

$$
\begin{aligned}
attConservationNoCreation(E, E', R, R') \quad = \quad & \forall (c', a') \in E'^2,\ hasAttribute(c', a') \\
& \Rightarrow \exists (c, a) \in E^2,\ eq(c, c') \wedge eq(a, a') \\
& \wedge\ hasAttribute(c, a)
\end{aligned}
$$

Immutabilité du type et du nom des attributs. Le nom et le type d'un attribut ne peuvent être modifiés au cours d'une reconfiguration (les valeurs des attributs peuvent par contre être modifiées).

$$
\begin{aligned}
attImmutability(E, E', R, R') \quad = \quad & \forall a \in E,\ \exists a' \in E',\ eq(a, a') \\
& \wedge\ name(a) = name(a') \\
& \wedge\ type(a) = type(a')
\end{aligned}
$$

L'ensemble de ces contraintes dynamiques s'ajoute aux contraintes statiques définies dans la section 4.3 pour définir l'ensemble des invariants du modèle Fractal $\mathcal{F}_P$.

5.2 Analyse des opérations de reconfiguration dans Fractal

Les modèles de composants qui possèdent des capacités de reconfiguration dynamique exposent différentes catégories d'opérations. Des opérations modifient la structure du système (opérations de reconfiguration structurelle) et des opérations qui modifient le comportement (ou état) du système (opérations de reconfiguration comportementale). Nous analysons l'ensemble des opérations primitives du modèle de composants Fractal pour en extraire une spécification à base de pré et post-conditions. Nous en déduisons un certains nombre de propriétés intéressantes du point de vue de la composition. Nous finissons par traduire les préconditions et postconditions sur les opérations pour les simuler et les implémenter dans des applications concrètes.

5.2.1 Spécification de la sémantique des opérations primitives

Fractal propose un ensemble restreint d'opérations de reconfiguration primitives dans les contrôleurs standards de la spécification. Ces opérations sont cependant composables et étendues par la création de nouveaux contrôleurs. Nous proposons une taxonomie de ces opérations primitives pour le modèle Fractal dans la figure 5.2.

Nous distinguons plus généralement deux catégories d'opérations primitives dans le cadre des reconfigurations dynamiques :

5. Ceci est vérifié par l'implémentation standard de l'interface `attribute-controller`.

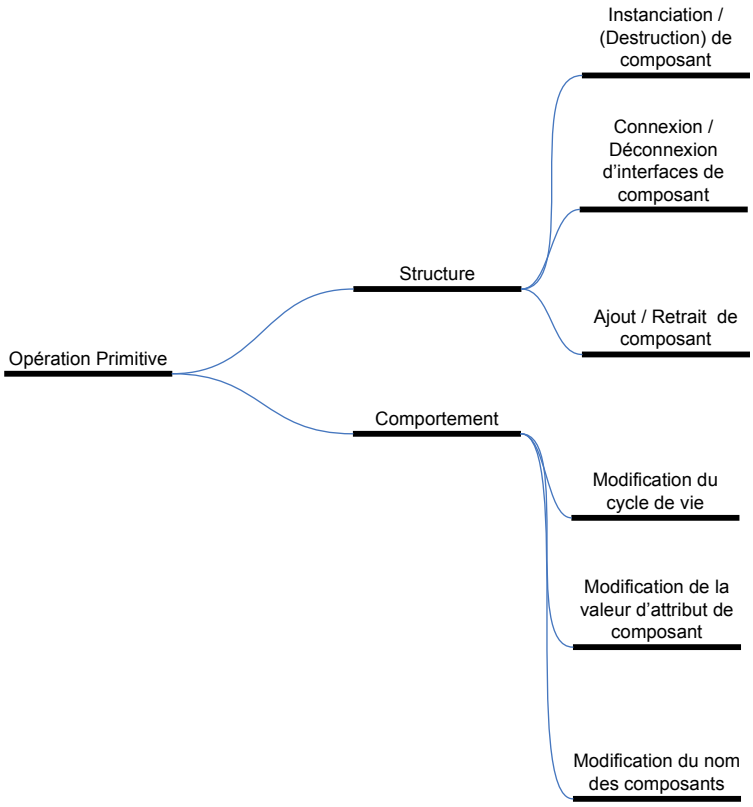

FIGURE 5.2 – Taxonomie des opérations de reconfigurations dans le modèle Fractal

– **Les opérations d'introspection** : elles comprennent les opérations accesseurs pour les propriétés des éléments architecturaux, et les opérations de navigation entre les éléments suivant les relations architecturales. Soit pour une configuration $\mathcal{A} = (E, R)$:

  Opérations accesseurs Une opération accesseur, étant donné un élément e de la configuration et le nom d'une propriété p, retourne la valeur x de cette propriété pour l'élément e. Elle est noté $e.p$.

$$\begin{aligned} op : \mathcal{F} &\longrightarrow Any \\ e &\longmapsto e.p \end{aligned}$$

  Opérations de navigation Une opération de navigation, étant donné un élément e et le nom r d'une relation $\mathcal{R}$, retourne l'ensemble des éléments en relation avec e. Elle est notée $e.r$.

$$\begin{aligned} op : E &\longrightarrow E \\ e &\longmapsto \bigcup_i e_i,\ \forall i\ e\mathcal{R}e_i \end{aligned}$$

– **Les opérations d'intercession** : elles modifient la topologie du graphe de configuration et les valeurs des propriétés des éléments. Une opération d'intercession op transforme une configuration $\mathcal{A} = (E, R)$ en une autre configuration $\mathcal{A}' = (E', R')$ avec $\mathcal{A}$ et $\mathcal{A}'$ conforme au modèle Fractal F. Une opération primitive de reconfiguration correspond à une transformation de graphe définie de manière générale comme :

$$\begin{aligned} op : \mathcal{F} &\longrightarrow \mathcal{F} \\ \mathcal{A} &\longmapsto \mathcal{A}' \end{aligned}$$

Ajout d'un élément architectural (ajout d'un sommet dans le graphe). Soit $e_i \in E$ l'élément ajouté, on a donc $E' = E \cup e_i$. Une telle opération sera notée : préfixe 'add-' + 'type d'élément'.

$$\begin{array}{rcl} op : \mathcal{F} & \longrightarrow & \mathcal{F} \\ (E,\ R) & \longmapsto & (E \cup e_i,\ R) \end{array}$$

Retrait d'un élément architectural (retrait d'un sommet dans le graphe). Soit $e_i \in E$ l'élément retiré, on a donc $E' = E \setminus e_i$. Une telle opération sera notée : préfixe 'remove-' + 'type d'élément'.

$$\begin{array}{rcl} op : \mathcal{F} & \longrightarrow & \mathcal{F} \\ (E,\ R) & \longmapsto & (E \setminus e_i,\ R) \end{array}$$

Ajout d'une relation architecturale (ajout d'un arc entre deux sommets dans le graphe). Soit $r_i \in R$ la relation ajoutée, on a donc $R' = R \cup r_i$. Une telle opération sera notée : préfixe 'connect-' + nom de la relation.

$$\begin{array}{rcl} op : \mathcal{F} & \longrightarrow & \mathcal{F} \\ (E,\ R) & \longmapsto & (E,\ R \cup r_i) \end{array}$$

Retrait d'une relation architecturale (retrait d'un arc entre deux sommets dans le graphe). Soit $r_i \in R$ la relation retirée, on a donc $R' = R \setminus r_i$. Une telle opération sera notée : préfixe 'disconnect-' + nom de la relation.

$$\begin{array}{rcl} op : \mathcal{F} & \longrightarrow & \mathcal{F} \\ (E,\ R) & \longmapsto & (E,\ R \setminus r_i) \end{array}$$

Modification de la valeur d'une propriété d'un élément architectural Soit p la propriété de l'élément $e_i \in E$ modifiée, v son ancienne valeur et v' sa nouvelle valeur, on a $E' = E$ et $R' = R$ car la topologie du graphe ne change pas et $\exists e_i \in E,\ e'_i \in E',\ id(e_i) = id(e'_i) \wedge e_i.p = v \wedge e'_i.p = v'$. Une telle opération sera notée : préfixe 'set-' + 'nom de l'élément' + 'nom de la propriété'.

$$\begin{array}{rcl} op : \mathcal{F} & \longrightarrow & \mathcal{F} \\ (E,\ R) & \longmapsto & (E',\ R'),\ E' = E \wedge R' = R \\ & & \wedge\ \exists e_i \in E,\ e'_i \in E',\ id(e_i) = id(e'_i) \wedge e_i.p = v \wedge e'_i.p = v' \end{array}$$

Nous ne considérerons par la suite dans notre analyse que les opérations d'intercession comme opérations de reconfiguration, seules susceptibles de rendre une configuration invalide par violation de contrainte.

Opérations de reconfiguration standard dans Fractal. Les opérations primitives décrites dans la spécification Fractal apparaissent dans le tableau 5.1 classées par contrôleur auquel elles appartiennent accompagnées de leur équivalence dans le langage de reconfiguration FScript (FScript n'étant pas typé statiquement, les paramètres des opérations FScript ne sont pas typés). L'objectif est de modéliser toutes ces opérations comme une transformation de graphe et donc de traduire directement leur effet sur le graphe de configuration.

Contrôleurs	Opérations dans l'API Java	Opérations FScript
ContentController	void addFcSubComponent(Component subComponent) void removeFcSubComponent(Component subComponent)	add(parent, child) remove(parent, child)
BindingController	void bindFc(String clientItfName, Object serverItf) void unbindFc(String clientItfName)	bind(clientItf, ServerItf) unbind(clientItf)
LifeCycleController	void startFc() void stopFc()	start(comp) stop(comp)
NameController	void setFcName(String name)	set-name(comp, name)
AttributeController	void setX(Object value)	set-value(attr, value)
Genericfactory	Component newFcInstance(Type type, Object controllerDesc, Object contentDesc)	new(definition)
Factory	Component newFcInstance()	new(definition)

TABLE 5.1 – Opérations de reconfiguration primitives dans Fractal (API Java et FScript)

Sémantique des opérations de reconfiguration. Nous utilisons les contraintes d'intégrité sous forme de préconditions et de postconditions pour spécifier la sémantique des opérations primitives de reconfiguration dans Fractal. La sémantique décrite est celle de la spécification Fractal, en cas de sous-spécification, la sémantique adoptée est celle de Julia avec la configuration standard des contrôleurs (spécifiée dans le fichier *julia-bundled.cfg*). Pour chaque opération, nous donnons son nom complet et un nom abrégé pour avoir une syntaxe plus concise.

Instanciation / destruction de composant. L'instanciation de composants dans Fractal passe par l'utilisation d'usines de composants qui sont elles-même des composants. Le modèle fait la distinction entre les usines génériques (fournissant l'interface *GenericFactory*) qui peuvent créer n'importe quel nouveau composant, et les usines standard (fournissant l'interface *Factory*) pour créer des composants d'un seul type. Il existe enfin une usine *bootstrap* du type *GenericFactory* qui ne nécessite pas de création explicite pour créer tout type de composants dont de nouvelles usines.

Les usines du type *GenericFactory* instancient de nouveaux composants par la méthode $newFcInstance$ qui nécessite un certain nombre de paramètres décrit dans la spécification dont le type du composant, un descripteur de membrane et de son contenu[6]. Fractal ADL offre aussi la possibilité d'instancier des composants à partir de leur définition ADL (type *Factory* de Fractal ADL). Nous ne voulons cependant pas restreindre l'opération d'instanciation à l'une ou l'autre des opérations proposées par Fractal ou Fractal ADL mais plutôt l'abstraire dans le cas général, nous avons pour cela défini le type générique $Definition$ qui par hypothèse contient les informations nécessaires à l'instanciation d'un nouveau composant.

- $add-component(Definition\ def_c, Component\ c)$ abrégée en new : instanciation d'un nouveau composant c (l'argument c contiendra l'instance du composant nouvellement créé par l'opération). L'opération ajoute un nouvel élément c de type *Component* dans le graphe. Cette opération suppose l'existence d'une définition def_c sous quelque forme que ce soit pour pouvoir instancier la nouveau composant. Cette définition peut être une définition ADL instanciable par une Factory ADL mais pas nécessairement, il peut également s'agir de n'importe quel descripteur qui peut être instancié par une factory comme une *GenericFactory* par exemple. Une précondition non exprimée et non exprimable dans notre modèle est la condition de validité de la définition passée en paramètre.

Instanciation d'un nouveau composant : $new(Definition\ def_c, Component\ c)$
Préconditions
Postconditions $E' = E \cup c$ $R' = R$

- Il faut noter que le modèle Fractal ne propose pas explicitement d'opération de destruction de composant du type $remove-component(Component\ c)$. Cette opération correspondrait à la suppression d'un composant d'une configuration et serait l'opération inverse de l'instanciation. L'opération n'existe pas non plus dans l'implémentation Julia, un composant isolé peut

6. cf. Fractal API : `http://fractal.objectweb.org/current/doc/javadoc/fractal/overview-summary.html`

cependant être déchargé de la mémoire par le mécanisme de *garbage collecting* en Java, si l'instance n'est plus référencée nulle part dans le système. Cette destruction n'est par contre pas explicitement invoquée et n'est pas garantie de fonctionner, des références peuvent en effet continuer d'exister même si le composant est arrêté, toutes ses interfaces déconnectées et sans composant parent. Le modèle OSGi [OSG] qui repose sur l'utilisation d'une plateforme Java supporte par contre cette opération à travers la désinstallation de « bundles »grâce à l'utilisation d'une hiérarchie de chargeur de classes. Les classes peuvent ainsi être déchargées à la demande [OSG].

Connexion / déconnexion d'interface de composant. Les reconfigurations des liaisons dans l'API Fractal sont gérées au niveau du *BindingController* par les méthodes *bindFc* et *unbindFc*. Nous définissons de la même façon deux opérations duales sur le graphe de configuration.

- $connect - hasBinding(Interface\ i_c, Interface\ i_s)$ abrégée en *bind* : connexion de deux interfaces i_c et i_s. L'opération vérifie la compatibilité des interfaces à connecter avant d'ajouter une relation *hasBinding* entre i_c et i_s.

Connexion de deux interfaces : $bind(Interface\ i_c, Interface\ i_s)$
Préconditions $i_c, i_s \in E$ $signature(i_s) <_i signature(i_c)$ $\neg\exists i'_s \in E,\ hasBinding(i_c, i'_s)$ $localBinding(i_c, i_s)$
Postconditions $E' = E$ $R' = R \cup \{hasBinding(i_c, i_s)\}$

- $disconnect - hasBinding(Interface\ i_c)$ abrégée en *unbind* : déconnexion d'une interface cliente i_c. L'opération vérifie notamment que le composant dont l'interface est à déconnecter est bien arrêté avant de supprimer une relation *hasBinding* entre i_c et l'interface serveur qui lui est connectée i_s.

Déconnexion de deux interfaces : $unbind(Interface\ i_c)$
Preconditions $i_c \in E$ $client(i_c)$ $\exists i_s \in E,\ hasBinding(i_c, i_s)$ $\forall (c, c') \in E^2,\ hasInterface(c, i_c) \Rightarrow stopped(c) \wedge (hasChild^{trans}(c, c') \Rightarrow stopped(c'))$
Postconditions $E' = E$ $R' = R \setminus \{hasBinding(i_c, i_s)\}$

Ajout / retrait de composant. Les reconfigurations du contenu des composants dans l'API Fractal sont gérées au niveau du *ContentController* par les méthodes *addFcSubComponent* et *removeFcSubComponent* que nous traduisons dans le graphe de configuration.

- $connect - hasChild(Component\ p, Component\ c)$ abrégée en *add* : ajout d'un sous-composant c dans le composant p. L'opération ajoute une relation *hasChild* entre p et c.

Ajout d'un sous-composant : $add(Component\ p, Component\ c)$
Préconditions $p, c \in E$ $\exists i \in E,\ hasInterface(p, i) \wedge contentCtrlItf(i)$ $\neg(p = c)$ $\neg hasChild^{trans}(c, p)$
Postconditions $E' = E$ $R' = R \cup \{hasChild(p, c)\}$

– *disconnect* − $hasChild(Component\ p, Component\ c)$ abrégée en *remove* : retrait du sous-composant c du composant p. L'opération supprime une relation *hasChild* entre p et c.

Retrait d'un sous-composant : $remove(Component\ p, Component\ c)$
Préconditions $p, c \in E$ $\exists i \in E,\ hasInterface(p, i) \wedge contentCtrlItf(i)$ $hasChild(p, c)$ $stopped(p) \wedge \forall c \in E,\ hasChild^{trans}(p, c) \Rightarrow stopped(c)$ $\forall (i, i', c') \in E^3,\ hasInterface(c, i) \wedge hasInterface(c', i') \wedge hasBinding(i, i') \Rightarrow c' \neq p \wedge$ $\neg hasChild(p, c')$
Postconditions $E' = E$ $R' = R \setminus \{hasChild(p, c)\}$

Modification du cycle de vie des composants. Les modifications du cycle de vie des composants sont gérées dans l'API Fractal par le $LifeCycleController$ avec les méthodes *startFc* et *stopFc*. Notre modélisation permet de supporter d'autres états que les deux états standard et supporter des extensions du cycle de vie de base des composants. La sémantique par défaut de deux états *stopped* et *started* est du type *suspend/resume*, la modélisation de cette sémantique liée à l'activité des composants n'est pas considérée ici. D'autre part, les opérations *startFc* et *stopFc* sont récursives dans l'implémentation Julia (i.e. elles s'applique à tous les sous-composants du composant considéré).

– *set* − $componentState(Component\ c, State\ state)$: modification de l'état du composant c. L'opération change la valeur de la propriété *state* du composant c. Elle est équivalente à un *startFc* ou *stopFc* non récursif suivant la valeur du paramètre (*started* ou *stopped*). Nous définissons dès lors les opérations recursives correspondantes *start* ou *stop*.

Arrêt d'un composant : $stop(Component\ c)$
Préconditions $c \in E$ $\exists i \in E,\ hasInterface(c, i) \wedge lifecycleCtrlItf(i)$
Postconditions $E' = E$ $\exists c' \in E',\ \forall c'' \in E',\ id(c') = id(c) \wedge stopped(c') \wedge hasChild(c', c'') \Rightarrow stopped(c'')$ $R' = R$

Démarrage d'un composant : $start(Component\ c)$
Préconditions $c \in E$ $\exists i \in E,\ hasInterface(c,i) \wedge lifecycleCtrlItf(i)$
Postconditions $E' = E$ $\exists c' \in E',\ \forall c'' \in E',\ id(c') = id(c) \wedge started(c') \wedge hasChild(c', c'') \Rightarrow started(c'')$ $R' = R$

Renommage de composant. Les composants peuvent être renommés grâce au *NameController* et la méthode $setFcName$ de l'API Fractal.

- $set - componentName(Component\ c, Name\ name)$ abrégée en $set - name$: modification du nom du composant c. L'opération change la valeur de la propriété *name* du composant c.

Renommage d'un composant : $set - name(Component\ c, Name\ n)$
Préconditions $c \in E$ $\exists i \in E,\ hasInterface(c,i) \wedge nameCtrlItf(i)$
Postconditions $E' = E$ $\exists c' \in E',\ id(c') = id(c) \wedge name(c') = n$ $R' = R$

Modification de la valeur d'attribut de composant. La valeur des attributs des composants peut être modifiés en passant par le contrôleur *AttributeController* avec des méthodes du type $setX$ ou X est le nom de l'attribut à modifier (la première lettre doit être mise en capitale). Nous lui faisons correspondre une opération dans le graphe.

- $set - attributeValue(Attribute\ a, Value\ v)$ abrégée en $set - value$: modification de la valeur de l'attribut a. L'opération change la valeur de la propriété v de l'attribut a.

Modification de la valeur d'un attribut : $set - value(Attribute\ a, Value\ v)$
Préconditions $a \in E$ $\exists (i,c) \in E^2,\ hasAttribute(c,a) \wedge hasInterface(c,i) \wedge attCtrlItf(i)$
Postconditions $E' = E$ $\exists a' \in E',\ id(a') = id(a) \wedge value(a') = v$ $R' = R$

On décrit l'ensemble des opérations primitives sur une configuration comme $\mathcal{O}$. L'ensemble des opérations primitives définies dans Fractal est alors le suivant :

$$\begin{aligned} \mathcal{F}_O = \quad \{ \quad & new, add, remove, \\ & bind, unbind, start, stop, \\ & set - name, set - value \ \} \end{aligned}$$

L'ensemble des préconditions et postconditions font partie des contraintes au niveau du modèle mais de nouvelles préconditions et postconditions peuvent être intégrées au sein d'un profil. Ces contraintes sont immutables tout comme les autres contraintes du modèle et des profils. Si les conditions sur les opérations de reconfiguration sont respectées, on en déduit que $\mathcal{F}_M \models \mathcal{A} \Rightarrow \mathcal{F}_M \models \mathcal{A}'$. Il faut noter qu'il n'existe pas de contraintes applicatives définies sous forme de pré ou postconditions, la sémantique des opérations est normalement la même pour tous les composants d'un système. Si une sémantique différente est nécessaire pour une opération sur un composant donné, une nouvelle opération doit être définie.

5.2.2 Propriétés des opérations primitives

Les opérations de reconfiguration modifient une configuration en manipulant les éléments architecturaux, leur propriétés et les relations entre les éléments. En tant que telles, elles peuvent posséder un certain nombre de propriétés classiques associés aux fonctions mathématiques. Nous nous intéresserons principalement aux propriétés utiles à la composition des opérations. Ces propriétés sont rarement explicitées dans la spécification Fractal, nous nous reposons donc en partie sur l'implémentation Julia pour les déduire.

Unicité du résultat et déterminisme. Etant donné deux configurations identiques, l'exécution de la même opération avec les mêmes arguments entraînera la même modification sur les configurations. Toutes les opérations primitives doivent être déterministes pour éviter les ambiguïtés dans leur sémantique.

Composition des opérations primitives. Les opérations primitives peuvent être composées pour former des opérations dite composites. Nous considérons ici que toute reconfiguration se traduit par l'exécution séquentielle d'opérations primitives. Par exemple, une opération de reconfiguration composite composée de n opérations transformera une configuration initiale $\mathcal{A}_0$ en une configuration finale $\mathcal{A}_n$:

$$\mathcal{A}_0 \overset{op_1}{\rightarrow} \mathcal{A}_1 \overset{op_2}{\rightarrow} \mathcal{A}_2 \overset{op_3}{\rightarrow} ... \overset{op_n}{\rightarrow} \mathcal{A}_n$$

Nous utilisons l'opérateur $\circ$ par analogie avec la composition de fonctions pour noter la composition séquentielle d'opération primitives. L'exemple $op_3 = op_2 \circ op_1$ signifie que l'opération op_3 est définie comme l'exécution séquentielle de op_1 puis de op_2 (op_1 est exécutée intégralement avant l'opération op_2). Les préconditions de op_2 doivent être satisfaire à l'issue de l'exécution de op_1. Les préconditions et postconditions de l'opération composite op_3 correspondent à la synthèse des préconditions et postconditions de op_1 et op_2.

Idempotence. L'idempotence est caractérisée pour une opération op par la relation suivante : $op = op \circ op$. Une opération idempotente a le même effet qu'on l'applique une ou plusieurs fois, nous rajoutons à la définition le fait que l'exécution est réalisée de manière strictement consécutive. Cette propriété est acquise pour des opérations indépendantes comme les opérations de lecture et d'écriture simples (ex : mises à jour dans les bases de données) mais elle ne l'est pas nécessairement pour des opérations plus complexes potentiellement interdépendantes comme les opérations de reconfiguration considérées. Le tableau de la table 5.2 donne l'idempotence pour les opérations primitives en fonction de leur sémantique définie précédemment. L'idempotence peut ne pas être valable quelques soient les arguments d'une opération, c'est le cas de l'opération d'instanciation *new* qui est uniquement idempotente dans le cas d'une usine de composant de type singleton qui retourne toujours la même instance pour les mêmes arguments, nous considérons alors que l'opération n'est pas idempotente de manière générale. L'idempotence est utile notamment pour certains algorithmes de recouvrement dans lesquels des opérations peuvent être exécutées plusieurs fois consécutivement. En effet, lors d'une défaillance, le processus de recouvrement va exécuter à nouveau les opérations qui n'ont pas été validée avant la défaillance et il peut arriver qu'une opération non validée ayant déjà été exécutée le soit une deuxième fois après la défaillance.

Inversibilité. La propriété d'inversibilité se traduit par la relation suivante : $\exists op^{-1} : op^{-1} \circ op = Id$ où Id est l'opération identité qui ne modifie rien dans la configuration. On a pour une opération composée $op_3 = op_2 \circ op_1$ la relation $op_3^{-1} = (op_2 \circ op_1)^{-1} = op_1^{-1} \circ op_2^{-1}$. Toutes les opérations ne sont pas nécessairement inversibles. Les opérations de modification des propriétés sont généralement leur propre inverse. D'autre part, l'opération inverse (ou réciproque) d'une opération primitive n'est pas nécessairement une opération primitive mais peut être une opération composite en fonction de la sémantique donnée à l'opération (ensemble des préconditions et des postconditions). Enfin, l'inverse d'une opération peut également dépendre de l'état de la configuration sur laquelle est exécutée l'opération.

Opérations	Idempotence
$add(Component\ p, Component\ c)$	
$remove(Component\ p, Component\ c)$	
$bind(Interface\ i_c, Interface\ i_s)$	
$unbind(Interface\ i_c)$	
$stop(Component\ c)$	X
$start(Component\ c)$	X
$set-name(Component\ c, Name\ name)$	X
$set-attr(Attribute\ a, Value\ value)$	X
$new(Definition\ d, Component\ c)$	

TABLE 5.2 – Idempotence des opérations de reconfiguration dans Fractal/Julia

Soit une configuration $\mathcal{A}$, par application de l'opération composite $op^{-1} \circ op$ sur $\mathcal{A}$, on obtient normalement $op^{-1} \circ op(\mathcal{A}) = \mathcal{A}$ suivant le schéma suivant où $\overset{op}{\rightarrow}$ désigne la reconfiguration par l'opération op :

$$\mathcal{A} \overset{op}{\rightarrow} \mathcal{A}' \overset{op^{-1}}{\rightarrow} \mathcal{A}$$

On a donc $op(\mathcal{A}) = \mathcal{A}'$ avec $\mathcal{A} = (E, R)$ et $\mathcal{A}' = (E', R')$. A l'exception de l'opération *new*, la relation $E = E'$ est vérifiée (égalité des identifiants), car aucun élément n'apparaît ni ne disparaît de la configuration, seules leurs propriétés et les relations architecturales peuvent être modifiées. On note dès lors p une propriété d'un élément de $\mathcal{A}$ et p' la propriété correspondante du même élément de $\mathcal{A}'$. Nous pouvons alors définir les opérations inverses des opérations primitives :

Inverse de l'opération *add*. L'inverse de *add* dépend de l'état initial du cycle de vie du composant parent p étant donné que l'opération inverse *remove* possède une précondition sur cet état.

$$\begin{aligned} \forall (p,c) \in Component^2,\ p.state = stopped &\Rightarrow add^{-1}(p,c) = remove(p,c) \\ p.state = started &\Rightarrow add^{-1}(p,c) = start(p) \circ remove(p,c) \\ &\quad \circ stop(p) \end{aligned}$$

Inverse de l'opération *remove*.

$$\forall (p,c) \in Component^2,\ remove^{-1}(p,c) = add(p,c)$$

Inverse de l'opération *bind*. L'inverse de *bind* dépend de l'état du cycle de vie du composant auquel appartient l'interface cliente i_c étant donné que l'opération inverse *unbind* possède une précondition sur cet état.

$$\begin{aligned} \forall (i_c, i_s) \in Interface^2, \exists c \in Component, \quad & hasInterface(c, i_c) \wedge c.state = stopped \\ & \Rightarrow bind^{-1}(i_c, i_s) = unbind(i_c, i_s) \\ & hasInterface(c, i_c) \wedge c.state = started \\ & \Rightarrow bind^{-1}(i_c, i_s) = start(c) \circ unbind(i_c, i_s) \\ & \circ stop(c) \end{aligned}$$

Inverse de l'opération *unbind*. L'inverse de *unbind* nécessite de connaitre l'interface serveur i_s à laquelle i_c était liée.

$$\forall (i_c, i_s) \in Interface^2,\ hasBinding(i_c, i_s) \in R \Rightarrow unbind^{-1}(i_c) = bind(i_c, i_s)$$

Inverse de l'opération *start*.

$$\forall c \in Component,\ start^{-1}(c) = stop(c)$$

Inverse de l'opération *stop*.

$$\forall c \in Component,\ stop^{-1}(c) = start(c)$$

Inverse de l'opération $set - name$**.**

$$\forall c \in Component, \quad (name, name') \in Name^2, \\ set - name^{-1}(c, name') = set - name(c, name)$$

Inverse de l'opération $set - value$**.**

$$\forall a \in Attribute, \quad (value, value') \in Value^2, \\ set - value^{-1}(a, value') = set - value(a, value)$$

Inverse de l'opération *new*. L'inverse de l'opération *new* n'est pas défini étant donné qu'il n'existe pas d'opération de destruction de composants.

L'inversibilité est une propriété servant à défaire les effets d'une opération de reconfiguration et est utile pour assurer son atomicité.

Commutativité. La commutativité est caractérisée pour deux opérations op_1 et op_2 par la relation $o_p1 \circ op_2 = op_2 \circ op_1$, elle est utile en cas de réordonnancement de l'exécution des opérations de reconfiguration primitives à l'intérieur d'un reconfiguration composite. Pour des raisons d'efficacité, ou de minimisation de l'indisponibilité d'une partie de l'application, il peut en effet être utile d'optimiser l'exécution de la reconfiguration en réordonnançant les opérations. Par exemple, les opérations d'instanciation qui ne nécessite pas l'arrêt d'une partie du système peuvent être exécutées en premier sans interférer avec le reste de la reconfiguration. La commutativité de deux opérations dépend notamment des éléments et des relations qu'elles modifient dans le graphe de configuration. Les opérations ne satisfont pas nécessairement à la propriété d'indépendance de leur représentations : l'exécution d'une opération op_1 peut valider ou invalider la précondition d'une opération op_2. Par exemple, pour un composant c qui possède une seule interface cliente i_1 qui est connectée à l'interface i_2 d'un autre composant, l'exécution d'une opération $unbind(i_1,\ i_2)$ puis $remove(c)$ est valide (toutes les interfaces du composant doivent être déconnectées pour l'enlever) mais l'exécution dans l'ordre inverse ne l'est pas. La commutativité entre opérations est ainsi déduite de la spécification de leur sémantique à base de préconditions et de postconditions.

Récursivité. Seules les opérations *start* et *stop* sont définies par construction comme récursives parmi les opérations primitives. Elles s'appliquent en effet récursivement à la fermeture transitive réflexive de la relation *hasChild* pour le composant en paramètre, i.e. au composant et à tous ses descendants. Cette sémantique correspond à celle de Julia.

5.2.3 Traduction des conditions sur les opérations de reconfiguration

De la même façon que pour les invariants de configurations du chapitre 4, nous traduisons également les préconditions et postconditions dans un premier temps en Alloy pour être capable de simuler les effets des opérations sur notre modèle et dans un second temps en FPath pour pouvoir les intégrer dans des applications réelles.

Simulation des opérations en Alloy. Nous utilisons les prédicats Alloy pour représenter les préconditions et postconditions sur les opérations de configurations (cf. Figure 5.3). Un prédicat est constitué d'une liste de contraintes qui doivent être toutes satisfaites pour que le prédicat soit vérifié. Nous pouvons modéliser les transitions entre états initial et final d'un système en introduisant en paramètres du prédicat les configurations avant et après l'exécution de l'opération de reconfiguration. Les prédicats sont simulés avec l'analyseur Alloy en utilisant la commande *run* pour un nombre limité d'instance du modèle.

Un exemple de simulation du prédicat *pred* pour n instances *Object* est le suivant :

```
run pred for n Object
```

L'ensemble des prédicats pour les opérations de reconfiguration sont intégrés dans un module séparé (*module fractal/reconfiguration*). Pour l'opération *add*, le prédicat correspondant est par exemple le suivant :

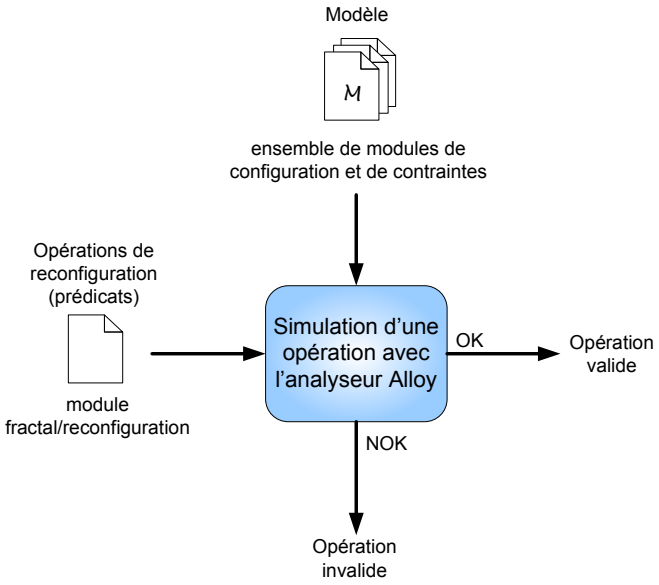

FIGURE 5.3 – Simulation des opérations dans Alloy

```
pred add [a1, a2: Configuration, p, c: Component] {
 // Preconditions
 (c + p)  in a1.components
 not p->c  in a1.child
 isComposite[a1, p]
 not c->p  in *(a1.child)

 // Elements
 sameElements[a1, a2]
 // Relations
 a2.child = a1.child + p->c
 a2.interface = a1.interface
 a2.attribute = a1.attribute
 a2.binding = a1.binding
 // Properties
 sameProperties[a1, a2]

 // Postconditions
 p->c  in a2.child
}
```

La configuration initiale est $a1$ et la configuration finale est $a2$. Nous utilisons le prédicat *isComposite* non définie ici qui teste la présence d'une interface *ContentController* pour un composant donné. La relation $*child$ est la fermeture réflexive transitive de la relation *child*. La spécification de l'ensemble des opérations primitives de reconfiguration est donnée en annexe A.2.1 et les contraintes dynamiques en annexe A.2.2.

Traduction des conditions sur les opérations en FPath. Une reconfiguration en Fractal peut soit être programmée directement avec l'API des contrôleurs ou grâce au langage de reconfiguration FScript. Il existe de même deux cas de spécification possibles des préconditions et des postconditions pour les reconfigurations dans les applications : en utilisant directement l'API Fractal, ou via l'utilisation du langage de reconfiguration FScript. Etant donné que l'interpréteur FScript utilise l'API Fractal pour l'invocation des opérations de reconfiguration dans Fractal, la première méthode de spécification au niveau de l'API est compatible avec l'utilisation de FScript.

Lors de l'exécution d'une opération de reconfiguration de l'API Fractal en Java, l'invocation est interceptée au niveau du contrôleur et les paramètres de l'opération sont automatiquement transfor-

més en noeuds FPath pour pouvoir être manipulés par l'interpréteur. Par exemple pour le retrait de composant, l'opération de l'API est :

```
void removeFcSubComponent(Component subComponent)
```

Le composant courant auquel appartient le contrôleur sur lequel est invoqué l'opération est automatiquement wrappé dans la variable FPath *this*, le paramètre *subComponent* est lui aussi wrappé dans une variable du type FPath *ComponentNode*. Les préconditions de l'opération peuvent ainsi être validées. Comme illustration, pour exécuter l'opération de retrait de composant, il doit être vérifié que le composant parent contient bien le sous-composant à retirer et que le parent est bien arrêté :

```
void removeFcSubComponent(Component subComponent) {
  ...
  // preconditions:
  size(intersection($subComponent, $this/child::*) == 1);
  stopped($this);
  ...
}
```

Dans le cas de FScript, de nouvelles opérations comportant des préconditions et des postconditions sont définies à partir des opérations de base de FScript pour le modèle Fractal. A chaque opération primitive correspond la même opération intégrant les contraintes. Nous intégrons les assertions dans FScript en définissant une nouvelle procédure *assert* qui prend en paramètre le résultat d'une expression FPath retournant obligatoirement un booléen. Par exemple pour l'opération FScript $add(p, c)$ définie comme l'ajout d'un composant Fractal dans un autre composant, une nouvelle opération $addSafe(p, c)$ est définie comme :

```
action addSafe(p, c) {
  //Preconditions
  assert($p/interface::content-controller)); -- p must be a composite component
  assert(not($p == $c)); // p must not equals c
  assert(size(intersection($p, $c/descendant::*)) == 0); -- p must not be a
                                                        -- descendant of c

  //Operation execution
  add($p, $c);

  //Postconditions
  assert(size(intersection($c, $p/child::*)) == 1); -- c must be a child of p
 }
```

La traduction des contraintes d'intégrité en FPath (invariants, préconditions et postconditions) permet ainsi de valider la configuration Fractal d'un système à l'exécution ainsi que les reconfigurations dynamiques du système.

5.3 Conclusion

Ce chapitre a été l'occasion 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 préciser la nature des reconfigurations dynamiques dans le modèle de composant Fractal.

Nous avons ensuite spécifié la sémantique des opérations de reconfiguration comme des opérations de transformation du graphe de configuration avec un ensemble de préconditions et de postconditions. Différentes propriétés liées à la composition des opérations sont ensuite étudiées pour les opérations primitives définies dans le modèle Fractal telles que la commutativité ou l'idempotence.

Ces contraintes sont traduites dans le langage de spécification Alloy pour simuler les effets des opérations de reconfiguration sur le modèle d'une configuration et elles sont implémentées en FPath pour pouvoir être vérifiées pendant l'exécution d'une application Fractal.

Chapitre 6

Une approche transactionnelle pour fiabiliser les reconfigurations dynamiques

Sommaire

Ce chapitre présente l'approche transactionnelle développée pour la fiabilisation des reconfigurations dynamiques dans les modèles de composants et plus particulièrement dans le modèle Fractal [LLC07]. Nous présentons dans la section 6.1 notre approche à base de transactions en montrant notamment l'utilité des propriétés classiques des transactions (propriétés ACID) pour fiabiliser les reconfigurations dynamiques. Nous définissons le concept de reconfiguration transactionnelle en nous reposant sur la définition déjà présentée des reconfigurations dynamiques dans les chapitres 4 et 5. Nous nous intéressons particulièrement au modèle de transactions mis en oeuvre et adapté dans ce contexte. Nous consacrons enfin plusieurs sections à la description des différentes propriétés ACID des reconfigurations transactionnelles : l'atomicité (section 6.2), la cohérence (section 6.3), l'isolation (section 6.4) et la durabilité (section 6.5).

6.1 Définition des reconfigurations transactionnelles

Les transactions sont un moyen de fiabiliser les reconfigurations dynamiques en assurant le maintien de la cohérence dans les systèmes reconfigurés. Nous définissons le concept de reconfiguration transactionnelle dans les architectures à composants et proposons un modèle de transactions en

conséquence. Le modèle de transactions est défini pour répondre aux caractéristiques et aux besoins spécifiques des reconfigurations dynamiques.

6.1.1 Motivation de l'approche transactionnelle

La fiabilité est un attribut essentiel de la sûreté de fonctionnement en tant que mesure de la continuité de la délivrance d'un service correct. Cette caractéristique participe notamment à la disponibilité des systèmes, disponibilité qui est un problème critique pour de nombreux systèmes informatiques. Par conséquent, dans le domaine du génie logiciel cette préoccupation est prise en compte tout le long du cycle de vie du logiciel et de ses évolutions. Pour faire évoluer les systèmes, nous considérons les reconfigurations dynamiques basées sur l'architecture [OMT98]. Plus particulièrement, nous utilisons les propriétés réflexives d'un modèle à composant comme Fractal [BCL+06] pour introspecter et modifier l'architecture des systèmes. Le composant sert ainsi à la fois d'unité d'assemblage et de reconfiguration dans les applications Fractal. Les reconfigurations dynamiques peuvent être structurelles (ajout, retrait de composant, etc.) ou comportementales (changement de l'état du cycle de vie des composants). Cependant, reconfigurer un système pendant son exécution peut le laisser dans un état incohérent et donc réduire sa fiabilité. Il est donc nécessaire de mettre en place des mécanismes spécifiques pour s'assurer de la fiabilité des évolutions des systèmes pendant leur exécution.

