N° d'ordre 0118

THÉSE / UNIVERSITÉ DE BRETAGNE SUD

Sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : Informatique

Présentée par

Franck Chauvel

Laboratoire : EA 2593
VALORIA – Equipe ALCC
Ecole Doctorale : Bretagne Sud

Méthodes et outils pour la conception de systèmes logiciels auto-adaptatifs

soutenue le 19 Septembre 2008, devant la commission d'examen

Composition du jury :

M. Jacques Malenfant / Président
Mme Isabelle Borne / Directeurs de Thèse
M. Jean-Marc Jézéquel
M. Jacques Malenfant / Rapporteurs
M. Lionel Seinturier
M. Thomas Ledoux / Examinateurs
M. Philippe Collet

Remerciements

Je remercie Mme Isabelle Borne et M. Jean-Marc Jézéquel pour m'avoir donné l'opportunité de faire cette thèse. Je les remercie de leur accueil, de leurs conseils, de leur confiance, et de leur soutien tout au long de ces 4 ans.

Je remercie ensuite M. Jacques Malenfant et M. Lionel Seinturier pour avoir accepté de rapporter cette thèse, ainsi que M. Thomas Ledoux et M. Philippe Collet pour avoir accepté de faire partie de la commission d'examen.

Je remercie naturellement tous les membres de l'équipe TRISKELL, du VALORIA et de l'équipe RAINBOW, et tout particulièrement ceux qui ont dû, de gré ou de force, partager un bureau ou une table avec moi, Olivier Defour, Erwan Drezen, Jean-Phillipe Thibault, Noël Plouzeau, Mohammed Belatar, Amed Saad, Cristina Vicente-Chicote, Mark Skipper, Julien DeAntoni et Alban Gaignard. On n'apprend jamais autant qu'en discutant. Je voudrais remercier spécialement, Olivier Barais pour ses conseils, ses remarques et le temps passé à réparer ma machine.

Je remercie également tous les membres du projet RNTL FAROS et spécialement ceux avec qui j'ai collaboré étroitement, Guillaume Waigner, Anne-Françoise Le Meur, Laurence Duchien, Michel Dao, Nicolas Rivierre, Jean Perrin, Bruno Traverson, Philippe Lahire, Philippe Collet, et Mireille Blay-Fornarino.

Je remercie aussi mes plus proches amis, Jean-Marie Hénaff, Christophe Maillard, Adrien Le Lann, pour toutes les parties de billard et de guitare.

Je remercie enfin mes parents, ma grande soeur, et mon amie pour tout le reste.

Table des matières

Chapitre 1

Introduction

Force est d'admettre que la technologie est omniprésente et incontournable aujourd'hui : la plupart des outils à notre disposition sont bardés d'électronique ou de systèmes informatiques. Une voiture de gamme moyenne intègre aujourd'hui par exemple plusieurs dizaines de processeurs contrôlant les essuie-glaces, les phares, les freins, les airbags etc. L'indice de pénétration de ces nouvelles technologies ne cesse de croître et là où la télévision aura mis vingt ans à se démocratiser, il n'en n'aura fallu que cinq à la téléphonie mobile.

Dans cette optique d'informatique ambiante, le développement de systèmes dits « ubiquitaires » ou pervasifs est incontournable. L'environnement de l'utilisateur est une multitude de systèmes logiciels et matériels enfouis et intégrés dans des objets familiers du monde réel : téléphone, montre, PDA, carte à puce, etc ... L'utilisateur a alors accès inconsciemment à tout un ensemble de services inter-connectés. En intégrant du même coup des systèmes distribués, hétérogènes, intelligents, le « saut technologique » associé à ces nouvelles technologies pose de nouvelles questions à la communauté du génie logiciel. Comment concevoir, développer, valider, déployer et maintenir de tels systèmes ?

Face à ces questions, la plupart des industriels a réagi de manière pragmatique en s'orientant vers des systèmes de plus en plus sophistiqués, capables de s'adapter, de se reconfigurer, voire même de s'auto-réparer dans certains cas. IBM a ainsi lancé en 2001 son *Autonomic Computing Initiative*, suivi par Hewlett Packard avec l'*Adaptive Enterprise Initiative* puis par Microsoft avec son *Dynamic System Initiative*. L'objectif de ces systèmes autonomes est de restreindre la configuration et la maintenance des systèmes à des objectifs de qualité ou de performance et de repousser dans les systèmes eux-mêmes les problèmes liés à l'hétérogénéité et à la distribution.

Les systèmes autonomes s'appuient sur le principe de base de l'adaptation : observation, décision, action. L'observation est nécessaire pour détecter les variations qui surviennent dans l'environnement (pris ici au sens large). La décision vise à sélectionner ou à concevoir une réaction à ces variations et l'exécution de cette même action doit tendre vers la satisfaction des objectifs prédéfinis.

Même si de tels systèmes ne sont pas encore une réalité tangible, leur conception reste une question de premier ordre pour le génie logiciel. Les travaux sur les systèmes sensibles au contexte intègrent des moyens d'observation et de prise en compte d'un environnement complexe et dynamique. Les recherches sur les systèmes auto-adaptatifs se focalisent sur les réactions à adopter face aux variations de l'environnement. Celles sur les systèmes auto-configurables ajustent la configuration en fonction de l'environnement. Chaque branche de la communauté des systèmes

« *auto-** » adresse un problème particulier.

La problématique de cette thèse est de proposer un environnement de conception et de développement pour les systèmes auto-adaptatifs qui prennent en compte l'observation, la décision et l'adaptation. Vis-à-vis des cycles de développement standards en génie logiciel, il s'agit de prendre en compte également les étapes de conception, de validation et de développement en proposant les outils appropriés pour ces trois étapes.

La contribution associée s'articule donc autour de ces trois étapes du cycle de développement et se concrétise dans l'outil TANGRAM. En terme de modélisation et de conception, TANGRAM permet de capturer l'observation, la décision et les réactions du système. L'observation est basée sur la réification de sondes extra-fonctionnelles dans les architectures à composants. La décision, basée sur l'utilisation de la logique floue, permet de décrire des systèmes de règles qualitatifs et composables. Enfin, les actions sont capturées par des procédures d'adaptation architecturales. TANGRAM offre également un outil de validation à travers la sémantique opérationnelle associée aux modèles architecturaux. Elle permet d'envisager leur simulation et donc une première validation *a priori* des systèmes auto-adaptatifs. Finalement, TANGRAM s'intègre dans la plate-forme Fractal sous la forme d'un contrôleur qui interprète directement les règles de décision dans l'architecture et permet ainsi d'obtenir un système auto-adaptatif exécutable.

A la suite de cette introduction, cinq chapitres détaillent cette contribution et les développements associés. Le chapitre 3 dresse un panorama des techniques existantes pour le développement de systèmes auto-adaptatifs. Ce panorama s'articule autour du processus d'adaptation, observation, décision, action en détaillant à chaque fois les principales techniques de conception, de validation, et de développement. Le chapitre suivant présente la manière dont le modèle TANGRAM capture les aspects spécifiques des systèmes auto-adaptatifs en se basant sur la réification de sondes extra-fonctionnelles ainsi que sur la logique floue pour décrire les règles d'adaptation. La chapitre 4 présente le modèle associé à l'outil TANGRAM et la sémantique opérationnelle qui lui est associée et qui sert de base à la validation *a priori* des systèmes auto-adaptatifs. Le chapitre 5 présente l'intégration des concepts de TANGRAM dans la plate-forme Fractal. Le chapitre 6 présente une étude de cas basée sur la conception, la validation *a priori* et le développement d'un serveur HTTP capable d'adapter son architecture et son comportement aux variations de son environnement. Finalement les deux derniers chapitres présentent respectivement quelques perspectives et concluent sur l'approche associée à l'outil TANGRAM.

Chapitre 2

Problématique et contexte

L'objectif de ce premier chapitre est de présenter un panorama de techniques existantes qui s'appliquent aux systèmes auto-adaptatifs. Les techniques présentées ici sont organisées selon trois axes : les techniques de conception et de modélisation, les techniques de vérification et de validation, et les techniques et les outils de mise en oeuvre.

2.1 Réalisation de systèmes logiciels classiques

Dès le début des années 70, le développement de larges systèmes logiciels est apparu comme un processus par étapes qui vise à construire un système conforme à ses spécifications. Pour mener à bien cette mission, différents cycles de développement ont été proposés tels que le cycle en cascade, le cycle en « V » et le cycle en spirale. Un des points communs entre ces différents cycles est qu'ils identifient généralement au moins trois phases : la conception, le développement, et la vérification et la validation. Cette section présente rapidement les grandes tendances actuelles de ces trois phases.

2.1.1 Conception et modélisation

Depuis le langage Simula-67 [29] qui a jeté les bases de l'approche orientée-objet, mais plus spécialement depuis le début des années 80, les langages à objets n'ont pas cessé d'évoluer tant sur le plan théorique que pratique. L'approche objet s'est vue appliquée dans de nombreux domaines : analyse, conception, base de données, etc. En 1997, l'*Object Management Group* a proposé un langage de modélisation unifié, UML (Unified Modeling Language [86, 85]) dédié à l'approche objet. Depuis sa première version, UML n'a cessé d'évoluer et de se complexifier, et intègre maintenant treize diagrammes permettant de capturer les différents aspects d'un système logiciel. L'approche objet, largement adoptée par la communauté industrielle par le biais d'UML, a contribué à répondre au problème de la réutilisation lors du développement, mais pas lors du déploiement ou lors de la maintenance.

Pour pallier ces problèmes récurrents, on a introduit la notion de « composant logiciel », inspirée des composants matériels et notamment de l'électronique. Un composant est une brique logicielle élémentaire pouvant être déployée et connectée avec d'autres pour former des briques plus complexes, à leur tour réutilisables. Là où un objet ne spécifie que les services qu'il fournit,

un composant spécifie également les services qu'il requiert de son environnement. Cette particularité, qui fait des composants des entités autonomes, se retrouve au centre de la définition donnée par Szyperski [106].

De nombreux travaux ont vu le jour pour formaliser les assemblages de composants sous le terme d'architecture logicielle, notamment avec les langages de description d'architecture (ADL) dont Medvidovic a donné une classification [72]. Les plus connus sont Wright [3, 2], Darwin [67], Rapide [66], ACME [42], π-ADL [16, 87], xADL [30], ABC/ADL [73], AADL [38] mais également UML qui intègre les concepts nécessaires pour représenter des architectures logicielles [71] et propose même un diagramme dédié. UML se différencie cependant des ADL car son objectif dépasse le simple cadre architectural : l'objectif est de capturer un système dans son ensemble (structure, données, comportement, etc). Toutefois, pour être utilisable dans un grand nombre de domaines d'application, la sémantique d'UML n'est pas complétement définie et comporte un certain nombre de points de variation sémantique, laissés à la charge de l'utilisateur. UML n'est donc pas un langage à proprement parler, mais plutôt un canevas pour des langages de conception.

A titre d'illustration, la figure 2.1 montre un exemple d'architecture logicielle décrit à l'aide d'un diagramme de structure composite issu d'UML. Il s'agit d'un système de capture vidéo permettant d'envoyer en temps réel une scène filmée par une caméra. L'architecture choisie pour ce système comprend :

- Une caméra (*CaptureDevice*), modélisée comme le périphérique d'acquisition, permet de capturer le son et l'image de la scène. Ces informations sont séparées en deux flux audio et vidéo distincts,
- Deux filtres (*AudioFilter* et *VideoFilter*) permettent de modifier ou de transformer les flux audio et vidéo : ajustement des couleurs, normalisation du son, etc.
- Un multiplexeur permet de combiner en un seul flux d'information les informations audio et vidéo
- Un émetteur transmet le flux d'information final sur le réseau en temps réel.