Le maintien de la cohérence dans un système est nécessaire à la garantie de sa fiabilité. Nous avons donc donné une définition de la cohérence d'un point de vue architectural pour les configurations Fractal et les reconfigurations (cf. chapitres 4 et 5) sous forme de contraintes d'intégrité. Il s'agit alors de garantir que cette cohérence est maintenue au cours des reconfigurations dynamiques, c'est à dire que l'ensemble des contraintes d'intégrité sont bien respectées pour un système donné. En effet, un système reconfiguré peut par exemple ne plus être conforme avec le modèle de composant choisi pour le concevoir et violer certaines contraintes d'intégrité. Ajouter un composant *C1* dans un autre composant *C2* est ainsi une reconfiguration invalide dans le modèle de composants Fractal si elle crée un cycle dans les hiérarchie de composants (i.e. si *C2* contient déjà *C1*). Par conséquent, la violation d'une contrainte d'intégrité constitue donc une faute dans le système reconfiguré et l'objectif est de pouvoir prévenir ou réparer ce type de faute.

Par ailleurs, le maintien de la cohérence doit être réalisé sous les hypothèses de base considérées pour les reconfigurations dynamiques :

- Les reconfigurations sont potentiellement concurrentes : plusieurs reconfigurations peuvent être exécutées en parallèle et il est donc nécessaire de les coordonner pour éviter les conflits si leurs effets concernent les mêmes éléments architecturaux du système. D'autre part, les reconfigurations se produisent pendant l'exécution du système, elles doivent donc être synchronisées avec le niveau fonctionnel.
- Les reconfigurations sont non-anticipées, ce qui signifie qu'elles ne sont pas nécessairement conçues et prévues au moment de la conception des systèmes. Elles sont donc d'autant plus susceptibles d'être des transformations incorrectes de l'état de ces systèmes et ne peuvent être validées qu'au moment de leur exécution.
- Les reconfigurations sont réparties, l'architecture du système reconfiguré peut être distribuée sur plusieurs machines physiques, ce qui doit être pris en compte par les mécanismes de maintien de la cohérence.

Différentes techniques permettent de garantir la fiabilité d'un système. Nous distinguons la prévention de fautes qui a pour objectif la détection et l'élimination des fautes avant l'exécution du système, et la tolérance aux fautes qui vise à réparer la faute une fois qu'elle s'est produite dans le système. Un des moyens répandus de fiabilisation par la tolérance aux fautes est le recouvrement d'erreurs par l'intermédiaire de transactions. Les transactions sont utilisées comme unités d'exécution pour assurer la cohérence des données dans de nombreux systèmes informatiques, particulièrement dans les SGBD (Systèmes de Gestion de Base de Données) [GMUW00]. Pour simplifier, une transaction peut être représentée par une séquence d'opérations de lectures et d'écritures sur un ensemble d'objets dans un système. Les transactions sont dotées de propriétés pour le support de la concurrence, du recouvrement d'erreurs, et de garantie de cohérence dans les systèmes distribués.

Les propriétés fondamentales des transactions, communément appelées « propriétés ACID »[GR92], permettent d'améliorer la fiabilité des reconfigurations dynamiques d'applications les rendant tolé-

rants aux fautes, i.e. la cohérence du système est maintenu malgré les reconfigurations concurrentes et les défaillances. Les transactions apportent ainsi aux reconfigurations dynamiques et aux systèmes reconfigurés le support du contrôle de la concurrence (Propriété I), le recouvrement en cas de panne (Propriétés A et D) et des garanties de cohérence (Propriété C). Les transactions constituent ainsi des unités de reprise (plus petite unité de traitement susceptible d'être annulée) et des unités de gestion de concurrence (unité d'exécution à synchroniser). Le contrôle de concurrence a pour objectif de synchroniser les accès concurrents aux objets du systèmes et repose sur le principe de sérialisabilité [GR92] des transactions. Le recouvrement dans un système transactionnel classique utilise la reprise sur défaillance pour garantir qu'en cas de défaillance dans le système (logicielle ou matérielle), les effets des transactions dont l'exécution est totale soient visibles dans le système alors que les effets des transactions exécutées que partiellement soient invisibles.

Les propriétés ACID peuvent se traduire dans le contexte des reconfigurations dynamiques de composants et nous proposons donc les définitions suivantes :

- *Atomicité.* Les opérations de reconfigurations sont exécutées et le système est reconfiguré ou bien le système n'est pas reconfiguré et demeure dans son été initial avant la l'exécution de la reconfiguration.
- *Cohérence.* Une reconfiguration est une transformation valide de l'état du système d'un état cohérent vers un autre état cohérent. Un système est dans un état cohérent si et seulement si il est conforme à nos critères de cohérence.
- *Isolation.* Les reconfigurations sont exécutées comme si elles étaient indépendantes. Les modifications d'une configuration ne sont visibles dans le système que lorsqu'elles sont validées.
- *Durabilité.* Le résultat d'une reconfiguration est permanent : quand une reconfiguration a réussi, l'état modifié du système (son architecture et l'état des composants) est sauvegardé afin de pouvoir être récupéré en cas de panne majeure (ex : une panne franche).

Le contrat de cohérence transactionnel [BCF+97] stipule normalement que pour garantir la cohérence des informations, le système transactionnel doit assurer les propriétés A, I, et D et l'utilisateur du système transactionnel la propriété C. Dans notre solution, l'utilisateur participe également à la définition des propriétés A, I et D. En effet, en définissant les opérations de reconfigurations, il peut leur associer des opérations de compensation (propriété A), il peut spécifier les opérations en conflits (propriété I) et leurs effets sur une configuration indiquant l'état du système qui doit être rendu permanent (propriété D). La cohérence (propriété C) dans notre définition est liée au respect des contraintes d'intégrité dans l'architecture du système. Ces contraintes peuvent être inhérentes au modèle de composant ou spécifiées par le programmeur.

6.1.2 Modèle des transactions pour les reconfigurations

Une transaction manipule de manière générale des « objets » (au sens large de ressource) sous forme de séquence d'opérations. Dans le cas simple d'une base de données, les opérations considérées sont des opérations primitives de lecture ou d'écriture sur les données. Le modèle de transactions le plus simple et le plus courant est le modèle de transactions plates [TGGL82]. Il est notamment utilisé dans les SGBD relationnels où il y a statistiquement beaucoup de petites transactions. De nombreux autres modèles de transactions ont été conçus pour répondre à des besoins applicatifs particuliers, c'est le cas par exemple des transactions emboîtées [Mos81] et des Sagas [GMS87]. Ces modèles en permettant de relâcher les propriétés d'atomicité et d'isolation des transactions sont utiles à l'exécution de transactions à longue durée de vie pour lesquelles un abandon complet d'une transaction est coûteux. Nous avons adopté le modèle de transactions plates pour les reconfigurations. En effet, les reconfigurations considérées sont généralement de courte durée de vie : ce sont des séquences de quelques opérations primitives qui impactent peu de composants du système. De plus, pour des raisons de fiabilité, le niveau d'isolation le plus élevé est nécessaire entre reconfigurations. Enfin, le modèle plat est simple et répandu.

Formalisation des reconfigurations transactionnelles. Après avoir spécifié les opérations de reconfiguration primitives et leur composition en dehors d'un contexte transactionnel dans le chapitre 5, nous nous proposons de définir le concept de reconfiguration transactionnelle. Formellement, une transaction T_i est représentée par un doublet $(E_i, <_i)$ tel que :

1. E_i est l'ensemble des événements associés à T_i, i.e. soit l'exécution d'une opération sur un objet, la validation ou l'abandon de la transaction (le début d'une transaction est implicite et identifié à l'occurrence de la première opération dans la transaction),
2. $<_i$ est une relation d'ordre partiel sur E_i et d'ordre total sur les opérations de E_i exécutées sur le même objet.

Les opérations de reconfigurations sont sémantiquement plus riches que des opérations de lecture et d'écriture mais elles peuvent cependant être aussi décomposées par analogie en opérations d'introspection et d'intercession même si la distinction n'existe pas dans le modèle Fractal et au niveau des contrôleurs de reconfiguration. Les objets manipulés par les reconfigurations transactionnelles sont les éléments définis par le modèle Fractal : composants, interfaces et attributs. Une hypothèse forte que les opérations de reconfiguration doivent respecter est l'indivisibilité : deux opérations invoquées simultanément sont exécutées l'une complètement avant l'autre.

Nous notons $op_i[x_k]$ l'opération de reconfiguration op dans la transaction T_i sur l'objet x_k du modèle Fractal. Une reconfiguration transactionnelle est alors définie comme une séquence d'opérations d'introspection et d'intercession sur les éléments du modèle Fractal. Si V_i et A_i représentent respectivement la validation et l'abandon d'une transaction T_i, alors $T_i = (\{\bigcup op_i[x_k] \cup V_i \cup A_i\}, <_i)$.

Une histoire (exécution concurrente) pour un ensemble de transactions $\{T_1, T_2, ..., T_n\}$ est définie par le doublet $(E, <)$ avec :

1. $E = \bigcup E_i$
2. $<$ relation d'ordre partielle sur E et totale sur l'ensemble des opérations $op_i[x]$ exécutées sur le même objet x
3. $\forall T_i,\ <_i \subseteq <$, i.e. l'ordre sur les événements dans E_i est conservé par la relation d'ordre $<$

Un exemple d'histoire comportant des opérations d'intercession et d'introspection dans deux transactions T_1 et T_2 est le suivant : $add_1(c_1, c_2)$; $interfaces_2(c_1)$; $bind_1(i_1, i_2)$

Mise à jour des transactions. Etant donné un modèle de transactions, il existe différents modes de mise à jour des transactions dans un système qui correspondent à la répercussion des effets des reconfigurations sur le système en train de s'exécuter. Deux modes ont été considérés : le mode de *mise à jour immédiate* et le mode de *mise à jour différée* [BCF+97]. Le mode de mise à jour influe notamment sur les techniques de reprise sur défaillance.

Dans la méthode de *mise à jour immédiate* (cf. Figure 6.1), les opérations de reconfiguration invoquées par le client de la reconfiguration sont exécutées directement sur le système à l'exécution pour en modifier les structures de données. Le système passe d'une configuration initiale A_0 à une configuration finale A_n après l'exécution d'une séquence d'opérations d'intercession. L'avantage de cette méthode est la rapidité d'exécution de la reconfiguration dans le système qui est faite sans opération préalable, d'où un gain en réactivité quand la reconfiguration est valide. Par contre, en cas d'annulation de transaction, les effets des opérations déjà exécutées doivent être annulés par des techniques de compensation qui peuvent diminuer le temps de disponibilité du système.

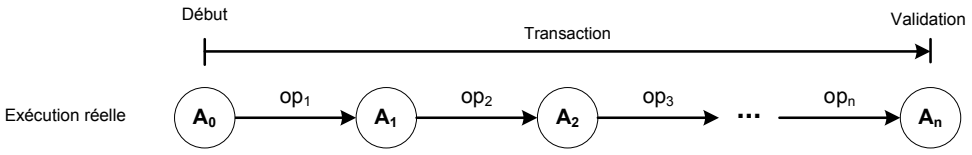

FIGURE 6.1 – Mise à jour immédiate

Dans le mode de *mise à jour différée* (cf Figure 6.2), une copie de l'état du système est réalisée pour chaque transaction, cet état appelé espace de travail est inaccessible aux autres transactions. La reconfiguration est exécutée dans un premier temps sur cet espace de travail le faisant passer d'une configuration $A_0^{'}$, copie de l'état A_0 du système, à une configuration finale $A_n^{'}$. L'état $A_n^{'}$ de la copie est reproduit après validation dans le système pour le mettre dans l'état obtenu A_n.

L'état est représenté sous forme d'un graphe de configuration présenté dans le chapitre 4. Plutôt que la copie complète du graphe de configuration, l'état est copié de manière paresseuse au fur et à mesure de l'exécution des opérations. Le graphe de configuration est donc construit progressivement suivant les besoins d'introspection du système et n'est ainsi pas nécessairement entièrement copié

en mémoire pour chaque transaction. Nous nommerons *simulation* l'exécution de la reconfiguration dans l'espace de travail et exécution réelle l'exécution des opérations sur le système réel. La recopie de l'espace de travail vers le système consiste en l'exécution en différée des opérations dans le système après la validation de la transaction.

La simulation permet ainsi de valider la cohérence du système à l'issue de la reconfiguration sans modifier réellement le système. D'autre part, la reconfiguration est « évaluée » lors de la simulation : l'ensemble des paramètres des opérations d'intercession sont évalués par introspection du système. Seules les opérations d'intercession sont donc exécutées lors de la recopie dans le système et pas les opérations d'introspection. Par exemple, l'histoire d'une transaction $add(c_1, c_2)$; $interfaces(c_1)$; $bind(i_1, i_2)$ une fois simulée élimine les opérations d'introspection pour ne garder que l'histoire $add(c_1, c_2)$; $bind(i_1, i_2)$ qui ne comporte que des opérations d'intercession.

La mise à jour différée est plus coûteuse en temps d'exécution pour une reconfiguration valide car elle nécessite une simulation sur l'espace de travail. Par contre, l'annulation d'une transaction ne requiert pas de modification du système mais seulement la destruction de la copie utilisée pour la simulation. De plus, il est nécessaire que toutes les opérations de reconfiguration utilisées soient modélisées comme des opérations de transformation du graphe de configuration (cf. chapitre 5). Dans le cas contraire, une opération ne peut être simulée et doit donc être exécutée directement dans le système en mode immédiat tout en étant associé à une opération inverse ou de compensation.

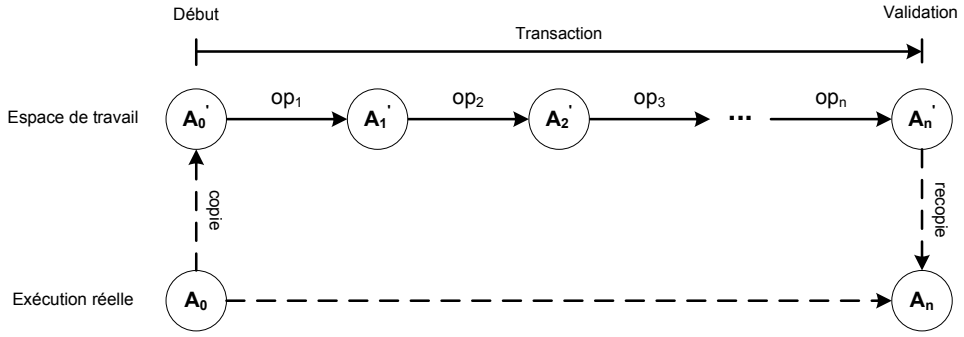

FIGURE 6.2 – Mise à jour différée

Le choix d'un mode de mise à jour ou de l'autre pour les reconfigurations dynamiques se fait ainsi essentiellement en fonction du taux potentiel d'abandon (ou réciproquement de validation) des transactions. Le taux d'abandon de transactions est le rapport entre le nombre de transactions abandonnées sur le nombre total de transactions exécutées (abandonnées et validées). Ce taux peut être estimé en fonction du nombre de reconfigurations dans le système et notamment du nombre de contraintes d'intégrité. Le nombre de reconfiguration augmente le risque de défaillances et plus les contraintes sont nombreuses, plus une reconfiguration risque de violer une contrainte du système. Si le nombre d'abandons est élevé, le coût en temps d'exécution de la mise à jour immédiate est important par rapport à la mise à jour différée en raison de l'annulation qui consiste alors à défaire les opérations. Par contre, si le taux d'abandon est faible, le surcoût est en défaveur de la mise à jour différée : l'étape de « simulation » s'ajoute effectivement à l'exécution des opérations dans le système pour les transactions valides.

Propagation des transactions. Différents types de propagation peuvent être envisagés lors de l'exécution d'une méthode transactionnelle si une transaction est déjà en cours d'exécution par la même entité active (processus ou thread). Etant donné que toute reconfiguration peut potentiellement mettre le système dans un état incohérent, le type de propagation choisi par défaut et implémenté pour les transactions de reconfiguration correspond à la politique *Required* (cf Figure 6.3) définie dans le standard J2EE pour les composants EJB (EJB Container Management Transaction [Jav]). Cette politique de propagation requiert qu'une opération soit toujours exécutée dans un contexte transactionnel. Lors de l'exécution d'une opération, si un contexte transactionnel existe, il sera propagé à l'opération, sinon un nouveau contexte sera créé. Une opération de reconfiguration (introspection ou intercession) isolée qui n'est pas incluse dans une transaction démarquée explicitement sera automatiquement associée à un nouveau contexte transactionnel pour respecter l'isolation. Cette politique peut cependant être modifiée pour optimiser les performances des opérations en lecture

seule notamment.

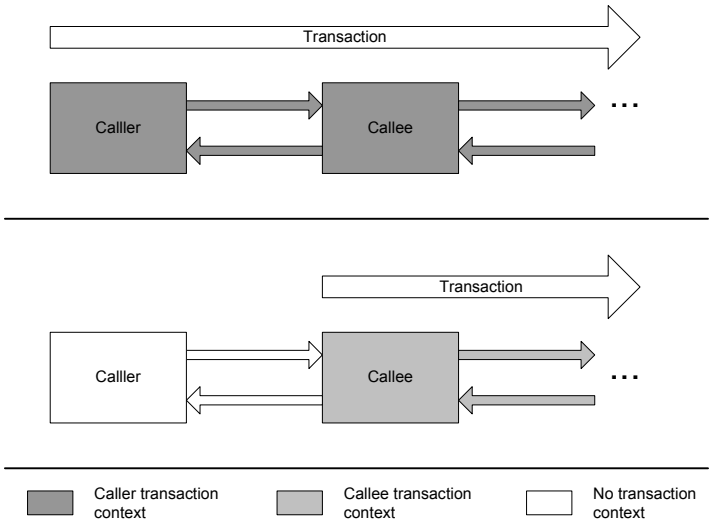

FIGURE 6.3 – Mode de propagation « Required » des transactions de reconfiguration

6.2 Atomicité des reconfigurations pour assurer la tolérance aux fautes

Une reconfiguration lorsqu'elle est une transformation invalide de la configuration d'un système doit être annulée et le système doit être remis dans l'état cohérent qu'il possédait avant la reconfiguration invalidée suivant la sémantique du « tout ou rien ». Pour garantir l'atomicité des reconfigurations transactionnelles dans des applications à composants réparties, nous utilisons un protocole de validation atomique adapté. Nous définissons également un modèle pour défaire les opérations qui sont exécutées dans le système au cours d'une transaction invalidée.

6.2.1 Protocole de validation atomique

L'issue d'une transaction consiste à décider soit de sa validation (« commit »), soit de son annulation (« abort »). Les effets de la transaction après sa validation sont définitifs alors que son annulation remet le système dans l'état avant le début de la transaction. Un protocole de validation atomique est un processus de décision de l'issue d'une transaction.

Rappel sur la validation atomique répartie. Dans le cas d'un système distribué, la garantie de la propriété d'atomicité globale dans le système repose généralement sur le mise en oeuvre d'un protocole de validation répartie, le protocole de validation atomique en deux phases (Two-Phase-Commit ou 2PC [TGGL82] est le plus utilisé. Le protocole (cf. section 2.3.2) comporte un première phase (phase de vote) au cours de laquelle le coordinateur envoie un message de demande de vote à tous les participants. Chaque participant répond au coordinateur en votant pour la validation ou pour l'abandon. Dans la seconde phase (phase de décision), le coordinateur reçoit les votes, puis informe les sites du résultat de sa décision.

Validation des reconfigurations transactionnelles. La distribution est une propriété gérée de façon transparente en Fractal : les composants d'une application peuvent être arbitrairement répartis sur plusieurs sites. L'état modifié par les reconfigurations transactionnelles est l'état représenté par une configuration, c'est à dire essentiellement l'architecture des composants. Cet état en mémoire principale est volatile et est analogue au cache pour les base de données, il est complété par l'état stocké en mémoire secondaire, stable (par exemple sur disque), que nous appelons état durable. L'état global considéré à l'instant courant pour un système est ainsi constitué par les structures de données en mémoire implémentant la configuration et l'état durable. Les reconfigurations transactionnelles

visent à garantir l'atomicité des changements de cet état grâce au mécanisme de validation. Il faut noter que les deux états ne sont pas nécessairement identiques et nécessitent une synchronisation, l'état en mémoire étant toujours le plus récent. Par ailleurs, si l'état en mémoire est distribué sur plusieurs sites, l'état durable peut être centralisé sur un disque unique sur un même site.

L'exécution d'une transaction est isolée en différentes branches. Nous considérons en élargissant la définition qu'un participant est une partie de l'application en charge de valider l'exécution d'une branche. L'implication de plusieurs participants au sein d'un transaction nécessite l'emploi d'un protocole de validation atomique répartie tel que le 2PC. Plusieurs choix s'offrent à nous pour le choix de la granularité et du nombre de participants dans une reconfiguration transactionnelle suivant que l'on considère le composant, le site ou l'application complète comme participant potentiel dans les reconfigurations transactionnelles (cf. Figure 6.4). :

- Chaque composant est responsable de la branche de la transaction consistant uniquement en l'exécution d'opérations primitives sur ses contrôleurs. Cette approche offre l'avantage de préserver la transparence de la distribution pour le recouvrement de panne. Par contre, chaque composant est un système transactionnel centralisé indépendant qui gère son propre état et le nombre de participants est donc multiplié.
- Chaque site constitue un participant suivant la définition traditionnelle. Cette solution nécessite de maintenir un état durable sur chaque site correspondant à l'état de l'ensemble des composants instancié sur le site et donc d'identifier les composants selon leur répartition physique.
- L'application est considérée globalement comme le seul et unique participant dans toutes les reconfigurations transactionnelles. Si les reconfigurations et l'état mémoire du système reconfiguré sont répartis, l'état durable est géré de manière centralisée sur un seul site. Cette approche permet de s'affranchir d'un protocole de validation réparti pour se contenter d'un protocole en une phase sans phase de vote (1PC [AGP98]). Elle passe cependant difficilement à l'échelle du fait de la gestion centralisé de l'état durable.

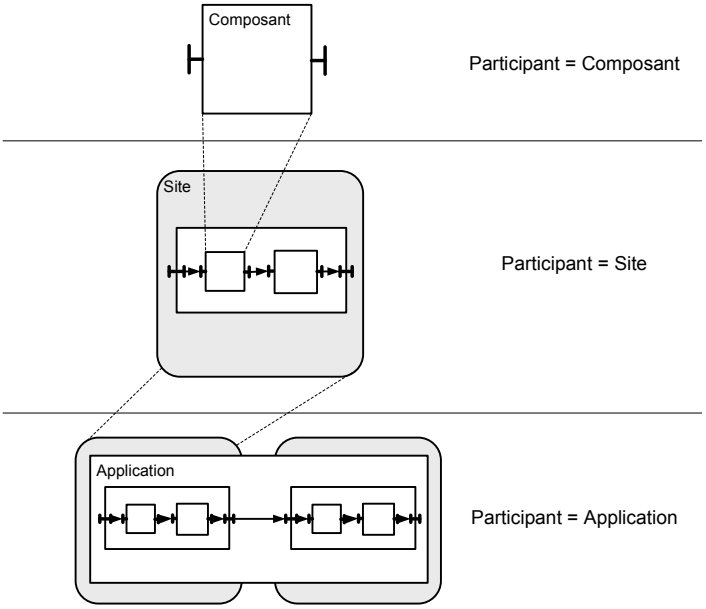

FIGURE 6.4 – Granularité des participant pour la validation des transactions

Nous optons pour le troisième choix car il facilite beaucoup le processus de validation tout en autorisant les reconfigurations réparties. Nous considérons un seul participant qui est donc de facto le coordinateur de toute transaction, i.e. il coordonne la validation ou l'abandon des transactions. L'architecture cible du système à reconfigurer est alors gérée comme une seule et unique ressource transactionnelle avec un seul gestionnaire de transactions, centralisé, dédié aux reconfigurations dynamiques. Le site à partir duquel les opérations sont invoquées est le client ou initiateur de la reconfiguration, et n'est pas nécessairement le site où est localisé le gestionnaire de transactions.

Le gestionnaire de transactions supporte également le protocole de validation à deux phases afin de pouvoir participer à des transactions globales impliquant plusieurs ressources transactionnelles, il

peut ainsi être coordonné comme participant d'une transaction distribuée par un autre gestionnaire de transactions jouant le rôle de coordinateur (cf. chapitre 7).

Limitations de la validation. Notre approche ne permet pas de garantir complètement l'atomicité globale en cas de défaillance de site. En effet, certaines structures de données constituant l'état en mémoire peuvent être réparties sur plusieurs sites (c'est typiquement le cas dans Julia) et elles sont donc corrompues lors de pannes de machines. Par contre, l'atomicité de l'état en mémoire secondaire (sur disque) est garantie. Pour retrouver un état cohérent, le système doit donc être réinstancié à partir de l'état persisté qui est le dernier état cohérent connu du système avant la panne. Etant donné que nous utilisons pas de phase de vote pour la validation, l'occurrence d'une défaillance de site pendant une reconfiguration est détectée implicitement par le dépassement du délai de garde des communications synchrones distribuées. L'invocation d'une opération de reconfiguration qui ne retourne pas ou retourne une erreur liée à un problème de communication réseau entraîne automatiquement l'abandon de la transaction.

6.2.2 Modèle de « undo »

L'exécution d'une reconfiguration transactionnelle revient à exécuter une composition d'opérations primitives de reconfiguration correspondant aux opérations définies dans les contrôleurs Fractal. Ces opérations sont soit des opérations d'introspection sans effets de bord ou des opérations d'intercession qui modifient le système. Pour assurer l'atomicité des transactions de reconfiguration, les effets des opérations exécutées dans une transaction sont défaits en cas d'abandon de la transaction. Plus précisément, seules les opérations d'intercession sont prises en compte lors de l'annulation de la transaction car ce sont les seules opérations susceptibles de modifier l'état du système, les opérations d'introspection sont des opérations en lecture seule qui ont juste besoin d'être isolées.

Dans le cas de la mise à jour immédiate des reconfigurations, deux cas se présentent suivant que les opérations considérées sont réversibles ou non. Quand les opérations sont réversibles, défaire une reconfiguration est équivalent à défaire la séquence d'opérations d'intercession dans l'ordre inverse de leur exécution (cf. Figure 6.5). Le modèle est ainsi linéaire puisqu'il ne nécessite pas de commutativité entre opérations. Chaque transaction maintient un journal en mémoire sous la forme d'un séquence d'opérations déjà exécutée. C'est ce journal qui est « défait »pour annuler une transaction.

Certaines opérations ne sont pas inversibles, c'est le cas de l'opération *new* pour laquelle l'opération inverse correspondrait à une opération de destruction de composant qui n'existe pas dans le modèle. Pour le cas particulier des opérations non réversibles, l'annulation de la reconfiguration nécessite un traitement spécifique en cas d'abandon de la transaction. Des opérations de compensation peuvent alors être associées à ces opérations, plus aucune garantie sur l'atomicité ne peut être alors donnée car l'état du système résultant de l'abandon peut ne pas être tout à fait identique à l'état d'avant l'exécution de la reconfiguration. La cohérence du système, au sens de la satisfaction de l'ensemble des contraintes d'intégrité, est par contre maintenue par la transaction.

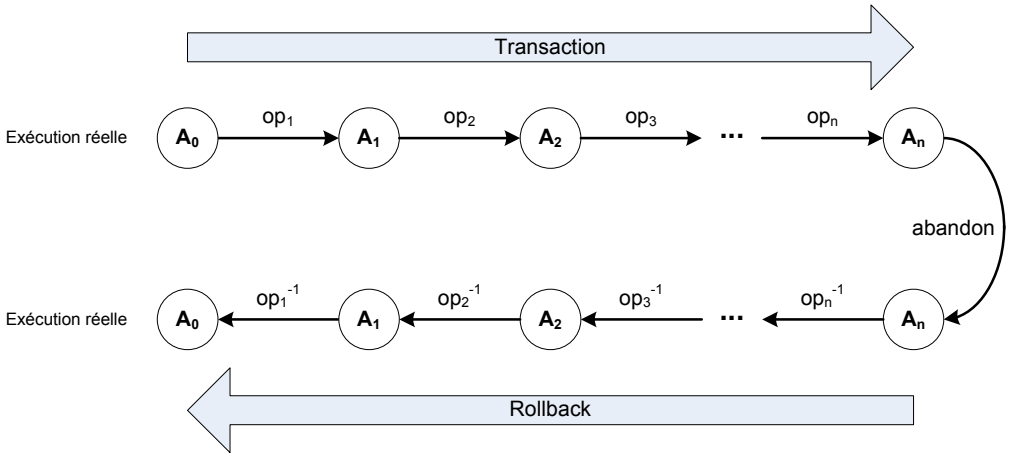

FIGURE 6.5 – Abandon de transaction en mise à jour immédiate

Les opérations primitives provenant de l'API Fractal sont modélisées pour préciser leur sémantique (cf. chapitre 5). Une bibliothèque extensible d'opérations est proposée (cf. Figure 6.6) où chaque opération d'intercession est associée à son opération inverse dans la sémantique que nous avons choisie (typiquement *bindFc* et *unbindFc* pour des interfaces de composant sont des opérations inverses). De plus, une opération peut conserver un état interne pour pouvoir être défaite. Par exemple, changer la valeur d'un attribut requiert de garder l'ancienne valeur de l'attribut pour défaire l'opération.

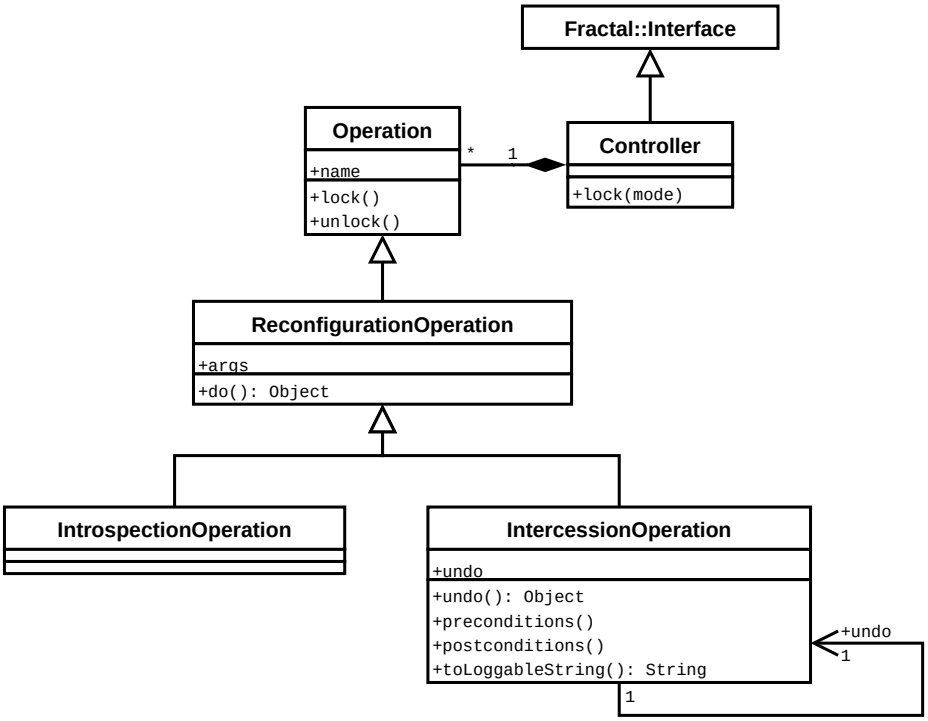

FIGURE 6.6 – Modèle d'opérations de reconfiguration

Le cas de la mise à jour différée (cf. Figure 6.7) est beaucoup plus simple car les effets des reconfigurations ne sont pas directement appliqués dans le système, l'annulation d'une transaction revient alors simplement à supprimer la copie de travail qui a servi pour la simulation de la reconfiguration.

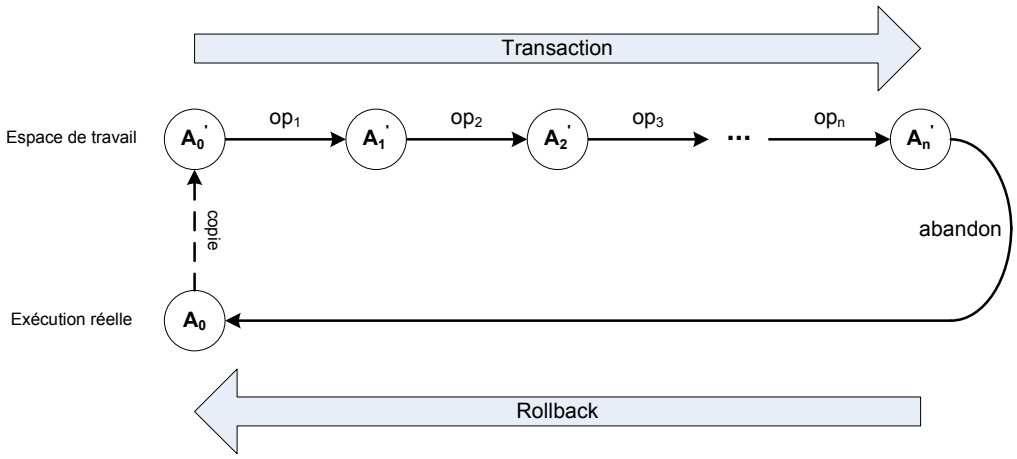

FIGURE 6.7 – Abandon de transaction en mise à jour différée

6.3 Cohérence du système définie par des contraintes d'intégrité

La propriété de cohérence dans le contrat de cohérence transactionnel est du ressort de l'application. Nous définissons la cohérence d'un système comme la satisfaction d'un ensemble de contraintes d'intégrité. Un modèle de contraintes permet de spécifier des contraintes à plusieurs niveaux d'abstraction. Les transactions permettent de maintenir cette cohérence au cours des reconfigurations dynamiques, toutes les contraintes devant être vérifiées au moment de la validation des transactions.

6.3.1 Modèle de contraintes

Dans notre approche, la cohérence d'un système repose sur le respect de contraintes d'intégrité analogues aux contraintes dans les SGBD. Une contrainte d'intégrité en bases de données est une assertion vérifiable à tout moment spécifiant les valeurs permises pour certaines données, éventuellement en fonction dŠautres données, et permettant dŠassurer la cohérence de la base de données. On peut citer par exemple l'unicité de clé ou encore la contrainte référentielle qui spécifie que toute valeur d'un groupe d'attributs d'une relation doit figurer comme valeur de clé dans une autre relation. Une contrainte d'intégrité dans notre contexte, telle que décrite dans le chapitre 4, est un prédicat portant sur l'assemblage des éléments architecturaux d'un système et sur l'état de ses composants . Nous voulons exprimer ces contraintes à la fois au niveau du modèle de composant et directement sur des applications. Par conséquent, une transaction de reconfiguration ne peut être validée que si le système reconfiguré est bien cohérent, c'est à dire si toutes les contraintes d'intégrité sont satisfaites.

Modèle. Le modèle de contraintes (cf. Figure 6.8) propose différents types de contraintes : des *invariants* sur les configurations, et des *préconditions* et des *postconditions* sur les opérations de reconfiguration (uniquement les opérations d'intercession). Les invariants peuvent concerner l'ensemble de la configuration du système reconfiguré. Les conditions peuvent notamment mais pas exclusivement porter sur les paramètres des opérations.

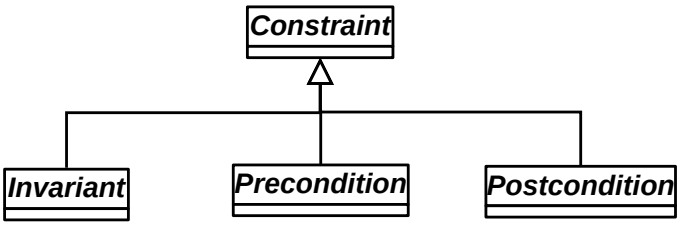

FIGURE 6.8 – Trois types de contraintes

Le modèle distingue également trois niveaux de spécification de contraintes (cf. Figure 6.9) qui correspondent à trois niveaux d'abstraction différents :

- **Le niveau modèle** : il s'agit d'un ensemble de contraintes génériques associées au modèle. Ces contraintes s'appliquent à tous les systèmes utilisant le modèle Fractal. Un exemple de ce type de contrainte est l'intégrité de la hiérarchie (les liaisons entre composants doivent respecter l'encapsulation des composants sans franchissement de membranes), ou encore l'absence de cycle dans la structure de l'application (un composant ne peut se contenir lui-même pour éviter les récursivités infinies). Nous avons ainsi spécifié la sémantique des opérations de reconfiguration du modèle avec des préconditions et des postconditions qui ne doivent pas être violées.
- **Le niveau profil** : consiste en des contraintes génériques pour raffiner le modèle de composant utilisable pour une famille d'applications. Le niveau profil doit être conforme au modèle de composant. Un profil peut par exemple interdire le partage de composants en spécifiant que tout composant ne peut avoir plus d'un composant parent à la fois.
- **Le niveau applicatif** : les contraintes applicatives sont spécifiques à une architecture donnée et sont appliquées directement aux éléments Fractal désignés par leurs noms. Le niveau applicatif est conforme au niveau profil et au niveau modèle. Des invariants peuvent concerner par exemple la cardinalité des sous-composants dans un composant composite ou encore interdire la

déconnexion d'interfaces de composants, etc. Ces contraintes sont spécifiées dans les définitions ADL des composants.

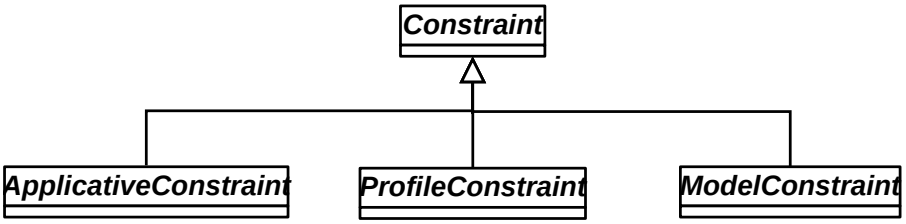

FIGURE 6.9 – Trois niveaux de contraintes

Une contrainte d'intégrité est ainsi caractérisée par son type et son niveau de spécification. Un invariant peut par exemple être spécifié au niveau du modèle, d'un profil, ou d'une application. Les pré et postconditions sur les opérations de reconfigurations sont par contre limitées aux deux premiers niveaux et ne peuvent être spécifiques à une application donnée.

Dynamicité. Toutes les contraintes du modèle et des profils sont définies statiquement avant l'exécution du système. Elles sont immutables et ne peuvent donc être modifiées, ajoutées, ou retirées dynamiquement. Les contraintes applicatives peuvent être par contre ajoutées ou enlevées dynamiquement sur un composant donné. Une contrainte applicative est toujours associée à une instance de composant qui contient donc une liste des contraintes applicatives qui le concernent directement. L'opération d'ajout et de retrait de contraintes n'est cependant pas considérée comme une opération de reconfiguration dynamique dans notre modèle car les contraintes n'apparaissent pas comme des éléments du graphe de configuration. L'ajout ou le retrait d'une contrainte applicative n'est pris en compte que pour les nouvelles transactions de reconfiguration et pas pour les transactions en cours d'exécution.

Spécification. Le langage FPath [Dav05] conçu pour naviguer dans les architectures Fractal est utilisé comme langage de contraintes pour exprimer les contraintes d'intégrité sur les configurations et les reconfigurations de systèmes pendant leur exécution (cf. chapitres 4 et 5). Un système ne doit pas être *surcontraint*, c'est à dire que les contraintes ne doivent pas être contradictoires entre elles. La non contradiction n'est pas directement vérifiée à l'exécution mais nécessite une étape de traduction (cf. chapitre 4) dans le langage de spécification Alloy [Jac02] .

6.3.2 Vérification des contraintes

Les contraintes une fois spécifiées définissent la cohérence d'une application dans notre modèle. Une transaction de reconfiguration doit donc respecter cette cohérence, i.e. assurer le respect de l'ensemble des contraintes d'intégrité lors de la validation des transactions. Il faut pouvoir valider à la fois les invariants de configuration et les conditions sur les opérations de reconfiguration.

Modèle d'exécution pour la vérification des contraintes. Nous distinguons plusieurs moments de l'exécution du système au cours desquels les contraintes peuvent être vérifiées :
- lors de la validation d'une transaction de reconfiguration,
- lors de l'exécution d'une opération (primitive ou non) de reconfiguration (avant et/ou après l'exécution),
- lors de l'instanciation d'une configuration de composants (juste après l'opération d'instanciation),
- lors du changement d'état du cycle de vie des composants correspondant à un point de synchronisation avec l'exécution fonctionnelle (ex : avant le passage de l'état *stopped* à l'état *started*),
- lors d'une demande explicite de vérification.

Pour gérer la cohérence avec les transactions de reconfiguration, chaque type de contraintes est associé à un modèle d'exécution obligatoire ou par défaut (cf. Figure 6.10). Ainsi les pré et postconditions sont vérifiées obligatoirement à chaque exécution d'opération de reconfiguration et

peuvent donc invalider une transaction avant la fin de son exécution. La vérification des invariants est plus souple, par défaut ils ne sont vérifiés que lors de la validation des transactions, ce qui implique qu'une reconfiguration peut temporairement violer des contraintes d'intégrité tant qu'elle n'est pas validée. Il est cependant possible d'imposer la vérification de certains invariants au plus tôt, i.e. après l'exécution de chaque opération de reconfiguration. La vérification au plus tôt peut être intéressante dans le cas d'une mise à jour immédiate des reconfigurations où les reconfigurations sont directement appliquées dans le système et où l'annulation doit être déclenchée avant la validation de la transaction. Quelque soit le moment de la vérification, quand une violation de contrainte est détectée, la transaction de reconfiguration ayant causé la violation est annulée.

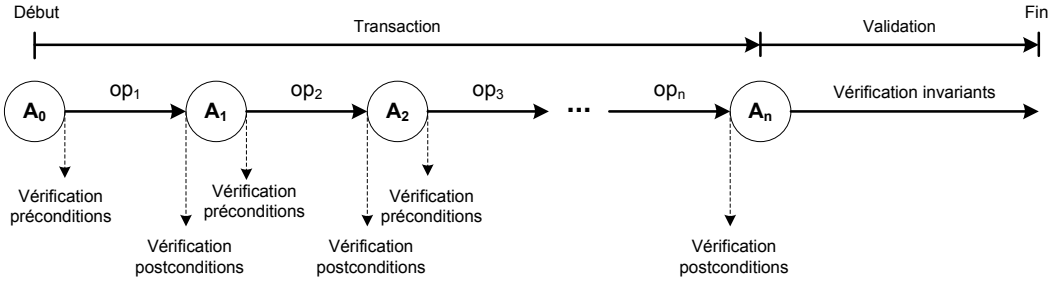

FIGURE 6.10 – Vérification des contraintes dans une transaction de reconfiguration

Activation/Désactivation de la vérification. Il est possible de relâcher la vérification de certaines contraintes dans le système pendant l'exécution. La désactivation se fait par type et par niveau de contraintes, il est donc possible par exemple de désactiver l'ensemble des pré et postconditions sur les opérations de reconfiguration au niveau modèle. Cette désactivation est utile dans une implémentation comme Julia [BCL+06] où la vérification des pré et postconditions au niveau modèle est déjà incluse dans le code des opérations des reconfigurations (mixins constituant les contrôleurs standard), la vérification de ces contraintes serait donc redondante. Pour les contraintes applicatives, chaque contrainte attachée à un composant est explicitement nommée afin de pouvoir être désactivée unitairement. L'activation et la désactivation des contraintes sont synchronisées avec l'exécution des transactions de reconfiguration, c'est à dire qu'elle ne seront prises en compte que pour les nouvelles transactions et pas pour les transaction en cours.

6.4 Isolation des reconfigurations pour gérer les reconfigurations concurrentes

L'isolation est une propriété des transactions gérée au sein des systèmes transactionnels par différentes méthodes de contrôle de concurrence. Le contrôle de concurrence a pour objectif d'assurer une exécution sérialisable de transactions dont l'exécution est concurrente. D'autre part, les reconfigurations sont exécutées dans un système pendant son exécution. Il est donc nécessaire de synchroniser reconfigurations et exécution fonctionnelle du système.

6.4.1 Stratégie de gestion de la concurrence

Plusieurs administrateurs (hommes ou machines) peuvent reconfigurer le même système en même temps. Par ailleurs une même reconfiguration peut être composée de plusieurs opérations en parallèle pour optimiser le processus de reconfiguration. Un première politique minimaliste de contrôle de concurrence proposé consiste à forcer l'exécution en série des transactions : une seule reconfiguration est exécutée à la fois tandis que les autres sont simplement placées dans une file d'attente (ordonnancement FIFO). Cet ordonnancement au niveau des transactions est adapté pour des niveaux de concurrence faible et permet d'éviter de définir les conflits entre opérations. La seconde politique s'intéresse à l'ordonnancement de l'exécution des opérations des transactions : les reconfigurations sont exécutées de manière concurrente et les accès aux éléments architecturaux Fractal dans le système doivent être synchronisés pour éviter les conflits.

Il existe deux principales méthodes de contrôle de concurrence (cf. section 2.3.2 : les méthodes pessimistes (ou de contrôle continu) et les méthodes optimistes (ou de contrôle par certification). Parmi ces deux méthodes, les méthodes pessimistes sont plutôt adaptées à la mise à jour immédiate où le coût d'abandon de transaction est plus élevé en cas de conflits [BHG87]). Les méthodes optimistes sont adaptées en cas de conflits peu nombreux entre transactions et à la mise à jour différée car les effets des transactions non validées ne sont pas directement appliqués dans le système. Nous avons choisi pour des raisons pratiques de ne considérer qu'une seule méthode de gestion de concurrence pour les reconfigurations transactionnelles mais les deux types de méthodes sont envisageables. Notre choix a porté sur une méthode pessimiste plutôt qu'optimiste principalement pour les raisons suivantes :

- Nous considérons deux modes de mises à jour pour les transactions. Or les méthodes optimistes sont difficilement conciliables avec le mode de mise à jour immédiate où la détection de conflit implique automatiquement l'abandon des transactions impliquées et donc perturbe le fonctionnement du système réel. Les méthodes pessimistes sont plus polyvalentes.
- Les techniques de verrouillage sont plus facile à implémenter que les techniques de certification et plus adaptées pour les systèmes avec de nombreuses transactions courtes et concurrentes. La plupart des systèmes transactionnels dont la majorité des SGBD implémente d'ailleurs un contrôle pessimiste.

Modèle de verrouillage. Nous adoptons pour la gestion de concurrence une approche pessimiste avec du verrouillage en deux phases strict (2PL) [GR92] afin de supporter un fort niveau de concurrence (correspondant au niveau d'isolation ANSI *serializable*) et qui permet de construire des exécutions strictes pour des transactions concurrentes. Dans le protocole 2PL, les verrous acquis pendant une transaction ne sont relâchés qu'au moment de la validation pour éviter les *pertes de mises à jours*. Plusieurs éléments du modèle correspondant aux noeuds FPath sont verrouillables avec un algorithme de verrouillage hiérarchique : le verrouillage d'un composant par exemple entrainera également le verrouillage de toutes ses interfaces.

Les verrous sont réentrants (une transaction peut acquérir plusieurs fois le même verrou) et il y a deux types de verrous : les verrous en lecture (ou partagés) et les verrous en écriture (ou exclusifs). Un contrôleur de verrouillage (*LockController*) est fourni pour chaque composant afin de verrouiller ses sous-éléments dans le graphe de verrouillage (cf Figure 6.11). Un verrou peut également être acquis par l'intermédiaire d'une expression FPath.

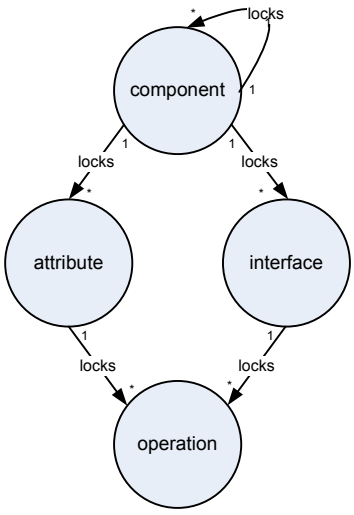

FIGURE 6.11 – Modèle de verrouillage hiérarchique dans Fractal

Nous utilisons ce modèle de verrouillage hiérarchique à fine granularité reposant sur la sémantique des opérations de reconfiguration pour éviter les conflits entre opérations. Une opération d'introspection incluse dans une transaction prendra des verrous en lecture (cf. table 6.1) pour prévenir les *lectures sales* et les *lectures non répétables* : une transaction ne devrait ainsi pas voir les modifications du système effectuées par d'autres reconfigurations non validées. Cependant, deux opérations dans

Verrou demandé \ Verrou détenu	Introspection	Intercession
Introspection	oui	non
Intercession	non	non

TABLE 6.1 – Compatibilité des verrous

deux transactions séparées doivent pouvoir introspecter le même élément en partageant un verrou en lecture. Par contre, une opération d'intercession va prendre un verrou en écriture sur les éléments qu'elle modifie afin d'éviter des modifications concurrentes. Par exemple, une opération d'addition d'un *composant B* dans un *composant A* prendra un verrou en écriture au niveau du *ContentController* du composant parent *A* sur les opérations suivantes : *addFcSubComponent* (intercession), *removeFcSubComponent* (intercession) et *getFcSubComponents* (introspection). L'opération d'introspection *getFcSuperComponents* dans le *SuperController* du composant fils *B* sera également verrouillée en écriture.

Limitations du modèle de verrouillage. Il faut noter que ce modèle de verrouillage n'est pas optimal. Nous n'avons en effet accès à l'état du composant qu'à travers les opérations de reconfigurations définies dans ses contrôleurs. Or le verrouillage ne se base que sur le nom des opérations et pas sur la valeur des paramètres de ces opérations. De ce fait, l'état verrouillé est plus large que l'état réellement accédé ou modifié. Par exemple, le verrouillage exclusif de l'opération *bindFc* pour connecter deux interfaces bloque la possibilité d'invoquer cette opération sur l'ensemble des interfaces clientes d'un composant. Cependant l'isolation optimale consisterait à pouvoir verrouiller la relation *bind* entre deux interfaces, c'est à dire l'opération *bindFc* uniquement pour une interface donnée en paramètre.

Gestion des interblocages. Le verrouillage des opérations pose le problème de la possibilité d'interblocage entre transactions. Lorsqu'une transaction T_1 est en attente d'un verrou sur opération op_1 en conflit avec un verrou déjà possédé par une transaction T_2 et que T_2 est en attente d'un verrou sur une opération op_2 en conflit avec un verrou possédé par T_1, les deux transactions se retrouvent bloquées, i.e. en interblocage, en attente de la libération d'un verrou possédé par l'autre. Une telle situation créé un circuit dans le graphe d'attente des transactions : $T_1 \rightarrow T_2$ et $T_2 \rightarrow T_1$. Nous maintenons un graphe de dépendance globale et centralisé entre toutes les transactions en attente et détectons les interblocages au fur et à mesure de la pose de verrous (détection continue). Lors de la détection d'un interblocage, une transaction doit être abandonnée afin que l'exécution des autres transactions puisse se poursuivre. Plusieurs politiques peuvent être suivies quand au choix de la transaction à abandonner, la politique choisie par défaut dans notre cas est de défaire la transaction qui créé le circuit dans le graphe.

6.4.2 Isolation du niveau fonctionnel

Le remplacement de composant (*hotswap* en Anglais) dans un système à l'exécution est une reconfiguration relativement classique mais complexe qui pose le problème de la gestion de l'état du composant remplacé. Pour deux composants C et C' et en supposant que le type de C' est un sous-type de C, remplacer C par C' nécessite dans un premier de figer l'état du composant C en arrêtant son exécution pour éviter qu'il ne continue d'évoluer au cours de la reconfiguration. Le composant C atteint ainsi un état figé, aussi appelé état « quiescent »[KM90]. Le problème d'un éventuel transfert d'état entre composants n'est pas abordé dans le cadre de l'isolation des transactions, une solution générique par l'ajout d'un nouveau contrôleur est proposée (cf. chapitre 7). En Fractal et plus particulièrement dans la sémantique adoptée par l'implémentation Julia, une telle opération de remplacement générique de composant est décomposée en plusieurs opérations primitives :

- Instanciation du nouveau composant C'
- Arrêt de tous les composants parents de C (l'opération est récursive et donc arrête C)
- Déconnexion des interfaces (serveurs et clientes) de C

- Retrait du composant C de tous ses parents
- Ajout du composant C' dans tous les composants anciens parents de C
- Connexion des interfaces de C'
- Démarrage des parents de C' (l'opération est récursive et donc démarre C)

La règle pour déconnecter une interface cliente d'une interface serveur est d'arrêter le composant auquel appartient l'interface. Pour retirer un composant d'un composant parent, l'ensemble des composants fils du parent doivent être arrêtés y compris le composant à retirer. De manière générale, les reconfigurations dynamiques doivent être synchronisées avec l'exécution fonctionnelle des systèmes reconfigurés, les états du cycle de vie des composants servent de point de synchronisation entre le niveau de contrôle et le niveau fonctionnel.

Synchronisation en Julia. Nous considérons comme modèle d'exécution un modèle à threads et comme modèle de communications entre interfaces de composants des communications synchrones. Ces choix correspondent à la sémantique de base utilisée dans Julia : l'activité dans les composants correspond à l'exécution de threads Java et les communications entre composants sont des invocations synchrones de méthodes Java. Un composant lorsqu'il est démarré dans son fonctionnement normal doit pouvoir (cf. Figure 6.12) :

- accepter des invocations d'opérations sur ses interfaces serveurs
- initier des invocations d'opération à partir de ses interfaces clientes
- servir des invocations d'opération sur ses interfaces serveurs, i.e. il transmet les invocations à ses interfaces clientes après un traitement potentiel
- posséder une activité interne sans invocation externe

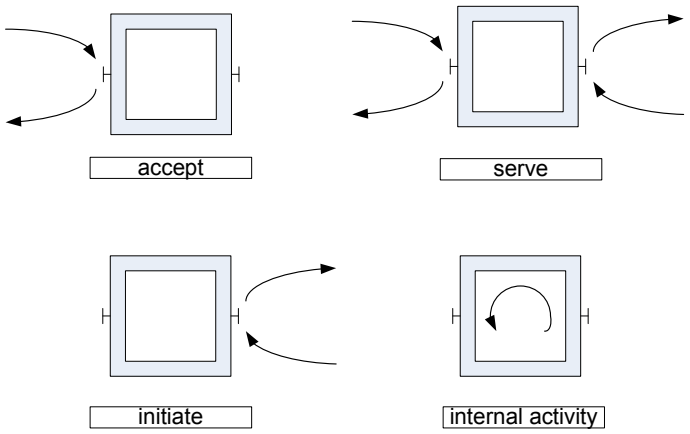

FIGURE 6.12 – Synchronisation de l'activité d'un composant

Pour atteindre l'état quiescent, le composant ne doit plus ni accepter de nouvelles invocations de méthodes en entrée ni initier d'invocations vers l'extérieur. Pour éviter les interblocages, il doit cependant pouvoir terminer les invocations de méthode en cours sur ses interfaces serveurs. Le composant dans l'état quiescent ne doit également plus avoir aucune activité interne susceptible de modifier son état.

En Julia, le protocole de synchronisation pour arrêter l'activité d'un composant et atteindre un état quiescent utilise une méthode de comptage de threads [Wer99]. Un compteur de thread sur chaque interface serveur du composant intercepte et compte les invocations de méthodes. Le compteur est incrémenté à l'aller et décrémenté au retour de l'appel de méthode. La synchronisation de l'arrêt du composant avec son exécution fonctionnel passe par successivement par deux états suite à l'invocation de l'opération *stopFc* :

1. Le composant est mis dans l'état *stopping* : les nouveaux appels de méthode en entrée sur les interfaces serveur du composant sont interceptés et bloqués. Le composant n'accepte plus d'invocations mais continue de servir les invocations en cours.
2. Le composant passe dans l'état *stopped* quand les compteurs de thread sur toutes ses interfaces serveurs sont à 0 : l'ensemble des invocations en cours sont terminés et le composant n'accepte

pas de nouvelles invocations.

Le redémarrage du composant arrêté provoque le déblocage des éventuelles invocations en attente sur les interfaces serveurs du composant.

Si ce protocole permet effectivement de bloquer les invocations entrantes tout en continuant de servir les invocations en cours, il ne permet pas de garantir que le composant n'a plus d'activité interne. Un composant peut ainsi continuer d'initier des appels sortants alors qu'il est arrêté. Comme il n'existe pas de contrainte sur l'activité dans les composants dans le modèle Fractal, il est de la responsabilité du programmeur du code « métier » du composant en Julia d'arrêter l'activité interne du composant (arrêt des threads d'exécution). Cet arrêt de l'activité peut être mené à bien lors de la notification du passage à l'état *stopped* à travers des méthodes de « callback ». Pour empêcher les invocations sortantes, nous ajoutons un intercepteur sur chaque interface cliente du composant pour bloquer les invocations. Dés lors, un composant dans l'état *stopped* ne pourra ni accepter, ni initier de nouvelles invocations qui sont bloquées en entrée et en sortie du composant.

Synchronisation des cycles de vie des composants. Le protocole de synchronisation dans Fractal et plus particulièrement dans Julia bloque les invocations sur les interfaces serveurs des composants lorsqu'ils ne sont pas démarrés pour les forcer à atteindre un état quiescent. Or, il peut s'avérer qu'un composant nécessite pour son initialisation et pour passer dans l'état *started* l'utilisation d'un service fourni par un autre composant dont le cycle de vie est dans un état indéterminé. La figure 6.13 montre un composant d'impression *Printer* qui, pour être démarré, nécessite d'aller chercher un pilote d'impression spécifique auprès d'un composant gestionnaire de pilotes *Driver*. L'invocation de la méthode *startFc()* sur son contrôleur de cycle de vie entraîne ainsi l'appel de la méthode *getDriver()* sur le composant *Driver* auquel il est connecté.

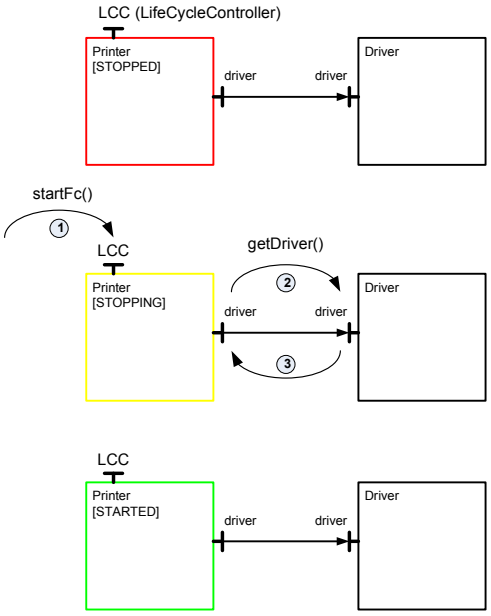

FIGURE 6.13 – Exemple de démarrage d'un composant avec une dépendance de cycle de vie

De manière général (cf. Figure 6.14), si un composant A est connecté à un composant B et qu'il requiert pour passer dans l'état *started* l'invocation d'une méthode $m()$ sur une interface fonctionnelle de B, il faut que ce dernier composant soit dans l'état *started* pour que l'invocation ne soit pas bloquée et que le composant A puisse être démarré. Nous proposons donc d'ajouter aux dépendances fonctionnelles classiques entre les interfaces de composants des *dépendances entre cycles de vie* des composants. Ces dépendances sont comme les dépendances fonctionnelles déclarés statiquement et sont immutables.

Un graphe de dépendances entre composants est maintenu à l'exécution du système et lors du démarrage récursif d'un composant et de ses sous-composants. Ce graphe est orienté, il permet donc en remontant dans le graphe de déterminer un ordre de démarrage des composants dans le système

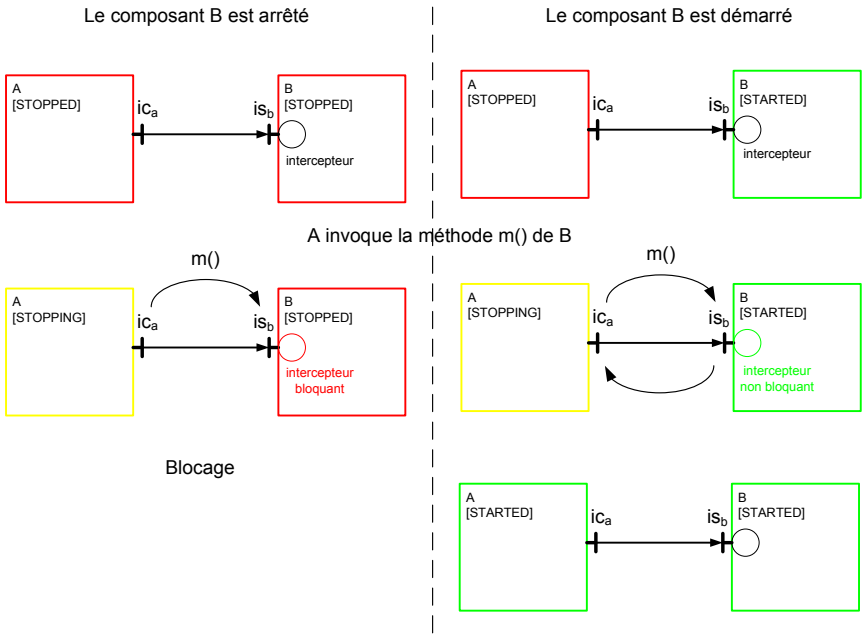

FIGURE 6.14 – Dépendance entre cycles de vie de composants

à respecter pour éviter les interblocages. Le graphe doit être et rester acyclique. Si un cycle entre dépendances fonctionnelles ne pose a priori pas de problème car les invocations fonctionnelles se font sur des composants démarrés, les cycles entre dépendances de cycle de vie peuvent entraîner des interblocages : il y aura par exemple interblocage si un composant A attend le démarrage du composant B pour être démarré ($A \rightarrow B$) et si B attend lui-même le démarrage de A pour être démarré ($B \rightarrow A$). L'instanciation de configurations de composants qui comportent de tels cycles est donc rejetée au moment de leur détection.

6.5 Durabilité des reconfigurations pour permettre la reprise en cas de défaillance

Les transactions sont un outil de tolérance aux fautes, c'est-à-dire un moyen de détecter et réparer de manière transparente des fautes dans un système et de le remettre dans un état cohérent. Le recouvrement est réalisé par un mécanisme de reprise où le système est ramené dans un état survenu avant l'occurrence de la défaillance et les transactions jouent le rôle d'unités de reprise. La propriété de durabilité en rendant les effets des transactions permanents est utile à la réparation des défaillances qui entraîne la perte le mémoire principale du système.

6.5.1 Modèle de défaillances

Une défaillance est susceptible de survenir au cours de l'exécution d'un reconfiguration transactionnelle, cette défaillance est soit due à une cause externe à la reconfiguration (panne de machine, du système d'exploitation, de JVM), soit la cause est interne et la reconfiguration est elle-même à l'origine de la défaillance (une reconfiguration invalide viole la cohérence du système reconfiguré). Les transactions apporte une solution pour le recouvrement en cas de défaillance ((cf. section 2.3.2)). Nous définissons le modèle de défaillances suivant auquel les transactions de reconfiguration doivent apporter une solution de recouvrement : l'abandon de transaction suite à la violation de la cohérence du système ou à un problème de sérialisabilité, et la défaillance de site en cas panne matérielle d'une machine ou des couches logicielles basses comme le système d'exploitation.

A chaque type de défaillance correspond une politique de recouvrement différente :

1. La transaction défaillante est annulée, elle est défaite à partir du journal en mémoire principale. Cette technique est dite de reprise « à chaud » car la réparation se fait entièrement en mémoire volatile.

2. Le système est remis dans son dernier état cohérent à partir des informations en mémoire secondaire. Un protocole du type faire/défaire est mis en oeuvre à partir du journal sur disque et du dernier point de contrôle. Cette technique est dite de reprise « à froid » car elle nécessite normalement le redémarrage de la machine défaillante.

Nous assimilons le problème de panne du réseau à une défaillance de site. Cette assimilation est valable du fait de la validation atomique centralisée. Il faut noter que l'abandon de transaction a une origine logicielle, il s'agit donc d'une défaillance logicielle. Nous qualifierons par opposition de défaillance matérielle la défaillance de site qui si elle n'a forcément qu'une origine matérielle se traduit par une panne de la machine considérée. Pour les défaillances matérielles, nous ne considérons que les pannes franches avec un comportement « tout ou rien » : soit la machine fonctionne correctement soit elle ne fonctionne plus du tout et il ne peut y avoir d'état intermédiaire corrompu ou de faute dite byzantine.

Pour un système centralisé sur un seul site, une défaillance de machine entraîne la perte de tout le système. En revanche, pour un système distribué dans laquelle les composants sont répartis sur plusieurs sites, seuls les composants localisés sur le site défaillant disparaissent du système. Une défaillance de site peut concerner soit le site contenant le gestionnaire de transactions ou n'importe quel autre site contenant des composants de l'application. Dans notre solution où il n'existe qu'un participant dans les transactions qui est aussi le coordinateur, les défaillances de sites sont équivalentes. En effet, comme le couplage entre composants sur les différents sites est fort, nous optons dans une approche simple de reprise constituant en le redémarrage global de l'application.

6.5.2 Reprise sur défaillances

La reprise sur défaillance de site pour les transactions repose sur la propriété de durabilité. La durabilité est réalisée pour les reconfigurations transactionnelles à travers deux mécanismes : la journalisation et les points de contrôle (*checkpointing* en Anglais). La journalisation consiste à maintenir sur disque un fichier séquentiel contenant un ensemble d'information nécessaires pour restaurer le système dans un état courant cohérent. Un point de contrôle est un point de synchronisation entre le contenu du journal et l'état du système stocké sur disque. L'état du disque est en effet en règle générale plus ancien que l'état du journal par rapport à la validation des transactions et le point de contrôle permet de garantir que l'état du disque à un instant donné est postérieur à un endroit précis du journal.

Journalisation des transactions. Les transactions pour servir d'unités de reprise sur défaillance nécessitent de pouvoir être défaite ou refaite à tout moment. Le journalisation est un moyen de rendre permanent les modifications effectuées dans le système. Nous utilisons la journalisation logique plutôt que physique car les reconfigurations décomposables en opérations primitives simples s'y prêtent bien : nous journalisons les opérations qui modifient l'état du système (différentiel d'état) et pas l'état modifié sous forme d'images qui seraient volumineuses. Le journal constitue l'état de référence en cas de défaillance de site pour restaurer le système.

Le protocole de journalisation utilisé est WAL (Write Ahead Logging) [GR92] avec points de contrôle : toutes les modifications sont d'abord écrites dans le journal avant d'être appliquées au système et les informations à la fois pour refaire et pour défaire la transaction sont contenues dans le journal. Nous garantissons ainsi qu'aucun effet produit par une transaction non validée ne peut être présent sur le disque. Le journal est un fichier centralisé contenant une séquence d'enregistrements auxquels sont associés un identifiant unique de transaction (« txid ») et une estampille de temps (« timestamp ») :

- les informations pour refaire et défaire les transactions : écriture des opérations qui sont exécutées, les opérations inverses ou de compensation pour défaire sont déduites automatiquement du modèle d'opérations. Le format est le suivant : $< timestamp\ D|U :\ txid,\ op_i[x] >$, ou D est mis pour « Do » et indique l'exécution d'une opération directe et U est mis pour « Undo » et correspond à une opération d'annulation. Pour le mode de mise à jour immédiate, l'annulation d'une transaction se traduit par l'exécution d'un ensemble d'opérations inverses ou de compensation, ces opérations sont également journalisées et leurs enregistrements sont ajoutés à la suite des opérations de la transaction.

– les informations liées au changement d'état des transactions : validation ou abandon. Le format est le suivant : $< timestamp\ E :\ txid,\ COMMIT|ROLLBACK >$

Un exemple simple d'enregistrement pour une opération ajout d'un composant *Server* dans un composant *ClientServer* est le suivant :

```
15812 D: TX3, add(/ClientServer, /Server)
```

Recouvrement avec points de contrôle. En plus de la journalisation des transactions, l'état du système est sauvegardé sur disque périodiquement au moment d'un point de contrôle. Les points de contrôle sont dits cohérents avec les transactions car aucune transaction ne doit être en cours d'exécution au moment de sa mise en oeuvre. L'état d'un système à composants qui est considéré consiste en la description de son architecture et l'ensemble de l'état fonctionnel de ses composants. Le point de contrôle au moment de la validation de la transaction permet de conserver le dernier état stable connu du système issu de la dernière reconfiguration. L'état structurel du système est sauvegardé sous forme de définitions ADL. L'état fonctionnel considéré pour un composant est limité dans notre implémentation par défaut à l'ensemble des valeurs de ses attributs. La valeur des attributs est persistée au moment de la sauvegarde.

En cas de perte de la mémoire principale, le protocole de recouvrement *Ne pas Défaire / Refaire* [GR92] est appliqué par le gestionnaire de transactions. Il n'est pas nécessaire de défaire les effets en cas de reprise à froid car aucun effet de transactions non validées ne peuvent se trouver sur le disque. Par contre, il est nécessaire de refaire les opérations des transactions validées qui n'ont pas eu le temps d'être reporté sur le disque.

En cas de reprise et de redémarrage du système, l'état des composants est automatiquement mis à jour avec le dernier état persisté au moment du dernier point de contrôle. Suivant la période des points de contrôle, des transactions peuvent avoir été exécutées dans le système et validées après la dernière sauvegarde, toutes les transactions qui ont été validées depuis le dernier point de contrôle sont donc refaites. Il est possible de synchroniser les points de contrôle avec chaque validation de transaction (cf. Figure 6.15). Dans ce cas précis, l'état persisté est toujours l'état cohérent issu de la dernière reconfiguration validée dans le système et il n'est pas nécessaire d'exécuter le protocole de recouvrement, la phase *Refaire* étant inutile.

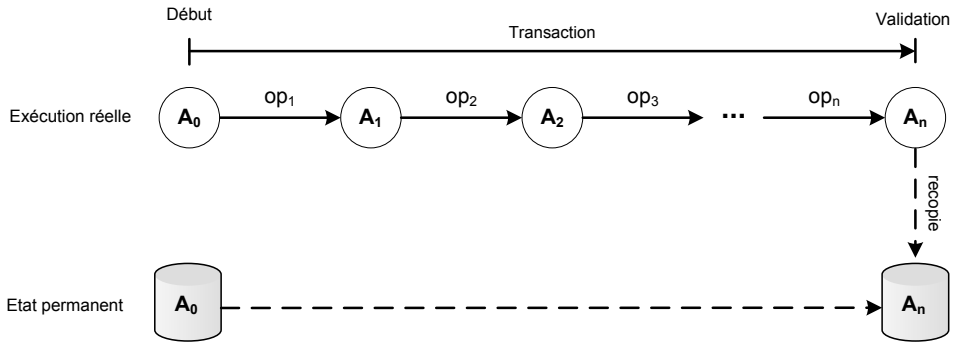

FIGURE 6.15 – Sauvegarde de l'état à la validation d'une transaction de reconfiguration

6.6 Conclusion

Notre solution pour la fiabilisation des reconfigurations dynamiques dans les architectures à composants passe par la définition d'un modèle de transactions adapté aux reconfigurations qui manipulent un système à l'exécution. Ce modèle tient compte des particularités des reconfigurations : brièveté, sémantique des opérations, architecture à base de composants. Le choix du mode de mise à jour des transactions (immédiate ou différée) est ainsi un problème important dans le cas de la reconfiguration dynamique de systèmes en ayant une influence sur leur disponibilité et sur le modèle de recouvrement des défaillances.

Les propriétés classiques des transactions (propriétés ACID) sont déclinées dans ce contexte particulier. Les transactions permettent ainsi de préserver la cohérence des systèmes définie par des

contraintes d'intégrité tout en autorisant des reconfigurations concurrentes. Elles permettent également de rendre les systèmes reconfigurés tolérants aux fautes. L'atomicité des reconfigurations est garantie en considérant l'architecture d'un système comme une seule et unique ressource transactionnelle au cours le validation dans un contexte réparti et grâce à un modèle pour défaire les opérations en cas d'annulation des transactions. Une approche pessimiste à base de verrouillage est adoptée pour le contrôle de concurrence entre reconfigurations pour éviter les conflits entre opérations. En plus de cette synchronisation entre les reconfigurations, il est nécessaire de synchroniser les reconfigurations avec l'exécution fonctionnelle des composants des systèmes. Une gestion des dépendances entre cycles de vie des composants a été proposé à cette occasion. Une fonctionnalité essentielle des transactions est la possibilité de reprise en cas de défaillance aussi bien logicielles (abandon de transaction) que matérielles (défaillance de site). Pour cela, un modèle de recouvrement est proposé utilisant l'atomicité des opérations de reconfiguration et la durabilité de l'état des architectures à composants reconfigurées. La durabilité mise en oeuvre fait appel à des techniques de journalisation et points de contrôle.

Chapitre 7

Architecture globale et modulaire des mécanismes de reconfigurations fiables

Sommaire

Le modèle Fractal et Julia, utilisés pour notre implémentation, ont été étendus pour fournir une sémantique transactionnelle aux opérations de reconfiguration. L'extensibilité du modèle peut reposer sur la définition de nouveaux contrôleurs pour les composants et sur une bibliothèque de composants génériques et réutilisables. Les mécanismes transactionnels reposent sur un composant global, le *gestionnaire de transactions*, qui est composé d'un certain nombre de composants, plus ou moins génériques, implémentant différentes fonctionnalités telles que la persistance, la gestion de concurrence ou encore la vérification de la cohérence des configurations. L'architecture du gestionnaire de transactions est modulaire par le fait que son architecture peut être construite à la carte en choisissant les différents composants qui le constituent, ces composants sont détaillés dans la section 7.1. D'autre part, notre extension consiste aussi en l'ajout de contrôleurs, décrits dans la section 7.2, dédiés aux différentes préoccupations des systèmes transactionnels comme les propriétés ACID des transactions. Nous décrivons enfin dans la section 7.3 l'ensemble des extensions apportées au langage et compilateur Fractal ADL pour mettre en oeuvre les reconfigurations transactionnelles.

7.1 Une architecture modulaire à base de composants

Le gestionnaire de transactions est le composant qui assure le support des reconfigurations transactionnelles au sein d'un système transactionnel. Ce composant est composé de plusieurs sous-composants (cf. Figure 7.1) implémentant les différentes propriétés des transactions dans le contexte

des reconfigurations dynamiques. Le composant central dans l'architecture est le moniteur transactionnel, ce composant gère essentiellement le protocole de validation des transactions.

Pour pouvoir bénéficier des propriétés d'autres fonctionnalités telles que la durabilité de l'état des composants ou encore le maintien de la cohérence par des contraintes d'intégrité, des composants optionnels peuvent être ajoutés dans l'architecture et connectés au moniteur transactionnel. Ces composants sont considérés comme des plugins pour le gestionnaire de transactions qui peut ainsi être construit à la carte suivant les besoins : un système transactionnel peut par exemple se passer de la durabilité implémentée par le composant *DurabilityManager* si le système transactionnel ne nécessite pas de tolérance aux pannes franches. De plus, certains de ces composants plugins disposent de plusieurs implémentations alternatives, pour la gestion de la concurrence (pessimiste ou optimiste) ou bien encore le support de la persistance (sérialisation dans des fichiers ou sauvegarde en base de données).

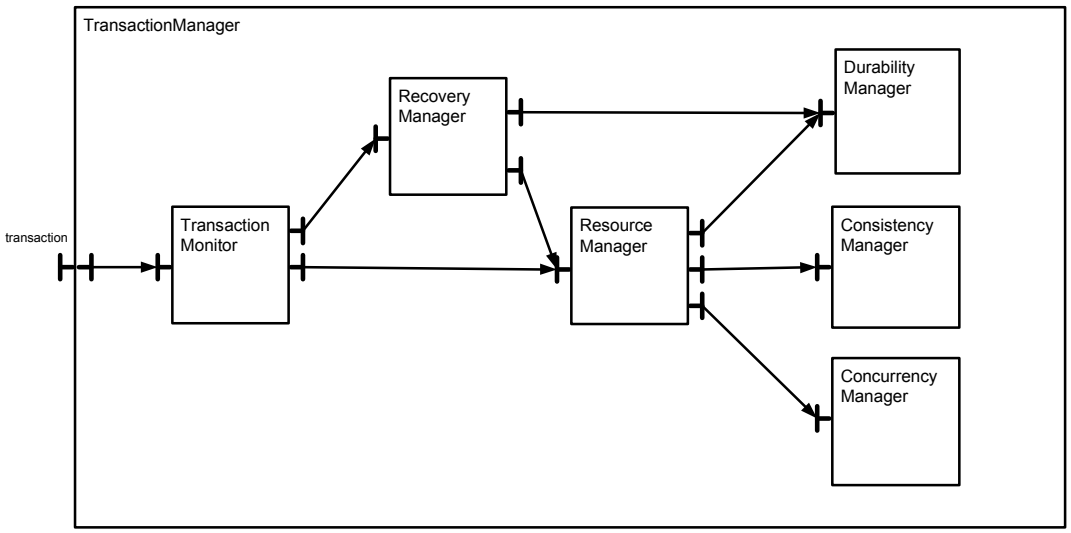

FIGURE 7.1 – Architecture du Gestionnaire de Transactions

7.1.1 Le moniteur transactionnel

Le moniteur transactionnel est le composant au coeur de l'architecture du système transactionnel, i.e. du gestionnaire de transactions et des composants de l'application dont les reconfigurations doivent être exécutées de manière transactionnelle. Il joue le rôle de coordinateur de l'exécution des transactions dans le système et de point d'entrée pour le gestionnaire de transactions (cf. Figure 7.2) et des composants qui le constituent. Le moniteur transactionnel implémente le modèle de transactions choisi pour les reconfigurations dynamiques. Ce composant gère ainsi la création, la démarcation, la propagation et la validation des transactions.

Le composant TxCommitProtocol. Ce composant implémente le protocole de validation des transactions de reconfiguration en vue de garantir l'atomicité globale de l'état des composants du système. Aucun protocole de validation n'est adapté à tous les contextes applicatifs. Dans notre solution, le gestionnaire de transactions est centralisé et le moniteur transactionnel est un singleton dans le système. Les composants de l'application dont les reconfigurations sont transactionnelles peuvent par contre être répartis mais nous ne considérons cependant qu'une seule ressource transactionnelle globale définie par le modèle de l'application qui est donc le seul participant dans toute transaction de reconfiguration. L'implémentation existante utilise l'API standard JTA en Java [JTA]. Le code de reconfiguration du client de la reconfiguration appelle explicitement l'interface JTA *UserTransaction* pour démarquer les transactions et pour récupérer une référence sur la transaction courante afin de pouvoir exécuter une reconfiguration transactionnelle. D'autre part, pour prévenir les cas d'interblocage ou simplement borner la durée d'une reconfiguration, un « timeout »paramétrable est associé à chaque transaction de reconfiguration, une transaction dont la durée dépasse le « timeout »est annulée.

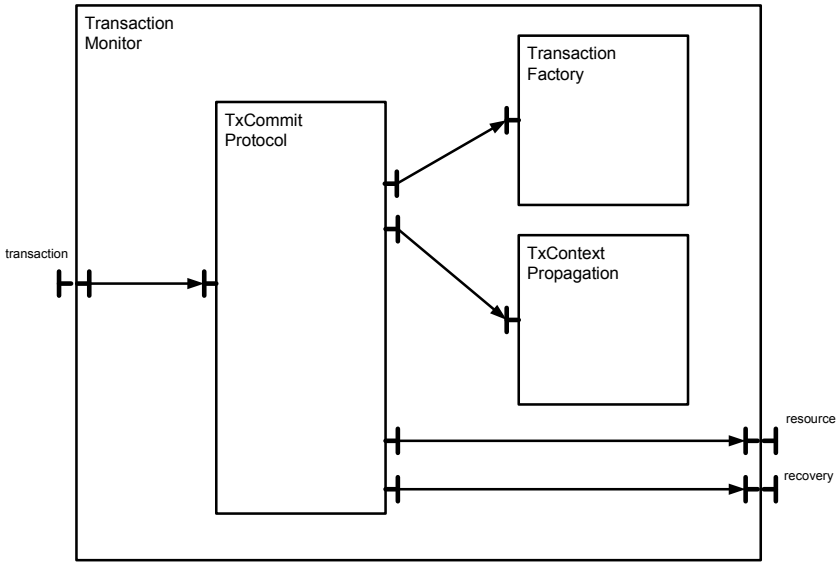

FIGURE 7.2 – Architecture du Moniteur Transactionnel.