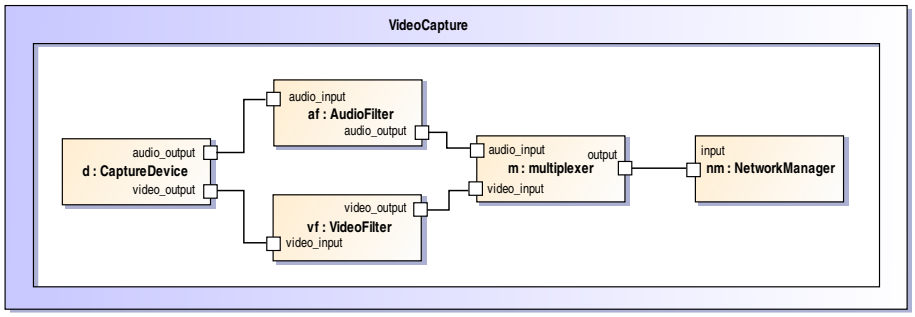

FIG. 2.1 – Une architecture possible pour un système de capture vidéo

Dans cette figure, les composants sont représentés par des boîtes connectées les unes aux autres. Chaque point de connexion ou port, est représenté par un carré qui le matérialise. L'un des avantages d'UML par rapport aux ADL est qu'il facilite la description de l'intérieur des composants (leurs données et leur comportement notamment) à l'aide des autres diagrammes disponibles : diagrammes de classes, de séquences, d'activités, d'états, etc.

2.1.2 Vers l'ingénierie des modèles

L'avènement de la programmation orientée-objet coïncide avec le développement des « intergicels » ou *middleware* en anglais. Conçus à l'origine pour résoudre des problèmes de distribution et d'interopérabilité entre systèmes et entre langages, ils sont devenus la technologie nécessaire pour faire communiquer des processus distants, puis des objets distants, et finalement pour déployer des architectures à base de composants. Dans la modèle en couche de l'OSI, ils interviennent juste au dessus de la couche transport, au niveau des couches session et présentation. Encore plus que les ADL, les intergiciels sont très nombreux et varient selon des besoins spécifiques (transactions, communication par messages, base de données, qualité de service, etc) sans pour autant être parfaitement compatibles. On parle alors de *middle-war* ou guerre du milieu en anglais.

Pour tenter de corriger ce problème, l'OMG a mis en avant un procédé de développement basé sur la notion de « modèle » qui vise à capitaliser sur les phases de conception, indépendamment du langage de modélisation utilisé. L'objectif de cette approche « orientée-modèle[1] » est de générer tout ou partie du code exécutable à partir d'un modèle initial, voulu purement logique et indépendant de toute technologie particulière. Ce processus s'applique particulièrement bien aux applications déployées sur des intergiciels, dont le code dépend généralement de celui qui est visé. Ce processus génératif est supporté par des transformations de modèle, qui permettent de transformer, de raffiner ou de compléter le modèle initial. Toutefois, l'approche orientée-modèle dépasse maintenant largement le simple cadre de la génération de code et s'applique à d'autres problèmes tels que le retro-conception (ou *refactoring* en anglais), la simulation, les langages métiers, etc. Sur l'exemple de la figure 2.1, cette approche nécessiterait d'intégrer progressivement des éléments de conception (héritage, association, appels externes) issus de l'intergiciel visé (CORBA, EJB, COM, etc) par transformations successives. Ainsi, en gardant le modèle initial intact, il devient possible de viser plusieurs plates-formes à l'aide de transformations adéquates et de résoudre, au moins partiellement, le problème des intergiciels.

Kermeta [77], par exemple, est une plate-forme libre dédiée à l'ingénierie des modèles. Kermeta est construit comme une extension du langage EMOF (Essential Meta-Object Facilities) [84]. Il s'agit d'un langage d'action permettant de spécifier la sémantique et le comportement associés à un métamodèle. Ce langage d'action est impératif et orienté-objets et est utilisé pour fournir une implantation des opérations définies dans un métamodèle. Kermeta est particulièrement adapté à la manipulation des modèles car il inclut à la fois des concepts objets et des concepts modèles.

Kermeta propose par exemple une notation graphique et une notation textuelle pour représenter les métamodèles utilisés dans la suite de cette thèse. Il permet d'implanter des algorithmes de transformation de modèles et offre également de nombreuses facilités : typage statique, héritage multiple, généricité, fermeture lexicale sur les collections, support des contrats et du langage OCL (Object Constraint Language). Enfin, Kermeta est compatible avec l'outil de méta-modélisation d'Eclipse EMF (Eclipse Modeling Framework) [13] ce qui permet d'utiliser Eclipse pour éditer, stocker et visualiser les modèles.

L'ingénierie des modèles a engendré divers modèles de composants supposés refléter les spécificités d'intergiciels particuliers tels que CORBA ou J2EE. Cependant, Lau et al. [63] en présentent une sélection allant de modèles généraux à des modèles visant des plates-formes propriétaires telles que Koala, ou Pecos par exemple.

[1] Communément appelée ingénierie des modèles, ou *Model Driven Engineering* en anglais

2.1.3 V & V : Vérification et Validation

Malheureusement, ces bonnes pratiques n'ont pas encore résolu la crise que le logiciel connaît depuis la fin des années 60. La conception de logiciels fiables passe nécessairement par des phases de vérification et de validation. La « vérification » [9] d'un système logiciel consiste à s'assurer de sa complétude, de sa cohérence, et de sa correction. Elle s'oppose dans la littérature à la « validation » qui assure que le système répond bien aux exigences des utilisateurs. De manière plus concise, la vérification s'assure que le système est bien construit, alors que la validation s'assure que le système construit est le bon. Regroupées sous l'appellation « V&V » elles forment un domaine trop vaste pour être résumé en quelques paragraphes et ceux qui suivent se cantonnent donc à la vérification.

Lorsque la vérification porte sur un modèle du système ou qu'elle ne nécessite pas l'exécution du système réel, on parle de « vérification statique ». De manière simple, il peut s'agir de relecture de code, mais également de vérifications automatiques ou semi-automatiques. De nombreuses méthodes de vérification existent, basées sur des graphes de contrôle, des graphes de flots de données, etc. La preuve de programme est également une solution pertinente pour s'assurer du bon comportement d'un système logiciel. La notion de modèle, présentée précédemment à propos de l'ingénierie des modèles, est une des techniques majeures pour intégrer et fédérer des vérifications formelles reposant sur des formalismes différents. A partir d'un modèle initial, UML par exemple, l'ingénierie des modèles permet de basculer (par transformation de modèles) vers des formalismes appropriés pour la vérification (automate, réseau de Petri, modèle markovien, etc), mais également d'exploiter et de formaliser les résultats issus de ces mêmes vérifications.

Par opposition, lorsque les outils de vérification nécessitent l'exécution du programme ou du système, on parle de « vérification dynamique ». L'idée est d'essayer de mettre en évidence des défaillances en testant le système avec des entrées pour lesquelles on connaît le résultat attendu. Pour cela, on tente de couvrir au mieux le domaine d'entrée en sélectionnant un certain nombre de données représentatives (cas limites, cas typiques, etc), puis on exécute le système avec ces mêmes données, et on compare le résultat (ou l'état du système) avec un oracle, qui connaît le résultat attendu. Une erreur est mise en évidence lorsque le résultat produit et l'oracle diffèrent, et une correction est alors nécessaire. Ce processus se répète jusqu'à ce que la confiance (généralement associée à un critère de couverture) soit suffisante. La figure 2.2 présente le principe général de la vérification, et les trois problèmes qui s'y rattachent, respectivement, la génération de données de test, la définition d'un oracle, et le critère d'arrêt.

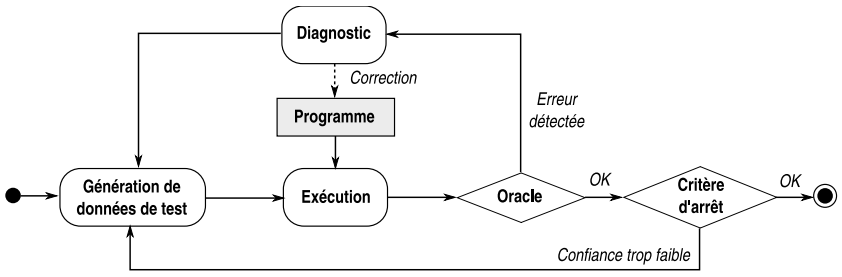

FIG. 2.2 – Principe général du processus de vérification dynamique

Cependant, l'exemple d'Ariane 5 [52] entre autres, a montré la nécessité d'embarquer une cer-

taine forme de vérification à l'exécution, les « contrats », pour garantir les interactions entre les composants et ainsi maximiser et sécuriser leur réutilisation. Cette notion de contrat, inspirée de la société moderne, reflète un accord entre plusieurs participants et définit les droits et les devoirs de chacun. Un contrat est une sorte d'oracle partiel utilisé à l'exécution pour détecter d'éventuels dysfonctionnements. Beugnard et al [8] distinguent quatre niveaux, allant de vérifications syntaxiques vers des vérifications de qualité.

Niveau syntaxique il s'agit de vérifier la cohérence syntaxique d'un système en s'assurant par exemple, que tous ses éléments sont bien définis ; il s'agit de techniques classiques liées à la compilation et à la vérification de type.

Niveau comportemental il s'agit de vérifier que le comportement attendu est bien celui qui est offert. On utilise par exemple, la vérification de pré et de post condition (Eiffel [74]).

Niveau protocolaire il s'agit de vérifier la coordination des systèmes, propriétés de sûreté, de vivacité, existence d'inter-blocages, etc. Ce type de vérification est généralement statique.

Niveau qualité il s'agit de vérifier des contraintes de qualité de services, ce qui englobe des problématiques de consommation de ressources, de temps, mais également de contraintes liées au domaine de l'application.

Il est important de noter que quel que soit le type de propriété, la vérification dynamique implique une capacité d'observation dédiée. La vérification du temps de réponse, par exemple, implique l'existence d'une horloge, celle de la mémoire libre d'une sonde mémoire, ou d'une technique de calcul quelconque.

2.2 Vers des systèmes auto-adaptatifs

2.2.1 Vers une définition commune

Avec l'avènement des systèmes mobiles qui sont confrontés à des environnements fluctuants, les systèmes « auto-adaptatifs » sont devenus un besoin et une réalité incontournables. Cependant, cette nouvelle branche du génie logiciel ne dispose pas encore de normes ou de standards, et il n'y a ni définition ni vocabulaire communs.

Cette section propose donc une définition qui permet de cerner les limites de notre étude, mais reflète également le regard que nous portons sur la conception et la réalisation des systèmes adaptatifs. Elle permet par ailleurs de définir le vocabulaire qui nous semble pertinent dans ce domaine. Cette définition est la suivante : un système « auto-adaptatif » est un système capable de modifier sa configuration ou sa structure interne pour prendre en compte les changements de son environnement et ainsi offrir un service d'une qualité optimale.

Le corps humain est un bon exemple de système adaptatif. Lorsque l'environnement change, celui-ci s'adapte pour préserver au mieux les fonctions vitales. Si la température augmente par exemple, le système de sudation se déclenche et maintient (dans une certaine mesure) une température interne acceptable. De manière plus concrète, le système de capture vidéo introduit précédemment peut intégrer de nombreuses adaptations. La qualité du flux vidéo par exemple (résolution spatiale, temporelle, couleurs, son, etc) peut être adaptée en fonction des ressources disponibles (bande passante, mémoire, CPU, etc) lors de la capture ou lors du rendu. Comme notre définition le stipule, l'objectif inhérent aux systèmes adaptatifs est de maintenir la qualité du service rendu quel que soit l'état de l'environnement. Il s'agit de stabiliser la qualité de service.