Le composant TransactionFactory. Le composant *TransactionFactory* est une usine de transactions reposant sur le patron de conception *Factory* [GHJV94] qui implémente le modèle de transactions choisi pour les reconfigurations dynamiques. Le modèle implémenté par défaut est un modèle plat (cf. chapitre 6.1) dans le lequel une transaction ne peut-être imbriquée et décomposée en sous-transactions. Ce modèle est le plus répandu dans les bases de données et est adapté à des transactions dont le temps d'exécution est court, ce qui est le cas général des reconfigurations. Chaque transaction possède un identifiant unique dans tout le système transactionnel.

Le composant TxContextPropagation. Différents types de propagation peuvent être envisagés lors de l'exécution d'une méthode transactionnelle si une transaction est déjà en cours d'exécution par la même entité active (processus ou thread). Le type de propagation choisi par défaut et implémenté correspond à la politique de démarcation *Required* du standard J2EE [Jav]. Cette politique de démarcation requiert qu'une opération soit toujours exécutée dans un contexte transactionnel. Lors de l'exécution d'une opération, si un contexte transactionnel existe, il sera propagé à l'opération, sinon un nouveau contexte sera créé. Plusieurs paramètres sont configurables dont l'attribut d' « autocommit »du composant, activé par défaut, qui indique que toute opération de reconfiguration non démarquée le sera automatiquement. Une opération de reconfiguration isolée constitue dans ce cas une transaction.

La propagation des transactions entre différents processus et donc entre machines dans le cas des applications distribuées nécessite le maintien d'un contexte transactionnel associé à l'entité active courante participant à la transaction (cf. Figure 7.3). Ce contexte transactionnel permet d'identifier la transaction courante sur chaque site.

Une application à base de composants Fractal distribuée en Java peut utiliser les liaisons composites de Fractal RMI, qui est une adaptation de Java RMI pour Fractal, pour implémenter les communications réparties entre interfaces de composants distants. Le protocole de communication dans Fractal RMI (ainsi que dans Java RMI) ne permet pas le passage d'un contexte quelconque lors de l'invocation d'une méthode distante. Le protocole de Fractal RMI a donc été modifié pour permettre le passage de ce contexte entre plusieurs JVM comme présenté dans la figure 7.4. Dans le cas des reconfigurations transactionnelles, ce contexte correspond à l'identifiant de la transaction associée au thread courant. Ainsi, si une reconfiguration concerne des composants répartis sur plusieurs sites, la même et unique transaction pourra être propagée sur les différents sites.

7.1.2 Le gestionnaire de ressources

Les composants sont la granularité de base pour la gestion des reconfigurations. Les reconfigurations en tant que préoccupation non fonctionnelle sont en effet définies dans les différents contrôleurs

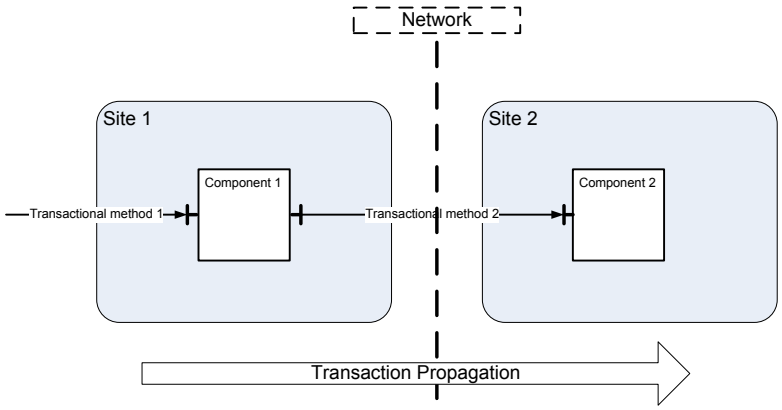

FIGURE 7.3 – Propagation des transactions entre composants répartis

Request Message				
Header				Payload
request identifier (4 bytes)	object key length (4 bytes)	object key data (variable length)	context (variable length)	payload (variable length)

Reply Message			
Header			Payload
request identifier (4 bytes)	exception flag (1 byte)	context (variable length)	payload (variable length)

FIGURE 7.4 – Protocole modifié de Fractal RMI

de chaque composant. La granularité pour la gestion des transactions de reconfiguration est donc aussi le composant en tant qu'unité de base pour le partitionnement de l'état global de l'application. L'état d'une application est défini comme l'ensemble des états de tous les composants qui la composent. Chaque composant est responsable de l'activation des transactions pour ses propres opérations de reconfiguration primitives.

Le composant *ResourceManager* 7.5 en tant que sous composant du *TransactionManager* est responsable de la gestion des ressources transactionnelles, c'est à dire l'ensemble des composants de l'application, de l'enregistrement des opérations de reconfiguration transactionnelles.

Le composant ResourceManager. Chaque composant « transactionnel »doit être enregistré auprès du gestionnaire de ressources. L'enregistrement d'un composant se fait par l'intermédiaire d'un contrôleur spécifique (*TransactionController*), cet enregistrement est fait automatiquement si le composant est instancié par une usine ADL transactionnelle. Par ailleurs, les opérations de reconfiguration exécutées doivent aussi être enregistrées dans le gestionnaire de ressources pour être associées à une transaction. La membrane d'un composant enregistré intercepte et notifie les invocations d'opérations de reconfiguration primitives au composant *ResourceManager*. Le composant *ResourceRegistration* ajoute alors les opérations invoquées au journal de la transaction courante. Ces opérations sont enregistrées au fur et à mesure de leur exécution et chaque opération est associée à un contexte transactionnel.

Chaque opération de reconfiguration primitive définie dans les contrôleurs des composants est représentée par un wrapper dans notre modèle d'opérations. Un wrapper d'opération est défini par une classe Java chargeable dynamiquement qui doit au minimum implémenter l'interface ReconfigurationOperation (cf. listing 7.1). Les wrappers sont obtenus grâce au composant *ResourceFactory* à partir du nom de l'opération et de ses arguments. L'ensemble des wrappers constitue une bibliothèque d'opérations contenue dans le composant *ResourceModel*. Une opération est ainsi définie par l'exécution d'une opération concrète dans le système, un ensemble de pré et postconditions, et par un ensemble de propriétés (dont l'idempotence par exemple). Une opération doit être sérialisable pour pouvoir être enregistrée à distance. Tout comme il est possible d'ajouter de nouvelles opérations

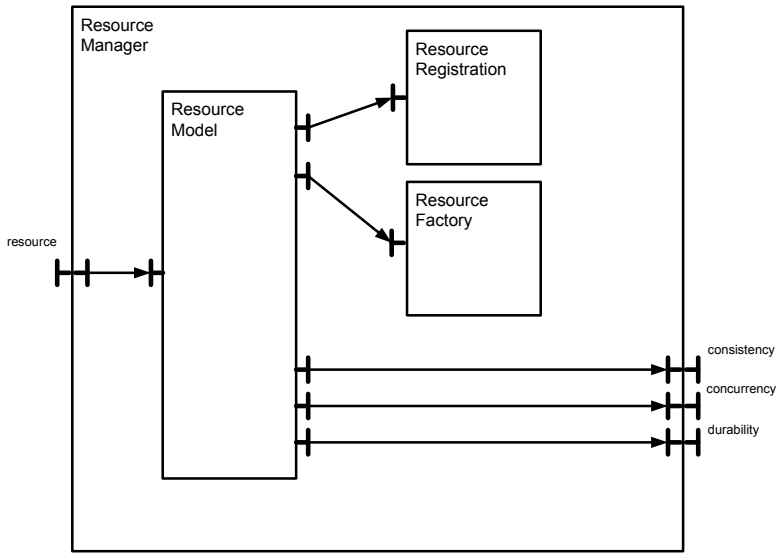

FIGURE 7.5 – Architecture du Gestionnaire de Ressources.

de reconfiguration dans le modèle Fractal par l'intermédiaire de nouveaux contrôleurs, de nouveaux wrappers d'opération peuvent être définis pour les nouvelles opérations.

```
1  public interface ReconfigurationOperation extends Serializable {

3    String getName();

5    String getControllerName();

7    Object apply() throws Throwable;

9    Object[] getArgs();

11   Object[] getTargets();

13   void preconditions() throws Exception;

15   void postconditions() throws Exception;

17   boolean isIntercession();

19   boolean isPrimitive();

21   Object getProperty(String name);

23   void setProperty(String name, Object value);

25 }
```

Listing 7.1 – Interface des opérations de reconfiguration

Une opération réversible doit implémenter l'interface *Undoable* (cf. listing 7.2). Les inverses des opérations standard sont initialisés à leur valeur par défaut dans la sémantique de base de Julia.

```
1  public interface Undoable {

3    Object undo() throws Throwable;

5    public void setUndo(Operation undo);

7    public Operation getUndo();
   }
```

Listing 7.2 – Interface pour les opérations inversibles

7.1.3 Le gestionnaire de recouvrement

Le composant *RecoveryManager* (cf Figure 7.6) a pour objectif d'implémenter la stratégie de recouvrement du système en cas de pannes. Concernant la stratégie de recouvrement, plusieurs type d'exceptions sont détectables, le système peut adapter le recouvrement en fonction du type d'exception rencontrée. La politique de recouvrement est donc flexible. L'implémentation actuelle considère deux politiques de recouvrement correspondant à deux types de fautes dans notre modèle de fautes : les fautes logicielles et les pannes franches.

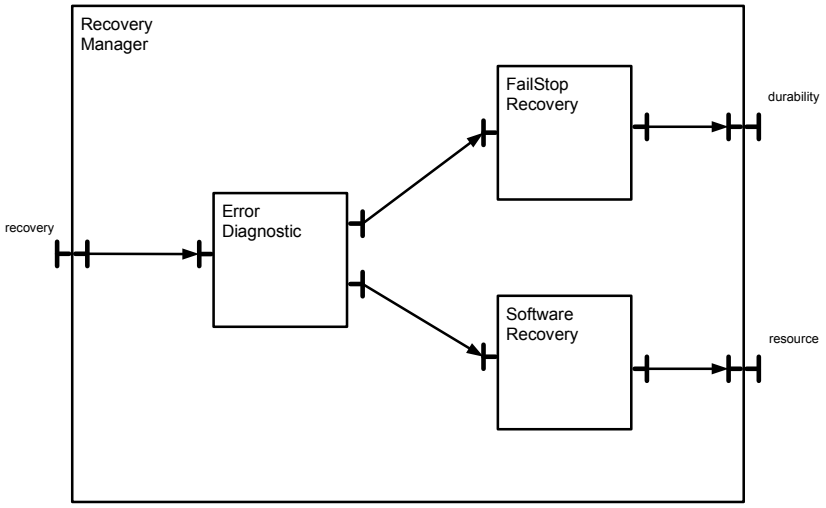

FIGURE 7.6 – Architecture du Gestionnaire de Recouvrement.

Le composant ErrorDiagnostic. Le composant de diagnostique d'erreur est en charge de déléguer la réparation de l'erreur rencontrée au cours d'une reconfiguration à un gestionnaire de recouvrement en fonction du type d'erreur détectée. Concernant la détection des fautes, le fonctionnement de l'implémentation basique du composant n'utilise pas de détecteurs de fautes spécifiques (heartbeat, ping, etc), le composant se contente de capturer les différentes exceptions levée lors de l'exécution des opérations de reconfiguration. Par défaut, toute exception à l'exécution d'une opérations de reconfiguration entraîne le recouvrement de la transaction, aussi bien les exceptions non capturées héritant de *RuntimeException* que les exceptions déclarées dans les signatures de méthode (ex : *IllegalBindingException* pour la méthode *bindFc*) [1]. Le diagnostique consiste alors en la transmission du traitement de la faute à un composant de recouvrement en fonction du type d'exception rencontré. Les exceptions liés à des problèmes de connexion du réseau lors de l'utilisation des liaisons Fractal RMI (*RemoteException*) sont ainsi traitées comme des pannes franches. Les exceptions rencontrées lors de violation de contraintes d'intégrité (*ConstraintViolationException*) sont traitées comme des faute logicielles.

Le composant SoftwareFaultRecovery. Le composant de traitement des fautes logicielles utilise le recouvrement sous forme de reprise par annulation (*rollback*) de la transaction qui a échoué. Dans le cas de la mise à jour immédiate (cf. chapitre 6), les opérations enregistrées dans le journal de la transaction sont défaites dans l'ordre inverse de leur première exécution. Cette reprise en défaisant les opérations est beaucoup moins intrusive dans un système à l'exécution que de réinstancier les composants impliqués dans la reconfiguration dans leur état antérieur suivant le principe des points de contrôle (checkpointing). Lors de l'annulation d'une transaction suite à une exception logicielle, le recouvrement est soit automatique et l'annulation de la transaction est exécuté avec notification au client de la reconfiguration (remontée d'une *RollbackException*), soit le recouvrement est manuel et dans ce dernier cas, le rollback de la transaction doit être explicitement invoqué par le client.

1. Il est cependant possible de surcharger ce comportement via la méthode *noRollBackFor* du composant *ErrorDiagnostic*.

Le composant FailStopFaultRecovery. L'implémentation d'une politique complète de réparation de panne franche n'est pas du ressort de ce composant. Celui-ci garantit seulement que pour une réinstanciation du système (*reboot*), l'état du du système réinstancié (architecture et états des composants) sera le même que celui d'avant la panne. Il utilise pour cela le composant gestionnaire de durabilité pour déterminer si un état cohérent existe pour les composants réinstanciés et sélectionne le dernier disponible.

L'interface du composant RecoveryManager (cf. listing 7.3) permet explicitement de refaire ou défaire une transaction à partir de son journal en mémoire ou à partir des enregistrement (*logs*), ou bien encore de déclencher un point de contrôle (*checkpoint*) du système.

```
public interface RecoveryManager {

  void recover(Transaction tx, Operation op, Throwable e) throws RecoveryException;

  boolean noRollbackFor(Operation op, Throwable th);

  void checkpoint(Transaction tx) throws Throwable;

  void undo(Transaction tx) throws Throwable;

  void redo(Transaction tx) throws Throwable;

  void undoLog(int txid) throws Throwable;

  void redoLog(int txid) throws Throwable;

}
```

Listing 7.3 – Interface du gestionnaire de recouvrement

7.1.4 Le gestionnaire de cohérence

L'objectif principal de l'utilisation des transactions pour les reconfigurations dynamiques est le maintient de la cohérence de l'état du système. Cette cohérence est définie par un ensemble de contraintes d'intégrité sous forme d'invariant de configuration et de préconditions et postconditions d'opérations de reconfiguration. La vérification de la cohérence dans le système est à la charge du composant *ConsistencyManager* (cf. Figure 7.7).

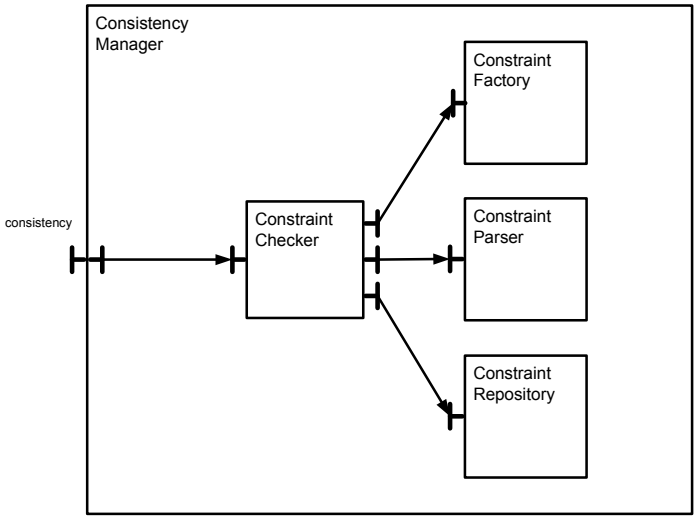

FIGURE 7.7 – Architecture du Gestionnaire de Cohérence.

Le composant ConstraintChecker. Le composant *ConstraintChecker* (cf. Figure 7.4) est le composant en charge de la vérification des contraintes d'intégrité dans le système. Les contraintes d'intégrité applicatives sont attachées à chaque composant du système par l'intermédiaire d'un contrô-

leur de contraintes(*ConstraintController*) tandis que les contraintes au niveau profil et modèle sont stockées dans le composant *ConstraintRepository*. Le composant *ConstraintFactory* est dédié à la création des contraintes suivant leur niveau. Pour que les contraintes sur un composant soient prise en compte lors d'une reconfiguration, le composant doit être enregistré auprès du *ConstraintChecker*, l'enregistrement peut être ou non récursif : les contraintes considérées peuvent être seulement celle du composant ciblé ou bien également celles de l'ensemble de ses sous-composants. Une contrainte est essentiellement désignée par une chaîne de caractères exprimé dans un langage dédié (cf. chapitre 4), le vérificateur de contraintes joue alors le rôle d'un interpréteur de ce langage de contraintes. La plupart des contraintes sont définies au moment du déploiement du système et ne sont plus modifiées à l'exécution : il peut dès lors être intéressant si l'implémentation du vérificateur de contraintes le supporte de compiler les expressions des contraintes. Lorsqu'une contrainte est ajouté au niveau du *ConstraintController* d'un composant, elle est alors automatiquement parsée par le composant *ConstraintParser* et stockée dans sa forme compilée. Cette forme compilée sera utilisée à chaque vérification pour éviter d'avoir à parser à nouveau les contraintes.

Le composant *ConstraintChecker* offre une API générique de vérification de contraintes. Une fois le composant enregistré dans le *ConstraintChecker*, plusieurs solutions sont offertes pour vérifier ses contraintes d'intégrité :

- Les contraintes sur un composant du système peuvent être vérifiées (même si le composant n'est pas enregistré) en invoquant explicitement l'API du *ConstraintChecker* avec le composant cible en paramètre.
- Lorsque le composant est enregistré, il est possible de procéder à une vérification de contraintes sur un composant du système en passant par son contrôleur de contraintes *ConstraintController*, la vérification auprès du *ConstraintChecker* se fait alors de manière transparente.

Le composant *ConstraintChecker* est par défaut un singleton dans le système mais plusieurs instances peuvent être réparties dans le système. Une répartition intéressante d'un point de vue optimisation pour un système réparti est d'instancier un composant de vérification par site et d'y enregistrer les composants colocalisés. Par ailleurs, plusieurs vérificateurs de contraintes différents avec des préoccupations différentes peuvent être utilisés pour un même composant. D'autre types d'assertions et contraintes que les contraintes d'intégrité de notre approche sont utilisables, des contrats au niveau fonctionnel pourraient ainsi être vérifiés après une reconfiguration pour valider la transaction. L'implémentation choisie dans le composant fournie pour la vérification des contraintes d'intégrité est un interpréteur FPath avec un « frontend »pour tenir compte notamment des spécificités des contraintes génériques.

Le *ConstraintChecker* est indépendant des mécanismes transactionnels. Il peut être utilisé en dehors des reconfigurations transactionnelles par des invocations explicites à son interface *ConstraintChecker*. Par ailleurs, un mixin complétant le *LifeCycleController* a été développé pour interagir avec le *ConstraintController* de chaque composant et ainsi vérifier le respect des contrainte au moment de son (re)démarrage (invocation de l'opération *startFc()*).

```
1  public interface ConsistencyManager {
     void check(Constraint constraint, Map context)
3        throws ConstraintViolationException;

5    Object parseContraint(Constraint constraint) throws ConstraintParsingException;

7    boolean makeConstraintAware(Component component, boolean recursive)
         throws ConstraintViolationException;
9
     void check(Component component, boolean recursive)
11       throws ConstraintViolationException, ConstraintParsingException;

13   boolean addGenericConstraint(Constraint constraint);

15   boolean removeGenericConstraint(Constraint constraint);

17 }
```

Listing 7.4 – Interface du gestionnaire de cohérence

7.1.5 Le gestionnaire de concurrence

Le composant *ConcurrencyManager* implémente la politique de gestion de concurrence du système transactionnel. On distingue généralement deux principaux types de stratégies de gestion de la concurrence dans un système transactionnel : les stratégies optimistes (ou par validation), et les stratégies pessimistes (ou par verrouillage) (cf. 6.4).

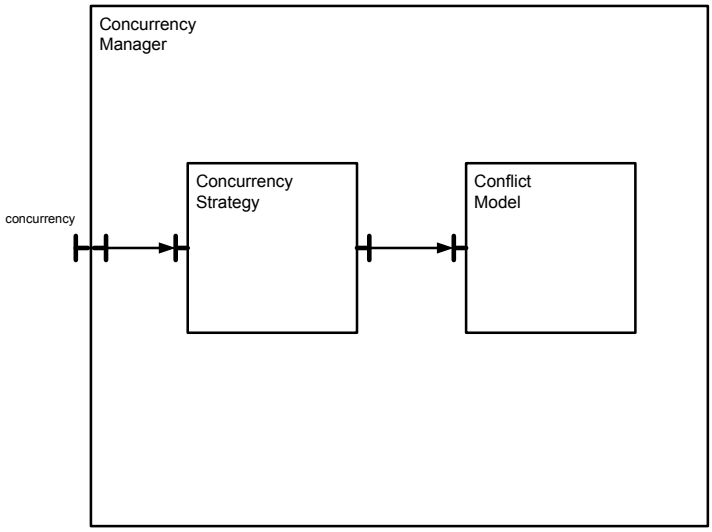

FIGURE 7.8 – Architecture du Gestionnaire de Concurrence.

Le composant ConcurrencyStrategy. Seule une stratégie de gestion de concurrence par verrouillage a été implémentée dans notre solution au sein du composant *ConcurrencyStrategy*. La granularité de base du verrouillage est l'opération de reconfiguration (introspection ou intercession) pour chaque composant, l'état des composants n'est en effet pas toujours accessible et donc verrouillable directement. Chaque opération de reconfiguration est potentiellement en conflit avec les autres opérations pour la mise à jour ou la lecture de l'état de l'application. Un graphe d'attente sur les verrous est maintenu entre les transactions en vue de détecter d'éventuel cycles, c'est-à-dire des interblocages. En cas de détection d'interblocage entre transactions de reconfiguration, le composant annule une des transactions en cause, le gestionnaire de recouvrement tente alors de refaire les transactions après un certain délai qui est configurable. Le nombre d'essais pour une transaction qui échoue est également paramétrable.

Le composant ConflictModel. Ce composant implémente les conflits entre les différentes opérations de reconfiguration. Ces conflits sont spécifiés dans un fichier de définition XML qui recense pour chaque opération l'ensemble des conflits avec les autres opérations primitives d'intercession et d'introspection (cf. listing 7.5). Le composant *ConflictModel* traduit ce fichier en une table de conflits pour en cas d'invocation d'opération verrouiller les autres opérations conflictuelles soit en lecture soit en écriture. La méthode *controlConflicts* de l'interface du gestionnaire de concurrence (cf 7.6) permet de déclencher suivant la stratégie adoptée, soit la validation, soit le verrouillage, pour la transaction et l'opération considérée. Le verrouillage des opérations se fait au niveau des contrôleurs de verrouillage (*LockController*) de chaque composant. L'invocation d'une opération de reconfiguration sur un composant peut entraîner le verrouillage d'opérations sur d'autre composants du système, ces composants sont alors désignés par une expression FPath.

```
1 <conflicts>
    <operation method="addFcSubComponent" controller="content-controller">
3     <conflict component="." interface="content-controller" lockMode="W">
        <operation name="*"/>
5     </conflict>
      <conflict component="./*" interface="super-controller" lockMode="W">
7       <operation name="*"/>
      </conflict>
```

```
  </operation>
  <operation method="getFcSubComponents" controller="content-controller">
    <conflict component="." interface="content-controller" lockMode="W">
      <operation name="addFcSubComponent"/>
      <operation name="removeFcSubComponent"/>
    </conflict>
    <conflict component="." interface="content-controller" lockMode="R">
      <operation name="getFcSubcomponents"/>
      <operation name="getFcInternalItf"/>
    </conflict>
  </operation>
  ...
</conflicts>
```

Listing 7.5 – Implémentation des conflits entre opérations

```
public interface ConcurencyManager {

  void controlConflicts(Transaction tx, Operation op) throws ConcurrencyException;

}
```

Listing 7.6 – Interface du gestionnaire de concurrence

7.1.6 Le gestionnaire de durabilité.

La durabilité des transactions par la persistance de l'état du système est du ressort du composant *DurabilityManager*. Ce composant est lui-même composé de plusieurs sous-composants : un composant de persistance de l'architecture du système, un composant de persistance de l'état fonctionnel des composants, et un composant de journalisation (log) des transactions. Tout composant du système peut être persisté, récursivement ou non, sous forme de définitions ADL mais seuls les composants déclarés comme persistant ont leur état fonctionnel persisté. Dans le système transactionnel, l'état d'un composant est persisté uniquement si il est impliqué dans une transaction de reconfiguration.

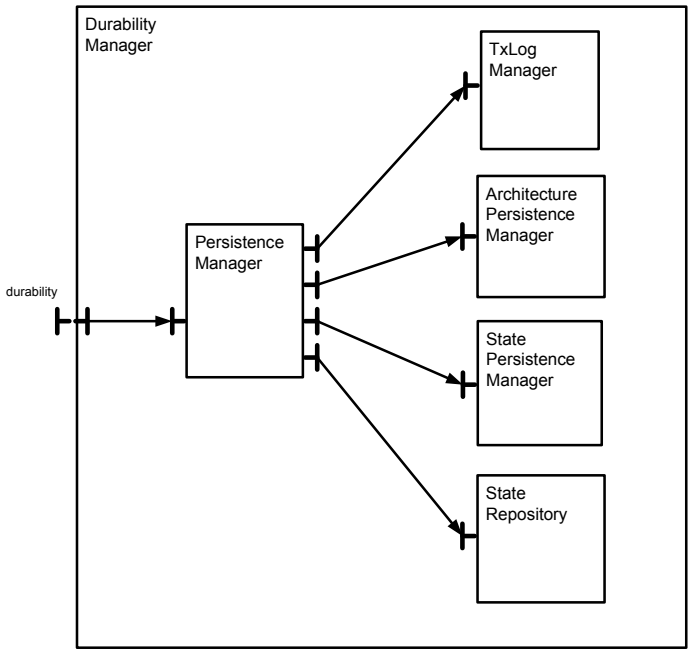

FIGURE 7.9 – Architecture du Gestionnaire de Durabilité.

Le composant ArchitecturePersistenceManager. Il est chargé de sérialiser les définitions ADL des composants reconfigurés. La persistance de l'architecture d'un composant est réalisée par

introspection de son architecture grâce aux opérations des contrôleurs Fractal. La définition ADL produit de la persistance est alors conservée dans deux fichiers différents : les informations mutables sont en effet séparées des informations immutables (par exemple les informations de typage, le typage étant immutable dans le modèle Fractal). Lors de l'instanciation d'un nouveau composant, l'ensemble des informations mutables et immutables est entièrement persistée dans des fichiers ADL. Pour d'autres types d'opérations de reconfiguration, seul la partie mutable du composant est persistés et remplace l'ancienne version du fichier. La sérialisation de l'ADL est extensible pour tenir compte d'éventuelles extensions du langage de Fractal ADL. Il est ainsi possible d'ajouter des méta-données dans les définitions ADL tel que l'état du cycle de vie des composants qui n'est normalement pas présent dans la définition standard des composants.

Les composant StatePersistenceManager. Ce composant permet de rendre durable l'état fonctionnel des composants. Plusieurs implémentations sont disponibles pour réaliser cet objectif. Une première implémentation utilise les mécanismes de sérialisation Java pour persister les objets déclarés persistant dans la définition ADL des composants. Une deuxième implémentation repose sur le mapping Objet-Relationnel (ORM) et l'utilisation d'une base de donnée. Le framework de persistance Hibernate[2] est utilisé dans cette implémentation, un fichier de mapping des objets correspondant à l'état fonctionnel du composant vers des tables de la base de données doit alors être défini pour chaque composant.

Le composant StateRepository. Ce composant est un composant centralisé et un singleton dans le système transactionnel. Il est à la fois rattaché au *PersistenceManager* et au composant *RecoveryManager* et sert d'endroit de stockage de l'état des composants du système et notamment des définitions ADL persistées des composants. Le composant *StateRepository* contient une base de données si l'implémentation retenue pour la persistence fonctionnelle est du type ORM. L'état persisté peut alors être récupéré en vue du recouvrement de panne.

Le composant TxLogger. Les différents événements des transactions sont journalisés sur support persistant (sur disque) dans un fichier de log. Les informations du journal comprennent une estampille temporelle, les identifiants uniques des transactions, leur démarcation ainsi que les différentes opérations de reconfiguration qui ont été exécutées pendant les transactions. Le format du journal suit la syntaxe FPath et les éléments Fractal sont désignés par leur chemin FPath dans l'architecture quand ils sont passés en argument des opérations de reconfiguration. Le chemin peut ne pas être unique en raison du possible partage de composants.

```
1  public interface PersistenceManager {
     void serializeArch(Component comp, boolean def, boolean type)
3        throws IOException, IllegalLifeCycleException;

5    File getSerializedArch(String componentName, int timestamp);

7    File getSerializedArch(String componentName, int timestamp);

9    void serializeState(Component comp, boolean recursive) throws IOException,
         IllegalLifeCycleException;
11
     void deserializeState(Component comp, boolean recursive) throws IOException,
13       IllegalLifeCycleException;

15   void persist(Component comp, String controller) throws Exception;

17   void log(Transaction tx, Demarcation dem);

19   void lod(Transaction tx, Operation op, Modifier, mod);

21 }
```

Listing 7.7 – Interface du gestionnaire de persistance

2. http ://www.hibernate.com

7.2 Des contrôleurs Fractal pour la gestion des transactions

La gestion des transactions est généralement considérée dans les différents modèles de composants (e.g., dans les EJB [DeM03]) comme un service technique orthogonal aux fonctionnalités métier d'une application. Dans le modèle Fractal, les contrôleurs sont des interfaces dédiées aux préoccupations qui ne relèvent pas directement du niveau fonctionnel, nous choisissons donc d'implémenter les différentes fonctionnalités des transactions pour chaque composant sous la forme de nouveaux contrôleurs Fractal.

7.2.1 Contrôleurs et propriétés ACID

La granularité considérée pour la gestion des transactions est le composant, de nouveaux contrôleurs sont ainsi définis pour gérer les transactions au niveau de chaque composant dans le système. Ces contrôleurs ont été ajoutés aux contrôleurs standard de Fractal pour implémenter les propriétés ACID des reconfigurations. Ils sont requis par les composants du système dont les reconfigurations doivent être transactionnelles. Les composants du gestionnaire de transactions utilise en effet ces nouveaux contrôleurs lors de l'exécution de transactions de reconfiguration. Leur implémentation est réalisée sous la forme de contrôleurs dans Julia (éventuellement à base de mixins) insérés dans les descripteurs de membranes de composants dans le fichier de configuration de Julia.

Le contrôleur de contraintes. Le *ConstraintController*, dont l'interface est donnée dans le listing 7.8, regroupe les fonctionnalités liées à la gestion de la cohérence d'une configuration et donc à la vérification de contraintes pour un composant donné. Il contient notamment la liste des contraintes applicatives associées au composant, contraintes que l'on peut ajouter ou enlever dynamiquement du composant. L'ensemble des contraintes est normalement initialisé lors de l'instanciation d'un composant. Ce contrôleur permet également de connecter le composant à un vérificateur de contraintes implémentant l'interface *ConsistencyManager* en charge de la validation des contraintes, par exemple avant la validation d'une transaction. Une exception du type *ConstraintViolationException* est retournée en cas de violation de contrainte. La vérification peut être soit uniquement pour le composant possédant le contrôleur, soit récursive à tous ses sous-composants. L'interface du contrôleur ne fait aucune hypothèse quand la forme des contraintes ni sur le moment où elles sont vérifiées. Ce contrôleur est optionnel si l'on n'utilise pas la vérification de contraintes pour valider la cohérence des reconfigurations.

```
1  public interface ConstraintController {
     void addConstraint(Constraint constraint, boolean recursive)
3        throws ConstraintViolationException;

5    boolean removeConstraint(Constraint constraint, boolean recursive);

7    Set<Constraint> getConstraints();

9    void setChecker(ConsistencyManager checker, boolean recursive)
         throws ConstraintViolationException;
11
     ConsistencyManager getChecker();
13
     void checkConstraints(boolean recursive)
15       throws ConstraintViolationException, ConstraintParsingException;
   }
```

Listing 7.8 – Interface du Contrôleur de contraintes

Le contrôleur de préconditions et postconditions d'opérations. Le *OperationController* (cf. listing 7.9) est complémentaire au *ConstraintController* et vérifie les préconditions et postconditions des opérations de reconfiguration avant et après leur exécution. Il procède par interception des méthodes de reconfiguration et peut être désactivé, la vérification est automatique. En cas de violation d'une condition, une exception du type *ConstraintViolationException* est levée.

```
public interface OperationController {
  void addCondition(Constraint constraint, Operation op, boolean recursive);

  boolean removeCondition(Constraint constraint, Operation op, boolean recursive);

  Set<Constraint> getConditions();

  Set<Constraint> getConditions(Operation op);

  void setChecker(ComponentConstraintChecker checker, boolean recursive)
      throws ConstraintViolationException;

  ComponentConstraintChecker getChecker();
}
```

Listing 7.9 – Interface du contrôleur de pré et postconditions sur les opérations

Le contrôleur d'état. Le *StateController* à travers son interface donnée dans le listing 7.10 permet de récupérer et d'initialiser l'état du composant, c'est-à-dire la description de son architecture interne et son état fonctionnel à travers des méthodes de sérialisation et de désérialisation. La persistance des données est déléguée à un composant gestionnaire de persistance implémentant l'interface *PersistenceManager* qui aura été enregistré au préalable auprès du composant. La persistance de l'état du composant peut ainsi être déclenchée lors de la validation d'une transaction et la persistance peut être réalisée sur le composant uniquement ou récursivement. L'interface de contrôle ne précise pas le format de persistance de l'état, l'objectif étant de pouvoir supporter plusieurs implémentations de la simple sérialisation Java à l'utilisation de mappings (Objetc Relational Mapping) vers des bases de données. L'état n'étant pas représenté explicitement, ce contrôleur ne permet pas de réaliser directement un transfert d'état entre deux instances de composants.

Un contrôleur générique de transfert d'état *StateTransferControlleur* (cf. listing 7.11) complémentaire au *StateController* est défini pour manipuler directement cet état par des accesseurs sous une forme à préciser dans les implémentations. Il est utile notamment dans le cas d'une reconfiguration comme le remplacement dynamique de composant durant lequel l'état de l'ancien composant doit être transféré vers la nouvelle instance. Comme rien ne suppose cependant que les états aient la même forme, une étape de transformation spécifique (étape de coercition) entre l'ancien état et le nouvel état peut être nécessaire.

Le *StateController* est optionnel si la persistance n'est pas activée. Une implémentation par défaut est fournie qui considère que l'état d'un composant est constitué par les valeurs de l'ensemble de ses attributs de configuration. Cette solution est cependant limitée car elle ne capture pas l'état complet des composants mais seulement l'état réifié sous forme d'attributs.

```
public interface StateController {
  void serializeState(boolean recursive) throws IOException,
      IllegalLifeCycleException;

  void deserializeState(boolean recursive) throws IOException,
      IllegalLifeCycleException;

  boolean cleanState();

}
```

Listing 7.10 – Interface du contrôleur d'état

```
public interface StateTransferController {
  Object getState();

  void setState(Object state);
}
```

Listing 7.11 – Interface du contrôleur de transfert d'état

Le contrôleur de verrouillage. Le *LockController* (cf. listing 7.12) sert à la gestion de la concurrence entre reconfigurations par un verrouillage hiérarchique (approche pessimiste) des composants. Les éléments verrouillables pour un composant sont ses interfaces (fonctionnelles et de contrôle), ses attributs et les opérations dans les interfaces. Le verrouillage étant hiérarchique, le verrouillage d'une interface de composant entraine donc par exemple le verrouillage de l'ensemble des opérations contenues dans cette interface. Le verrouillage peut être récursif pour un composant et impliquer le verrouillage de tous ses sous-composants. La sémantique du verrouillage est au niveau du contrôleur est simple, les verrous sont réentrants et de deux types : partagés (en lecture) ou exclusifs (en écriture), leur granularité élémentaire est la méthode. Aucune hypothèse n'est faite au niveau de l'interface sur le modèle d'activité et donc sur la forme des entités actives qui peuvent verrouiller les éléments. Cependant dans la pratique et dans notre implémentation, les entité actives considérées sont des threads. Le *LockController* interagit avec un composant implémentant l'interface *ConcurrencyManager* qui gère la politique de verrouillage des transactions (maintient d'un graphe globale d'attente des verrous, gestion des deadlocks, etc.). Ce contrôleur est optionnel en cas de désactivation de la gestion de concurrence des reconfigurations.

```
1  public interface LockController {
     void lockInterface(String itfName, LockMode mode) throws DeadLockException,
3        NoSuchInterfaceException;

5    void unlockInterface(String itfName) throws NoSuchInterfaceException;

7    void lockAttribute(String itfName, LockMode mode) throws DeadLockException;

9    void unlockAttribute(String itfName);

11   void lockComponent(LockMode mode) throws DeadLockException;

13   void unlockComponent();

15   void lockMethod(String itfName, String methodName, LockMode mode)
         throws DeadLockException, NoSuchInterfaceException;
17
     void unlockMethod(String itfName, String methodName)
19       throws NoSuchInterfaceException;
   }
```

Listing 7.12 – Interface du contrôleur de verrouillage

Le contrôleur de transactions. Le *TransactionController* (cf. listing 7.13) est indispensable au bon fonctionnement des reconfigurations transactionnelles. Il permet d'activer ou non la prise en charge des transactions pour les reconfigurations dans un composant. S'il est activé, il intercepte les invocations des méthodes de reconfiguration et les enregistre auprès d'un composant implémentant l'interface *ResourceManager*. Les données fournies pour l'enregistrement consistent en l'opération réifiée, un modificateur d'interception et un contexte. Le modificateur d'interception caractérise le mode d'interception : une interception avant l'exécution de la méthode (*PRE*), après l'exécution de la méthode (*POST*) ou bien une remontée d'exception lors de l'exécution (*FAILED*). Le contexte contient notamment les éventuelles exceptions levées par l'opération de reconfiguration. A charge ensuite du composant implémentant *ResourceManager* de gérer la journalisation de ces enregistrements.

```
   public interface TransactionController {
2    void registerTransactional(ResourceManager rm, boolean recursive);

4    void setTransactional(boolean transactional);

6    boolean isTransactional();

8    void register(Operation op, Modifier mod, Object context);
   }
```

Listing 7.13 – Interface du contrôleur de transactions

7.2.2 Interception des invocations d'opérations de reconfiguration

Pour pouvoir enregistrer les opérations de reconfiguration avec le *TransactionController*, il est nécessaire d'intercepter le flot d'exécution au niveau de ces opérations. Le mécanisme d'interception dans Julia n'est disponible que pour les interfaces fonctionnelles et les intercepteurs standard ne sont pas utilisables sur les interfaces de contrôle. Des intercepteurs spécifiques aux contrôleurs sont donc rajoutés à la configuration standard de Julia. Ces intercepteurs jouent le rôle d'un proxy pour chaque interface de contrôle et reportent toute erreur qui pourrait se produire durant l'exécution d'une opération et conduire à l'annulation de la transaction (cf Figure 7.10).

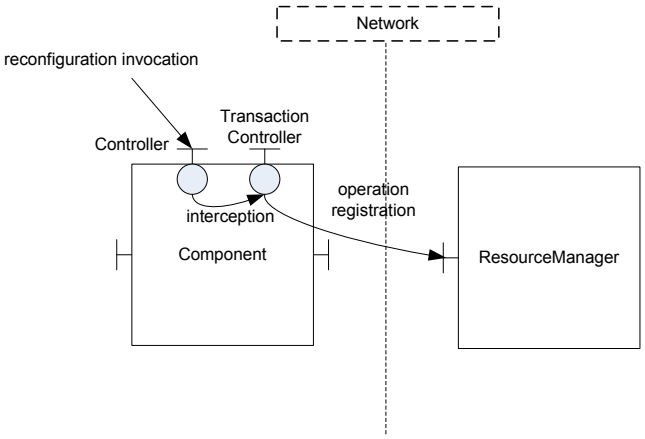

FIGURE 7.10 – Interception dans les contrôleurs Fractal

Quand une exception est levée par le code de l'opération Fractal, la reconfiguration est éventuellement annulée et l'erreur est notifiée au client de la reconfiguration. Par exemple, une opération de reconfiguration $m()$ est interceptée avant et après son exécution tandis que les exceptions sont capturées et enregistrées par le *TransactionManager* :

```
1 void m () {
    try {
3     register(m, PRE);
      impl.m();
5     register(m, POST);
    } catch(Exception e) {
7       register(m, FAILED, e);
    }
9 }
```

Listing 7.14 – Interception d'une méthode de reconfiguration

Génération statique d'intercepteurs. Les contrôleurs standard dans Julia sont implémentés par des mixins, les intercepteurs se présentent donc aussi sous la forme de mixins [DS95]. Un ensemble de mixins d'interception pour chaque contrôleur est donc fourni. Si le composant comporte de nouveaux contrôleurs de reconfiguration personnalisés, un générateur de mixin d'interception permet de générer automatiquement le code d'interception nécessaire pour rendre les opérations transactionnelles. Le générateur prend en paramètre l'interface de contrôle dont il faut générer l'intercepteur. Les mixins générés doivent alors être intégrés dans la configuration de la membrane des composants tel que le *TransactionalNameMixin* dans l'implémentation du *NameController* :

```
(name-controller-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    NameControllerImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.control.name.BasicNameControllerMixin
    org.objectweb.fractal.julia.UseComponentMixin
    org.objectweb.fractal.julia.control.name.UseNameControllerMixin
```

```
    com.francetelecom.fractal.reconfiguration.interceptors.TransactionalNameMixin
  ))
)
```

Génération dynamique d'intercepteurs. La méthode de génération précédente présente l'avantage de ne pas modifier les mécanismes de génération des mixins Julia mais est relativement peu flexible. C'est pourquoi un nouveau générateur d'interface de contrôle basé sur l'*InterfaceClassGenerator* de Julia a été conçu pour pouvoir intercepter dynamiquement les méthodes dans les interfaces de contrôle.

7.3 Extensions de l'ADL Fractal pour les reconfigurations transactionnelles

Afin de pouvoir spécifier les nouvelles préoccupations liées aux propriétés ACID des transactions pour les reconfigurations, le langage de description d'architecture Fractal ADL a été étendu avec de nouveaux modules. Par exemple, la définition de la cohérence qui passe par la spécification de contraintes d'intégrité est intégrée dans le langage ADL et la prise en charge de cette extension du langage implique une modification du compilateur. Le langage est basé sur une syntaxe XML décrit par une DTD et un module est constitué par la définition d'un élément XML. Une extension de Fractal ADL consiste donc en l'extension de la DTD (grammaire) du langage et en la fourniture des composants du compilateur (cf Figure 7.11) nécessaire à la compilation du nouveau module : un composant pour parser le nouveau module et un composant de compilation dans le « frontend »de Fractal ADL, et un composant de génération de code ou d'interprétation dans le « backend ». Tous les modules d'extension sont optionnels.

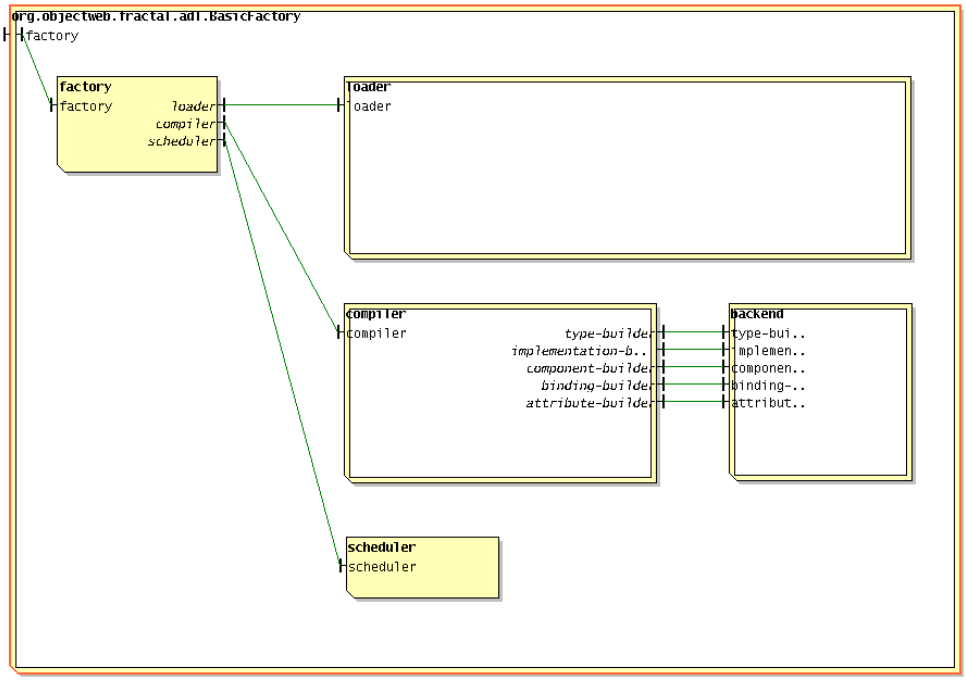

FIGURE 7.11 – Architecture du compilateur Fractal ADL

L'ensemble des nouveaux modules et des modules modifiés est intégré dans une nouvelle usine ADL dont la définition (*TransactionalFactory*) étend la définition de l'usine standard (*BasicFactory*) en intégrant les nouveaux composants. Ce nouveau type d'usine étendue doit être utilisée pour instancier des composants dont les reconfigurations doivent être réalisées de manière transactionnelle et qui utilisent les différentes extensions du langage Fractal ADL.

Par ailleurs, l'usine ADL est également modifiée pour intercepter les opérations de création de nouveaux composants de la même façon que pour l'interception des contrôleurs de reconfiguration

décrite dans la section 7.2.2. En outre, l'usine étendue enregistre automatiquement les nouveaux composants instanciés auprès des différents composants du gestionnaire de transaction tels que le composant de gestion de ressource (ResourceManager), enregistrement nécessaire pour que leurs reconfigurations soient réalisées de manière transactionnelle. L'utilisation de définitions ADL classiques non étendus est possible avec la nouvelle usine de même que l'utilisation d'une usine ADL standard est utilisable dans le système transactionnel. Cependant et dans ce dernier cas, les opérations d'instanciation ne seront pas transactionnelles et l'enregistrement des composants dans le gestionnaire de transaction devra être réalisé « manuellement »grâce aux contrôleurs des composants du système.

7.3.1 Les modules de spécification de contraintes

Les contraintes d'intégrité sont le moyen de spécifier la cohérence des configurations. Il est possible de les déclarer dans les définitions ADL des composants à travers l'utilisation de deux modules différents en fonction du type de contraintes : le module *<constraints>* pour les contraintes applicatives spécifiques à un composant donné, et le module *<style>* pour les contraintes de profil qui s'applique de manière générique à l'ensemble des composants d'une application.

Le module *<constraints>*. Les contraintes applicatives sont déclarées directement dans la définition ADL des composants à l'aide du nouveau module de contraintes, plusieurs contraintes peuvent ainsi être déclarées qui porteront sur le composant dont la définition encapsule directement les contraintes.

A titre d'illustration, nous pouvons spécifier deux invariants structurels sur un simple composant *ClientServer* (Figure 7.12) précisant qu'il doit toujours contenir au moins un composant fournissant l'interface *service* et interdisant son partage, ces contraintes seront définies dans la définition ADL du composant comme suit (lignes 14 à 20 du listing 7.15) :

```
1  <definition name="ClientServer">
     <interface name="main" role="server" signature="java.lang.Runnable" />
3    <component name="client">
       <interface name="main" role="server" signature="java.lang.Runnable" />
5      <interface name="service" role="client" signature="Service" />
       <content class="ClientImpl" />
7    </component>
     <component name="server">
9      <interface name="service" role="server" signature="Service" />
       <content class="ServerImpl" />
11   </component>
     <binding client="this.main" server="client.main" />
13   <binding client="client.service" server="server.service" />
     <constraints>
15     <!-- Contrainte : un sous-composant doit fournir l'interface "service" -->
       <constraint value="size(./child::*[./interface::service])==1" />
17     <!-- Contrainte : le composant ne doit pas être partagé -->
     <constraint value="no_sharing(.)" />
19   </constraints>
   </definition>
```

Listing 7.15 – Exemple de contraintes intégrées dans une définition ADL

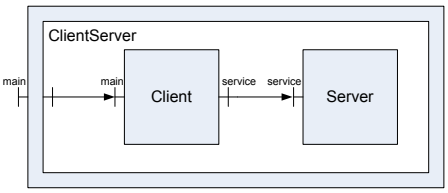

FIGURE 7.12 – Une simple application client-serveur en Fractal

Le module *<style>*. Les profils se déclarent comme les contraintes applicatives dans les définitions ADL des composants. Plusieurs profils peuvent être déclarés dans la même définition d'un

composant. Les contraintes des profils s'additionnent dans ce cas. S'il est préférable de déclarer les profils dans le composant « top-level »de l'application, des sous-composants peuvent aussi déclarer l'utilisation de profils qui viendront simplement s'ajouter aux autre profils existants. La définition d'un profil est effectué dans un fichier du type *profil.spec* qui doit regrouper l'ensemble des nouvelles contraintes qui s'appliqueront à tous les composants de l'application récursivement depuis le composant « top-level », et ce quelque soit l'endroit où est déclaré le profil. Le module se déclare de la façon suivante dans la définition d'un composant :

```
<style file="my_profile.spec" />
```

7.3.2 Les modules liés à la persistance des données

La persistance des composants passe par l'extension du module de gestion des attributs dans l'ADL, et la définition de deux nouveaux modules : <*persistence*> et <*lifecycle*>. Ces extensions permettent de préciser quels sont les composants persistants dans une configuration et quelles sont les données à persister.

Extension du module <*attribute*>. Les attributs des composants peuvent être déclarés comme persistants dans leur définition ADL grâce au module étendu de gestion des attributs. Lors de la persistance du composant, les attributs concernés sont persistés sur support physique. Par ailleurs, les types des attributs ne sont plus limités aux types primitifs présents en Java mais peuvent être également tout type (ou classe) Java. Le type apparaît désormais comme un attribut du module, cet attribut est optionnel lorsque le type est primitif et peut être inféré par le compilateur. D'autre part, la valeur des attributs ne pouvant pas toujours être spécifiée aisément dans le cas de types complexes, cette valeur n'est plus nécessairement présente dans la définition. Si l'attribut n'est pas déclaré comme persistent et aucune valeur n'est donnée, la valeur par défaut pour le type considéré lui est attribuée. Lors de l'instanciation du composant par l'usine ADL, les attributs du composants déclarés comme persistants sont initialisés avec leur dernière valeur persistée si elle existe.

```
<attributes signature="itf.MyAttributes">
  <attribute name="header" type="impl.Data" persistent="true" />
  <attribute name="count" type="int" value="0" persistent="true" />
</attributes>
```

Le module <*lifecycle*>. Ce module est utilisé pour représenter le cycle de vie qui fait partie de l'état du composant au moment de sa persistance. L'état du cycle de vie est représenté par une chaîne de caractère, les deux états de base du modèle Fractal sont gérés par le module : $STARTED$ et $STOPPED$. Le composant est positionné dans l'état du cycle de vie correspondant, au moment de son instanciation par une usine ADL. La définition du module se fait de la manière suivante :

```
<lifecycle value="STARTED" />
```

Le module <*persistence*>. Le module de persistance des données est spécifique à la persistance par sérialisation sous forme de fichier de l'état des composants. Il indique simplement le chemin d'accès vers le répertoire contenant les données sérialisées pour le composant concerné. Ce chemin d'accès peut être local mais peut également être un répertoire distant accédé via le protocole HTTP. Ce module n'est pas obligatoire pour la persistance de l'état des composants si tous les composants ont le même répertoire de persistance. Le répertoire de sérialisation par défaut est alors celui qui est spécifié comme attribut de configuration dans le composant de gestion de persistance. Le module se déclare comme tel dans la définition d'un composant :

```
<persistence dir="generated" />
```

L'implémentation du module gère deux types de fichiers, les fichiers de sérialisation de l'architecture des composants (fichiers de définitions Fractal ADL) et les fichiers de sérialisation de l'état fonctionnel des composants (fichier de sérialisation Java doté de l'extension « .save »). Le nom de ces fichiers respecte par défaut la convention de nommage suivante : le chemin du composant dans

la hiérarchie avec comme séparateur le caractère '_' suivi de l'extension du fichier. En cas de partage de composant et pour conserver un chemin unique, le chemin choisi correspond au plus petit chemin dans l'ordre lexicographique. Le nom des fichiers correspondant au composant *client* contenu dans le composant *ClientServer* sera donc *ClientServer_client.save* pour son état fonctionnel et *ClientServer_client.fractal* pour sa définition ADL.

7.3.3 Modification de modules existants

Initialisation des attributs. Le module de gestion des attributs <*attribute*> a été modifié pour simplifier la configuration des composants en surchargeant la valeur de leurs attributs définis dans les définition ADL. En effet, les fichiers des définitions ADL des composants sont dans la pratique et pour des composants développés en Java embarqués dans des archives (« jar »), or il est normalement préférable de ne pas modifier ces archives même pour des raisons de configuration. La solution « standard »pour une application réutilisant des composants avec de nouvelles valeurs d'attributs est de créer et d'utiliser de nouvelles définitions ADL qui héritent des anciennes définitions tout en redéfinissant les valeurs des attributs des composants. Cette solution est cependant peu pratique car verbeuse dans le cas ou de nombreux composants ont des attributs configurables. La solution adoptée utilise un fichier de configuration externe passé en paramètre de l'usine ADL. Ce fichier dont la syntaxe est du type « attribut - valeur »permet de surcharger simplement les valeurs par défaut des attributs définies dans les fichiers ADL sans avoir besoin de modifier ces mêmes fichiers.

La désignation des attributs doit respecter la convention de nommage suivante : chemin du composant dans la hiérarchie séparé par le caractère '.' suivi du nom de l'attribut. La désignation de l'attribut *header* du composant *server* contenu dans le composite *ClientServer* et son initialisation avec la chaîne de caractères '− >' se feront donc comme suit dans un fichier de configuration :

```
ClientServer.server.header ->
```

Le chemin du fichier est passé au moment de l'instanciation d'un composant dans le contexte de l'usine ADL à l'aide la propriété *fractaladl.attributes.config* :

```
Map context = new HashMap();
context.put("fractaladl.attributes.config", "fractal_attributes.properties");
Factory factory = FactoryFactory.getFactory(BACKEND, context);
Component component = (Component) fact.newComponent(DEFINITION, new HashMap());
```

Listing 7.16 – Code d'instanciation d'un composant avec usine ADL modifiée

Gestion des dépendances liées au cycle de vie. La version de base de Fractal ADL ne permet que de déclarer des dépendances fonctionnelles entre interfaces de composants sans spécifier d'ordre d'initialisation de ces composants pour satisfaire ces dépendances. Nous distinguons donc parmi les interfaces clientes celles qui ont besoin que le composant dont l'interface serveur lui est connectée soit démarré. Ces nouvelles dépendances liées au cycle de vie (cf. chapitre 6) se traduisent dans les définitions ADL des composants par l'ajout de l'attribut booléen et optionnel *init* dans le module <*interface*>, il n'est valable que pour les interfaces clientes. Lorsque cet attribut est positionné à *true* (sa valeur par défaut est *false*), il indique une dépendance de cycle de vie entre le composant possédant l'interface et le composant dont une des interfaces est connectée à cette interface :

```
<interface name="service" role="client" signature="itf.Service" init="yes"/>
```

Le module détecte également les cycles entre les dépendances de cycle de vie pour éviter les interblocages lors du démarrage des composants du système. En cas de cycle, une erreur est remontée lors du chargement des définitions ADL des composants.

7.4 Conclusion

L'introduction des transactions pour la gestion des reconfigurations dynamiques passe par la définition d'un certain nombre d'extensions du modèle Fractal et de son implémentation Julia, aussi bien sous la forme de nouveaux contrôleurs que d'une bibliothèque de composants pour la construction d'un gestionnaire de transactions, ou encore des extensions d'outils tels que Fractal ADL.

De nouveaux contrôleurs sont définis et ajoutés aux contrôleurs standard pour que les composants d'une application puissent supporter des reconfigurations transactionnelles. Il s'agit notamment de la prise en charge au niveau de chaque composant des contraintes et de la persistance de son état. D'autre part, pour que les opérations de reconfigurations soient bien incluses dans des transactions, il est nécessaire de pouvoir intercepter les invocations sur les contrôleurs.

Notre gestionnaire de transactions est doté d'une architecture modulaire consistant en différents composants correspondant aux différentes propriétés ACID des transactions de reconfigurations. Le composant central est le moniteur transactionnel implémentant le modèle de transactions autour duquel peuvent venir se greffer les composants optionnels de gestion de cohérence avec des contraintes, de persistance et de concurrence. Ces composants peuvent également être utilisés sans intégration du moniteur transactionnel pour bénéficier de leurs fonctionnalités en dehors d'un contexte transactionnel. Le gestionnaire de transactions peut ainsi être construit à la demande suivant les propriétés et fonctionnalités recherchées.

Finalement, un ensemble de nouveaux modules de Fractal ADL est fourni pour intégrer les préoccupations transactionnelles, que sont notamment la spécification de contraintes et la persistance, au niveau de la définition de l'architecture des composants. Le langage et le compilateur ont été étendus : une nouvelle usine ADL transactionnelle a été réalisée pour instancier des composants nécessitant le support des reconfigurations transactionnelles.

Chapitre 8

Evaluation des contributions

Sommaire

Ce chapitre a pour objectif de présenter des éléments d'évaluation et des expérimentations des mécanismes de reconfigurations transactionnelles dans le modèle de composants Fractal. Nous décrivons dans un premier temps (section 8.1) la mise en oeuvre concrète dans une application Fractal des reconfigurations transactionnelles, ainsi que leur intégration dans l'interpréteur de langage de reconfiguration FScript. Une brève analyse des performances des reconfigurations transactionnelles sur un exemple basique est fournie dans la section 8.2. Plusieurs scénarios applicatifs autour de serveurs HTTP et d'un cluster de serveurs J2EE (section 8.3) sont définis pour intégrer les reconfigurations transactionnelles dans des boucles autonomiques dans lesquels sont mis en avant les bonnes propriétés des transactions pour la fiabilisation des reconfigurations. Une dernière section (section 8.4) est dédiée à l'intégration des préoccupations de fiabilité avec les reconfigurations transactionnelles et de la sécurité grâce à une solution de contrôle d'accès sur les opérations de reconfigurations.

8.1 Mise en oeuvre des reconfigurations transactionnelles

Cette section a pour premier objectif d'expliquer la mise en oeuvre des reconfigurations transactionnelles dans un exemple d'application classique du type « Hello World ». Un second objectif est de présenter l'intégration du gestionnaire de transactions comme *backend* de l'interpréteur du langage de reconfiguration FScript [DLLC09].

8.1.1 Un « HelloWorld transactionnel »

Nous abordons ici la mise en oeuvre des reconfigurations transactionnelles dans une application simple développée avec Fractal ADL et Julia. L'application est conçue à la base sans les mécanismes transactionnels qui sont intégrés par la suite en tant qu'éléments de configuration, toute application Fractal développée en Julia peut ainsi bénéficier directement des reconfigurations transactionnelles de manière non intrusive.

Développement et administration des systèmes. Avant d'aborder la mise en oeuvre des reconfigurations transactionnelles sur un exemple, nous présentons les différents acteurs qui joue un rôle au cours du cycle de vie d'un système à base de composants Fractal. Le diagramme de contexte de la figure 8.1 recense cinq acteurs principaux qui interviennent au cours du cycle de vie du système. Le premier acteur est à l'origine de la spécification du modèle de composants Fractal utilisé pour le développement de l'application. L'architecte non-fonctionnel définit les propriétés non fonctionnelles attendues, plus particulièrement pour les reconfigurations dynamiques. Il choisit par exemple les propriétés transactionnelles des reconfigurations (concurrence et persistance) et définit la cohérence de l'architecture à haut niveau par l'intermédiaire d'un profil de contraintes. L'architecte fonctionnel spécifie essentiellement l'assemblage des composants sous forme par exemple de définitions Fractal ADL, il intègre des contraintes applicatives au niveau de l'architecture des composants. Le développeur fonctionnel intervient pour implémenter le code métier des composants en faisant abstraction de la préoccupation de reconfiguration du système. Enfin, le dernier acteur est l'administrateur qui supervise le système pendant son exécution et est notamment amené à le reconfigurer dynamiquement pour le faire évoluer.

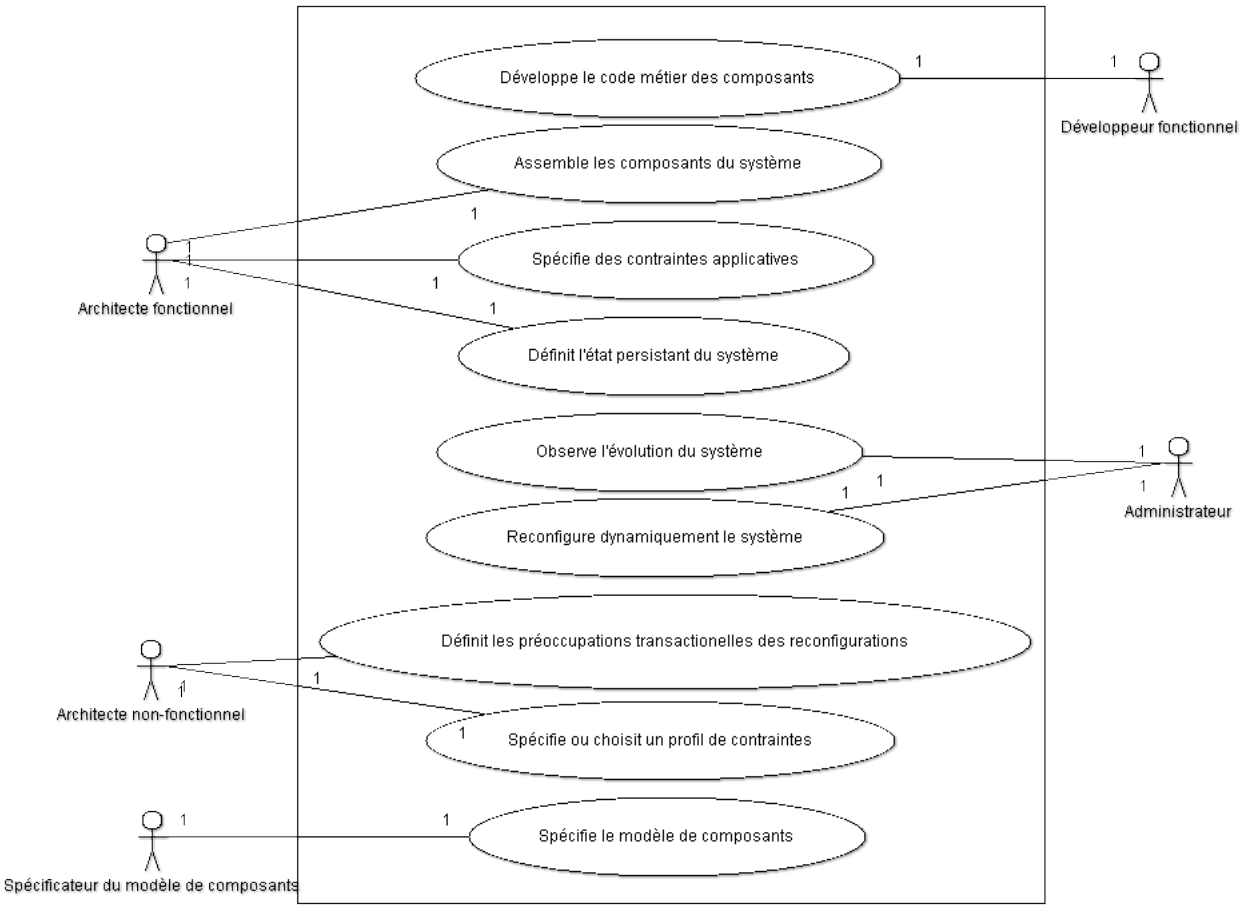

FIGURE 8.1 – Diagramme de contexte pour le développement et l'administration d'une application Fractal

Description de l'application. L'application choisie pour illustrer l'utilisation des reconfigurations transactionnelles est une simple application du type *HelloWorld*. Le composant *HelloWorld* (cf. Figure 8.2) est un composant composite contenant deux sous-composants primitifs : un composant *Client* et un composant *Server*. Les composants *HelloWorld* et *Client* offrent tous les deux une interface *main* du type *java.lang.Runnable*. Le composant *Server* offre une interface *printer* du type *Printer* (cf. listing 8.1) qui permet d'imprimer des messages sur la sortie standard. Il possède un attribut *header* qui est une chaîne de caractères s'affichant en préfixe des messages imprimés. La description de l'architecture du composant *HelloWorld* est donnée par sa définition en fractal ADL (cf. listing 8.2). L'invocation de la méthode *run()* de l'interface fonctionnelle *main* du composant *HelloWorld* affiche le message « -Hello World ! »(la valeur de l'attribut *header* du composant *Server* est fixée à « - »par défaut dans la configuration).

```
public interface Printer {
  void print(String msg);
}
```

Listing 8.1 – Interface serveur du composant Server

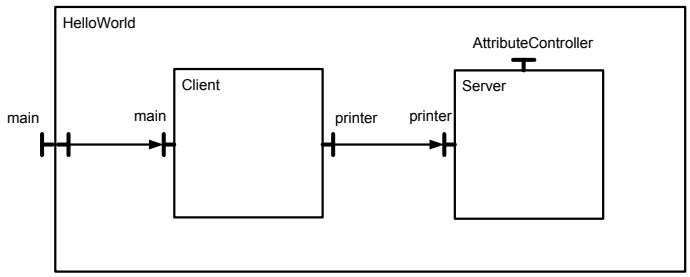

FIGURE 8.2 – Architecture Fractal du composant HelloWorld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath://
    org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="HelloWorld">
 <interface name="main" role="server" signature="java.lang.Runnable" />
 <component name="Client">
  <interface name="main" role="server" signature="java.lang.Runnable" />
  <interface name="printer" role="client" signature="Printer" />
  <content class="ClientImpl" />
 </component>
 <component name="Server">
  <interface name="printer" role="server" signature="Printer" />
  <content class="ServerImpl" />
  <attributes signature="PrinterAttributes">
   <attribute name="header" value="-" />
  </attributes>
 </component>
 <binding client="this.main" server="Client.main" />
 <binding client="Client.printer" server="Server.printer" />
</definition>
```

Listing 8.2 – Définition Fractal ADL du composant HelloWorld

Mise en oeuvre des reconfigurations transactionnelles. Nous souhaitons rendre les reconfigurations transactionnelles dans l'application *HelloWorld* déployée de manière centralisée. Tous les composants du gestionnaire de transactions sont utilisés dans leur configuration par défaut et toutes les fonctionnalités des transactions (recouvrement, contraintes, concurrence et persistance) sont donc activées.

Une usine ADL transactionnelle est créée pour pourvoir instancier les composants, le chemin d'un fichier de configuration peut être passé dans le contexte de l'usine pour surcharger la valeur des attributs des composants instanciés :

```
Map context = new HashMap();
Component fact = FractalTx.newTxFactory(context);
```

Cette usine transactionnelle est configurée par défaut avec le fichier de configuration (« julia_tx.cfg ») qui ajoute l'ensemble des nouveaux contrôleurs pour les transactions décrits dans le chapitre 7 (section 7.2).

Un composant *TransactionManager* est créé pour gérer les reconfigurations transactionnelles :

```
Component tm = FractalTx.newTransactionManager();
```

L'usine ADL est enregistrée auprès du *TransactionManager* pour que les opérations d'instanciation de composants soient transactionnelles et que les composants nouvellement instanciés bénéficient de reconfigurations transactionnelles :

```
FractalTx.setTransactional(fact, tm);
```

Enfin, le nouveau composant est instancié à partir de l'usine ADL, l'opération d'instanciation si elle n'est pas démarquée sera automatiquement et par défaut incluse dans une transaction comme c'est le cas dans le code suivant :

```
Component helloworld = (Component) fact.newComponent(
  "HelloWorld", null);
```

Une fois le composant créé, il est possible d'effectuer des reconfigurations transactionnelles complexes du composant *HelloWorld* en récupérant une référence sur l'interface *UserTransation* du composant *TransactionManager* et en démarquant explicitement les transactions (cf. listing 8.3).

```
UserTransaction tx = FractalTx.getUserTransaction(tm);
try {
  // Début de la reconfiguration transactionnelle
  tx.begin();
  // Invocations d'opérations de reconfiguration
  Component client = Fractal.getContentController(helloworld)
    .getFcSubComponents()[0];
  Component server = Fractal.getContentController(helloworld)
    .getFcSubComponents()[1];
  ContentController csCC = Fractal.getContentController(helloworld);
  BindingController clientBC = Fractal.getBindingController(client);
  clientBC.unbindFc("printer");
  csCC.removeFcSubComponent(server);
  csCC.addFcSubComponent(server);
  // Validation de la transaction
  tx.commit();
} catch (Throwable th) {
  try {
    // Déclenchement du rollback en cas d'erreur
    tx.rollback();
  } catch (SystemException se) {
    se.printStackTrace();
  }
}
```

Listing 8.3 – Exemple de reconfiguration d'un composant HelloWorld

Par défaut, la transaction n'est pas automatiquement annulée en cas d'échec de la reconfiguration, le « rollback » est donc ici invoqué explicitement. Ce comportement est cependant paramétrable comme décrit dans le chapitre 7 (section 7.1), il est en effet possible de configurer un « rollback » automatique pour certaines exceptions levées lors de la reconfiguration.

Pour intégrer les préoccupations de persistance et de vérification de contraintes dans l'ADL, la modification préalable à effectuer dans l'ADL est le changement de la DTD standard pour la DTD modifiée prenant en charge les extensions du langage :

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://com/francetelecom/fractal/reconfiguration/tx.dtd">
```

Il est alors possible d'intégrer des contraintes d'intégrité dans les définitions ADL et de spécifier quels sont les attributs persistants du composant défini comme expliqué dans le chapitre 7 (section 7.3 sur les extensions de l'ADL Fractal).

8.1.2 Intégration dans une chaîne de validation FScript

Dans Fractal, les reconfigurations peuvent être exprimées soit directement à l'aide de l'API de reconfiguration dans le langage d'implémentation du modèle, soit avec le langage dédié de reconfigurations FScript [DLLC09]. Une reconfiguration dans Julia est ainsi exprimée en Java. Une des différences principales entre les deux modes de spécification, outre la concision de FScript par rapport à un langage de programmation généraliste comme Java, est le fait que FScript est volontairement restreint à la manipulation des éléments du modèle de composants sans autres effets de bord.

L'objectif de nos travaux est ainsi de pouvoir intégrer les reconfigurations transactionnelles dans l'interpréteur FScript et ce de manière transparente pour l'utilisateur du langage FScript : aucune nouvelle primitive au niveau du langage n'est ajoutée pour la gestion des transactions, seule l'architecture de l'interpréteur est modifié. D'autre part, la gestion des erreurs de reconfiguration est rendu transparente pour le développeur FScript, le code de gestion d'erreur n'apparaît pas les scripts de reconfiguration, le recouvrement d'erreur étant automatique.

Modification de l'architecture de FScript. L'interpréteur FScript (dans sa version 2) est implémenté sous forme de composants Fractal. Son architecture est présentée dans la figure 8.3. Le composant est constitué principalement d'un *frontend* pour le chargement des scripts et d'un *backend* contenant l'interpréteur proprement dit. Une usine ADL standard (composant adl-factory) est également intégré dans l'architecture pour l'instanciation de composants.

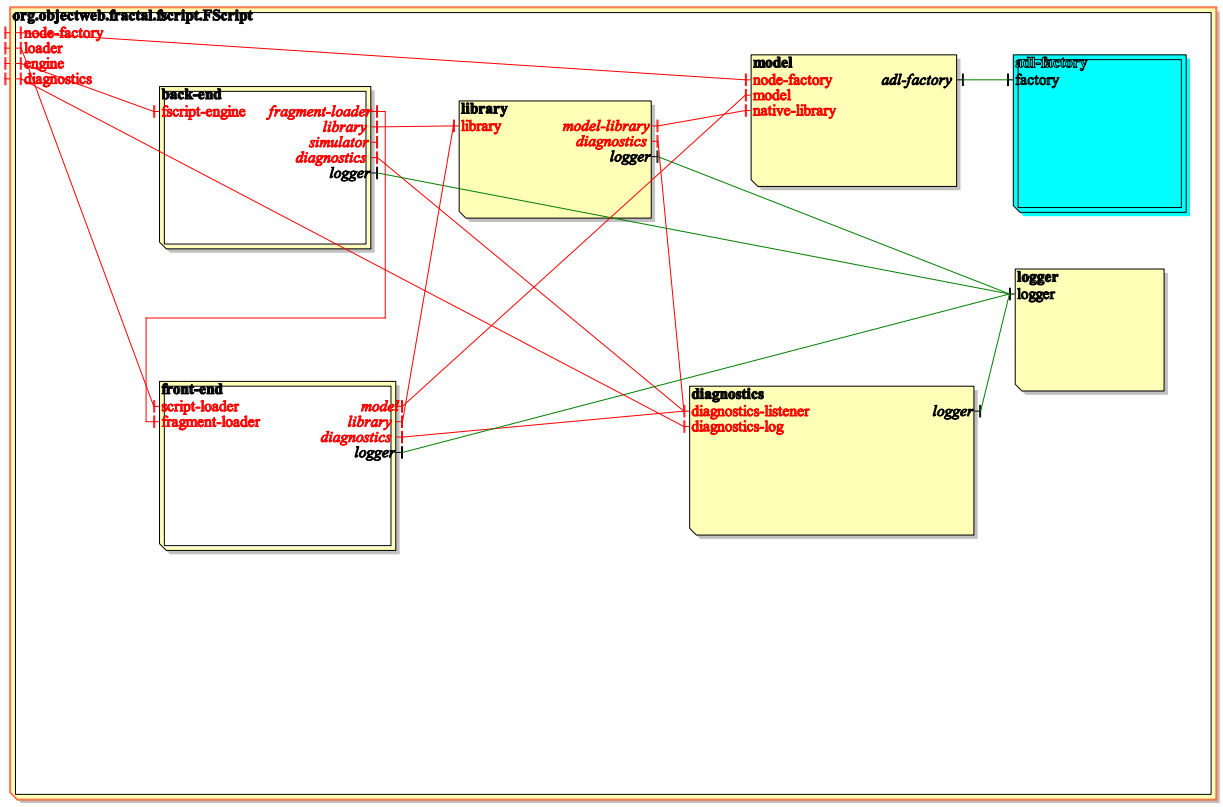

FIGURE 8.3 – Architecture Fractal de FScript v2

Pour réaliser des reconfigurations transactionnelles avec FScript, il est nécessaire de connecter notre « backend » transactionnel à l'architecture de l'interpréteur. Une première modification consiste en le remplacement de l'usine ADL standard (en bleu sur la figure 8.3) par notre usine ADL transactionnelle de façon à ce que les composants instanciés par l'interpréteur le soit de manière transactionnelle et que leurs éventuelles futures reconfigurations le soient également. D'autre part, il est nécessaire d'ajouter et de connecter notre gestionnaire de transaction dans le *backend* de FScript. Cette connexion se fait au niveau du composant *driver* qui possède une interface optionnelle dont la signature correspond à la norme JTA (interface *TransactionManager*).

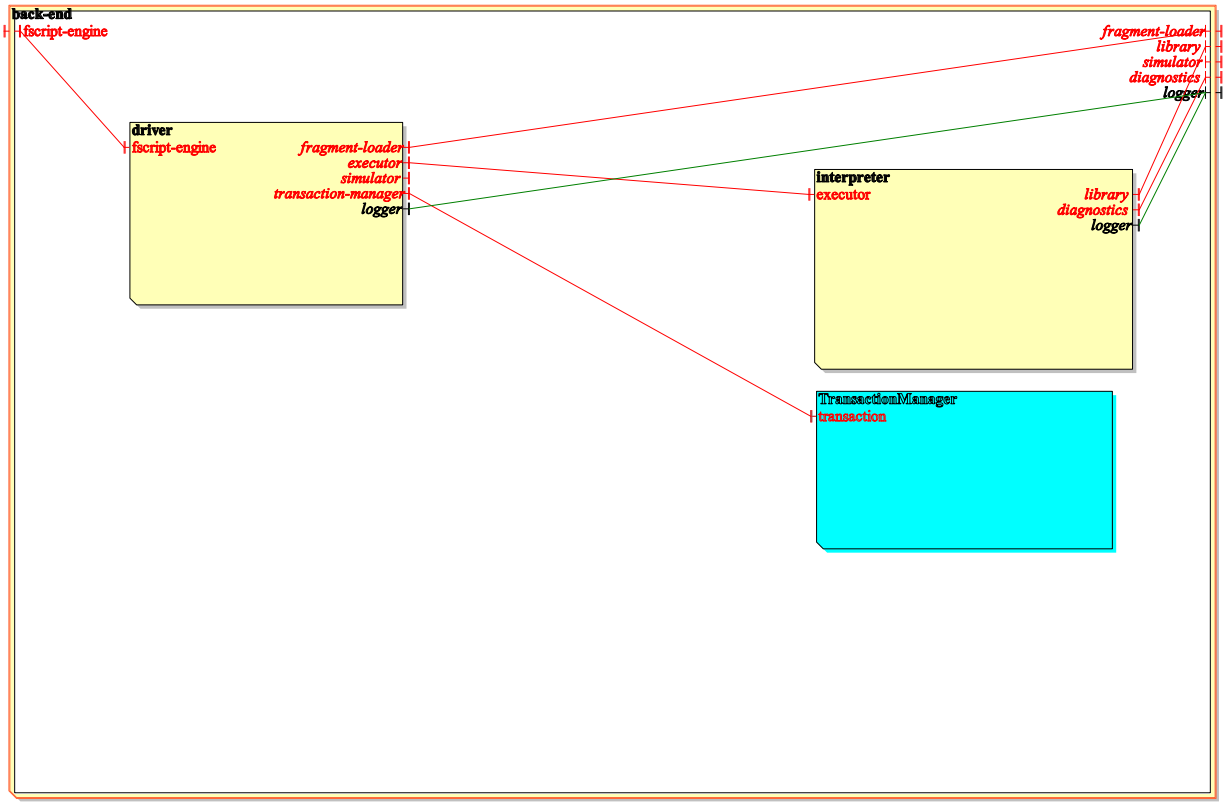

FIGURE 8.4 – Intégration du gestionnaire de transactions dans le backend de FScript

Démarcation des transactions. La démarcation des transactions dans FScript est automatique et transparente contrairement à l'utilisation de l'API en Java où elle elle doit être explicite. Une action FScript est ainsi automatiquement exécutée dans une transaction. FScript permet l'imbrication d'actions de reconfiguration, les actions dans notre modèle de transactions sont linéarisées pour correspondre au modèle de transactions plat : c'est l'action de plus haut niveau qui délimite la transaction. Il faut cependant noter qu'un modèle de transactions plus complexe avec transactions emboîtées [Pro02] pourrait cependant être envisagé pour refléter l'imbrication des actions en FScript.

Exemples de reconfigurations transactionnelles avec FScript. L'interpréteur FScript détecte et rejette les script qui comportent des erreurs de syntaxes mais il est possible de définir des reconfigurations invalides qui violent la cohérence du système reconfiguré. Nous définissons par exemple deux actions de reconfigurations avec FScript pour illustrer l'utilisation des reconfigurations transactionnelles et plus spécifiquement la transparence du recouvrement en cas de faute logicielle. Chaque action est exécutée dans une transaction grâce au gestionnaire de transactions connecté à l'interpréteur.