Trois capacités essentielles caractérisent un système adaptatif : l'*observation*, la *décision* et l'*intro-action*[2].

Observation Un système adaptatif est un système qui observe son environnement et en détecte les changements. D'un point de vue technique, cette capacité se traduit par l'existence de sondes (logicielles ou matérielles) permettant de mesurer les propriétés pertinentes de l'environnement.

Décision En fonction de l'état de l'environnement, mais également fonction de sa configuration actuelle (capacité d'introspection), un système adaptatif décide par lui-même de la nouvelle configuration à adopter pour fonctionner de manière optimale.

Intro-action Pour modifier sa propre configuration, le système adaptatif manipule et modifie les éléments qui le constituent. Ces « intro-actions » sont des actions *de* l'intérieur *sur* l'intérieur. Il s'agit d'une capacité d'introspection active.

Ces trois capacités reflètent également le cycle de l'adaptation : observation – décision – action à la base de l'automatisme et de la théorie du contrôle. Par ailleurs, la capacité d'intro-action inhérente à ces systèmes, implique un système à l'architecture clairement définie en terme d'éléments identifiés et interchangeables. Les architectures à composants sont donc prédisposées à supporter le développement de ces systèmes.

2.2.2 Trois générations de systèmes adaptatifs

L'un des principes de base du génie logiciel est l'anticipation. Concevoir un système, c'est, en effet, anticiper toutes les situations auxquelles le système devra faire face et anticiper le comportement relatif du système. En d'autres termes, un bon concepteur doit pouvoir anticiper à la fois les changements de l'environnement et les configurations du système. Or dans le cas des systèmes adaptatifs simples, la complexité de l'environnement engendre un très grand nombre de configurations possibles, qu'il est difficile d'anticiper et coûteux de concevoir séparément. L'idéal serait donc de concevoir un système qui infère lui-même les configurations à utiliser en fonction des variations de l'environnement. Plus généralement encore, il est difficile d'anticiper les états possibles de l'environnement et le système idéal doit inférer également les états critiques de son environnement et peut-être même inférer comment le mesurer.

Ce problème d'anticipation traduit l'existence de trois générations potentielles de systèmes adaptatifs, allant de systèmes statiques vers des systèmes inférant à la fois la configuration adéquate mais également les mesures pertinentes à effectuer sur l'environnement.

1. Les systèmes de première génération sont statiques, ou pseudo-adaptatifs. L'architecture d'un système statique n'évolue pas tandis que celle d'un système pseudo-statique supporte quelques évolutions implémentées de façon *ad hoc*. La plupart des systèmes temps réel actuels sont conçus de cette manière, par manque d'outils et de techniques appropriées.
2. Les systèmes de seconde génération sont capables d'inférer l'architecture la plus pertinente vis-à-vis de l'état de leur environnement. Ils intègrent pour cela un certain nombre de capteurs prédéfinis qui leur permettent de détecter les changements d'environnement. Certains systèmes mobiles commencent tout juste à intégrer ce type de fonctionnalités.

[2]Ce mot n'existe pas en français, mais s'oppose par construction, à l'introspection. On trouve parfois dans la littérature anglophone le terme *intercession* mais son équivalent français n'a pas réellement de rapport.

3. Les systèmes de troisième génération sont capables de découvrir un certain nombre de capteurs disponibles dans leur environnement, d'apprendre à les utiliser pour adapter au mieux leur architecture et satisfaire ainsi différents objectifs prédéfinis. Cette vision, qui relève de l'*autonomic computing* [55, 54] est celle vers laquelle tendent les systèmes multi-agents, les systèmes sensibles au contexte, les systèmes intelligents, etc.

Une nouvelle branche du génie logiciel, centrée sur les systèmes adaptatifs est donc en train de se dessiner. Elle intègre des technologies issues de différents domaines de l'informatique, principalement du génie logiciel et de intelligence artificielle mais également de l'informatique répartie. Il s'agit réellement d'une discipline émergente au sens premier du terme et une communauté scientifique commence à s'organiser. L'apparition d'ateliers tels SelfMAN (créé en 2005), M-ADAPT créé en 2007 et associé à la conférence ECOOP, MoDeLs@runtime créé en 2006 lors de la conférence MoDeLs, SEAMS créé en 2006 au sein de la conférence ICSE, ARAMIS et WCAT créés en 2008 lors de la conférence ASE, reflètent cette tendance. Des conférences ainsi que des journaux apparaissent également tels que SASO (1^{ere} édition en 2007) et TAAS (journal).

Comme nous allons le montrer par la suite, les systèmes produits actuellement sont des systèmes de première génération. La principale raison à cela est le manque d'outils dédiés à la réalisation de systèmes adaptatifs. Concevoir un système fiable, capable d'observer, de décider et de s'adapter n'est pas encore chose aisée. La suite de ce premier chapitre dresse un état de l'art sur ces trois problèmes que sont l'observation, la décision, et l'intro-action.

2.3 Observation et systèmes sensibles au contexte

2.3.1 Modélisation des données observables

S'adapter en fonction de l'environnement nécessite de le sonder pour connaître ses caractéristiques. Un système adaptatif inclut donc nécessairement un ensemble de moyens de mesure dédiés à son environnement. Avec l'apparition de systèmes mobiles et de l'informatique *ubiquitaire*, la notion de systèmes sensibles (ou *Context-aware System* en anglais) prend de plus en plus d'importance. Naturellement, la notion de « contexte » n'a pas de définition communément acceptée. Dey [35] présente un bref résumé des différentes définitions et propose : « *Le contexte correspond à l'ensemble des informations qui peuvent être utilisées pour caractériser la situation*[3] *d'une entité (logicielle, matérielle, humaine, etc)* ». Les systèmes sensibles sont clairement des systèmes auto-adaptatifs mais ont généralement pour objectif d'améliorer l'interface utilisateur, en observant son comportement.

Panorama des approches existantes

La solution la plus simple pour modéliser le contexte d'une application consiste à utiliser un ensemble de couples « clé-valeur ». Les différentes données observables sont alors accessibles via l'étiquette qui leur est associée. La simplicité de cette méthode a permis de mettre en place des systèmes efficaces de découverte dynamique de services [98] par exemple. Des modèles de données plus structurés ont également été proposés avec le *Composite Capabilities/Preferences Profile* du W3C et le *Resource Description Framework* (RDF) [62] conçus tous deux comme des schémas XML.

[3] A prendre ici, au sens large, de conjoncture, d'état, de circonstance, de situation

D'autres approches encore utilisent des modèles logiques pour alimenter directement des bases de faits exploitées par des systèmes experts (voir section 2.4.1). Ces systèmes utilisent les mécanismes d'inférence fournis par la logique sous-jacente pour dériver de nouveaux faits ou pour raisonner sur le contexte. On peut noter les travaux de McCarthy qui proposent une première modélisation formelle du contexte [70], ceux de Akman et Surav qui réutilisent la théorie des situations [1], ou ceux plus récents, de Kaenampornpan qui propose une théorie des activités [53].

Des approches orientées-objet ont également été proposées, à l'aide d'UML par exemple [101], mais également à l'aide du langage ORM (*Object Modeling Role*) [48] qui s'avère très efficace pour générer des modèles relationnels et les bases de données qui leur sont associées. Cependant, parmi les différents états de l'art sur les modèles de données contextuelles [102, 10, 4], les approches à base d'ontologies [103, 45, 92] semblent les plus utilisées.

Utilisation d'ontologies

A l'instar des modèles entité-relation ou des modèles objets, les ontologies décrivent les différents concepts d'un domaine et les relations qui les lient. Si la différence entre un modèle orienté-objet et une ontologie reste floue [58], les ontologies, issues de l'Intelligence Artificielle, sont fréquemment couplées à des systèmes de raisonnement automatique. Chen et al. [23] intègrent par exemple dans leur intergiciel *CoBrA* un moteur leur permettant de raisonner sur les ontologies. L'objectif est de combiner une base de faits intégrée dans le système, aux données contextuelles mesurées par les capteurs pour maximiser les capacités de raisonnement. Cette approche, particulièrement utilisée pour la géo-localisation d'individus permet par exemple de confronter une localisation mesurée (i.e à l'aide de badge RFID) avec une localisation calculée (i.e à l'aide d'emploi du temps).

Une ontologie possible pour modéliser le contexte du système vidéo est représentée par la figure 2.3 à l'aide d'un diagramme de classe UML. Elle permet, entre autres, de capturer facilement les préférences de l'utilisateur en terme de qualité par exemple. Le système peut essayer de maximiser la dynamique du flux vidéo (nombre d'images par seconde) ou sa qualité (résolution des images). La figure 2.3 représente les données qui peuvent entrer en jeu lors de l'adaptation. Il est naturel d'y modéliser à la fois les ressources physiques telles que la mémoire, le CPU, ou la bande passante, mais également des ressources logiques à la disposition du système comme les composants logiciels disponibles par exemple. Surtout, adopter une approche de haut niveau, comme le suggère la littérature sur les systèmes sensibles, permet de capturer directement des éléments tels que les intentions de l'utilisateur, ou celles de l'administrateur dans le cas de notre service de capture vidéo.

Il faut admettre cependant que la modélisation des données n'est plus réellement un problème difficile, mais qu'il faut maintenant être capable d'exploiter ces données et de les interpréter. C'est d'ailleurs, l'un des manques des approches orientées-objet pourtant avancées en terme de modélisation : elle ne permettent pas de raisonner sur les données. A l'inverse, le monde des ontologies, plus mature sur le raisonnement, n'a pas encore convergé en terme de modélisation.

2.3.2 Vérification et validation de systèmes sensibles

Comme nous l'avons évoqué précédemment, l'objectif des systèmes sensibles est de tenir compte des fluctuations de l'environnement. La nécessité d'approfondir cette connaissance de l'environnement implique d'intégrer continuellement de nouveaux capteurs distants spécifiques

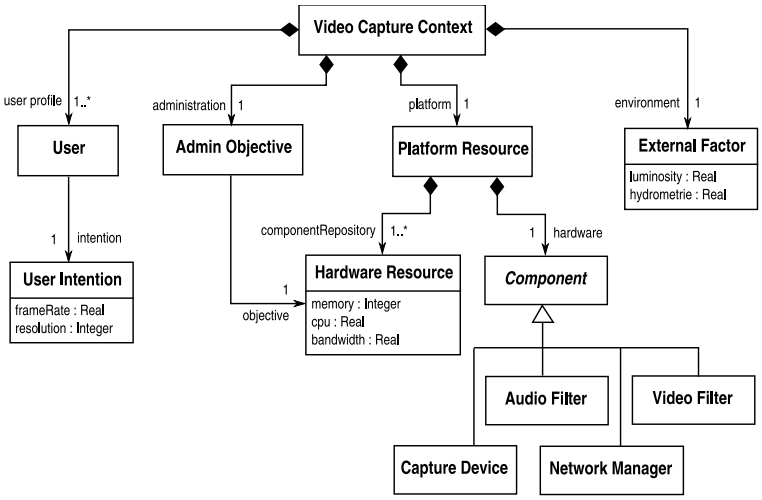

FIG. 2.3 – Ontologie décrivant les données contextuelles décrites à l'aide du langage UML

(géolocalisation, préférences de l'utilisateur, position dans l'espace, temps, etc). L'observation d'un environnement complexe et dynamique est donc devenu un problème à part entière nécessitant ses propres techniques de validation et de vérification. Comment fournir une description de l'environnement cohérente, correcte et complète lorsque les informations sont issues depuis de multiples capteurs distants et généralement redondants ? Naturellement, du fait de la relative jeunesse des techniques d'acquisition et d'agrégation de données contextuelles, les techniques de test dédiées sont encore peu nombreuses.