Le premier exemple d'action prend en paramètres deux composants et les ajoute réciproquement l'un dans l'autre :

```
action strictly-invalid(c1, c2) {
  add($c1, $c2);
  add($c2, $c1);
}
```

Le modèle Fractal interdit les cycles dans la hiérarchie des composants donc cette action de reconfiguration est toujours annulée par le gestionnaire de transactions quelque soit les composants $c1$ et $c2$ passés en paramètre. En effet, l'exécution de l'action est invalidée pour cause de violation de contrainte d'intégrité au niveau du modèle.

Le second exemple d'action qui prend également deux composants en paramètres est le suivant :

```
action conditionally-valid(c1, c2) {
  add($c1, $c2);
```

```
  bind($c1/internal-interface::foo,
4       $c2/interface::bar);
}
```

Cette action n'est valide et donc autorisée par le gestionnaire de transactions que si les composants $c1$ et $c2$ respectent certaines conditions :

- $c2$ doit être un composant composite,
- $c1$ doit avoir une interface foo, $c2$ une interface bar,
- les types des interfaces foo et bar doivent être compatibles.

L'annulation des reconfigurations est donc automatique avec le backend transactionnel de FScript et la préoccupation transactionnelle est introduite sans modification au niveau du langage mais requiert juste une modification architecturale de l'interpréteur.

8.2 Evaluation de performances

La problématique des performances des reconfigurations transactionnelles, même si elle n'est pas absente de nos préoccupations ne constitue cependant pas une priorité absolue. La fiabilité est en effet une propriété plutôt coûteuse en performances et il s'agit alors de choisir un compromis acceptable entre fiabilité et performance.

8.2.1 Empreinte mémoire

Nous ne nous intéressons en terme d'empreinte mémoire qu'à la taille des bibliothèques Java nécessaire pour l'exécution d'une application avec reconfigurations transactionnelles. La taille de l'archive du gestionnaire de transactions est donnée dans le tableau 8.1. Cette évaluation ne tient pas compte des dépendances aux différentes bibliothèques Fractal nécessaires à l'instanciation et au fonctionnement du gestionnaire de transactions qui est développé avec des composants spécifiés dans des définitions en Fractal ADL. Il est cependant possible de se passer de ces dépendances en générant statiquement le code d'assemblage des composants avec l'usine Fractal ADL et en précompilant les membranes des composants avec Julia.

Module	Taille Mémoire (Ko)
Moniteur transactionnel + recouvrement logiciel	142,1
Gestionnaire de transactions complet	249,6

TABLE 8.1 – Empreinte mémoire du gestionnaire de transactions

Il faut noter que la modularité de l'architecture du gestionnaire de transactions se traduit par une modularité au niveau de l'organisation du code. La solution transactionnelle minimale embarque ainsi simplement un moniteur transactionnel et le recouvrement de défaillances logicielles, elle ne contient pas la gestion de la persistance, de la concurrence et des contraintes. L'empreinte mémoire peut apparaître comme relativement élevée au regard de l'empreinte mémoire du seul runtime Julia (40 Ko pour la v2.5.1) et même de Fractal ADL (145 Ko pour la v2.1.7) mais il faut noter que nous ne ciblons pas particulièrement les applications embarquées et que le code n'est pas optimisé pour ce type de plateformes.

8.2.2 Temps d'exécution

L'objectif des tests est de constater le surcoût en temps d'exécution des mécanismes transactionnels dans le cas où la reconfiguration dynamique est réalisée avec succès. Dans cette hypothèse optimiste, les mécanismes transactionnels n'interviennent pas pour le recouvrement des transactions. Nous comparons donc pour chaque test le temps d'exécution de la reconfiguration dynamique d'une application basique avec ou sans transaction. Nous faisons le rapport entre les deux temps d'exécution, le temps d'exécution sans transaction servant de témoin de comparaison.

L'application utilisée pour tous les tests est le composant *HelloWorld* décrit dans la section 8.1.1. Les scénarios suivants ont été définis pour tester l'impact des mécanismes transactionnels sur le temps d'exécution des reconfigurations dynamiques :

1. *Instanciation.* Une seule instanciation du composant *HelloWorld* à partir de sa définition ADL.
2. *Reconfiguration Java locale.* 100 reconfigurations consécutives et locales du composant *HelloWorld* déjà instancié consistant en la déconnexion du composant *Server* du composant *Client*, le retrait du *Server* du composant *HelloWorld*, puis son ajout et enfin sa reconnection (le code Java de la reconfiguration est donné dans le listing 8.3).
3. *Reconfiguration Java distribuée.* 10 reconfigurations réparties identiques au scénario 2 sauf que le composant *Server* est situé dans une autre JVM.
4. *Reconfiguration FScript.* 100 reconfigurations identiques à celles du scénario 2 mais en utilisant FScript au lieu de l'API Java. Le code FScript du script correspondant est le suivant :

```
action reconf(cs) {
  server = $cs/child::server;
  client = $cs/child::client;
  unbind($client/interface::printer);
  remove($cs, $server);
  add($cs, $server);
  bind($client/interface::printer, $server/interface::printer);
}
```

Dans un premier temps, le gestionnaire de transactions est testé sous sa forme minimale pour isoler l'impact en performances des fonctionnalités des transactions. Par conséquent, la durabilité (persistance de l'état des composants), le contrôle de concurrence (verrouillage) et la vérification de la cohérence (vérifications des contraintes d'intégrité) sont désactivées. L'application ne comporte donc aucune contrainte ni attribut persistant. Seule la journalisation mémoire est active pour l'annulation des transactions en cas d'abandon. Le mode de mise à jour choisi pour tous les tests est la mise à jour immédiate (cf. chapitre 6, section 6.1) avec effet direct des reconfigurations dans le système.

L'ensemble des tests de performance est réalisé sur une même machine. Les tests de reconfiguration distribuée utilisent plusieurs JVM instanciée sur la même machine. L'environnement matériel de test est le suivant :

- Processeur : Pentium M 1,86 GHz
- Mémoire : 1 Go
- Système d'exploitation : Microsoft Windows XP SP2
- JVM : JRE version 1.6.0_11

Les résultats des tests sont résumés dans le tableau 8.2. Nous constatons pour tous les scénarios un surcoût plus ou moins important des reconfigurations transactionnelles par rapport aux reconfigurations sans transactions. Le surcoût relativement important en temps d'exécution pour une reconfiguration simple (scénario 2) est principalement dû aux interceptions de chaque opération de reconfiguration par le contrôleur de transactions et la mise en oeuvre des transactions par le moniteur transactionnel (journalisation mémoire). Nous remarquons que ce surcoût est plus faible dans les cas de l'opération d'instanciation (scénario 1), des reconfigurations distribuées (scénario 3) et avec FScript (scénario 4). Cette atténuation est due au temps plus long de l'exécution des opérations de reconfiguration dans ces cas de figure alors que le surcoût de l'utilisation des transactions reste sensiblement le même que pour le scénario 2 et est donc proportionnellement plus faible.

Plusieurs hypothèses sont à considérer pour relativiser ce surcoût en performance des reconfigurations transactionnelles :

- Le taux de reconfiguration dans le système : les reconfigurations peuvent intervenir plus ou moins fréquemment dans le système. Dans nos exemples, les invocations d'opérations de reconfigurations dynamiques sont moins nombreuses dans le système que celle d'opérations fonctionnelles, leur impact en terme de performance globale est donc plus limité.
- Le taux de disponibilité : si une reconfiguration échoue, le système devient indisponible et seuls les mécanismes de tolérance aux fautes apportés par les transactions permettent de le remettre dans un état disponible.

Cependant, il faut noter que pour certains systèmes le besoin en réactivité peut primer sur le besoin en fiabilité apporté par les transactions. La vitesse d'adaptation du système et donc la vitesse d'exécution des reconfigurations est alors déterminante pour réagir à un changement rapide de contexte d'exécution.

Scénarios de test	Sans transaction (1)	Avec transactions (2)	Rapport (2)/(1)
1.Instanciation	219	344	1,57
2.Reconf. Java locale	37	112	3,03
3.Reconf. Java distribuée	1297	1985	1,53
4.Reconf. FScript	157	250	1,59

TABLE 8.2 – Comparaison des temps d'exécution (en ms) avec et sans reconfigurations transactionnelles

Ajout de fonctionnalités. Nous souhaitons tester les différentes fonctionnalités des reconfigurations transactionnelles et nous reprenons le scénario *2* précédant de reconfiguration locale programmée en Java. Ce scénario est choisi car il est le plus simple et pour ne pas introduire de surcoûts liés à d'autre facteurs tels que la distribution ou l'utilisation de FScript. Nous définissons quatre nouveaux scénarios en activant les différentes fonctionnalités du gestionnaire de transactions séparément :

1. Ajout de contraintes : nous ajoutons la vérification des contraintes d'intégrité ainsi qu'une contrainte applicative simple sur le composant *HelloWorld* pour restreindre le nombre de ses sous-composants à deux :

   ```
   size(./child::*)==2
   ```

2. Activation de la gestion de concurrence : les opérations de reconfigurations verrouillent les éléments reconfigurés suivant une stratégie pessimiste.
3. Activation de la persistance : l'état des composant est persisté suite à la validation des reconfigurations transactionnelles pour implémenter la propriété de durabilité. Aucun attribut n'est déclaré persistant dans la définition du composant *HelloWorld*, donc seule l'architecture est persistée. Le composant ne possède de toute façon qu'un attribut *header* de type primitif (cf. listing) dont la valeur apparaît dans les définitions ADL persistées.

Les résultats des tests sont présentés dans le tableau 8.3. Nous donnons toujours le rapport à titre de comparaison avec une exécution des reconfigurations sans transactions, le temps de référence de l'exécution sans transaction pour le calcul du rapport est celui donnée dans le tableau 8.2 pour le scénario 2 (37 ms).

Scénarios de test	Avec transactions	Rapport (avec tx)/(sans tx)
1.Contraintes	161	4,35
2.Concurrence	177	4,78
3.Persistance	2715	73,38

TABLE 8.3 – Comparaison des temps d'exécution (en ms) des reconfigurations transactionnelles suivant les fonctionnalités

Les résultats des performances avec la vérification d'une contraintes simple sans violation sont données dans la première ligne du tableau 8.3. Ce test est réalisé avec une contrainte particulière n'est pas forcément représentatif du surcoût de la vérification de contraintes en général, la complexité et le nombre de contraintes pouvant être très variable. Le test donne tout de même un aperçu du surcoût non négligeable de l'ajout de cette fonctionnalité dans le système transactionnel.

Le coût du contrôle de concurrence (ligne 2 du tableau) est du même ordre que celui de la vérification de contrainte sur cet exemple. La méthode de gestion de concurrence est en effet une méthode pessimiste par verrouillage qui ajoute donc le coût de la pose des verrous requis pour chaque opération de reconfiguration primitive et de la vérification de l'absence d'interblocage.

L'activation de la persistance de l'état des composants pour la durabilité des transactions est très couteuse en temps d'exécution (ligne 3 du tableau). En effet, par défaut l'état des composants

est persisté à chaque validation de transaction. Ce paramètre est cependant réglable en fixant un nombre de transactions supérieur à un avant chaque point de contrôle. Il faut donc relativiser ce résultat car le test mesure le pire cas en terme de temps d'exécution, le meilleur cas serait de réaliser la persistance de l'état qu'après l'exécution des 100 reconfigurations du scénario. Le pire cas est tout de même particulièrement intéressant car il permet de conserver une synchronisation parfaite entre l'état mémoire et l'état persisté sur disque, ce qui permet en cas de défaillance de site de ne pas avoir à exécuter le protocole de recouvrement car l'état sur disque est bien le dernier état cohérent du système issu de la dernière transaction validée.

8.2.3 Comparaison des modes de mise à jour

Nous comparons maintenant les différences en temps d'exécution entre les deux modes de mise à jour étudiés et implémentés pour les transactions (cf. chapitre 6, section 6.1) : la mise à jour immédiate et la mise à jour différée. Le scénario testé est toujours le même scénario consistant en 100 reconfigurations Java locales et le gestionnaire de transaction est utilisé dans sa configuration minimale (sans contrainte, concurrence et persistance). La comparaison est faite à la fois dans le cas de la validation des transactions et dans le cas de l'abandon. Pour forcer l'abandon des transactions, nous générons volontairement une exception dans le code de la reconfiguration avant le « commit » de la transaction (cf. listing 8.3), la transaction est alors annulée. Les résultats des tests apparaissent dans le tableau 8.4. Nous conservons à titre de témoin l'exécution du scénario sans transaction dans le cas des transactions validées. Cependant, le scénario n'est pas exécutable sans transaction dans le cas de l'abandon car la transaction ne peut alors être annulée, ce test impossible est représenté par un « X » dans le tableau. Nous donnons également dans le tableau le rapport des temps exécutions entre les deux mises à jour avec validation et avec abandon de transaction.

Reconfiguration	Sans transaction (1)	Immédiate (2)	Différée (3)	Rapport (3)/(2)
1.Avec validation	45	113	135	1,19
2.Avec abandon	X	173	143	0,83

TABLE 8.4 – Comparaison des temps d'exécution (en ms) pour deux mode de mises à jour différents

Les résultats montrent que l'exécution d'un transaction valide est plus efficace pour la mise à jour immédiate que la mise à jour différée. Ceci est du au coût de l'opération de simulation mise en oeuvre avant l'exécution de la reconfiguration dans le système pour la mise à jour différée contrairement à la mise à jour immédiate où la reconfiguration est directement exécutée dans le système. Le résultat est inversé pour le cas de l'abandon des transactions. En effet, l'annulation de la transaction suppose de défaire les opérations exécutées directement dans le système pour la mise à jour immédiate. Par contre, pour la mise à jour différée, les modifications du système n'apparaissent que lors de la validation de la transaction, l'annulation suppose donc seulement l'arrêt de la simulation. Un critère important en terme de performances du choix du mode de mise à jour est ainsi le taux estimé d'abandon ou de validation des transactions.

8.3 Reconfigurations dynamiques fiables pour l'informatique autonomique

L'informatique autonomique (en anglais « autonomic computing ») [KC03] vise à réaliser des systèmes capables de s'auto-administrer. Cette section présente la mise en oeuvre de plusieurs scénarios autonomiques qui illustrent les mécanismes de reconfigurations transactionnelles à l'intérieur de boucles de contrôle. Ces scénarios ont été expérimenté dans un premier temps sur de simples serveurs Web. Un scénario a ensuite été mis en oeuvre dans le contexte d'un cluster de serveurs d'application J2EE. L'objectif est d'effectuer une évaluation qualitative des reconfigurations transactionnelles en montrant l'utilisation des contraintes d'intégrité, de la reprise à chaud et de la reprise à froid des reconfigurations transactionnelles à partir d'expérimentations applicatives.

8.3.1 Une boucle autonomique générique

L'ensemble des scénarios mis en oeuvre utilise une boucle de contrôle autonomique pour différentes préoccupations : auto-protection, auto-optimisation et auto-réparation. Dans la boucle de contrôle présentée dans la figure 8.5, nous distinguons les fonctions de monitoring, d'analyse et décision et enfin d'action. Cette boucle générique est déclinée dans les différents scénarios implémentés. L'action sur le système dans le retour de la boucle est réalisée par des reconfigurations dynamiques constituées de plusieurs opérations qui agissent sur les composants du système en cours d'exécution. Les reconfigurations transactionnelles rendent le système et la boucle de contrôle tolérants aux fautes aussi bien logicielles comme la violation de contraintes d'intégrité que matérielles comme les pannes franches. Les propriétés d'atomicité et de durabilité des transactions permettent de revenir dans un état correct du système avant la faute.

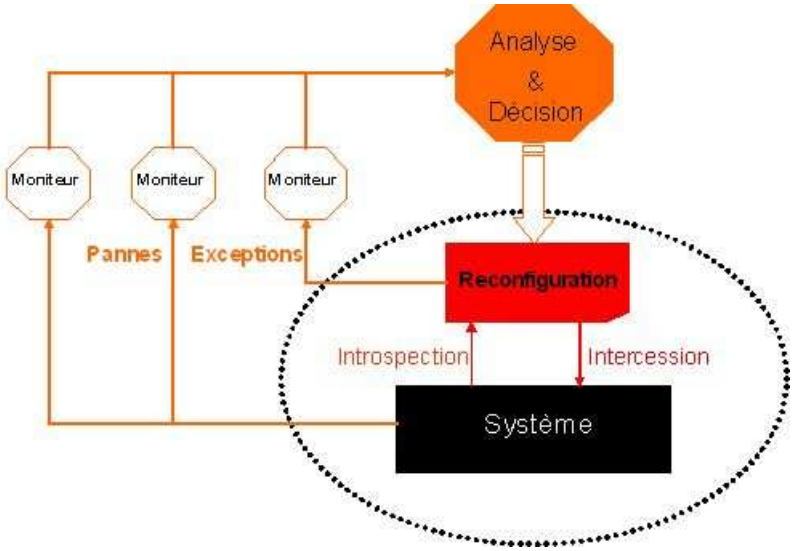

FIGURE 8.5 – Boucle d'auto-réparation transactionnelle

Monitoring. Nous distinguons différents types de moniteurs qui détectent les fautes qui surviennent dans le système :

- Détecteur de panne franche : les pannes franches sont détectées grâce à un moniteur qui peut reposer soit sur le principe du « heart beat », du « ping » ou du timeout lors d'invocations d'opérations.
- Détecteur de contraintes architecturales : le moniteur transactionnel détecte les violations de contraintes architecturales qui peuvent survenir pendant les reconfigurations.
- Détecteur de fautes fonctionnelles : la détection des fautes fonctionnelles dépend très fortement de l'application considérée. Il faut donc spécifier les valeurs sur lesquelles se fait l'observation et aussi le moment où on peut considérer qu'une faute est détectée.

Analyse et décision. L'analyse et la décision se font au niveau d'un gestionnaire autonomique qui interprète les types de fautes récupérées par les moniteurs. Suivant la faute, un plan de réparation va être choisi puis donné à l'exécution.

- Dans le cas des pannes franches, la réparation consiste à réparer l'application par une réinstanciation soit sur la même machine soit sur une nouvelle machine.
- Dans le cas de violation de contraintes architecturales, le gestionnaire peut soit choisir d'élaborer son propre plan d'action, soit laisser le gestionnaire de transactions annuler la reconfiguration pour revenir dans un état cohérent.
- Dans le cas de fautes fonctionnelles, le plan d'action est spécifique aux éléments constituants le système.

Action. Un plan de réparation est une reconfiguration exécutée dans le système. La reconfiguration est exprimée en langage FScript et est exécuté par interpréteur associé au gestionnaire de transactions pour les reconfigurations.

8.3.2 Auto-protection mémoire d'un serveur Web

Objectifs du scénario. Ce scénario consiste en l'auto-protection d'une application contre une surcharge mémoire de la JVM qui l'héberge. Il ne fait pas intervenir l'ensemble des mécanismes transactionnels des reconfigurations (recouvrement de fautes) mais seulement la vérification de contraintes d'intégrité, c'est à dire la propriété de cohérence des transactions. L'objectif est simplement de montrer l'utilisation d'une contrainte applicative et de sa vérification à l'extérieur des reconfigurations et sans l'usage des transactions. Nous réutilisons pour cela le composant *ConsistencyManager* séparément du gestionnaire de transactions, nous prouvons ainsi la réutilisabilité et la modularité des composants de notre solution. Le composant *ConsistencyManager* décrit dans le chapitre 7 (section 7.1) permet de vérifier des contraintes d'intégrité dans les architectures des composants pendant leur exécution. Il est utilisé dans le cadre des reconfigurations transactionnelles pour valider la cohérence du système à l'issue de reconfigurations.

Description de l'application. Ce scénario autonomique repose sur l'utilisation d'une application simple développé en Fractal : le serveur *Comanche*. *Comanche* est un serveur HTTP basique qui fait partie de la distribution logicielle Fractal et qui est implémenté sous forme de plusieurs composants Fractal en Julia. Il fournit au client le fichier correspondant à l'URL qu'il a demandée. L'architecture d'un composant *Comanche* est composée de sept composants primitifs organisés dans des composants composites (cf Figure 8.6) :

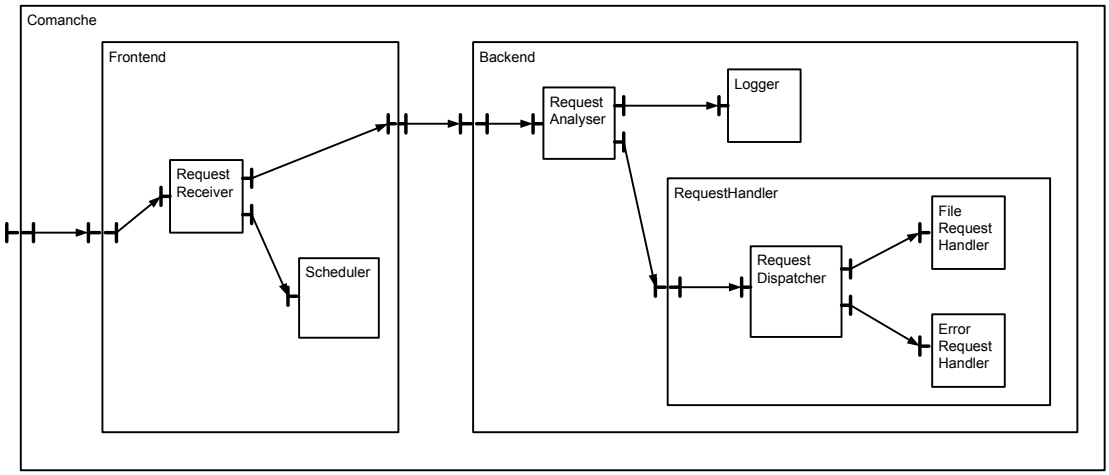

FIGURE 8.6 – Architecture du serveur web Comanche

Le *FrontEnd* reçoit les requêtes des clients, les ordonnance puis les envoie au *Backend* pour les traiter. Il comprend les composants :

- *RequestReceiver* : qui prend en paramètre la requête des clients.
- *Scheduler* : prend en paramètre une tâche à ordonnancer dans un seul thread ou en multi-thread.

Le *BackEnd* analyse la requête, la journalise et exécute la requête. Il comprend les composants :

- *RequestAnalyser* : lit le flot d'entrée pour obtenir l'URL.
- *Logger* : journalise le résultat des requêtes
- *RequestHandler* est lui même un composant composite contenant le *RequestDispatcher*, le *FileHandler* qui essaie de trouver et de renvoyer le fichier qui correspond à l'URL au client et le *ErrorHandler* qui renvoie au client un message d'erreur si le fichier n'est pas trouvé.

L'application du scénario est le serveur *Comanche* dans lequel nous introduisons un composant *Cache* pour accélérer le temps de traitement des réponses. Le cache garde en mémoire les requêtes et les réponses associées afin de les retourner en cas de requêtes identiques. L'emplacement du composant *Cache* se situe entre les composants *RequestDispatcher* et *FileHandler* comme le montre la figure

8.7 : si le cache ne contient pas la réponse à la requête, il transmet la demande au *FileHandler* pour trouver le fichier correspondant et le met en mémoire au passage, sinon il transmet le fichier en mémoire directement au client.

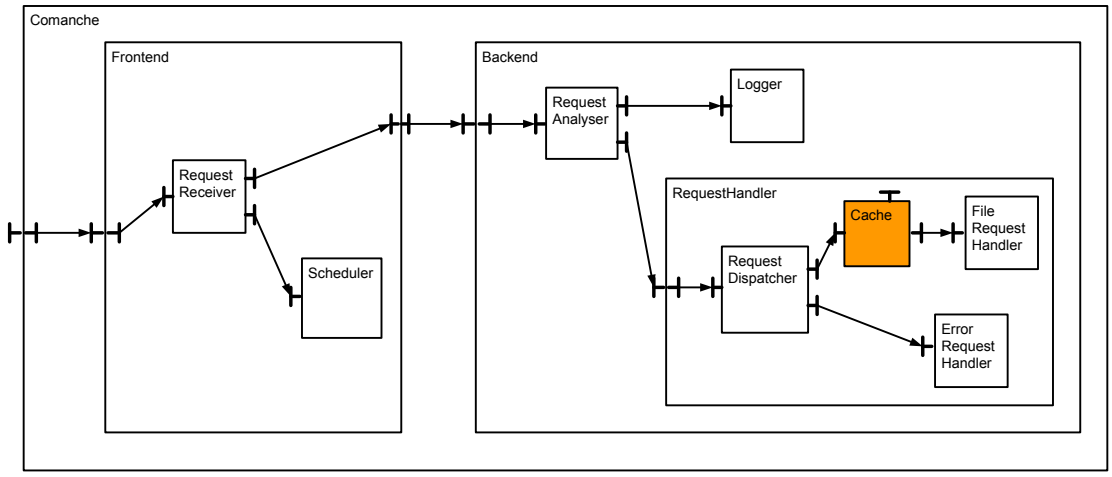

FIGURE 8.7 – Ajout d'un cache dans l'architecture de Comanche

Le composant *Cache* ajouté possède les attributs *CacheSize* pour pouvoir obtenir à tout moment la quantité de mémoire utilisée et *MaxCacheSize* pour pouvoir fixer la taille de mémoire maximale utilisable pour stocker les requêtes. Le cache effectue alors un remplacement de type FIFO des fichiers en mémoire pour que cette valeur ne soit pas dépassée au cours de son remplissage.

Description du scénario. Le but de l'auto-protection est d'éviter le dépassement du seuil fixé pour la quantité de mémoire utilisée dans la JVM : une JVM (Java Virtual Machine) est une machine virtuelle Java qui fournit un environnement nécessaire pour l'exécution du bytecode java. La JVM sur laquelle s'exécute notre application ne doit pas utiliser plus que la quantité *XMX* de mémoire qui lui est attribuée à sa création. En cas de dépassement, une exception de type *OutOfMemoryException* est levée et le fonctionnement de l'application peut être perturbé voire arrêté. Un nouveau composant de gestion mémoire (MemoryManager) est introduit dans l'application *Comanche* (cf. Figure 8.8). Il possède un attribut *seuil* correspondant à un pourcentage du paramètre *XMX* de la JVM dans laquelle le composant est instancié.

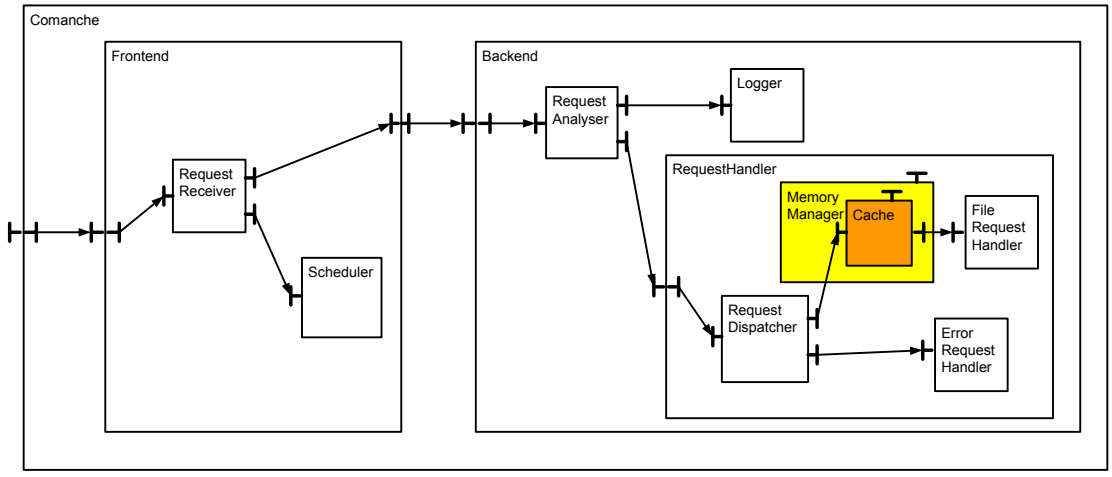

FIGURE 8.8 – Architecture de *Comanche* avec cache et gestionnaire de mémoire

Réalisation. La principale cause d'augmentation de la consommation mémoire de notre application est le remplissage du cache du serveur. Le cache peut utiliser une grande quantité de mémoire

pour stocker les données, dans ce cas la quantité de mémoire maximale attribuée *XMX* à la JVM peut facilement être dépassée. La configuration de la taille maximale du cache représentée par l'attribut *maxCacheSize* du composant *Cache* permet ainsi de limiter sa consommation mémoire. Nous fixons alors un seuil (en pourcentage de la valeur *XMX*) pour pouvoir minimiser les risques de déclencher une surcharge mémoire. La contrainte est ainsi spécifiée au niveau du composant *Cache* :

```
<definition name="Cache">
  ...
 <constraint value="./Cache@maxCacheSize < .@seuil " />
</definition>
```

Le composant *MemoryManager* réalise l'ensemble de la boucle d'auto-protection telle que décrite dans le diagramme de la figure 8.9.

- Observation : la contrainte d'intégrité spécifiant la condition de dépassement du seuil mémoire est évaluée à intervalles réguliers (mode polling) par le composant *ConsistencyManager* qui est un sous-composant du composant *TransactionManager* (cf. chapitre 7, section 7.1). La période de temps pour l'évaluation de la contrainte est paramétrable dans le gestionnaire de mémoire.
- Analyse et décision : le composant de gestion mémoire est chargé de récupérer le résultat de l'évaluation de la contrainte mémoire. En cas de violation, la taille maximale du cache est réajustée pour limiter la quantité de mémoire utilisée par le serveur qui doit repasser sous le seuil maximum. Dans le cas contraire, la taille maximale du cache est optimisée pour occuper tout l'espace mémoire disponible.
- Exécution : la valeur de l'attribut *maxCacheSize* est régulé en fonction de la mémoire disponible dans la JVM et du seuil défini dans la contrainte. La diminution de sa valeur peut entraîner la suppression de certaines données du cache en utilisant l'algorithme FIFO.

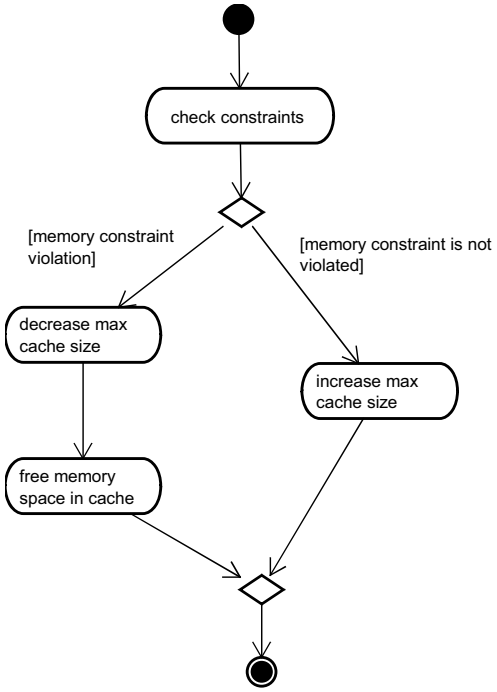

FIGURE 8.9 – Diagramme d'activité du scénario

Pour simuler les requêtes des clients, nous utilisons une injection de charge automatique. L'injection de charge consiste à injecter des requêtes HTTP différentes dans le serveur pour faire augmenter la taille du cache jusqu'à dépasser la limite fixée par le seuil dans la contrainte. L'injecteur de charge utilisé est un simple client HTTP qui envoie des requêtes à période fixe sur une série de fichiers de taille déterminée.

Conclusion. Ce scénario d'auto-protection permet de montrer la modularité et la réutilisabilité d'un composant parmi les différents composants de l'architecture du gestionnaire de transactions

pour les reconfigurations. En effet, la boucle autonomique utilise le composant de vérification de cohérence (*ConsistencyManager*) en dehors de toute reconfiguration pour déclencher l'adaptation de l'application. Ce composant est utilisé pendant la phase d'observation (monitoring) du serveur pour vérifier la violation d'une contrainte d'intégrité applicative. Cette contrainte applicative est intégrée dans la définition ADL de l'application. La violation de contraintes, au lieu d'invalider une reconfiguration, comme dans le cas classique des reconfigurations transactionnelles, est ici détectée par un gestionnaire autonomique particulier dédié à la gestion de la consommation mémoire. Ce gestionnaire réagit de manière spécifique sans mettre en oeuvre les mécanismes d'abandon de transaction.

8.3.3 Auto-optimisation d'un cluster de serveur Web

Objectifs du scénario. Ce scénario implémente une boucle d'auto-optimisation de la gestion de la charge dans un cluster de serveur HTTP. Le recouvrement logiciel (ou reprise à chaud) des reconfigurations transactionnelles est ici mise en oeuvre en cas de violation d'une contrainte d'intégrité applicative. Cette contrainte permet de limiter le nombre d'instances de serveurs dans le cluster. Les propriétés d'atomicité et de cohérence des transactions sont utilisées dans ce scénario.

Description de l'application. L'application retenue pour ce scénario est un cluster de serveurs web *Comanche* (cf. Figure 8.10). Les requêtes des clients sont orchestrées par un composant *Scheduler* instancié sur machine séparée qui réalise la répartition de charge entre les différentes instances du cluster, c'est-à-dire qu'il répartit les requêtes des clients sur les différents serveurs pour ne pas les surcharger (algorithme du tourniquet ou « round robin »). Chaque serveur *Comanche* est instancié sur une machine différente, les machines du cluster sont réifiées sous forme de composants Fractal dans lesquelles les composants *Comanche* sont insérés. Le composant Cluster regroupe l'ensemble des machines actives dans le cluster, un certain nombre de machine peuvent ne pas être utilisée et disponible pour l'instanciation d'un nouveau serveur. Le composant *ClusterComanche* au plus haut niveau de la hiérarchie regroupe l'ensemble des machines ainsi que le répartiteur de charge.

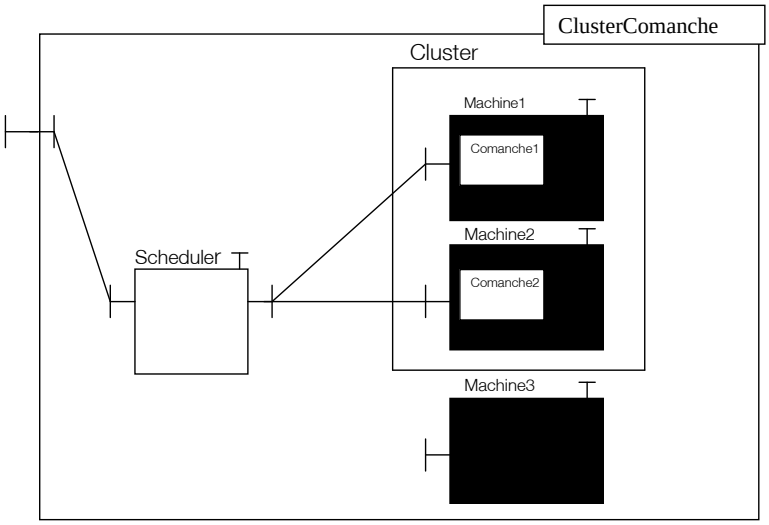

FIGURE 8.10 – Architecture du cluster *Comanche*

Description du scénario. Un cluster de serveur *Comanche* est mise en place sur un ensemble de machines. Le facteur retenu pour l'optimisation est le taux de charge global du cluster correspondant à la quantité de requêtes soumises au cluster par unité de temps. Si le taux de charge dépasse un certain seuil, il est possible d'instancier d'autres serveurs sur les ressources disponibles. Une ressource disponible étant une machine présente dans le système mais non encore utilisée dans la répartition de charge, un seul serveur est instancié par machine. Un composant de gestion de charge *LoadManager* colocalisé avec le composant *Scheduler* (cf. Figure 8.11) est chargé d'observer le taux de charge par rapport au nombre de serveurs utilisés dans le cluster, les deux paramètres étant exposés sous forme

d'attribut respectivement dans les composants *Scheduler* et *Cluster*. Le *LoadManager* instancie de nouveaux serveurs sur des machines disponibles *Comanche* pour équilibrer la charge si elle devient trop importante pour que le cluster puisse rester performant.

Le cluster dispose d'un ensemble de machines disponibles pour éventuellement augmenter le nombre de serveurs. Les opérations pour instancier un nouveau *Comanche* sont les suivantes. En cas de dépassement de seuil de charge, le *LoadManager* recherche une machine dans le cluster, instancie un nouveau serveur sur la machine, le connecte au composant *Scheduler* et met à jour le nombre de serveurs dans le cluster.

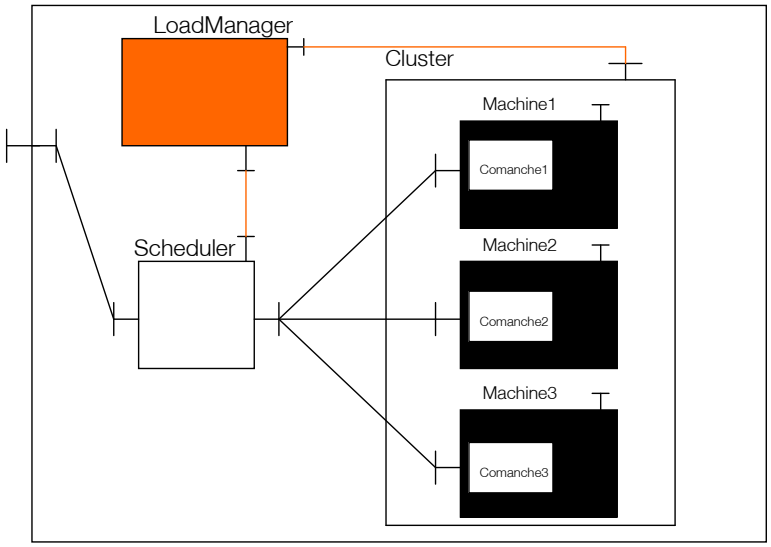

FIGURE 8.11 – Cluster Comanche après reconfiguration

Réalisation. Les ressources (machines) disponibles peuvent être en fait partagées entres différentes applications. Nous souhaitons que ces ressources ne soient pas toutes utilisées pour le *ClusterComanche*. Une contrainte d'intégrité est alors spécifiée pour imposer l'utilisation d'un nombre limité de machines par le cluster (la chaîne *≶* remplace le caractère < en XML) :

```
1 <definition name="Cluster">
  ...
3 <constraint value=".@nbServers &lg;= 2"/>
  ...
5 </definition>
```

Le composant *LoadManager* réalise la boucle d'auto-optimisation décrite dans le diagramme de la figure 8.12. Il utilise les reconfigurations transactionnelles pour l'instanciation de nouveaux serveurs dans le cluster. Le composant de gestion de transactions décrit dans le chapitre 7 (section 7.1) est localisé sur la machine du *Scheduler*. La boucle autonomique est ainsi la suivante :

- Observation : le taux de charge global du cluster est mesuré au niveau du *Scheduler*, le taux de charge par serveur en est déduit en fonction du nombre de serveurs dans le cluster.
- Analyse et Décision : Si le taux de charge par serveur dépasse un seuil fixé (en nombre de requêtes par seconde), la décision d'instanciation d'un nouveau serveur est prise pour prendre en compte cette augmentation de charge et redimensionner le cluster.
- Exécution : une opération d'instanciation et de connexion d'un nouveau serveur est réalisée de manière transactionnelle. Si la contrainte de limitation du nombre de serveurs dans le cluster est violée, l'opération est annulée par le gestionnaire de transactions.

L'instanciation du nouveau serveur utilise le déploiement à distance de Fractal ADL avec Fractal RMI pour la gestion de la distribution. Le code de reconfiguration pour redimensionner le cluster est spécifié en FScript :

```
1 action reconf(clusterComanche, newMachine) {
    $cluster = $clusterComanche/cluster;
```

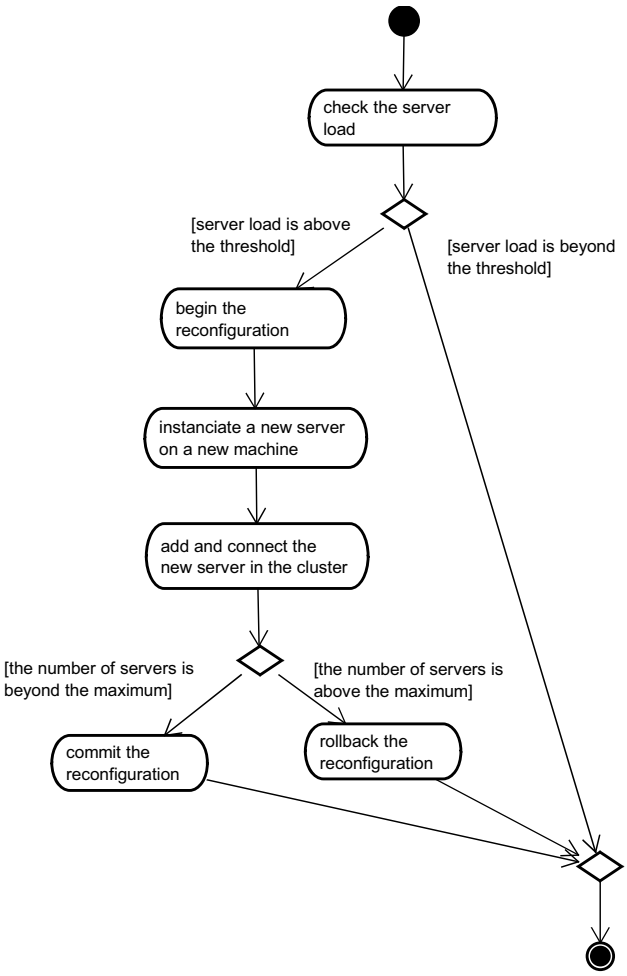

FIGURE 8.12 – Diagramme d'activité du scénario

```
3   $scheduler = $clusterComanche/scheduler;
    newServer = adl-new('comanche.Comanche');
5   add($newMachine, $newServer);
    add($cluster, $newMachine);
7   start($newMachine);
    cbind($scheduler, 'schedule', $newMachine/interface::schedule); -- schedule is a
        collection interface
9 }
```

Le cluster initial comporte un ordonnanceur (*Sheduler*) en frontend sur une machine et deux serveurs chacun sur une machine différente dans le backend. L'injection de charge se fait en deux temps : un premier client HTTP qui effectue des requêtes périodiques est lancé sans que la charge dépasse le seuil de redimensionnement. Un deuxième client est lancé au bout d'un certain temps, la charge du cluster est ainsi normalement doublée de manière à déclencher le redimensionnement du cluster par ajout d'une machine. Or la contrainte qui fixe le nombre de serveur à moins de deux est violée, la reconfiguration consistant en l'ajout du nouveau serveur est alors annulé.

Conclusion. Ce scénario d'auto-optimisation est une expérimentation des reconfigurations transactionnelles dans un contexte distribué. L'intégration des transactions est réalisée de manière transparente et non intrusive, le gestionnaire de transactions une fois instancié gère l'ensemble des reconfigurations dynamiques dans l'application. Nous montrons l'utilisation d'une contrainte d'intégrité applicative sous forme d'invariant structurelle pour modéliser une limitation de disponibilité des ressources (les machines dans le cluster). Enfin, une défaillance avec abandon de transaction est détectée suite à une violation de contrainte et réparée automatiquement grâce à la reprise à chaud des transactions. La reconfiguration invalide est annulée (propriété d'atomicité) pour remettre l'architecture

de l'application dans son état initial et cohérent.

8.3.4 Auto-réparation de panne franche dans un cluster de serveur Web

Objectifs du scénario. Ce scénario met en oeuvre une boucle d'auto-réparation de panne franche dans un cluster de serveur HTTP. La panne franche considérée est limitée à un crash JVM pour simplifier le scénario, ce dernier permet de mettre en oeuvre la reprise à froid des reconfigurations transactionnelles dans le cadre d'un cluster de serveurs web. Les propriétés d'atomicité et de durabilité des transactions sont mises en avant au cours de l'exécution de ce scénario.

Description de l'application. L'application dans ce scénario est la même que celle utilisée dans le scénario précédent (cf. Figure 8.10) : un cluster de serveurs *Comanche*. Les composants *Comanche* sont initialement dépourvus de cache lors de l'instanciation du cluster. Chaque machine du cluster ne contient qu'une unique JVM dans laquel est instancié un seul serveur.

Description du scénario. Nous considérons un cluster de serveurs *Comanche*. Nous supposons que suite à la volonté d'un administrateur de faire évoluer les fonctionnalités du cluster, l'ensemble des serveurs *Comanche* sont reconfigurés dynamiquement pour y introduire un composant *Cache* (l'opération est décrite en 8.3.2), cette reconfiguration est réalisée de manière transactionnelle. Au bout d'un certain temps, un des serveurs *Comanche* du cluster ne répond plus (cf. Figure 8.13). Le composant *Scheduler* envoie les requêtes aux serveurs pour être traitées, s'il ne reçoit pas de réponse au bout d'un temps déterminé, il considère que le serveur est en panne et le retire du cluster. Le fait que nous ne considérions d'un crash de JVM implique la machine contenant le serveur défaillant est toujours active et disponible pour l'instanciation d'un serveur. Le serveur *Comanche* est alors réinstancié sur cette même machine dans une nouvelle JVM.

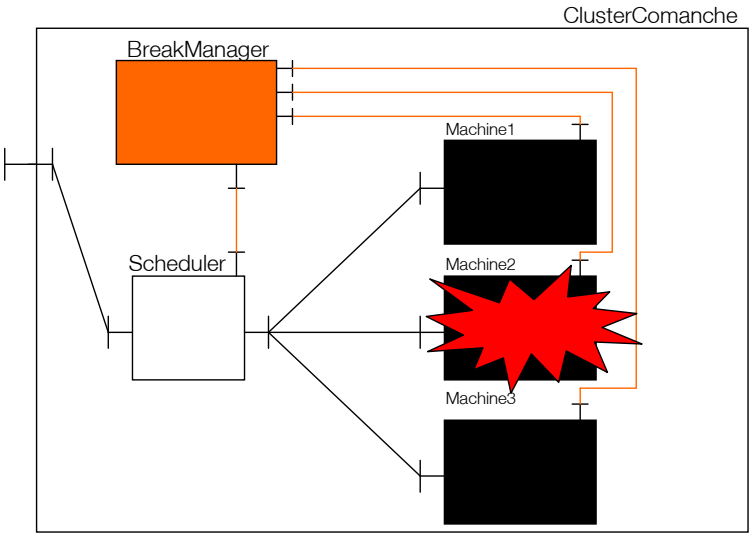

FIGURE 8.13 – Panne franche dans le Cluster Comanche

Réalisation. Le composant *BreakManager* est chargé de la réparation des serveurs défaillants dans le cluster. Le boucle d'auto-réparation est décrite dans la figure 8.14.

- Observation : le composant *Scheduler* transfert les requêtes aux différents serveurs avec un « timeout »pour chaque requête. Si un bout de ce temps, il ne reçoit pas de réponse d'un serveur, il considère que le serveur est en panne. Il notifie alors le *BreakManager* de la panne et modifie la répartition de charge en conséquence.
- Analyse et Décision : le composant *BreakManager* prend en compte l'information du *Scheduler* et choisit le plan de réparation.
- Exécution : le serveur défaillant est retiré du cluster et le serveur est réinstancié sur la même machine. Grâce à la propriété de durabilité des transactions, le serveur est réinitialisé dans l'état

dans lequel il était au cours de la dernière reconfiguration. Le serveur est connecté au *Scheduler* pour être intégré dans la répartition de charge, son architecture et son état fonctionnel ont en effet été persistés.

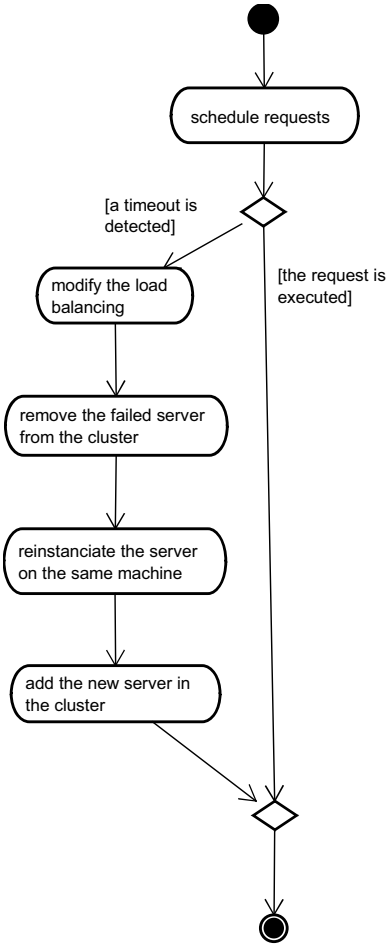

FIGURE 8.14 – Diagramme d'activité du scénario

La panne franche est simulée par l'arrêt d'une JVM sur une machine du cluster contenant un serveur. Lors de la réinstanciation du serveur, le composant est remis dans le dernier cohérent à l'issue de la dernière reconfiguration. Or le composant a été reconfiguré dynamiquement avec l'ajout d'un composant cache avant la panne donc son architecture ne correspond plus à son architecture initiale spécifiée dans sa définition en Fractal ADL. La propriété de durabilité des reconfigurations transactionnelle permet de retrouver le dernier état, particulièrement l'architecture, du composant *Comanche* et donc de le réinstancier dans sa configuration avant la panne, i.e. avec un composant *Cache* dans son architecture.

Conclusion. Le principale intérêt de cette expérimentation est de montrer l'utilisation de la propriété de durabilité des reconfigurations transactionnelles lors de défaillance de site. La durabilité permet grâce notamment à la persistance de l'architecture des composants reconfigurés de remettre l'application dans son état avant l'occurrence d'une panne de machine. Le protocole de recouvrement est automatiquement mise en oeuvre pour réaliser une reprise à froid (cf chapitre 6 section 6.5) lors de la réinstanciation des composants qui était sur le site défaillant.

8.3.5 Auto-réparation d'un server Java EE dans un cluster

Objectifs du scénario. Nous présentons dans ce scénario l'utilisation des reconfigurations transactionnelles dans le cadre d'un cluster de serveur Java EE[1]. Ce scénario consiste en la réparation d'un serveur défaillant à l'intérieur du cluster. Le type de défaillance détectée et réparée est la surcharge mémoire d'un serveur. L'opération de réparation consiste à redémarrer le serveur pour réinitialiser son occupation mémoire, cette opération de reconfiguration est exécutée dans une transaction et doit respecter des contraintes d'intégrité définissant la cohérence de l'architecture du cluster et des serveurs.

Description de l'application. Le scénario se déroule dans un environnement Java EE et plus particulièrement dans un cluster de serveurs d'applications JOnAS. Une application de commerce en ligne appelée SOAPSOO est déployé sur ce cluster. Nous présentons donc successivement le standard Java EE, le serveur d'applications JOnAS, le cluster JOnAS, et l'application SOAPSOO.

La spécification Java EE de Sun [Jav] (anciennement J2EE) définit une architecture et des interfaces pour développer et déployer des serveur d'applications Internet distribuées en Java reposant sur une architecture multi-tiers. JOnAS [JOn] est un serveur d'application open source développé au sein du consortium OW2 qui implémente la certification J2EE 1.4 de Sun et qui peut facilement être déployé en cluster. Les clusters Java EE sont de plus en plus fréquemment déployés dans les systèmes d'informations d'entreprises pour faire face à des besoins croissants en termes de disponibilité et de passage à l'échelle. Dans beaucoup de cas, la complexité des architectures distribuées soulève des problèmes importants d'administration qui ne peuvent plus raisonnablement être gérés manuellement :

- Le modèle N-tiers (tiers web, métier et données), les nombreuses couches logicielles (OS/-JVM/Java EE/application) et l'aspect distribué d'un cluster augmente considérablement les informations à observer et à gérer dans le système.
- La configuration et l'optimisation des capacités fournies par les composants de l'intergiciel ainsi que la complexité de Java EE nécessite la prise en compte de nombreux paramètres pour lesquels un outil reprenant des concepts de l'informatique autonomique est utile. L'objectif d'un tel outil est particulièrement la réduction des coûts de telles architectures.

Les principaux apports d'une solution autonomique pour l'administration de cluster sont :

- la reduction des risques des erreurs humaines d'administration
- l'amélioration du temps de réponse pour éliminer les problèmes dans le système
- l'amélioration de la disponibilité des applications (en minimisant les interruptions de service)
- l'amélioration de la performance globale du système

Pour les besoins du scénario, l'application SOAPSOO est déployée sur le cluster JOnAS. SOAPSOO est une application Web de catalogue en ligne reposant sur les technologies Java EE. L'application est développée utilisée opérationnellement par France Telecom/Orange pour la qualification de serveurs Java EE. Elle gère des articles de deux types (matériel et service) associés à des catalogues et des contrats. L'utilisateur peut à travers une interface Web consulter, modifier, créer ou supprimer des articles, des catalogues et des contrats (see Figure 8.16). Les données de l'application sont persistées dans une base de donnée relationnelle grâce à un mapping objet-relationnel. La connexion à l'application requiert une authentification avec login et mot de passe.

Description du scénario. Ce scénario est utilisé pour réparer une faute transiente dans une instance JOnAS. Une faute transiente est par exemple une fuite mémoire ou une surcharge mémoire, et une méthode commune pour réparer ce type de faute est le redémarrage du serveur d'application (reboot). De plus, nous démontrons dans ce scénario certaines capacités des reconfigurations dynamiques utilisées pour l'adaptation du système dans la plateforme Selfware. En effet, il est possible d'ajouter des contraintes d'intégrité sur l'architecture du cluster qui ne doivent pas être violées pendant les reconfigurations. Le service de reconfiguration assure ensuite que les reconfigurations dynamiques déclenchées par les gestionnaires autonomiques pour réparer le système sont fiables et respectent les contraintes. L'application web SOAPSOO décrite dans la section précédente est déployée sur un cluster de serveurs JOnAS 4.8.

1. Ce scénario est basé sur une plateforme développé dans le cadre du projet RNTL Selfware auquel nous avons participé : http ://sardes.inrialpes.fr/selfware/index.php

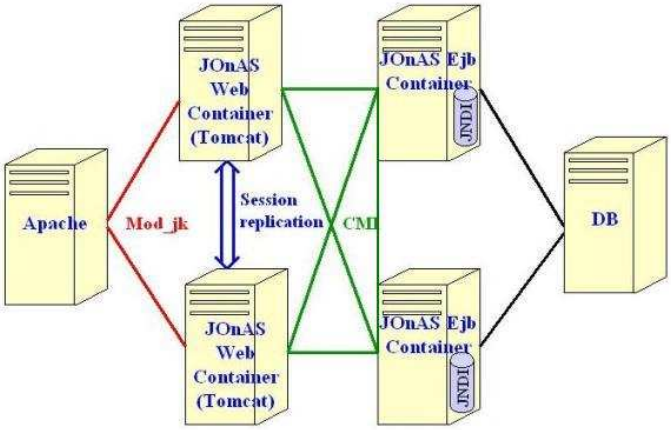

FIGURE 8.15 – Cluster de serveurs J2EE Jonas

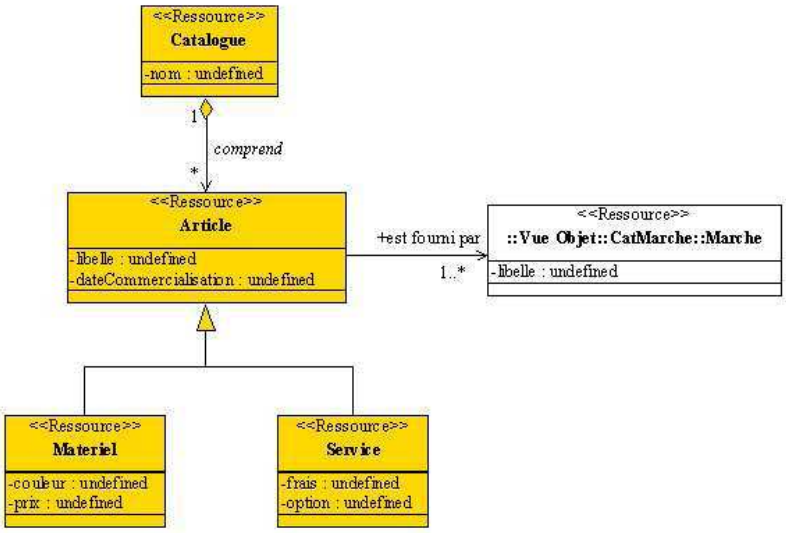

FIGURE 8.16 – Modèle de données de l'application de catalogue en ligne SOAPSOO

Pour implémenter ce scénario, nous utilisons deux services additionnels dans la plateforme Selfware : le service de vérification de contraintes et le service de reconfiguration. Avec le service de vérification de contraintes, il est possible de vérifier que l'architecture du système est conforme au contraintes d'intégrité spécifiées. Les contraintes peuvent ainsi être spécifiées au niveau de l'architecture globale du cluster ou au niveau de chaque noeud du cluster. Nous avons identifié quelques exemples de contraintes :

Sur un cluster :
- Unicité du nom d'une instance JOnAS dans le domaine d'administration du cluster.
- Unicité de l'instance JOnAS maître dans le domaine pour gérer le cluster.
- Séparation entre les tiers Web et EJB sur différents noeuds.

Sur un noeud :
- Disponibilité des ressources système (mémoire, CPU) : une configuration d'instance JOnAS ou une application donnée peut requérir un minimum garanti de quantité de ressource pour pouvoir être exécutée.
- Unicité des ports réseaux entre les instances JOnAS.
- Nombre maximum/minimum d'instances JOnAS sur un même noeud.

Le service de reconfiguration est utilisé pour reconfigurer dynamiquement le système avec le langage FScript. L'interpréteur FScript est combiné à notre gestionnaire transactionnel pour garantir les propriétés ACID des reconfigurations. Si une reconfiguration viole des contraintes dans le système cible, la reconfiguration est annulée et le système est remis dans son dernier état consistent.

Réalisation. Nous choisissons de réparer une surcharge mémoire dans une instance JOnAS qui peut mener à un crash de la JVM en redémarrant le serveur localement (i.e. la JVM). Nous mettons une contraintes sur chaque noeud du cluster en rapport avec la quantité minimum de mémoire disponible nécessaire pour redémarrer le noeud.

La boucle autonomique (cf. Figure 8.17) utilisée pour réparer la faute transiente est composée des éléments suivants :
- Sondes : l'outil MBeanCmd fourni par JASMINe [JAS] est utilisé pour monitorer les instances JOnAS dans les clusters et permet de récupérer les exceptions du type OutOfMemoryException levée par les JVM dont la mémoire est en surcharge.
- Contrôleur : le RebootManager souscrit aux événements JMX levés par le MBeanCmd et quand il est notifié de la surcharge mémoire, il décide de redémarrer les serveurs d'applications fautifs.
- Actionneurs : le ReconfigurationService exécute le plan de réparation comme une transaction de reconfiguration. La transaction est composée des opérations de reconfiguration suivantes :

1. arrêter le serveur : invocation de l'opération « stop »du LifeCycleController sur le composant qui wrappe le serveur ;
2. tuer le serveur : après l'étape précédente, le serveur est dans un état « stopped » mais est toujours présent, d'où la nécessité de terminer l'exécution du serveur ;
3. redémarrer le serveur fautif sur la même machine.

Lors de la validation de la reconfiguration, le service de vérification de contraintes vérifie que la quantité de mémoire disponible dans le noeud est au-dessus du seuil minimum fixé par la contrainte sur le noeud :
- si oui, l'action de réparation est exécutée jusqu'à sa fin ;
- sinon, le service de reconfiguration annule l'action de réparation et notifie l'administrateur de la violation de contrainte et de l'impossible réinstanciation du serveur fautif sur le noeud. Une extension possible du scénario serait de proposer un nouveau plan de réparation comme par exemple la réinstanciation du serveur fautif sur un autre noeud où suffisamment de mémoire est disponible.

Conclusion. Ce scénario a été l'occasion d'expérimenter les reconfigurations transactionnelles dans un contexte industriel Java EE. L'apport de notre contribution sur les reconfigurations dynamiques au scénario consiste à introduire une contrainte d'intégrité applicative au niveau d'un serveur d'application et à vérifier que cette contrainte est bien respectée lors de la validation de la reconfiguration du serveur. La reconfiguration appliquée est ici le redémarrage du serveur suite à un dépassement mémoire et la contrainte permet de valider le redémarrage que si la mémoire disponible sur la machine

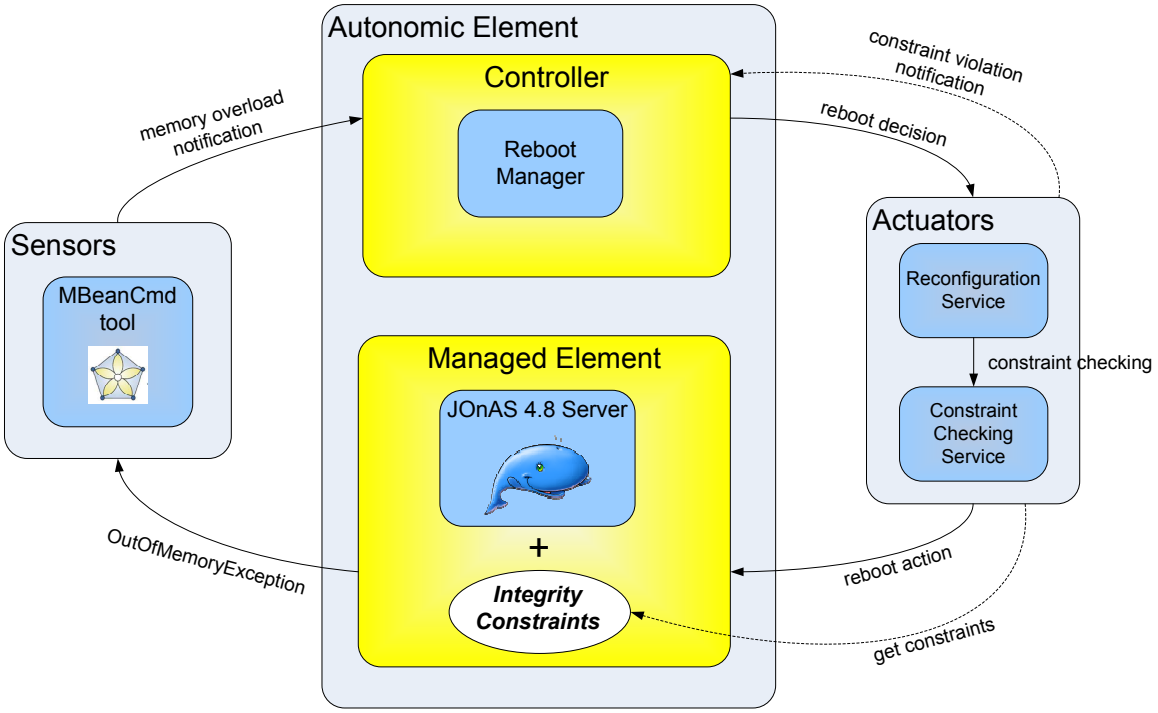

FIGURE 8.17 – Scénario de reconfiguration transactionnelle dans un cluster J2EE

est suffisante pour l'opération de « reboot ». Dans le cas contraire, la transaction est abandonnée suite à la violation de la contrainte et la reconfiguration est annulée.

–

8.4 Contrôle d'accès pour les reconfigurations dynamiques et transactionnelles

La fiabilité et la sécurité sont souvent associées comme attributs de la sûreté de fonctionnement. Nous nous proposons donc de mettre en oeuvre les mécanismes de reconfigurations transactionnelles pour répondre à un besoin de sécurité. Nous définissons dans un premier temps un modèle de contrôle d'accès pour l'exécution des reconfigurations dynamiques de composants. Nous associons dans un second temps la vérification des permissions sur les opérations de reconfigurations au recouvrement de ces reconfigurations en cas de violation de permission. Un scénario simple de reconfiguration sur un serveur HTTP est mise en oeuvre pour montrer l'association des deux préoccupations.

8.4.1 Un modèle de sécurité pour les opérations de reconfiguration

Les propriétés attendues d'un modèle de sécurité pour assurer sa cohérence sont la confidentialité et l'intégrité. Autrement dit, l'information ne doit être accessible qu'aux personnes autorisées et le caractère intègre de l'information doit être assuré. Parmi les différentes politiques de sécurité, les politiques d'accès basées sur les rôles [JBSK04] (RBAC ou Role-Based Access Control) sont largement utilisées dans les entreprises et les administrations. Elles reposent sur le concept de rôle. Un rôle représente un ensemble d'actions et de responsabilités associées à une activité particulière.

Nous avons choisi de concevoir et d'implémenter une politique de sécurité du type RBAC pour les reconfigurations dynamiques (cf. Figure 8.18). Ce type de politique répond en effet le mieux à la manière dont nous voulons représenter l'information de la politique de sécurité dans les systèmes à base de composants : différents utilisateurs dans un système peuvent posséder des permissions particulières pour reconfigurer ou introspecter le système.

Dans notre approche, la mise en oeuvre du contrôle d'accès se fait au niveau de l'API Fractal, nous attribuons à chaque contrôleur d'un composant des permissions en lecture et/ou écriture. Un

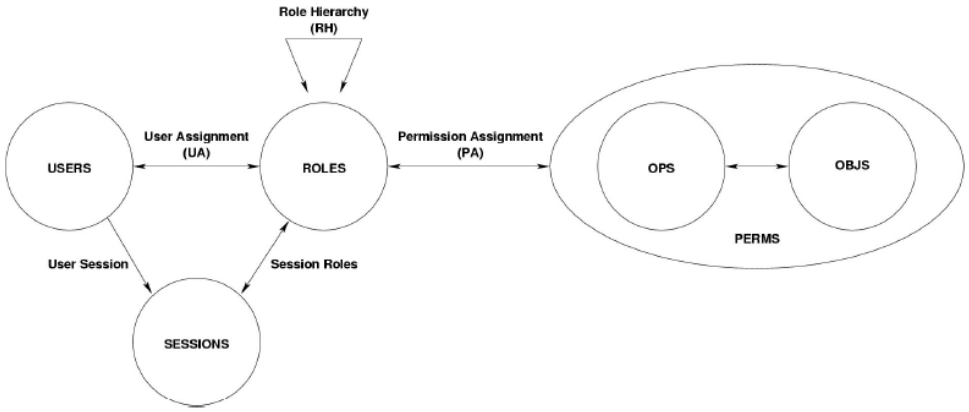

FIGURE 8.18 – Modèle RBAC

droit en lecture équivaut à autoriser les opérations d'introspection pour un contrôleur donné, un droit en écriture les opérations d'intercessions. En accord avec les concepts RBAC, les groupes définissent des ensembles d'objets (l'ensemble des composants auxquels s'applique la politique de sécurité), et les rôles des ensembles de sujets (les utilisateurs identifiés par des composants). Chaque utilisateur appartient à un ou plusieurs groupes et dispose d'un ou plusieurs rôles. Un rôle confère des droits d'exécution pour chaque opération de reconfiguration. Par conséquent, un utilisateur aura accès à une opération primitive de l'API Fractal si et seulement si les droits associés à son rôle lui permettent l'exécution des opérations correspondantes.

L'administration de la sécurité du système est basée sur un ensemble de règles ou permissions associées à chaque composant. Ce modèle respecte les concepts du modèle RBAC :

- la définition de hiérarchies de rôles (les droits d'exécution sont différents pour chaque rôle),
- le principe du moindre privilège (un composant aura le strict minimum des privilèges nécessaires pour pouvoir accomplir sa tâche)
- la séparation des responsabilités (pour éviter d'avoir des conflits de rôles).

Chaque composant identifié par son nom possède par défaut un groupe et un rôle au moment de sa création. Chaque rôle a des permissions pour pouvoir accéder aux opérations de reconfiguration de l'API Fractal. Notre modèle de sécurité peut est résumé par les relations de la figure 8.19.

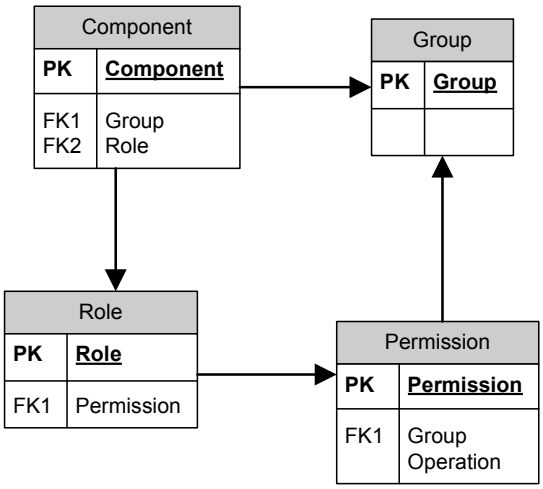

FIGURE 8.19 – Schéma relationnel du modèle RBAC pour les reconfigurations

La politique de sécurité pour un système donné est spécifiée dans des fichiers XML chargé au démarrage par un composant de sécurité (*AccessControlMonitor*). Ce composant est responsable de l'application du contrôle d'accès lors de l'invocation d'opérations de reconfiguration sur les compo-

sants du système.

8.4.2 Evaluation des reconfigurations transactionnelles et sécurisées

Objectifs du scénario. L'objectif de ce scénario est d'évaluer l'utilisation des mécanismes transactionnels pour les reconfigurations couplés à un modèle de contrôle d'accès sur les opérations de reconfigurations. L'exécution d'une reconfiguration au cours de laquelle une opération non-autorisée est exécutée entraîne l'annulation de la transaction. Nous utilisons ainsi le recouvrement logiciel des transactions lors de violation de permissions à la place des contraintes d'intégrité, prouvant ainsi la modularité de l'architecture du gestionnaire de transactions.

Description de l'application. Nous utilisons le serveur HTTP *Comanche* dans son architecture de base décrit dans la section 8.3.

Description du scénario. Le scénario consiste à ajouter dynamiquement une fonctionnalité de cache au serveur sous la forme d'un nouveau composant. Cette modification est réalisée par un administrateur dont les droits ne permettent pas d'ajouter de nouveaux composants dans l'application. Les opérations effectuées pour introduire le cache dans le serveur web *Comanche* en cours d'exécution :

- Arrêt du Composant *RequestDispacher* avec lequel va se lier le cache.
- Destruction de la liaison avec le *RequestDispacher*.
- Ajout du composant *Cache* dans l'architecture.
- Liaison du cache avec le *FileRequestHandler* avec qui il va communiquer pour trouver les fichiers qu'il n'a pas en mémoire.
- Liaison avec le *RequestDispacher*.
- Démarrage du cache.
- Démarrage du *RequestDispacher*.

La reconfiguration du serveur *Comanche* est réalisée par le script FScript suivant :

```
action add-cache(comanche) {
 rh = $comanche/descendant::rh;
 rd = $rh/child::rd;
 frh= $rh/child::frh;
 inth0 = $rd/interface::h0[client(.)];
 stop($rd);
 unbind($inth0);
 cache = adl-new('comanche.Cache');
 add($rh, $cache);
 bind($inth0, $cache/interface::r);
 bind($cache/interface::rh, $frh/interface::rh);
 start($rd);
}
```

Interception des opération primitives. Chaque opération primitive de reconfiguration est interceptée par un composant de contrôle d'accès *AccessControlMonitor*, l'administrateur est réifié dans notre modèle sous forme de composant, l'identité de l'administrateur correspond à l'identité du composant qui le réifie. L' *AccessControlMonitor* vérifie donc que le composant client de la reconfiguration (appelant lors de l'invocation d'opération) possède bien les droits d'exécution. Le scénario de reconfiguration contient les étapes suivantes :

- Lecture et stockage de la politique de sécurité (fichiers XML).
- Création du composant correspondant à l'administrateur du système.
- Création des composants à partir de leur définition ADL.
- Reconfiguration du système (invocation d'opération de reconfiguration dont l'opération d'instanciation de composant)
- Vérification des droits de l'utilisateur pour l'opération considérée
- Reconfiguration du système en cas d'autorisation, annulation de la reconfiguration sinon et remontée d'une exception du type *AccessControlException*.

Réalisation. En accord avec les concepts RBAC, les groupes définissent des ensembles d'objets (entités passives comme des fichiers), et les rôles des ensembles de sujets (entités actives comme des threads). Dans notre modèle, objets et sujets sont représentés par des composants. Nous définissons deux rôles dans la politique de sécurité RBAC pour l'accès aux reconfigurations dans le système : un rôle *administrator* possédant tous les droits et un rôle *user* dont les droits sont restreints. Les rôles *administrator* et *user* sont associés respectivement aux composants *Administrator* et *User* qui vont implémenter et déclencher la reconfiguration de l'application. Un seul groupe *comanche* est créé pour regrouper l'ensemble des composants du serveur. Les permissions sont donc attribués aux deux rôles pour la reconfiguration des composants appartenant au groupe *comanche*, c'est à dire tous les composants de l'application. Nous attribuons au rôle *user* tous les droit d'exécution sur les opérations de reconfiguration du groupe *comanche* hormis pour le contrôleur *ContentController* pour lequel nous interdisons les opérations d'intercessions (la lettre R indique un droit en lecture seulement) :

```
<role name="User">
      <controller name="ContentController">
        <access type="R" />
      </controller>
      ...
</role>
```

Le composant *User* dans le serveur invoque l'action FScript $add-cache$ sur le composant *Comanche*. Dans la terminologie du contrôle d'accès, le sujet est le composant *User* et l'objet est le composant *Comanche* qui est reconfiguré. Le composant *User* ne possède pas les droits d'exécution pour les opérations d'ajout et de retrait de composant d'après la politique de sécurité décrite précédemment. Le script de reconfiguration s'exécute donc jusqu'à rencontrer l'opération *add* et la violation des droits d'exécution entraîne l'annulation de la transaction de reconfiguration :

```
action add-cache(comanche) {
 rh = $comanche/descendant::rh;
 rd = $rh/child::rd;
 frh= $rh/child::frh;
 inth0 = $rd/interface::h0[client(.)];
 stop($rd);
 unbind($inth0);
 cache = adl-new('comanche.Cache');
 add($rh, $cache); // Accès interdit => annulation de la reconfiguration
 ...
}
```

La même reconfiguration exécutée par le composant *Administrator* s'exécute par contre normalement, le composant ayant tous les droits d'exécution.

Conclusion. Nous démontrons avec l'expérimentation de ce scénario que la définition de la cohérence pour les reconfigurations transactionnelles peut reposer sur d'autres types de contraintes que les contraintes d'intégrité définis dans le chapitre 4. La propriété de cohérence peut ainsi être lié à des concepts de sécurité à la place ou en complément des problèmes de fiabilité. Nous définissons ainsi des permissions dans un modèle de contrôle d'accès de type RBAC pour valider ou invalider des reconfigurations transactionnelles de la même façon que nous utilisons les contraintes d'intégrité avec des invariants de configuration et des pré et postconditions sur les opérations de reconfiguration. La violation d'une permission d'accès sur une opération primitive à l'intérieur d'une reconfiguration complexe entraîne ainsi l'annulation de toute la reconfiguration.

8.5 Conclusion

Dans ce chapitre, les mécanismes de reconfigurations transactionnelles ont été évalués quantitativement sur un exemple simple de type *HelloWorld* avec le surcoût des différentes fonctionnalités des transactions. Tout en restant utilisable de manière indépendante, notre gestionnaire de transactions a été intégré avec succès dans un outil existant : l'interpréteur de langage de reconfiguration FScript. Il est ainsi possible de spécifier des reconfigurations en FScript et tout en bénéficiant des bonnes propriétés (propriétés ACID) des transactions pour l'exécution des reconfigurations en Fractal.

De plus, l'apport de notre solution pour la fiabilisation des reconfigurations dynamiques dans les architectures à composants a été montré notamment dans le domaine de l'informatique autonomique à travers plusieurs expérimentations. Un premier scénario a été présenté pour réaliser de l'auto-protection mémoire dans un serveur HTTP et présenter la mise oeuvre des contraintes en dehors d'un contexte transactionnel. Un deuxième scénario réalise l'auto-optimisation contrainte de la charge dans un cluster de serveurs HTTP avec la démonstration de l'abandon de transaction en cas de violation de la cohérence du système. Un troisième scénario est conçu pour la réparation de panne franche dans un cluster de serveurs web, l'expérimentation fait appel à la propriété de durabilité des reconfigurations transactionnelles. Un dernier scénario autonomique dans un contexte J2EE permet de montrer l'auto-réparation avec reconfigurations transactionnelles dans un cluster de serveur d'application JOnAS. Enfin, nous avons montré la modularité de l'architecture de notre solution en intégrant des mécanismes de contrôle d'accès aux reconfigurations transactionnelles.

Chapitre 9

Conclusion

Sommaire

Ce chapitre de conclusion vient clore la présentation de nos contributions. Nous rappelons brièvement dans la section 9.1 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 9.2 propose un certain nombre de perspectives à plus ou moins long terme de nos travaux.

9.1 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, l'objectif de cette thèse était de concevoir et de mettre en oeuvre des mécanismes de fiabilisation des reconfigurations dynamiques dans les architectures à composants. Les reconfigurations dynamiques sont un moyen de faire évoluer les systèmes sans les arrêter totalement en préservant donc leur disponibilité. Nous avons considéré les hypothèses suivantes pour les reconfigurations :

- Les reconfigurations sont *dynamiques*. Elles se produisent pendant l'exécution du système.
- Les reconfigurations sont *concurrentes*. Plusieurs reconfigurations peuvent être exécutées en même temps sur le même système.
- Les reconfigurations sont *non-anticipées*. Une reconfiguration n'est pas nécessairement définie et prévue au moment de la conception et du déploiement du système.

Cependant, les reconfigurations sont des transformations de l'état des systèmes qui peuvent à ce titre rendre ces derniers incohérents et dans l'incapacité de fournir les fonctionnalités attendues. 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.

Synthèse des contributions. Pour fiabiliser les reconfigurations dynamiques, nous sommes plus particulièrement intéressés au modèle de composants Fractal, modèle réflexif et dynamique. Notre démarche peut se résumer en trois étapes. Maintenir la cohérence des systèmes au cours de leurs évolutions suppose d'avoir une définition précise de la cohérence dans les architectures à composants. La première étape a donc consisté en la définition de la cohérence des systèmes à base de composants, de leur architecture (ou configuration) et de leurs évolutions (ou reconfigurations). La deuxième étape a été de concevoir les mécanismes de maintien de la cohérence des systèmes sujets à des reconfigurations dynamiques. La troisième et dernière étape a été la réalisation d'une architecture pour la mise en oeuvre des mécanismes de reconfigurations fiables. A ces trois étapes correspondent nos trois principales contributions :

- une modélisation des configurations et des reconfigurations pour définir la cohérence des systèmes. Nous avons défini un modèle de cohérence à l'aide de contraintes d'intégrité sous forme

d'invariants de configuration et de préconditions et de postconditions sur les opérations de reconfigurations. Ces propositions ont été réalisées dans le modèle de composants Fractal.
- une approche transactionnelle pour maintenir la cohérence des systèmes reconfigurés et les rendre tolérants aux fautes. Nous avons défini un modèle de transactions adapté au contexte des reconfigurations dynamiques qui permet de supporter la concurrence des reconfigurations et le recouvrement de défaillances aussi bien logicielles que matérielles.
- une architecture modulaire pour la gestion des reconfigurations transactionnelles dans les applications à base de composants Fractal. Le modèle transactionnel pour les reconfigurations dynamiques est implémenté sous la forme d'un canevas logiciel à composants, d'extensions du modèle Fractal (nouveaux contrôleurs) et de ses outils (Fractal ADL). La modularité de l'architecture permet de choisir les fonctionnalités transactionnelles ou d'en changer d'implémentation.

Enfin, ces contributions ont été expérimentées dans des scénarios autour de l'informatique autonomique et intégrées dans FScript, un interpréteur de langage dédié pour les reconfigurations dans Fractal.

9.2 Perspectives

Nos contributions constituent une réponse au problème de la fiabilisation des reconfigurations dynamiques dans les architectures à composants. Cependant, nos travaux pourraient bénéficier de quelques améliorations pour corriger certaines limitations ou pour élargir les perspectives d'utilisation des reconfigurations fiables. Les perspectives à envisager concernent aussi bien des problèmes d'optimisation de l'exécution des reconfigurations que des propriétés supplémentaires à apporter à notre modèle de transactions.