Xu et Cheung [115] répertorient différents problèmes possibles qui peuvent amener à la découverte d'anomalies dans les données contextuelles.

- La nature même de certaines mesures les rend très rapidement obsolètes. La pertinence de la localisation d'un individu par exemple dépend naturellement de sa vitesse de déplacement.
- La redondance des mesures et leur confrontation peuvent faire surgir des problèmes sémantiques : unités différentes, sémantiques différentes, etc.
- La distribution des sources d'informations sur un réseau dynamique implique des données contextuelles potentiellement partielles. La précision d'une localisation GPS dépend par exemple d'une bonne communication avec les satellites : si celle-ci est inexistante alors l'information de localisation n'est pas disponible.

Panorama des approches existantes

Xu *et al.* [115, 116, 117] proposent une méthode statique de détection d'incohérences dans des données contextuelles. A l'aide de la logique du premier ordre, ils modélisent les données contextuelles comme un ensemble de données spatiales, temporelles, et descriptives. Ils décrivent alors des opérateurs d'équivalence et d'inclusion qui leur permettent de décrire des règles de cohérence. Ces règles sont ensuite utilisées pour vérifier la cohérence d'un contexte donné.

Dans une perspective de vérification dynamique, Tse *et al.* [108] utilisent le test métamorphique pour vérifier les systèmes sensibles. L'idée du test métamorphique [120] est de lier de manière sémantique une série de données de tests. A partir d'une donnée de test qui n'a pas néces-

sairement provoqué d'erreur, on génère une autre donnée liée à la première et l'on vérifie la validité d'une propriété métamorphique prédéfinie sur les deux résultats. Par exemple, pour tester une fonction périodique f, on peut, à partir de n'importe quelle donnée x, tester $f(x + T)$ où T est la période, et ainsi vérifier que l'on obtient bien le même résultat. Appliquée au test des systèmes sensibles, cette approche fait varier les données contextuelles en entrée et vérifie la réaction du système.

Wang *et al.* [114] automatisent la génération de données de test pour les systèmes sensibles. Leur approche, structurelle, détecte dans le code les points où les données contextuelles sont utilisées pour déclencher des traitements particuliers (*Context Aware Program Point*). Ces CAPP sont supposés isolés dans des *Context Handler*, c'est-à-dire des méthodes dédiées à la gestion du contexte. A l'aide d'un graphe de flot de données incluant les CAPP, ils génèrent des cas de tests qui en sensibilisent un maximum. Cette approche semble la plus prometteuse actuellement en terme de test dynamique mais repose sur l'hypothèse forte de la présence de *Context Handler* qui n'est pas vraie dans le cas de systèmes intelligents.

Illustrations des techniques de V & V liées aux systèmes sensibles

L'apparition d'une incohérence dans les données contextuelles est souvent liée à la confrontation de mesures physiques et logiques. Par exemple, dans le cas du système vidéo, deux techniques peuvent être utilisées pour mesurer la luminosité. D'un coté, on peut utiliser un capteur dédié qui mesure de manière précise la luminosité ambiante, de l'autre on peut l'évaluer plus grossièrement en fonction de l'heure courante. L'utilisation conjointe de ces deux solutions peut amener une contradiction si une lumière artificielle éclaire la scène capturée. Ce genre de contradiction peut être facilement éliminé par agrégation et filtrage des données dans les cas simples. Cependant, dans les cas où le contexte implique des données plus complexes (topologie, localisation, etc), l'utilisation de techniques citées précédemment s'impose.

Le problème le plus courant est celui de la dynamicité de l'environnement qui rend incomplet sa description. Dans le cas de la capture vidéo si le système a à sa disposition une description de la qualité perçue par l'utilisateur (nombre d'images par seconde ou *frame rate* en anglais), cette information lui est fournie par son client et n'est pas calculée localement. Si une défaillance survient chez ce dernier et que cette information n'est plus émise, alors les données contextuelles du système sont incomplètes.

2.3.3 Observation et systèmes sensibles au contexte

De nombreuses plates-formes ont été proposées pour faciliter le développement de systèmes sensibles. Elles offrent un accès transparent aux différentes informations contextuelles, masquant par exemple l'accès à des sondes matérielles. La plupart exploite un modèle du contexte issu des approches présentées dans la section 2.3.1.

Panorama des approches existantes

Comme l'explique Chen [23] trois principales architectures sont utilisées pour l'acquisition de données, allant de l'accès direct aux sondes matérielles à des réseaux de capteurs dédiés en passant par l'intégration de services dédiés dans des intergiciels. Toutefois, à l'exception de COSMOS [27] peu d'approches intègrent les notions de composant et d'architecture.

Parmi les architectures les plus simples, le système WildCat [32] développé pour Fractal offre une interface de haut niveau permettant d'accéder de manière homogène à des capteurs matériels ou logiciels.

CoBrA [23] propose un intergiciel dédié à l'accès aux informations contextuelles qui masque en grande partie la complexité de tels accès. D'autres travaux existent tels que CORTEX [112] construit autour de la notion d'objet sensible ou SOCAM [46] qui est également un intergiciel centralisé dédié aux données contextuelles.

Parmi les approches distribuées, LeWYS [15] est une plate-forme de mesure et de surveillance développée pour les applications distribuées. Dédiée aux applications J2EE, elle permet de collecter des données hétérogènes. Cette approche ne permet pas de filtrer et de combiner les données contextuelles comme le fait Phoenix [95, 96] qui offre un langage de description des données observables incluant quelques opérateurs (variations, amplitude, etc.). *Context Toolkit* [97] utilise également une architecture distribuée, intégrant des agrégateurs et des filtres permettant de construire des données contextuelles complexes.

Agrégation de données contextuelles avec COSMOS

Une autre approche, basée sur Fractal a été proposée dans [27]. L'idée est de représenter l'acquisition de données contextuelles sous la forme d'une composition de sources de données. Cette approche repose sur l'utilisation du pattern *composite* et sur la capacité du modèle Fractal à partager des composants entre plusieurs composites. Assez naturellement, cette approche permet de définir ses propres sources et ses propres compositions de données, mais n'intègre pas une description de haut niveau des données, et reste donc très liée à l'implémentation.

La figure 2.4 présente le principe de COSMOS appliqué à notre exemple de capture vidéo. L'acquisition du contexte se présente en trois niveaux. Le premier représente l'information que l'on veut obtenir, ici, une métrique permettant de décider de la nécessité d'une reconfiguration. Le second niveau décrit comment interpréter les données de base, c'est-à-dire comment combiner les différentes métriques. Enfin, le dernier niveau montre les mesures physiques ou logiques faites par le système.

COSMOS n'intègre pas directement une notation graphique comme peut le laisser entendre la figure 2.4. Chacun des éléments est en fait un composant Fractal composé des éléments qu'il requiert et qui implémente si besoin la combinaison de plusieurs métriques. Cependant, en offrant un certain nombre de sondes et d'opérateurs de base, COSMOS intègre élégamment l'accès au contexte dans une plate-forme à composants.

2.4 Prise de décision automatique et systèmes experts

2.4.1 Des systèmes experts pour la prise de décision

Le cœur d'un système adaptatif est un moteur de décision, c'est-à-dire à dire un élément capable de décider des modifications à apporter à l'architecture en fonction des observations que les capteurs du système lui envoient. Cette capacité de raisonnement est cruciale, car elle influe directement sur la pertinence des adaptations effectuées. Cependant, alors que les techniques liées au raisonnement et à la décision automatique arrivent à maturité, les systèmes adaptatifs intègrent toujours des éléments développés de façon *ad hoc*.

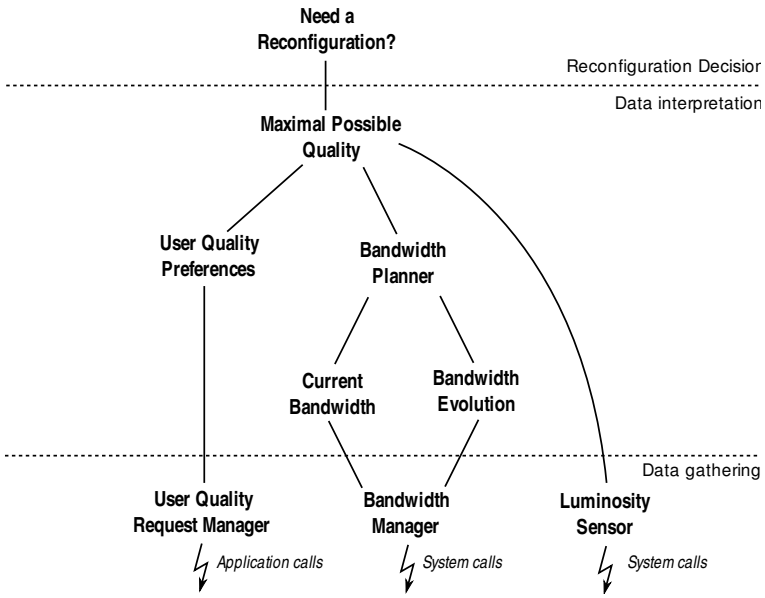

FIG. 2.4 – Modélisation des données et de leur interprétation pour la plate-forme COSMOS

Panorama des techniques de décision automatique

La capacité de prendre des décisions caractérise entre autre l'intelligence humaine, obligeant ainsi la communauté de l'Intelligence Artificielle à proposer des méthodes et des outils pour la décision automatique. Parmi ces différentes techniques, on peut citer les tables de décision et les approches basées sur des calculs d'utilité, les systèmes experts, les algorithmes de satisfaction de contraintes, les réseaux de neurones, les approches floues (contrôle et raisonnement), ou des combinaisons.

Les tables de décision [7] sont les systèmes les plus simples. Elles opposent structurellement (sous forme de tables) différents critères de décision à différents choix possibles. Utilisées dans le cas des systèmes adaptatifs, elles opposeraient ainsi l'état de l'environnement aux différentes architectures possibles, chaque case du tableau indiquant la pertinence (ou non) d'une architecture par rapport à l'état de l'environnement. Ces tables sont généralement couplées à des fonctions d'utilité qui permettent une analyse plus fine de la situation. Chaque décision possible se voit associer un certain nombre de conséquences (valuées) et il s'agit alors de maximiser l'utilité, calculée à l'aide des fonctions du même nom.

Une autre solution utilisée est le « système expert » [51] introduit dans les années soixante-dix aux Etats-Unis. Il s'agit de modéliser sous la forme de faits et de règles les connaissances utiles d'un expert. A partir de cette base de faits, le système utilise alors les règles pour produire de nouveaux faits ou pour prendre des décisions. La puissance d'un système expert dépend de la logique utilisée : logique des prédicats, logique des défauts, logiques modales non-monotones, etc. La principale difficulté est alors de capturer l'expertise et de la traduire sous forme de règles.

L'apprentissage à l'aide de réseaux de neurones est également utilisé pour résoudre des problèmes décisionnels. Il s'agit de faire « apprendre » au système un certain nombre de cas de base et de lui donner les moyens d'inférer d'autres cas, avec un certain degré d'erreur (que l'on veut

minimal). L'un des problèmes liés à ce type de solution, est qu'il est parfois délicat de comprendre *a posteriori* pourquoi le système a pris telle ou telle décision.

Logique floue et contrôle flou

Prenons le cas des approches dites « floues » et plus précisément du contrôle flou qui est utilisé par la suite. La notion de contrôle décrit généralement de manière naturelle des problèmes de commande, de pilotage ou de régulation, très présents en automatique. Contrôler la pression des freins d'une voiture pour en contrôler la vitesse ou ajuster la puissance d'un four pour maîtriser sa température sont de bons exemples. Plus généralement, il s'agit de contrôler un certain nombre de valeurs de sortie en fonction d'un certain nombre de valeurs en entrée. Une grande partie des techniques de contrôle utilisent des modèles mathématiques (équations différentielles par exemple) et sont donc parfaitement adaptées aux systèmes ayant un petit nombre de variables qui se prêtent facilement à une modélisation formelle. Le cas des systèmes adaptatifs nécessite de contrôler l'architecture logicielle en fonction de propriétés nombreuses et structurées et ne se prête donc pas aussi bien à ce genre de modélisation.

La logique floue a permis d'aborder ces problèmes moins intuitifs de manière élégante en proposant de modéliser le savoir-faire et l'expertise qu'un opérateur humain déploie dans ce genre de situation. L'une des particularité du raisonnement humain est son manque de précision : calculer la racine carrée de 23, par exemple, n'est pas aisé pour un humain mais est trivial pour une machine. A l'inverse, évaluer globalement la trajectoire d'un ballon pour l'intercepter est possible pour un humain, mais délicat pour une machine. L'idée de la logique floue est là : raisonner de manière globale et qualitative au lieu de raisonner de manière précise et quantitative.

Pour raisonner de manière qualitative, il faut décrire les qualificatifs associés aux grandeurs que l'on manipule ou sur lesquelles on raisonne. Pour cela, la logique floue utilise la notion « d'ensembles flous » dont l'appartenance est graduelle par opposition aux ensembles classiques dont l'appartenance est stricte. En logique floue, un élément peut appartenir « plus ou moins » à un ensemble. Ce degré d'appartenance est représenté par une fonction (d'appartenance) nommée μ généralement. La figure 2.5 représente ainsi les trois ensembles flous(nommés mauvaise, correcte, excellente), choisis pour qualifier la bande passante dans le cas du système vidéo. Chacun de ces ensembles est défini par une fonction d'appartenance de telle sorte qu'une bande passante de 6000 ko/s sera considérée comme excellente à 20% et correcte à 60%.

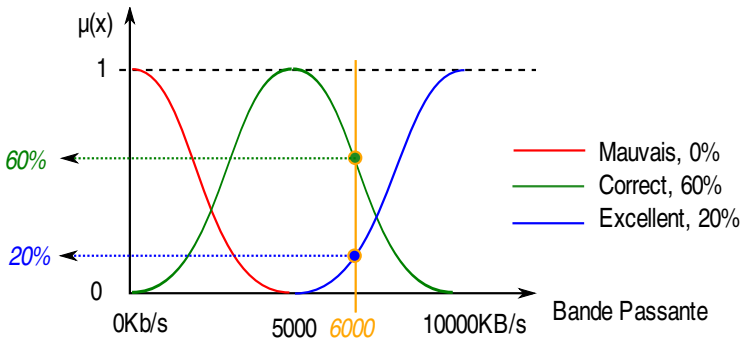

FIG. 2.5 – Définition d'une terminologie floue

A l'aide de ces ensembles flous, il devient alors possible de décrire une opération de contrôle sous la forme de règles simples telle que : « *Si la bande passante est mauvaise alors la résolution des images est petite* ». La logique floue offre également un mécanisme pour exploiter de telles règles de décision qualitatives. Ce mécanisme repose sur le principe du contrôle flou et s'articule en trois étapes distinctes :

La fuzzification Dans le cas des règles floues, il s'agit de mesurer le degré d'appartenance d'une variable à un ensemble flou. On s'intéresse à la partie gauche des règles : pour la règle 1 par exemple, on va calculer le degré d'appartenance de la propriété « *bandwidth* » à l'ensemble représenté par le terme « *medium* ». La figure 2.6 explique cela de façon graphique : la charge réelle est mesurée à 300 ko/s et la fonction d'appartenance indique que cette bande passante est donc » *medium* » à 45%.

L'inférence floue Cette notion consiste à considérer que le degré d'appartenance mesuré dans la partie gauche d'une règle peut être réutilisé tel quel dans sa partie droite. Par exemple, puisque la fuzzification de la règle 1 a établi que la charge est médium à 45%, le principe d'inférence floue indique donc que le taux de compression sera « *medium* » à 45% également.

La défuzzification Une fois que la fuzzification et l'inférence floue ont été appliquées sur toutes les règles, chaque propriété peut prendre plusieurs valeurs floues (ou degré d'appartenance). Pour les deux règles utilisées dans la figure 2.6, le taux de compression est donc « *medium* » à 45% (règle 1) et « *weak* » à 20% (règle 2). Pour calculer la valeur réelle correspondante (une quantité de mémoire), on agrège les deux aires correspondantes, et on calcule la projection du centre de gravité de l'aire résultante sur l'axe correspondant. On trouve donc un taux de compression de 41.23 dans l'exemple de la figure.

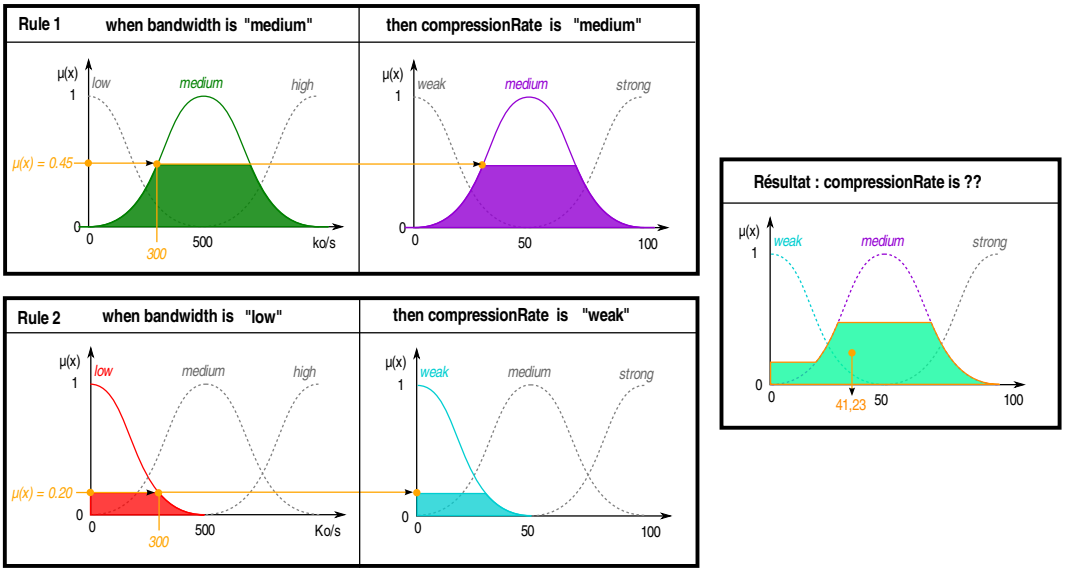

FIG. 2.6 – Processus d'inférence floue appliqué à un ensemble de règles

2.4.2 Vérification de systèmes experts

Comme nous l'avons évoqué dans la section 2.4.1, un système expert raisonne automatiquement sur des faits à l'aide de connaissances de base et de règles qui lui permettent de déduire de nouveaux faits. Ces connaissances de base ainsi que les règles de déduction, ne sont pas fournies par l'utilisateur, mais font partie intégrante du système et sont donc directement responsables de sa qualité, au même titre que l'algorithme de raisonnement sous-jacent. Du point de vue de la vérification, il est donc nécessaire de tester également ces connaissances pour s'assurer de leur pertinence, et par la même des conclusions produites par le système. Une erreur détectée peut être causée par une incohérence dans les règles, ou une erreur dans l'algorithme utilisé.

Le problème de la génération de données de test ne concerne donc que les données à la portée de l'utilisateur, c'est-à-dire les faits de base mais ne diffère pas réellement du problème classique. Le domaine des données d'entrée est naturellement très vaste mais c'est souvent la difficulté d'exprimer un critère de validité qui rend leur génération difficile.

Le problème de l'oracle est lui particulièrement délicat dans le cas des systèmes experts et est lié à leur utilisation. En règle générale, l'utilisation d'un système expert se justifie face à des problèmes clairement non triviaux : problèmes multi-critères, problèmes non linéaires, diagnostic, planning, etc. L'objectif est de capturer l'expertise d'un humain sous la forme de connaissances et de laisser l'algorithme résoudre le problème. La contre-partie de cette apparente facilité est que tout repose sur la modélisation des connaissances, qui, si elle est erronée, aboutira à des résultats généralement plausibles, mais sous-optimaux ou inefficaces. Du point de vue de la vérification, l'expression d'un oracle pour un système intelligent est un problème délicat : l'utilisation des solutions issues de l'IA étant justifiée par l'absence de solutions triviales. Il y a là, une sorte de paradoxe qui oppose IA et procédure de vérification.

Le problème du critère d'arrêt est également important. Dans le cas des systèmes experts, il est généralement lié à la couverture des règles ou des faits.

Panorama des techniques existantes

Pour certains systèmes, et notamment les systèmes d'aide au diagnostic médical, il a été évidemment impossible de se passer d'une procédure de vérification et de validation. La littérature sur le sujet propose plusieurs états de l'art pertinents [81, 56, 107] qui permettent d'identifier les principales approches.

Les vérifications statiques d'une base de règles aboutissent à la vérification de trois propriétés : la cohérence, la complétude et la correction. La cohérence garantit que les différentes règles du système se réfèrent à un ensemble précis et fermé. Il s'agit principalement d'un problème syntaxique. La complétude, elle, garantit que pour un problème donné, il n'existe pas de situation que le système ne saura résoudre par manque de règles. La correction, finalement, vérifie que le système ne comporte pas un ensemble de règles erronées, de règles contradictoires, de règles circulaires, de règles inclusives, etc.

Les méthodes dynamiques sont généralement liées à la validation d'un ensemble de règles. Comment s'assurer qu'un ensemble de règles cohérent, complet et correct répond bien au problème posé ? Parmi les méthodes possibles, on peut tout d'abord confronter le système à des cas connus de la littérature, ou à des cas pré-établis par des experts humains. Ces cas jouent alors directement le rôle d'un oracle. Cette approche rejoint le test de Turing [109] qui consiste à utiliser un troisième expert pour ordonner, selon leur pertinence, des solutions issues à la fois du sys-

tème expert et d'un expert humain. On peut également définir un métamodèle des règles utilisées c'est-à-dire un ensemble de super règles traduisant les grands principes de décision. On peut enfin utiliser une approche statistique pour détecter d'éventuels problèmes dans la répartition et l'utilisation des règles.

Illustration des problèmes de V&V dans les systèmes experts

Prenons le cas de notre système de capture vidéo « intelligent » et du calcul du taux de compression (*Compression Rate*) en fonction de la bande passante disponible (*bandwidth*). Chacune de ces deux grandeurs peut être qualifiée à l'aide des termes *low*, *medium*, *high*, on propose donc les deux règles suivantes :

1. bandwidth is low $\longrightarrow$ compression rate is high
2. bandwidth is low $\longrightarrow$ compression rate is medium
3. bandwidth is high $\longrightarrow$ compression rate is low

On remarque que le règles 1 et 2 se contredisent et rendent incorrect cet ensemble de règles. De plus, rien n'est spécifié quant aux valeurs moyennes de la bande passante et l'ensemble n'est donc pas complet. Pour qu'il soit complet et correct, il faut par exemple remplacer la règle 2 par « bandwidth is medium $\longrightarrow$ compression rate is medium ». Le nouvel ensemble des règles ainsi obtenu décrit finalement que le taux de compression doit évoluer à l'inverse de la bande passante. Cette relation d'évolution inverse reflète le sens général voulu et peut être capturée dans un métamodèle utilisé ensuite pour de la validation. Par exemple, ce métamodèle, appliqué à l'ensemble de règles initial permet de détecter que la règle 2 ne traduit pas l'évolution inverse souhaitée.