Optimisation de la disponibilité. Une première perspective possible est liée à la sémantique des opérations décrite dans le chapitre 5 et aux changements d'état du cycle de vie des composants qui servent à synchroniser les reconfigurations avec l'exécution fonctionnelle (cf. section 6.4). Certaines opérations d'intercession nécessitent en effet d'arrêter les composants pour les reconfigurer, cet arrêt se traduit par un blocage de l'exécution fonctionnelle des composants concernés et donc une indisponibilité temporaire pour les fonctionnalités du système implémentées par les composants arrêtés. Il serait possible d'analyser la forme canonique des reconfigurations, i.e. la séquence des opérations primitives qui les constituent, et de réordonnancer les opérations de manière à minimiser le nombre d'opérations entre deux opérations d'arrêt et de démarrage de composants (opérations *stop* et *start*) pour limiter le temps d'indisponibilité du système reconfiguré. Par exemple, une opération d'instanciation d'un nouveau composant ou une opération d'ajout d'un composant dans un autre n'ont pas de précondition sur l'état du cycle de vie des composants du système et peuvent donc être exécutées en priorité avant les autres opérations dans une reconfiguration composite. Ce réordonnancement des opérations de reconfiguration pour être valide devra cependant tenir compte de leur sémantique, en particulier de leur commutativité éventuelle pour respecter leurs préconditions et postconditions d'exécution.

Parallélisation de l'exécution. Une deuxième perspective vise à optimiser les reconfigurations en parallélisant leur exécution. Dans un contexte distribué ou multiprocesseurs, il peut en effet être intéressant d'exécuter certaines opérations en parallèle pour maximiser le temps d'exécution des reconfigurations. La gestion de la distribution est une préoccupation gérée de manière transparente au niveau du modèle Fractal et dans l'implémentation en Java avec Fractal RMI mais il est utile pour des raisons d'optimisation de tenir compte de la répartition des composants du système sur les différents sites. Un premier travail consisterait à essayer de séparer, avec d'éventuels réordonnancements, une reconfiguration distribuée en des sous-ensembles d'opérations dont les exécutions sont indépendantes et concernent des sites différents. Une fois cette séparation effectuée, la reconfiguration est exécutée par lots (*batch execution*) en parallèle sur les sites concernés par la reconfiguration. Le temps d'exécution des reconfigurations serait ainsi optimisé par la parallélisation de l'exécution des opérations primitives.

Passage à l'échelle. Le passage à l'échelle en terme de nombre de composants et de distribution n'a pas été étudié prioritairement dans nos travaux. Le choix de la granularité des participants dans les transactions et du protocole de validation ne sont pas conçu pour des systèmes massivement répartis. En effet, l'architecture du gestionnaire de transactions et le protocole de validation atomique utilisé sont centralisés (cf. section 6.2). Il faudrait notamment pousser l'étude des deux autres solutions concernant la granularité des participants (le composant ou le site) en tenant spécifiquement compte de la préoccupation du passage à l'échelle en terme de nombre de communications, de la gestion plus ou moins répartie de l'état du système et des techniques de recouvrement adaptées.

Modèle de gestion de concurrence enrichi. Nous avons considéré pour le contrôle de concurrence l'utilisation d'une méthode pessimiste avec verrouillage. Comme mentionné dans la section 6.4, les méthodes pessimistes sont adaptées pour des reconfigurations de courte durée de vie très concurrentes, ce qui est le cas de la plupart des reconfigurations architecturales. Dans le cadre plus générale de notre architecture modulaire du gestionnaire de transactions pour les reconfigurations dynamiques (cf. chapitre 7), il pourrait cependant être intéressant de proposer une solution pour l'implémentation d'un contrôle de concurrence optimiste pour certaines reconfigurations lorsque le niveau de concurrence est faible. La certification [BHG87] permettrait de s'affranchir du coût du verrouillage à chaque opération, par contre elle nécessiterait une étape d'analyse des conflits avant la validation des reconfigurations transactionnelles. Grâce à la modularité de l'architecture du gestionnaire de transactions, le modèle optimiste pourrait être évalué et comparé au modèle pessimiste en fonction de plusieurs facteurs comme le niveau de concurrence entre reconfigurations. De ces résultats, nous déduirions ainsi des heuristiques pour déterminer le choix optimal d'un modèle ou l'autre en fonction du type de reconfigurations envisagé.

Processus de vérification continue de la cohérence. Notre démarche de fiabilisation des reconfigurations dynamiques repose sur la définition de la cohérence des systèmes à base de composants par un ensemble de contraintes d'intégrité (cf. chapitre 4). Cependant le processus de spécification et de vérification des contraintes est divisée en deux principales étapes. Une première étape nécessite la spécification des contraintes dans un langage de spécification dédié, Alloy, pour garantir la cohérence du modèle des configurations, des reconfigurations et des contraintes entre elles. Il s'agit de déterminer la non-contradiction entre contraintes pour s'assurer de l'existence de configurations vérifiant les contraintes. Cette vérification est cependant limitée aux contrainte du modèle et des profils et exclue les contraintes applicatives. Dans un second temps, les contraintes sont implémentées dans un langage de navigation dans les architectures Fractal, FPath, afin de pouvoir les implémenter et les vérifier dans des systèmes à l'exécution. Une perspective intéressante serait donc de concevoir une chaîne de vérification intégrée qui ne suppose pas la spécification des contraintes dans les deux langages. Une possibilité est de concevoir un traducteur automatique du langage Alloy vers le langage FPath (ou réciproquement), seule la spécification en Alloy (ou en FPath) restant obligatoire à effectuer. Une autre possibilité serait de se passer complètement d'un des deux outils de vérification (l'analyseur Alloy et l'interpréteur FPath) et d'implémenter toute la vérification dans un seul et même outil, cela suppose soit de modifier l'analyseur Alloy pour y intégrer des capacités d'introspection et d'intercession dans des systèmes développés en Fractal/Julia en réalisant notamment un mapping entre des fonctions définies en Alloy et les opérations de l'API Fractal, soit d'ajouter dans l'interpréteur FPath la capacité de vérifier la non-contradiction entre contraintes d'intégrité via par exemple un solveur SAT.

Validation statique des reconfigurations. Les reconfigurations transactionnelles sont un moyen d'introduire des mécanismes de tolérance aux fautes au cours de l'évolution dynamique des systèmes à base de composants. La tolérance aux fautes consiste à détecter les fautes une fois qu'elles se sont produite et à les réparer de manière transparente. Une autre technique, dite de prévention de fautes, consiste à empêcher l'occurrence de certaines fautes avant leur apparition. Cette dernière technique de vérification au plus tôt, avant l'exécution des reconfigurations dans le système, pourrait reposer sur des analyses statiques du code de reconfiguration, les fautes considérées étant les violations de contraintes d'intégrité [DLG+08]. Les analyses statiques sont plus facile à mettre en oeuvre sur un langage dédié comme FScript que sur des reconfigurations programmées dans un langage généraliste comme Java. L'intégration de notre gestionnaire de transactions avec l'interpréteur FScript a été réalisée avec succès (cf. section 4.1), une extension possible serait donc d'intégrer des techniques

d'analyse statique au niveau de l'interpréteur pour filtrer les scripts de reconfiguration invalides lors de leur chargement. Un certain nombre de scripts peuvent en effet être rejeté sans même avoir connaissance de l'architecture cible du système à reconfigurer. Par exemple et comme présenté dans la section (cf. section 4.1), une reconfiguration qui crée un cycle dans l'architecture est invalide quelque soit le système sur lequel elle est appliquée car le modèle Fractal interdit ces cycles. L'intérêt d'une étape d'analyse statique serait donc d'empêcher l'exécution de ce type de reconfiguration, sachant à l'avance qu'elle produirait une violation de la cohérence du système.

Généralisation à d'autres modèles de composants. Notre modélisation des configurations et des reconfigurations est basé sur l'utilisation de systèmes à base de composants Fractal. De même notre implémentation des reconfigurations transactionnelles a été réalisée avec l'implémentation Julia du modèle Fractal. Une première perspective serait dans un premier temps de considérer d'autres implémentations du modèle Fractal telle que Think [FSLM02], implémentation en langage C pour le développement de composants embarqués et de systèmes d'exploitation. Il faudrait alors notamment mener une réflexion sur les éléments communs en terme de modélisation et les nouvelles préoccupations en terme de performance pour le modèle de transactions et son implémentation. Par ailleurs et pour aller au delà du modèle Fractal, notre approche de fiabilisation des reconfigurations dynamiques pourraient être généralisée et appliquées à d'autres modèles de composants qui possèdent des capacités similaires de reconfiguration. Le modèle réflexif OpenCOM [CBG+04], SCA [SCA], ou encore OSGi [OSG] sont ainsi des cibles potentielles pour intégrer nos travaux. Cette généralisation nécessiterait un effort de modélisation des concepts de ces différents modèles en terme d'éléments architecturaux et de relations entre ces éléments comme cela a été fait pour Fractal (cf. chapitre 4). Il faudrait également spécifier la sémantique des opérations de reconfigurations (cf. chapitre 5. Le modèle de transactions décrit dans le chapitre 6 devrait par contre rester sensiblement le même, les opérations de reconfigurations restent en effet assez similaires entre les modèles (modification de la structure, du cycle de vie, etc.). L'architecture du gestionnaire de transactions présentée dans le chapitre 7 est générique donc transposable plus ou moins directement à d'autres modèles de composants suivant leurs propriétés notamment en terme de support de la hiérarchie.

Reconfiguration dynamique du gestionnaire de transactions. L'architecture du gestionnaire de transactions pour les reconfigurations est conçue et implémentée sous forme de composants Fractal. Nous avons montré l'intérêt de l'utilisation de certains composants séparément du gestionnaire (utilisation du composant de gestion de la cohérence dans la section 8.3). Notre architecture est modulaire car elle permet le choix des fonctionnalités des transactions qui se traduit par le choix des composants et de leurs implémentations dans la configuration instanciée du gestionnaire de transactions. Par contre, nous n'avons pas considéré la reconfiguration dynamique du gestionnaire de transactions lui-même grâce aux propriétés du modèle Fractal. Cette reconfiguration dynamique permettrait par exemple de changer et d'adapter dynamiquement le modèle de cohérence, de concurrence, ou encore de recouvrement en fonction du contexte d'exécution : niveau de concurrence, longueur des reconfigurations, taux de validation, etc.

Troisième partie

Annexes

Annexe A

Spécification des (re)configurations Fractal en Alloy

A.1 Modélisation des configurations

A.1.1 Configurations

```
module fractal/configuration

/*
* Modelling of Fractal configurations
*/


// Modelling of architectural elements
abstract sig Element {
  id: Id // Each Element has an identifier
}

// Modelling of element properties (mutable and immutable)
abstract sig Property {}

abstract sig Id extends Property {}

fact elementSanity {
  Id in Element.id // All Ids are associated to an Element
}

// Interface modelling
sig Interface extends Element {
  name: ItfName,
  visibility: Visibility,
  signature: Type,
  role: Role,
  cardinality: Cardinality,
  contingency: Contingency
}

abstract sig ItfName, Role, Visibility, Cardinality, Contingency extends Property {}

abstract sig Type extends Property {
  subtypes: set Type
}

fact typeSanity {
  all type: Type | not type in type.^subtypes // No cycle in the subtyping relation
  Type in (Interface.signature + Attribute.type)
}

lone sig Client, Server extends Role {}

lone sig External, Internal extends Visibility {}

lone sig Singleton, Collection extends Cardinality {}

lone sig Mandatory, Optional extends Contingency {}

fact interfaceSanity {
  ItfName in Interface.name
  Role in Interface.role
```

```
54    Visibility in Interface.visibility
      Cardinality in Interface.cardinality
      Contingency in Interface.contingency
    }

59  // Attribute modelling
    sig Attribute extends Element {
      name: AttName,
      type: Type
    }
64
    abstract sig AttName, Value extends Property {}

    fact attributeSanity {
      AttName in Attribute.name
69  }

    // Component modelling
    sig Component extends Element {}

74  sig Name extends Property {}

    abstract sig State extends Property {}

    lone sig Started, Stopped extends State {}
79
    // A Configuration is a set of architectural elements, of relations between elements
    // and of element properties
    sig Configuration {
      // Architectural elements
84    components: set Component,
      interfaces: set Interface,
      attributes: set Attribute,
      // Architectural relations
      child: components -> components,
89    interface: components one -> interfaces,
      attribute: components one -> attributes,
      binding: interfaces -> lone interfaces,
      // Mutable element properties
      name: components -> one Name,
94    state: components -> one State,
      value: attributes -> one Value
    }

    fact configurationSanity {
99    Component in Configuration.components
      Interface in Configuration.interfaces
      Attribute in Configuration.attributes
    }


    module fractal/controller

3   /*
    * Modelisation of controller interfaces for Fractal configurations
    */

    open fractal/configuration
8
    // An interface is either a control or a functional interface
    abstract sig ControllerItf, FunctionalItf extends Interface {}

    // All controllers are server interfaces
13  fact controllerSanity {
      Interface = ControllerItf + FunctionalItf
      Interface in ControllerItf => Interface.role = Server
    }

18  // List all possible controllers
    abstract sig AttributeControllerItf, BindingControllerItf, ContentControllerItf,
        LifeCycleControllerItf,
      NameControllerItf, SuperControllerItf, ComponentItf, FactoryItf extends ControllerItf {}
```

A.1.2 Contraintes d'intégrité statiques

```
    module fractal/static_integrity

    /*
    * Integrity constraints to define consitency of static Fractal configurations
5   */
```

```
open fractal/configuration
open fractal/util
open fractal/controller

// Identifiers are unique
fact uniqueId {
  all aElt1, aElt2: Element | (aElt1.id = aElt2.id) => (aElt1 = aElt2)
}

// Two interfaces belonging to the same component and with different visibility
// must not have the same name
fact uniqueItfNameVis {
  all aConf: Configuration, aItf1, aItf2: Interface {
    ( aItf1 != aItf2 && ~(aConf.interface)[aItf1] = ~(aConf.interface)[aItf2]
      && (aItf1.visibility = aItf2.visibility)) => (aItf1.name != aItf2.name)
  }
}

// Two attributes belonging to the same component and with different visibility
// must not have the same name
fact uniqueAttrName {
  all aConf: Configuration, aAtt1, aAtt2: Attribute {
    ( aAtt1 != aAtt2 && ~(aConf.attribute)[aAtt1] = ~(aConf.attribute)[aAtt2] )
      => (aAtt1.name != aAtt2.name)
  }
}

// A component which contains subcomponents is a composite
fact parentsAreComposite {
  all aConf: Configuration, aComp: Component {
    aComp in ~(aConf.child)[Component] => isComposite[aConf, aComp]
  }
}

// A component which has an internal interface is a composite
fact internalItfs {
  all aConf: Configuration, aItf: Interface {
    (aItf.visibility in Internal) => isComposite[aConf, ~(aConf.interface)[aItf]]
  }
}

// Each internal interface has a dual external interface
fact dualItfs {
  all aConf: Configuration, aItf: Interface |
    (aItf.visibility = Internal) => one dualItf: Interface | isDualItf[aConf, aItf, dualItf
        ]
}

// There is no cycle in the component hierarchy of a configuration
fact noCycle {
  all aConf: Configuration | no aComp: Component {
    aComp->aComp in ^(aConf.child)
  }
}

// A binding is from a client interface to a server interface
fact bindingDirection {
  all aConf: Configuration, cItf, sItf: Interface {
    cItf->sItf in aConf.binding => (cItf.role in Client && sItf.role in Server)
  }
}

// Two interface can be bound if their type are compatible
fact bindingType {
  all aConf: Configuration,cItf, sItf: Interface {
    cItf->sItf in aConf.binding => isBindingTypeCompatible[cItf, sItf]
  }
}

// Mandatory client interfaces must be bound for a component to ba started
fact mandatoryBindings {
  all aConf: Configuration, aItf: Interface {
    let clientItfOwner = ~(aConf.interface)[aItf] {
      ( aConf.state[clientItfOwner] = Started && not aItf in ControllerItf
        && aItf.role = Client && aItf.contingency = Mandatory)
        => (aItf in ~(aConf.binding)[Interface])
    }
  }
}

// Bindings must respect the component hierarchy
fact localBindings {
  all aConf: Configuration, cItf, sItf: Interface {
    cItf->sItf in aConf.binding => isLocalBinding[aConf, cItf, sItf] && not isDualItf[aConf
        , cItf, sItf]
```

```
  }
}


// A component has at least th following controllers:
// Component, NameController, LifeCycleController and SuperController
fact standardControllers {
  all aConf: Configuration, aComp: Component {
    one aItf1: Interface | (aComp->aItf1 in aConf.interface) && (aItf1 in ComponentItf)
    one aItf2: Interface | (aComp->aItf2 in aConf.interface) && (aItf2 in NameControllerItf
        )
    one aItf3: Interface | (aComp->aItf3 in aConf.interface) && (aItf3 in
        LifeCycleControllerItf)
    one aItf4: Interface | (aComp->aItf4 in aConf.interface) && (aItf4 in
        SuperControllerItf)
  }
}

/* These constraints are already defined in Configuration */
/*
// An interface has an unique component owner
fact uniqueItfOwner{
  all aConf: Configuration, itf: Interface | #(~(aConf.interface))[itf] = 1
}

// An attribute has an unique component owner
fact uniqueAttrOwner {
  all aConf: Configuration, att: Attribute | #(~(aConf.attribute))[att] = 1
}

// A client interface can be at most be bound to one server interface
fact bindingCardinality {
  all aConf: Configuration, itf: Interface | #(~(aConf.binding))[itf] = 1
}
*/


module fractal/util

/*
* Utility predicates and functions for Fractal configurations
*/

open fractal/configuration
open fractal/controller

// Comparaison of elements between two configurations
pred sameElements[initial, final: Configuration] {
  final.components = initial.components
  final.interfaces = initial.interfaces
  final.attributes = initial.attributes
}

// Comparaison of relations between two configurations
pred sameRelations[initial, final: Configuration] {
  final.child = initial.child
  final.interface =initial.interface
  final.attribute = initial.attribute
  final.binding = initial.binding
}

// Comparaison of properties between two configurations
pred sameProperties[initial, final: Configuration] {
  final.name = initial.name
  final.state = initial.state
  final.value = initial.value
}

// Comparaison of configurations
pred sameConfigurations[initial, final: Configuration] {
  sameElements[initial, final]
  sameRelations[initial, final]
  sameProperties[initial, final]
}

// Subtyping relation
pred isSubTypeOf(type1, type2: Type) {
  type1 in type2.*subtypes
}

// Subtyping relation for interfaces
pred isSubInterfaceTypeOf(itf1, itf2: Interface) {
  itf1.name = itf2.name
  itf1.role = itf2.role
  itf1.role = Server => (isSubTypeOf[itf1.signature, itf2.signature]
    && itf2.contingency = Mandatory => itf1.contingency = Mandatory)
```

```
    itf1.role = Client => (isSubTypeOf[itf2.signature, itf1.signature]
      && itf2.contingency = Optional => itf1.contingency = Optional)
    itf2.cardinality = Collection => itf1.cardinality = Collection
}

// Subtyping relation for components
pred isSubComponentTypeOf(conf: Configuration, c1, c2: Component) {
  all itf1: conf.interface[c1] {
    itf1.role = Client => one itf2: conf.interface[c2] {
      isSubInterfaceTypeOf[itf1, itf2]
    }
  }
  all itf2: conf.interface[c2] {
    itf2.role = Server => one itf1: conf.interface[c1] {
      isSubInterfaceTypeOf[itf1, itf2]
    }
  }
}

// Type compatibilty for interface bindings
pred isBindingTypeCompatible(itf1, itf2: Interface) {
  not itf1 in ControllerItf
  not itf2 in ControllerItf
  itf1.role != itf2.role
  itf1.role = Client => isSubTypeOf[itf1.signature, itf2.signature]
  itf1.role = Server => isSubTypeOf[itf2.signature, itf1.signature]
}

// Binding is a normal binding
pred isNormalBinding[aConf: Configuration, cItf, sItf: Interface] {
  cItf.role in Client
  cItf.visibility in External
  sItf.role in Server
  sItf.visibility in External
  one cc, cs, cp: Component {
    cc->cItf in aConf.interface
    cs->sItf in aConf.interface
    cp->cc in aConf.child
    cp->cs in aConf.child
  }
}

// Binding is an export binding
pred isExportBinding[aConf: Configuration, cItf, sItf: Interface] {
  cItf.role in Client
  cItf.visibility in Internal
  sItf.role in Server
  sItf.visibility in External
  one cc, cs: Component {
    cc->cItf in aConf.interface
    cs->sItf in aConf.interface
    cc->cs in aConf.child
  }
}

// Binding is an import binding
pred isImportBinding[aConf: Configuration, cItf, sItf: Interface] {
  cItf.role in Client
  cItf.visibility in External
  sItf.role in Server
  sItf.visibility in Internal
  one cc, cs: Component {
    cc->cItf in aConf.interface
    cs->sItf in aConf.interface
    cs->cc in aConf.child
  }
}

// Binding locality
pred isLocalBinding[aConf: Configuration, itf1, itf2: Interface] {
  ~(aConf.interface)[itf1] != ~(aConf.interface)[itf2]
  isNormalBinding[aConf, itf1, itf2] || isExportBinding[aConf, itf1, itf2] ||
      isImportBinding[aConf, itf1, itf2]
}

// Component is composite
pred isComposite [aConf: Configuration, aComp: Component] {
  one aItf: Interface {
    (aComp->aItf in aConf.interface) && (aItf = ContentControllerItf)
  }
}

// Interface duality
pred isDualItf [aConf: Configuration, itf1, itf2: Interface] {
  itf1 in aConf.interfaces
```

```
    itf2 in aConf.interfaces
    itf1.visibility != itf2.visibility
    ~(aConf.interface)[itf1] = ~(aConf.interface)[itf2]
    itf1.name = itf2.name
    itf1.signature = itf2.signature
    itf1.role != itf2.role
    itf1.cardinality = itf2.cardinality
    itf1.contingency = itf2.contingency
  }

  // Test if a component has no binding inside a parent component
  pred notBoundInParent[aConf: Configuration, childComp, parentComp: Component] {
    all itf1, itf2: Interface, aComp: Component {
      (aComp->itf1 in aConf.interface && childComp->itf2 in aConf.interface
        && (itf1->itf2 in aConf.binding || itf2->itf1 in aConf.binding) )
        => ( aComp != childComp && aComp in aConf.child[parentComp] )
    }

  }
```

A.2 Modélisation des reconfigurations

A.2.1 Opérations primitives de reconfiguration

```
module fractal/reconfiguration

/*
* Modelling of primitive reconfiguration operations (only intercession)
*/

open fractal/configuration
open fractal/util
open fractal/static_integrity
open fractal/dynamic_integrity

// Create a new component in a configuration
pred new [initial, final: Configuration, newComp: Component] {
  // Preconditions
  not newComp in initial.components

  // Elements
  final.components = initial.components + newComp
  initial.interfaces in final.interfaces
  initial.attributes in final.attributes
  // Relations
  final.child = initial.child
  final.interface[Component - newComp] = initial.interface[Component]
  final.attribute[Component - newComp] = initial.attribute[Component]
  final.binding = initial.binding
  // Properties
  final.name[Component - newComp] = initial.name[Component]
  final.state[Component - newComp] = initial.state[Component]
  initial.value in final.value

  // Postconditions
  newComp in final.components
}

// Add a component in a composite component
pred add [initial, final: Configuration, childComp, parentComp: Component] {
  // Preconditions
  (childComp + parentComp) in initial.components
  not parentComp->childComp in initial.child
  isComposite[initial, parentComp]
  not childComp->parentComp in *(initial.child)

  // Elements
  sameElements[initial, final]
  // Relations
  final.child = initial.child + parentComp->childComp
  final.interface = initial.interface
  final.attribute = initial.attribute
  final.binding = initial.binding
  // Properties
  sameProperties[initial, final]

  // Postconditions
  parentComp->childComp in final.child
}
```

```
//Remove a component from a composite component
pred remove [initial, final: Configuration, childComp, parentComp: Component] {
  // Preconditions
  (childComp + parentComp) in initial.components
  parentComp->childComp in initial.child
  isComposite[initial, parentComp]
  initial.state[ *(initial.child)[parentComp] ] = Stopped
  notBoundInParent[initial, childComp, parentComp]

  // Elements
  sameElements[initial, final]
  // Relations
  final.child = initial.child - parentComp->childComp
  final.interface =initial.interface
  final.attribute = initial.attribute
  final.binding = initial.binding
  // Properties
  sameProperties[initial, final]

  // Postconditions
  not parentComp->childComp in final.child
}

// Bind two component interfaces
pred bind [initial, final: Configuration, clientItf, serverItf: Interface] {
  // Preconditions
  (clientItf + serverItf) in initial.interfaces
  not clientItf->serverItf in initial.binding
  isBindingTypeCompatible[clientItf, serverItf]
  isLocalBinding[initial, clientItf, serverItf]
  not isDualItf[initial, clientItf, serverItf]

  // Elements
  sameElements[initial, final]
  // Relations
  final.child = initial.child
  final.interface =initial.interface
  final.attribute = initial.attribute
  final.binding = initial.binding + clientItf->serverItf
  // Properties
  sameProperties[initial, final]

  // Postconditions
  clientItf->serverItf in final.binding
}

// Unbind two component interfaces
pred unbind [initial, final: Configuration, clientItf: Interface] {
  // Preconditions
  clientItf in initial.interfaces
  initial.binding[clientItf] != none
  let clientItfOwner = ~(initial.interface)[clientItf] |
    initial.state[ *(initial.child)[clientItfOwner] ] = Stopped

  // Elements
  sameElements[initial, final]
  // Relations
  final.child = initial.child
  final.interface =initial.interface
  final.attribute = initial.attribute
  final.binding = initial.binding - clientItf->initial.binding[clientItf]
  // Properties
  sameProperties[initial, final]

  // Postconditions
  final.binding[clientItf] = none
}

// Change a component name
pred setName [initial, final: Configuration, aComp: Component, newName: Name] {
  // Preconditions
  aComp in initial.components
  newName != initial.name[aComp] // to choose a new name

  // Elements
  sameElements[initial, final]
  // Relations
  sameRelations[initial, final]
  // Properties
  final.name = initial.name - aComp->initial.name[aComp] + aComp->newName
  final.state = initial.state
  final.value = initial.value

  // Postconditions
  final.name[aComp] = newName
```

```
}

// Change an attibute value
pred setValue [initial, final: Configuration, aAtt: Attribute, newValue: Value] {
  // Preconditions
  aAtt in initial.attributes
  newValue != initial.value[aAtt] // to choose a new value

  // Elements
  sameElements[initial, final]
  // Relations
  sameRelations[initial, final]
  // Properties
  final.name = initial.name
  final.state = initial.state
  final.value = initial.value - aAtt->initial.value[aAtt] + aAtt->newValue

  // Postconditions
  final.value[aAtt] = newValue
}

// Change a component state
pred setState [initial, final: Configuration, aComp: Component, newState: State] {
  // Preconditions
  aComp in initial.components
  newState != initial.state[aComp] // to choose a new state

  // Elements
  sameElements[initial, final]
  // Relations
  sameRelations[initial, final]
  // Properties
  final.name = initial.name
  final.state = initial.state - aComp->initial.state[aComp] + aComp->newState
  final.value = initial.value

  // Postconditions
  final.state[aComp] = newState
}

// Stop a component recursively
pred stop [initial, final: Configuration, aComp: Component] {
  all aChild: *(final.child)[aComp] {
    setState[initial, final, aChild, Stopped]
  }
}

// Start a component recursively
pred start [initial, final: Configuration, aComp: Component] {
  all aChild: *(final.child)[aComp] {
    setState[initial, final, aChild, Started]
  }
}
```

A.2.2 Contraintes d'intégrité dynamiques

```
module fractal/dynamic_integrity

/*
* Integrity constraints to define consitency of dynamic Fractal reconfigurations
*/

open fractal/configuration

// A reconfiguration is a state transformation between two configurations
fact  configurationLimit {
  #Configuration <= 2
}

// A reconfiguration operation is a simple evolution of a configuration
fact validEvolution {
  all aConf1, aConf2: Configuration {
    (aConf1.components in aConf2.components
      && aConf1.interfaces in aConf2.interfaces
      && aConf1.attributes in aConf2.attributes
    ) ||
    (aConf2.components in aConf1.components
      && aConf2.interfaces in aConf1.interfaces
      && aConf2.attributes in aConf1.attributes)
  }
}
```

```
// Interfaces are conserved for a given component
fact itfConservation {
  all aConf1, aConf2: Configuration, aComp: Component {
    (aComp in aConf1.components && aComp in aConf2.components)
      => (aConf1.interface[aComp] = aConf2.interface[aComp])
  }
}

// Attributes are conserved for a given component
fact attConservation {
  all aConf1, aConf2: Configuration, aComp: Component {
    (aComp in aConf1.components && aComp in aConf2.components)
      => (aConf1.attribute[aComp] = aConf2.attribute[aComp])
  }
}
```

Bibliographie

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *SIGSOFT '93 : Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 9–20, New York, NY, USA, 1993. ACM.

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering Methodologies*, 4(4) :319–364, 1995.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava : Connecting software architecture to implementation. In *International Conference on Software Engineering, ICSE 2002*, Orlando, Florida, USA, May 2002.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[AGP98] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-phase commit : Does it make sense ? In *ICPADS '98 : Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, page 182, Washington, DC, USA, 1998. IEEE Computer Society.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. Technical Report Number : CMU-CS-97-144.

[BCF+97] Jérôme Besancenot, Michèle Cart, Jean Ferrié, Rachid Guerraoui, Philippe Pucheral, and Bruno Traverson. *Les systèmes transactionnels : concepts, normes et produits*. Hermes collection Informatique, 1997.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, May 2004. Springer-Verlag.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12) :1257–1284, 2006.

[BCS03] Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stéfani. The Fractal component model. Technical report, The ObjectWeb Consortium, September 2003. version 2.0.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHQS05] F. Boyer, D. Hagimont, V. Quema, and J.-B. Stefani. Architecture-based autonomous repair management : Application to j2ee clusters. In *ICAC '05 : Proceedings of the Second International Conference on Automatic Computing*, pages 369–370, Washington, DC, USA, 2005. IEEE Computer Society.

[BJC05] Thais Batista, Ackbar Joolia, and Gordon Coulson. Managing dynamic reconfiguration in component-based systems. In *2nd European Workshop on Software Architectures (EWSA 2005)*, 2005.

[Car82] C Carter, W. A time for reflection. In *12th IEEE Int. Symp. on Fault Tolerant Computing*, page 41, Santa Monica, California, June 1982.

[CBG+04] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, November 2004.

[CDP+99] Fabio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon S. Blair, and Geoff Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In *OORaSE*, pages 79–98, 1999.

[Che02] Xuejun Chen. Extending rmi to support dynamic reconfiguration of distributed systems. In *ICDCS '02 : Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 401, Washington, DC, USA, 2002. IEEE Computer Society.

[CND+04] Pierre Cointe, Jacques Noyé, Rémi Douence, Thomas Ledoux, Jean-Marc Menaud, Gilles Muller, and Mario Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. *RSTI L'Objet*, 10(4), 2004.

[Coi06] Pierre Cointe. *Les langages à objets.* pub.vui, December 2006. Chapitre de l'Encyclopédie de l'informatique et des systèmes d'information.

[COM] COM Specification. *http ://www.microsoft.com/com/.*

[CSST98] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection : Bridging the gap between a running system and its architectural specification. In *Proceedings of Reengineering Forum '98*, 1998.

[Das02] Eric M. Dashofy. An infrastructure for the rapid development of xml-based architecture description languages. In *In Proceedings of the 24th International Conference on Software Engineering (ICSE2002*, pages 266–276. ACM Press, 2002.

[Dav05] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation.* PhD thesis, Université de Nantes / École des Mines de Nantes, July 2005.

[DC01] Jim Dowling and Vinny Cahill. The K-Component architecture meta-model for self-adaptive software. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan*, volume 2192 of *Lecture Notes in Computer Science*, pages 81–88. AITO, Springer-Verlag, September 2001.

[DCC01] Jim Dowling, Vinny Cahill, and Siobhàn Clarke. Dynamic software evolution and the K-Component model. In *Workshop on Software Evolution, OOPSLA 2001*, 2001.

[Dea07] Alan Dearle. Software deployment, past, present and future. *FOSE '07 : 2007 Future of Software Engineering*, pages 269–284, 2007.

[DeM03] Linda G. DeMichiel. Enterprise javabeans specification, version 2.1. Sun Microsystems Specification, November 2003.

[Dij82] E. W. Dijkstra. Ewd 447 : On the role of scientific thought. *Selected Writings on Computing : A Personal Perspective*, pages 60–66, 1982.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[DL06a] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive Fractal components. In *5th International Symposium on Software Composition (SC'06)*, Vienna, Austria, March 2006.

[DL06b] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[DLG+08] Pierre-Charles David, Marc Léger, Hervé Grall, Thomas Ledoux, and Thierry Coupaye. A multi-stage approach for reliable dynamic reconfigurations of component-based systems. In *Distributed Applications and Interoperable Systems (DAIS'08)*, 2008.

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath & fscript : Language support for navigation and reliable reconfiguration fractal architectures. *Annals of Telecommunications - Special issue on Software Components - The Fractal Initiative*, 2009.

[DS95] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the International Conference on Functional Programming (ICFP'95)*, pages 262–273, 1995.

[DvdHT01] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In David Garlan, Jeff Kramer, and Alexander L. Wolf, editors, *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM.

[EJB] EJB Specification. *http ://java.sun.com/products/ejb/.*

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK : A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, 2002.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10) :46–54, 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Professional Computing Series. Addison-Wesley, October 1994.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, third edition, 2005.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3) :249–259, 1987.

[GMUW00] H Garcia-Molina, J Ullman, and J Widom. *Database System Implementation.* Prentice Hall, New Jersey, 2000.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme : architectural description of component-based systems. In *Foundations of component-based systems*, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : The Language and Its Implementation.* Addison-Wesley, 1983.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering : Putting the Pieces Together (ACM Press).* Addison-Wesley Professional, June 2001.

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts, February 1995.

[HW04] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 594–603, May 2004.

[Jac02] Daniel Jackson. Alloy : a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2) :256–290, 2002.

[JAS] JASMINe Project. *http ://wiki.jasmine.objectweb.org/xwiki/bin/view/Main/WebHome.*

[Jav] Java EE 5 Specification. *http ://java.sun.com/javaee/technologies/javaee5.jsp.*

[JBCG05] Ackbar Joolia, Thais Batista, Geoff Coulson, and Antonio Tadeu A. Gomes. Mapping adl specifications to an efficient and reconfigurable runtime component platform. In *WICSA '05 : Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 131–140, Washington, DC, USA, 2005. IEEE Computer Society.

[JBSK04] Borka Jerman-Blazic, Wolfgang Schneider, and Tomaz Klobucar. *Security And Privacy In Advanced Networking Technologies (Nato Science Series/ Computer and Systems Sciences)*. IOS Press, Inc., 2004.

[JOn] JOnAS Project. *http ://wiki.jonas.objectweb.org/xwiki/bin/view/Main/WebHome*.

[JTA] JTA Specification. *http ://java.sun.com/products/jta/*.

[KBC02] Madjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Adapting applications on the fly. In *17th IEEE International Conference Automated Software Engineering 2002*, Edinburgh, UK, September 2002. IEEE & ACM.

[KC03] Jeffrey Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1) :41–50, January 2003.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, chapter Chapter 5 and 6. MIT Press, 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem : Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11) :1293–1306, 1990.

[KM07] Jeff Kramer and Jedd Magee. Self-managed systems : an architectural challenge. In *Future of Software Engineering (FOSE'07)*, pages 259–268, Washington, DC, USA, May 2007. IEEE Computer Society. in conjunction with ICSE'07.

[lap92a] *Dependability : A Unifying Concept for Reliable, Safe, Secure Computing*, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[Lap92b] Jean-Claude Laprie. *Dependability Basic Concepts and Terminology*. Springer Verlag, Vienna, Austria, 1992.

[LCL06] M. Léger, Thierry Coupaye, and T. Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors, *Langages et modèles à objets (LMO'06)*, pages 21–36. Hermès Lavoisier, 2006.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21 :336–355, 1995.

[LLC07] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the Fractal component model. In *Proceedings of the 6th workshop on Adaptive and reflective middleware (ARM'07)*, page 6, New York, NY, USA, 2007. ACM.

[LOQS07] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. pages 209–219, 2007.

[LP76] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *ICSE '76 : Proceedings of the 2nd international conference on Software engineering*, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155, New York, USA, 1987. ACM SIGPLAN, ACM Press.

[MBC02] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Formaware : Framework of reflective components for managing architecture adaptation. In Alberto Coen-Porisini and André van der Hoek, editors, *SEM*, volume 2596 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2002.

[MBC03] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Constraining architectural reflection for safely managing adaptation. In *Middleware Workshops*, pages 139–143. PUC-Rio, 2003.

[MBC04] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with FORMAware. In *ICDCSW '04 : Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[McI68] Douglas McIlroy. *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, chapter Mass-Produced Software Components, pages 138–155. Scientific Affairs Division, NATO, Garmisch, Germany, October 1968.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[Med96] Nenad Medvidovic. Adls and dynamic architecture changes. In *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*, pages 24–27, New York, NY, USA, 1996. ACM Press.

[Med99] Nenad Medvidovic. A language and environment for architecture-based software development and evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53, 1999.

[MEH04] Mark W. Maier, David Emery, and Rich Hilliard. Ansi/ieee 1471 and systems engineering. *Syst. Eng.*, 7(3) :257–270, 2004.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992.

[Mit00] R Mitchell, S. *Dynamic Configuration of Distributed Multi-media Components.* PhD thesis, University of London, 2000.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96 : Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[Mon01] Robert T. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, January 2001. Originally published October, 1998.

[Mos81] E. B. Moss. Nested transactions : An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.

[MS08] Philippe Merle and Jean-Bernard Stefani. A formal specification of the fractal component model in alloy. Technical report, INRIA Research Report, 2008.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[.NE] .NET Specification. *http ://www.microsoft.com/net/.*

[OCL05] OCL 2.0 Specification. *http ://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan, April 1998.

[Ore98] Peyman Oreizy. Issues in modeling and analyzing dynamic software architectures. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, Sicily, Italy, June 1998.

[OSG] OSGi Specification. *http ://www.osgi.org/Specifications/HomePage.*

[Pro02] Marek Procházka. Advanced transactions in component-based software architectures. Technical report, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranske namest i 25, 118 00 Prague 1, Czech Republic, 2002.

[PSDC08] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Thierry Coupaye. A component-based and aspect-oriented model for software evolution. In *Int. J. Comput. Appl. Technol.*, volume 31, pages 94–105, Geneva, Switzerland, 2008. Inderscience Publishers.

[Que05] Vivien Quema. *Vers l'exogiciel : une approche de la construction d'infrastructures logicielles radicallement adaptables.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, December 2005.

[RHMRM04] Roshanak Roshandel, André Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering Methodologies*, 13(2) :240–276, 2004.

[SCA] SCA Specification. *http ://www.ibm.com/developerworks/library/specification/ws-sca/.*

[SF96] Jonathan M. Sobel and Daniel P. Friedman. An introduction to reflection-oriented programming. In *Proceedings of Reflection'96*, 1996.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *11th annual ACM Symposium on Principles of programming languages*, pages 23–35, Salt Lake City, Utah, USA, 1984.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TGGL82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3) :323–342, 1982.

[TJ06] Pierre Wadier Tahar Jarboui, Marc Lacoste. A component-based policy-neutral authorization architecture. In *Conférence Française sur les Systèmes d'Exploitation (CFSE)*, 2006.

[TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for gui software. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering*, pages 295–304, New York, NY, USA, 1995. ACM.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, June 2000.

[Wer99] Miguel Alexandre Wermelinger. *Specification of Software Architecture Reconfiguration.* PhD thesis, Universidade Nova de Lisboa, 1999.

[WHW+04] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004*, pages 2–9, Los Alamitos, CA, USA, May 2004.

[Wor07] World Wide Web Consortium. XML path language (XPath) version 2.0. W3C Recommendation, January 2007. `http ://www.w3.org/TR/xpath20/`.

[WSO01] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the corba component model. pages 557–571, 2001.