2.4.3 Support de la décision à l'exécution

L'intégration de moteur de décision dans les intergiciels n'est pas encore chose courante. La plupart des plate-formes pour les systèmes sensibles utilise un mécanisme de « *call back* » pour déclencher les procédures d'adaptation prédéfinies lors de dépassements de seuil. A l'exception des plus novatrices, elles n'intègrent pas encore de moteur de décision à proprement parler.

Panorama des approches existantes

Garlan *et al.* [41] proposent Rainbow, une plate-forme d'exécution où les décisions d'adaptation sont prises sur un modèle architectural du système en cours d'exécution. Ce modèle est tenu à jour par un lien causal qui répercute les modifications réelles de l'architecture sur le modèle et répercute les modifications du modèle sur la plate-forme. Le mécanisme de décision repose sur un principe de violation/réaction associé à des invariants architecturaux. Chaque invariant est associé à une réaction (ou stratégie) décrite de manière impérative sur le modèle. A l'exécution, lorsque le système détecte la violation d'un invariant il déclenche l'exécution de la stratégie associée, et le lien causal entre le modèle et la plate-forme répercute les modifications sur l'application réelle.

De le même façon Batista et al. [6] tentent de combler l'écart entre les ADL et les intergiciels. Ils proposent un environnement d'exécution nommé Plastik qui permet de reconfigurer la plate-forme OpenCOM à partir de descriptions issues de l'ADL ACME/Armani [75]. A partir d'un style architectural prédéfini et décrit dans ACME/Armani, plusieurs configurations peuvent être dérivées et répercutées sur la plate-forme. Les reconfigurations sont déclenchées lors de l'évolution

de propriétés mesurées sur la plate-forme, mais Plastik, comme Rainbow, n'intègre pas de moteur de raisonnement à proprement parler.

QualProbes [65] est également un intergiciel doté d'une capacité de décision. Il intègre un moteur de logique floue qui lui permet de configurer dynamiquement les paramètres de qualité des applications qui y sont déployées. Ainsi, chaque application explicite les règles qui contrôlent ses paramètres de qualité et l'intergiciel se charge d'exécuter ces règles et de calculer des valeurs pertinentes pour les paramètres en fonction des ressources disponibles. L'idée développée dans QualProbes s'avère tout à fait pertinente mais ne prend pas en compte les reconfigurations dynamiques en terme architecturaux (ajout et suppression de composants ou de connecteurs).

La robotique développe de nombreux intergiciels spécifiques tels que Miro [60, 110] qui intègre des moteurs de décision. Cependant comme dans le cas de QualProbes, ces modules de contrôle ne sont utilisés que pour paramétrer des propriétés locales (vitesse, rotation, puissance), et ne sont pas utilisés pour contrôler l'architecture du système.

MADAM : Reconfiguration dynamique à l'exécution

Floch *et al.* [39] proposent MADAM un intergiciel permettant de reconfigurer dynamiquement un système à l'exécution. MADAM intègre un moteur de décision basé sur un calcul d'utilité qui lui permet de choisir la meilleure configuration possible en fonction de l'environnement. L'idée de base de MADAM est de décrire une collaboration de composants comme une connexion entre des types de composants où chaque type peut être doté de plusieurs implémentations nommées « variantes ». Cette description en terme de types se rapproche de la notion de style architectural proposé par [14] mais n'est pas aussi expressive.

A chaque variante est associé un certain nombre de propriétés qui permettent à l'intergiciel de faire son choix parmi les variantes. Ces propriétés sont numériques et décrivent donc aussi bien des aspects techniques (mémoire, bande passante), que des aspects plus abstraits tels que la sécurité, la disponibilité, etc. Ces propriétés peuvent également être calculées en fonction du contexte par le biais de fonctions décrites de manière impérative. A l'exécution, l'intergiciel calcule une sélection optimale de variantes par rapport à l'environnement grâce à un calcul d'utilité sur les propriétés attachées aux différentes variantes.

Appliquée à l'exemple du système de capture vidéo, cette approche permet de définir différents filtres vidéo et audio, différents périphériques de capture et différents gestionnaires de réseau. A chacun de ces éléments sont associées des propriétés dérivées, calculées en fonction de l'environnement. Par exemple, pour le filtre vidéo, on peut spécifier son impact sur le temps de réponse et faire de même pour les autres variantes de l'application. La figure 2.7 présente l'architecture du lecteur vidéo associée aux variantes possibles et aux propriétés qui leur sont attachées.

A l'exécution le système calculera une sélection optimale des variantes, en se basant sur le calcul des propriétés spécifiées. Si l'approche est intéressante, elle reste figée dans le carcan des types de composant. Dans le cas du lecteur vidéo, il semble évident que le système doit être capable de sélectionner lui-même la bonne combinaison de filtres, et pas uniquement le filtre le plus pertinent.

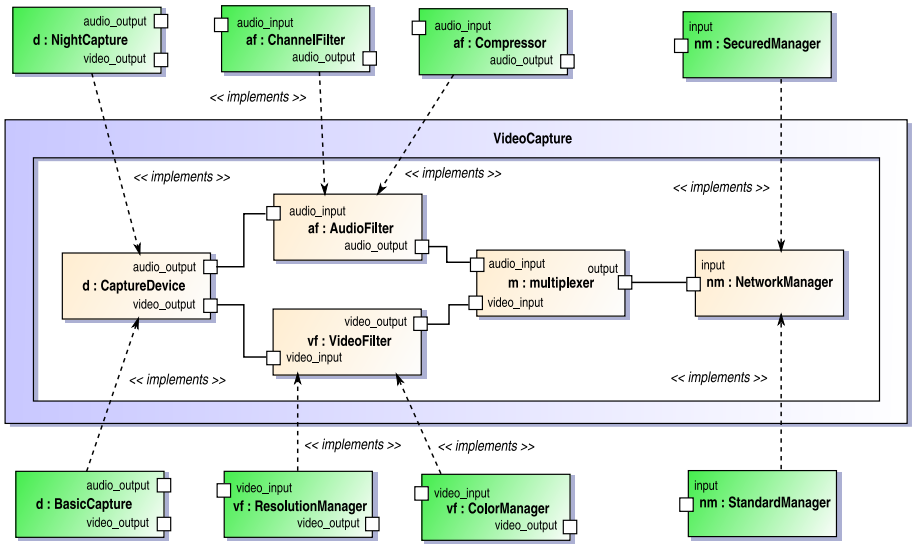

FIG. 2.7 – Approche de MADAM appliquée au système de capture vidéo

2.5 Architecture et systèmes intro-actifs

La troisième caractéristique des systèmes auto-adaptatifs est l'intro-action, c'est-à-dire la capacité, pour un élément donné, de modifier sa propre configuration ou sa propre structure. Cette section présente les techniques utilisées pour modéliser, valider et déployer les systèmes intro-actifs.

2.5.1 Modélisation de systèmes intro-actifs

La question de la modélisation des architectures logicielles a été abordée de deux manières. D'une part, les ADL ont proposé des langages dédiés aux architectures logicielles. Généralement issus du paradigme à composants, ils permettent de décrire de manière graphique ou textuelle un système logiciel sous la forme d'un assemblage de composants inter-connectés. D'autre part, l'ingénierie des modèles s'est également penchée sur les architectures logicielles et propose différents « métamodèle à composants ». Il s'agit de capturer de manière abstraite (sans considération syntaxique) les concepts de base utilisés ici. ADL et métamodèles à composants sont, très proches, au moins sur les concepts qu'ils définissent.

Panorama des approches existantes

Barais propose dans sa thèse [5] une classification des ADL prenant en compte le support des reconfigurations dynamiques. Même si cette liste n'est pas exhaustive, elle illustre bien la tendance des ADL à ne décrire que les aspects statiques. Dans cette optique, Oreizy [88] présente un état de l'art des ADL dynamiques. Les métamodèles à composants sont catalogués par Lau dans [63] et la plupart ne supporte pas l'évolution dynamique. La tableau 2.1 récapitule les caractéristiques d'introspection et d'intro-action de ces principaux outils.

COM+ puis .NET [47] sont les modèles de composant initialement proposés par Microsoft. Les composants, ne sont pas réellement modélisés, mais sont plutot des objets dotés, par héritage, de capacités standards (introspection, persistance, etc). Les composants COM sont initialement des artefacts binaires et déployables sur la plate-forme *Windows* sous la forme de bibliothèques dynamiques répertoriées et accessibles dans la base de registre.

Sun Microsystems fournit, en complément du langage Java, un modèle de composant logiciel, les *JavaBeans*. Initialement conçus pour maximiser la réutilisation d'éléments graphiques, ce modèle a engendré une plate-forme d'exécution pour les composants métiers nommée *Enterprise JavaBeans* [34]. L'objectif est de fournir un support à l'exécution de composants côté serveur, portables et distribués. Les composants JavaBeans offrent des interfaces d'introspection permettant d'accéder et de découvrir des méta-données mais également des interfaces d'intro-action permettant de gérer le cycle de vie des composants.

CCM [83] est un modèle de composant standardisé par l'OMG. Dans le modèle de composant abstrait, un composant possède des attributs et expose à son environnement des services synchrones fournis ou requis (*facettes* et *receptacles*) ainsi que des sources et des puits d'événements (services asynchrones). Les composants CCM offrent également des interfaces d'introspection et d'intro-action qui permettent de contrôler et de reconfigurer dynamiquement les connexions entres les composants.

Le modèle Fractal [11, 12] est un modèle de composant hiérarchique. Concis, chaque composant expose à l'extérieur des interfaces métiers fournies ou requises mais également des interfaces dites de contrôle permettant de gérer des problématiques extra-fonctionnelles (configuration, cycle de vie, etc.). Ce modèle est décrit plus en détails dans la section 2.5.3 page 31.

SOFA [90] est un autre modèle de composant très proche de Fractal. Les architectures SOFA sont vues comme des assemblages hiérarchiques de composants primitifs ou composites lorsqu'ils sont constitués de composants et des connecteurs qui les relient. SOFA permet également de capturer une partie du comportement des composants sous la forme de protocoles et permet donc de vérifier les compositions de composants : propriété de sûreté, détection d'inter-blocages, etc. La reconfiguration dynamique dans SOFA est dédiée à la mise jour d'un composant lorsque de nouvelles versions sont disponibles.

Le modèle de composant Koala [111] a été proposé par Philips pour modéliser les systèmes logiciels embarqués dans les téléviseurs. L'objectif est de minimiser les coûts de développement en maximisant la réutilisation de composants développés en interne. Les composants Koala exposent également des interfaces fournies ou requises mais la connexion entre les composants peut être assurée par des modules dédiés, formant une sorte de « colle logique » entre les composants.

Pecos [80] est un modèle de composant dédié à la conception et à la mise en œuvre de systèmes embarqués. Pecos distingue trois types de composants : les composants passifs, qui s'opposent aux composants actifs car ils ne possèdent pas leur propre flot d'exécution et les composants événementiels dont le comportement est déclenché par des événements. Ces derniers sont utilisés pour modéliser des éléments matériels.

Pin [50] est un autre modèle de composants où chaque composant expose des « fiches » (*Pin* en anglais) à son environnement. Chaque fiche peut être requise ou fournie et décrit les données qui peuvent y transiter. Les comportements de composant ainsi que les connexions entre les composants se font à l'aide d'un algèbre de processus, ce qui permet de vérifier la validité des assemblages. Rien n'est fourni cependant pour spécifier des reconfigurations dynamiques dans un assemblage de composants.

Wright [3] a été un des premiers ADL ayant été proposé. Un composant en Wright est une unité de traitement abstraite localisée et indépendante dont la sémantique est un processus. La description d'un composant contient deux parties importantes : l'interface, c'est-à-dire les services exposés et la partie calcul, c'est à dire les processus qui décrivent le comportement. Wright a été étendu [2] pour pouvoir prendre en compte la dynamicité dans les architectures. La politique de reconfiguration est définie à l'aide d'un langage *ad hoc* basé sur la notion événement-condition-action.

ACME [42] a pour buts principaux de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADL, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADL. Les principaux concepts sont les composants, les connecteurs et les systèmes. Le composant représente l'unité de traitement ou de donnée d'une application. Le connecteur représente l'interaction entre les composants. Il s'agit d'un médiateur de communication qui coordonne les connexions entre composants. Le système représente la configuration d'une application, c'est-à-dire l'assemblage structurel entre les composants et les connecteurs. ACME ne fournit aucun moyen de spécifier de manière simple et claire la dynamique d'un système ; en effet, le système est décrit comme un ensemble d'instances de composants liées entre elles par des connecteurs.

AADL [38] est à l'origine défini par différents partenaires du domaine de l'avionique, mais il s'adresse dans les faits à tous les systèmes embarqués temps réel, et pas uniquement aux systèmes avioniques. AADL considère deux niveaux de description du composant. Le premier, le type, correspond à l'interface fonctionnelle du composant : ce qui est visible des autres composants. Le second, l'implémentation, décrit le contenu du composant. Le modèle est hiérarchique, le contenu d'un composant consiste à définir ses sous-composants, les connexions en son sein et les délégations. AADL permet de définir différent modes de fonctionnement (modes dégradés par exemple) mais ne permet pas réellement de décrire comment basculer d'un mode à l'autre.

π-ADL [16, 87] est un ADL dont la sémantique repose sur le π-calcul. Il est dédié aux architectures dynamiques et supporte donc les modifications architecturales. Plus de précision sont apportées dans la suite de ce chapitre.

Le tableau 2.1 reflète le principal défaut de ces approches. A l'exception de Fractal qui intègre un langage dédié, les reconfigurations dynamiques sont intégrées dans le code même du système sous la forme de primitives dédiées, ce qui ne facilite pas la réutilisation ainsi que la séparation des préoccupations.

Illustration des techniques de reconfiguration dynamique

Le cas de l'ADL π-ADL est intéressant car c'est l'un des rares à permettre de représenter à la fois les données, les traitements et la structure des architectures logicielles. Le code du système de capture vidéo utilisé pour décrire le système vidéo est présenté par le listing 2.1. Par mesure de simplicité, nous avons omis la description des comportements. Dans π-ADL la description des adaptations architecturales se fait par raffinements d'une architecture principale, comme le montre l'architecture *VideoCapture* qui décrit la différence avec l'architecture originale. Le déclenchement des modifications se fera alors dans le code, à l'aide d'alternatives principalement, ce qui n'est pas complètement satisfaisant du point de vue architectural.

Listing 2.1 – Modélisationdu système de capture vidéo à l'aide de π-ADL

```
1 component CatureDevice is abstraction(){
```

	Dynamicité	Introspection	Intro-action	Langage dédié
COM/COM+	Possible (code)			
CCM	Possible (code)			
EJB	Possible (code)	Oui (Java)	Oui	
Fractal	Primitives dédiées	Oui (Java)	Oui	FScript
SOFA/DCUP	Version Update			
Koala	Non supportée			
Pin	Non supportée			
Pecos	Non supportée			
Wright	Primitives dédiées			
ACME	Primitives dédiées			
AADL	Modes			
π-ADL	Primitives dédiées			
UML	Non supportée	Oui		

TAB. 2.1 – Support de la dynamicité dans les ADL et les modèles à composants

```
        connection audio_output is out (string)
        connection video_output is out (string)
    behaviour is { }
}
component AudioFilter is abstraction(){
        connection audio_input is in (string)
        connection audio_output is out (string)
    behaviour is { }
}
component VideoFilter is abstraction(){
        connection video_input is in (string)
        connection video_output is out (string)
    behaviour is { }
}
component multiplexer is abstraction(){
        connection audio_input is in (string)
        connection video_input is in (string)
        connection output is out (string)
    behaviour is { }
}
component NetworkManager is abstraction(){
        connection input is in (string)
   behaviour is { }
}
architecture VideoCapture is abstraction(){
   behaviour is compose {
                    captureD        is CatureDevice ()
   and              audioF          is AudioFilter ()
   and              VideoF          is VideoFilter ()
   and              multiP          is multiplexer ()
   and              NetworkM        is NetworkManager ()
    } where {
        captureD::audio_output unifies audioF::audio_input,
```

```
35          captureD::video_output unifies videoF::video_input,
            audioF::audio_output unifies multiP::audio_input,
37          videoF::video_output unifies multiP::video_input,
            multiP::output unifies NetworkM::input
39      }
    }
41
    architecture VideoCapture1 refines VideoCapture using
43  {
      connections separates videoF::video_output from multiP::video_input,
45    and connections separates captureD::video_output from videoF::video_input
    }
```

2.5.2 Validation de systèmes intro-actifs

Panorama des approches existantes

La plupart des ADL construits sur des bases formelles intègrent des mécanismes de vérification statique sur les architectures dynamiques. En fonction du formalisme sous-jacent au langage, différents types de vérification sont possibles, vérification de propriété de sûreté, de vivacité, détection d'inter-blocage, etc. De par l'aspect « modèle » des ADL et des modèles à composant, il semble que la vérification dynamique des intro-actions n'ait pas encore été explorée.

Du point de vue purement architectural, la validation et la vérification de la dynamique des architectures à composants se rapprochent des problématiques d'évolution du logiciel. L'évolution se rapporte généralement à une dynamique sur le long terme et concerne principalement l'ajout de nouvelles fonctionnalités. Différents travaux existent quant à la validation et à la vérification de l'évolution des architectures logicielles. Parmi ces travaux, plusieurs utilisent la notion de styles architecturaux [69, 14] pour contraindre les évolutions possibles. Cette approche est tout à fait valide dans le cas des systèmes dynamiques et peut être vue comme la définition d'invariants architecturaux de haut-niveau, les intro-actions devant respecter ces invariants.

Vérification d'architecture dynamique à l'aide de π-AAL

L'outil ArchWare inclut un certain nombre de langages permettant de modéliser les architectures logicielles. Parmi ceux là, π-ADL et π-AAL permettent respectivement de décrire et d'analyser les architectures. Le listing 2.2 présente le code π-AAL permettant de contraindre le style architectural *Pipes and Filters* qui s'applique au système de capture vidéo.

Listing 2.2 – Modélisation d'un style *Pipes and Filters* à l'aide de π-AAL

```
1  PipeFilter is style extending Component where {
2          styles {
                   InputPort is style extending Port
4                  OutputPort is style extending Port . . .
                   Filter is style extending Component . . .
6                  Pipe is style extending Connector . . .
           }
8          constraints {
                   OnlyPipeOrFilter is constraint {
10                         to style Instance.connectors apply {
                                   forall ( c | c in style Pipe ) and
```

```
12                                          to style Instance.components apply {
                                                    forall ( c | c in style Filter
                                                        )
14                  },

16                  NoPipeConnection is constraint {
                            to style Instance.connectors apply
18                                  forall ( p1 , p2 | p1 in style Pipe and p2 in
                                        style Pipe implies not attached ( p1 , p2 )
                                         ) ,
                    },
20
                    OneToOneConnection is constraint {
22                          to style Instance.connectors apply {
                                    forall ( f | to f.port s apply
24                                          exists ( fp | to style Instance.
                                                connectors apply
                                                    exists ( [0..1] p | to p.p or
                                                        ts apply exists ( pp | pp
                                                        attached to fp )
26                                          )
                                    )
28                          )
                    }
30          }
    }
```

2.5.3 Plate-forme d'exécution pour les systèmes intro-actifs

Le support des intro-actions est une caractéristique nécessaire pour un système adaptatif. Dans une architecture à composants, il s'agit d'en modifier dynamiquement la topologie, c'est-à-dire de modifier les connexions entre les composants, mais également de supprimer, de créer ou de remplacer les composants.

Panorama des techniques actuelles

Pour faire face au besoin croissant de flexibilité lié au développement de systèmes mobiles et embarqués, les technologies sous-jacentes doivent maintenant intégrer des mécanismes spécifiques. Comme le montre [43], les principaux mécanismes sont le support d'applications nomades, le support d'information de QdS, le partage de ressources, le support d'adaptations dynamiques, etc. Une partie de la communauté des intergiciels s'est donc attachée à concevoir des outils réflexifs, c'est-à-dire des intergiciels supportant la reconfiguration dynamique des applications qui y sont déployées. Parmi les différents travaux on peut noter OpenCOM [24, 28], PUKAS [49], et DynamicTAO [59].

Chefrour et André proposent ACEEL (*self-Adaptive ComponEnts modEL*)[22] qui repose sur le patron de conception Stratégie [40] et sur un mécanisme de notification par événement pour détecter les variations de l'environnement. Il comporte un méta-niveau pour le contrôle de l'adaptation selon une politique écrite à part dans un langage de script.

Segara et André proposent MoleNE (*MObiLE Networking Environement*) [99, 68] qui est un *framework* destiné à faciliter l'écriture d'applications adaptatives dans le contexte des systèmes

mobiles et distribués à grande échelle. Pour cela, MolèNE utilise des techniques réflexives pour séparer le code fonctionnel des fonctionnalités réalisant l'adaptation. Les stratégies d'adaptation sont spécifiées au niveau méta et sont mises en œuvre par un contrôleur spécialisé. MolèNE assure également le transfert d'état à l'aide du patron de conception *Memento*.

Grondin *et al.* proposent MaDcAr [44] (*Model for Automatic and Dynamic Component Assembly Reconfiguration*), un modèle abstrait de moteur d'assemblage de composants qui permet la construction d'applications auto-adaptables. L'adaptation dynamique consiste à faire passer l'application d'un assemblage de composants à un autre sans nécessairement figer toute l'application. Le concepteur de l'application doit fournir la description des fonctionnalités de l'application sous forme de configurations alternatives, c'est-à-dire une spécification de l'ensemble des assemblages valides. Le choix de l'assemblage à réaliser se fait selon une politique d'adaptation, également spécifiée par le concepteur de l'application. Il s'agit d'un ensemble de règles qui, selon le contexte d'exécution (par exemple, l'état du réseau), génère un ensemble de contraintes. La résolution de ce problème de satisfaction de contraintes débouche sur l'assemblage le plus approprié au contexte courant.

DynamicTAO [59] est un intergiciel réflexif construit comme un extension de TAO ORB. Il intègre des fonctionnalités d'introspection et d'intro-action qui permettent à une application de modifier sa configuration et par conséquent les connexions entre les différents composants. Pour cela, il rend accessible à l'exécution les entités qui le constituent et offre un certain nombre d'opérations permettant de modifier ces entités.

DART [93] est une plate-forme de développement dont le but est de faciliter l'écriture d'applications distribuées. Par rapport à des systèmes comme CORBA, DART va plus loin en permettant l'adaptation dynamique des applications aux conditions d'exécution. Le système utilise pour cela une combinaison de techniques réflexives et de surveillance (monitoring) de l'environnement d'exécution.

K-Component [37] utilise un système de notification pour construire des systèmes sensibles au contexte. La logique de l'adaptation (sorte de politique d'adaptation) est décrite de manière déclarative à l'aide du langage ACDL sous la forme de contrats. Ces contrats stipulent les conditions associées aux événements et les opérations d'adaptation architecturales qui doivent être déclenchées par ces mêmes événements.

FORMAware [76] est un intergiciel réflexif qui combine une représentation explicite de l'architecture et des méta-informations qui permettent d'en contraindre les évolutions. Ces contraintes et les vérifications associées préservent la cohérence de l'architecture vis-à-vis d'un style architectural donné. FORMAware est principalement accessible sous la forme d'une API et n'est pas couplé à un langage de description de haut niveau ce qui en faciliterait pourtant l'utilisation.

OpenCOM suit une approche similaire et fournit un intergiciel « boîte blanche » à l'aide de mécanismes d'introspection et d'intro-action. Il intègre un modèle de composant réflexif lui permettant de manipuler l'architecture des applications et ainsi de supporter les reconfigurations dynamiques modifiant la topologie de l'application (en terme de composants et de connecteurs).

Huang *et al.* [49] proposent une implémentation de la plate-forme J2EE intégrant des capacités de réflexion et d'auto-optimisation. PUKAS dispose des procédures d'intro-actions communes (ajout, suppression, connexion, etc) mais n'est pas à proprement parler un modèle réflexif comme l'est OpenCOM. De plus, PUKAS est nativement couplé à ABC/ADL [73] ce qui permet de manipuler une architecture en cours d'exécution via un modèle ABC.

Fractal/FScript : une plate-forme pour la reconfiguration dynamique

La plate-forme Fractal [11, 12] est un autre exemple de plate-forme d'exécution possible pour les systèmes adaptatifs. En plus de l'implémentation de référence en Java, Fractal dispose de plusieurs implémentations qui diffèrent par leurs objectifs : Think [91] est une implémentation dédiée au développement de systèmes d'exploitation, Dream [64] est spécialisée dans les systèmes asynchrones, AOKell [100] intègre les notions d'aspects et de composants, etc.

Les composants Fractal fournissent et requièrent des services regroupés dans des interfaces. Les interfaces requises peuvent alors être connectées à des interfaces fournies, formant ainsi un assemblage de composants qui peut lui-même être encapsulé dans un composant dit « composite », et donc réutilisé dans un assemblage de plus haut niveau. L'une des particularités des composants Fractal est qu'ils fournissent également des interfaces de contrôle (*Controller* dans la terminologie Fractal) qui permettent de configurer certains aspects extra-fonctionnels tels que le cycle de vie, la topologie interne des composites, etc. Une autre particularité du modèle Fractal est de permettre le partage de composants entre plusieurs composants composites là où les autres modèles utilisent un relation de contenance stricte.

La figure 2.8 présente l'architecture du système de capture vidéo dans le monde Fractal. La notation utilisée, spécifique à Fractal, représente les interfaces sous la forme de « T ». Par convention, les interfaces fournies sont présentées en rouge alors que les interfaces requises sont présentées en vert. Les interfaces représentées en bleu sont les interfaces de contrôle. Les plus courantes sont : *AC, Attribute controler* qui permet de modifier la configuration locale d'un composant, *BC, Binding controler* qui permet de modifier les connexions d'un composant, *CC, Content Controler* qui permet de modifier les composants contenus dans un composant composite.

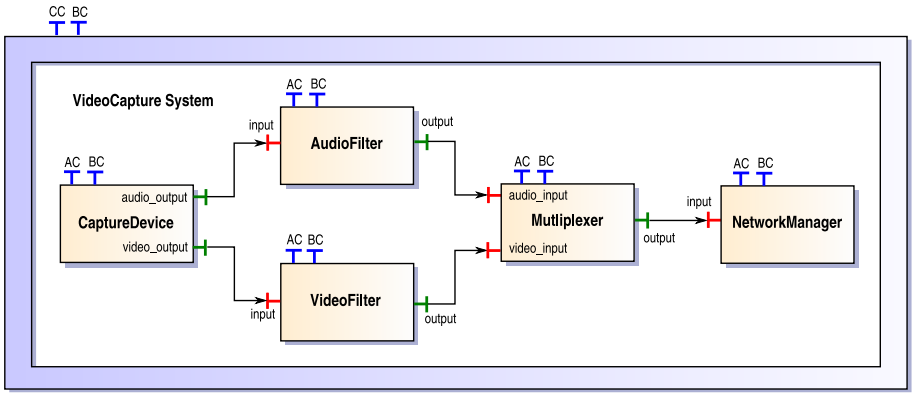

FIG. 2.8 – Une architecture Fractal pour le système de capture vidéo

Fractal, comme OpenCOM est construit sur un modèle à composants réflexif qui inclut les notions de composant, d'interface, et de connecteur (*binding*). Comme pour OpenCOM et dynamicTAO, on peut alors décrire des reconfigurations dynamiques à l'aide des capacités d'introspection de Fractal. Cependant, Fractal dispose d'un langage de reconfiguration dédié qui évite la description de procédures réflexives verbeuses et compliquées. FScript [31] offre les primitives nécessaires pour modifier les architectures, en instanciant (resp. supprimant) des composants, en créant (resp. supprimant) des connecteurs (*bindings* en anglais) entre les interfaces, ou en créant

(resp. supprimant) des interfaces. Il intègre également un langage d'expressions dédié aux architectures Fractal qui permet de référencer facilement les différents éléments architecturaux à la manière des expressions XPath utilisées pour naviguer dans les documents XML.

Le listing 2.3 présente un exemple de reconfiguration architecturale. Celle-ci a pour but de remplacer le filtre audio utilisé par un filtre audio qui limite le nombre de canaux utilisés. Plus précisément, il s'agit de déconnecter les deux interfaces du filtre audio en cours d'utilisation, d'instancier un nouveau filtre, de l'ajouter dans le système de capture vidéo, de le connecter aux autres composants, puis de le démarrer. L'un des intérêts de FScript est qu'il offre des mécanismes pour assurer le bon déroulement des reconfigurations et laisse l'architecture dans un état cohérent lorsqu'un problème survient.

Listing 2.3 – "Une action de reconfiguration FScript pour mettre en place un filtre audio différent"

```
action swicthAudioFilter(root)
{
        unbind($root/child::audioFilter/interface::output);
        unbind($root/child::device/interface::audio_output);

        newFilter = new("org.audioFilter.ChannelFilter");
        set-name($newFilter, "audioFilter");
        add($root, $newFilter);
        bind($root/child::device/interface::audio_output, $newFilter/interface
            ::input);
        bind($newFilter/interface::ouput, $root/child::multiplexer/interface::
            audio_input);
        start($newFilter);
}
```

De nombreux outils et extensions ont été développés autour de Fractal, dont trois sont particulièrement pertinents dans le cas des systèmes adaptatifs : ConFract [26, 25] pour le support de composants contractualisés ainsi que WildCat [32] et COSMOS [27] qui permettent d'accéder à des informations contextuelles.

2.6 Synthèse et positionnement

Le tableau 2.2 récapitule les principales contributions présentées dans les sections précédentes. Elles sont réparties selon deux axes : selon le cycle d'adaptation (observation, décision, introaction) et selon le cycle de développement (conception, validation, exécution). Les contributions sont précédées soit d'un nom, soit d'un sigle désignant un ensemble d'approches. Les sigles utilisés sont SE pour Systèmes Experts, LF pour Logique Floue, RN pour Réseau de Neurones, KV pour Key-Value, L pour Logique, OO pour Orienté-Objet, O pour Ontologie, MT pour Test Métamorphique, SRV Vérification Statique des Règles, DS pour Données Statistiques, MR pour *metarule* et CAPP pour *Context Aware Program Point*.

Du point de vue de la conception et de la modélisation, ce tableau montre tout d'abord que les trois étapes du cycle de l'adaptation ont été abordées dans la littérature. Les descriptions du contexte, du principe de décision et de l'architecture sous-jacente sont dès lors possibles. Cependant, aucune notation ou méthode n'a intégré proprement ces différents éléments. UML par exemple, s'il permet de décrire à la fois les données contextuelles et l'architecture du système, ne permet

	Observation	**Décision**	**Intro-Action**
Conception	KV[98], XML [62, 57], L [70, 1, 53], OO [101, 48], O [103, 45, 92],	SE, LF [119, 104], RN,	ADL [67, 66, 73, 42, 16, 87, 30, 38, 71]
V& V	L [115, 116, 117], MT [108], CAPPS [114]	SRV [105, 78, 79] ST [61, 82], MR [94, 89]	ADL [67, 66, 42, 87, 30, 38]
Implémentation	CosMos [27], LeWYS [15], RCSM [118, 113], WildCat [32], Phoenix [95, 96]	Rainbow [41], QualProbes [65], Miro [110]	Fractal/FScript [12, 31] OpenCOM [24, 28], DTAO [59], PUKAS [49]

TAB. 2.2 – Positionnement et topologie des contributions autours des systèmes adaptatifs

ni de décrire des intro-actions architecturales, ni de décrire une technique de raisonnement sur les données.

Ce tableau montre ensuite combien la validation est difficile pour les systèmes adaptatifs. Elle est d'autant plus difficile que les systèmes adaptatifs intègrent un moteur de décision intelligent. En effet, l'observation d'un contexte vaste et dynamique limite les approches de test dynamique et complexifie les approches plus statiques. De plus, l'injection d'un moteur de décision limite l'utilisation d'approche structurelle de type « boîte-blanche » et l'utilisation d'intro-action rend plus délicates les vérifications statiques sur les modèles architecturaux.

Enfin, en terme de plate-forme d'exécution, là encore, aucune solution n'intègre proprement des capacités d'observation, de décision et d'intro-action. La plate-forme Fractal intègre par exemple un langage d'intro-action ainsi qu'une extension (COSMOS ou WildCat) dédiée à l'observation du contexte mais ne dispose pas d'un moteur de décision capable de faire le lien entre les deux. PUKAS également offre à la fois des mécanismes de décision et d'intro-action mais ne dispose pas de technologies dédiées à l'observation de l'environnement.

Si les différents problèmes présentés dans le tableau ont tous été abordés, aucune approche n'intègre les solutions proposées. La contribution des travaux présentés ici est donc de jeter les bases de l'intégration en proposant un mécanisme de conception pour les systèmes adaptatifs, un mécanisme de validation basé sur la simulation ainsi que les mécanismes nécessaires à leur support sur la plate-forme Fractal.

Chapitre 3

Modélisation d'architectures auto-adaptatives

Les travaux présentés dans le chapitre précédent ne permettent pas de modéliser de manière cohérente les différents aspects d'un système adaptatif. Les ADL dynamiques, par exemple, ne permettent ni de représenter la prise de décision, ni de représenter l'accès aux données du contexte. Ce chapitre présente comment intégrer l'observation, la décision et l'intro-action dans des modèles à composants de type UML.

3.1 Vers des architectures intro-actives

3.1.1 Portée et régie des intro-actions

Portée des intro-actions

Dans le cas des modèles à composants hiérarchiques, la portée des intro-actions est capitale pour préserver la réutilisation des architectures adaptatives. De nombreux ADL, dont celui de Fractal notamment, permettent de traverser les différents niveaux d'encapsulation de l'architecture et de modifier ainsi directement des composants enfouis tout au fond du système. Si cette idée semble pragmatique, elle brise directement les principes d'encapsulation et d'isolation nécessaires à la réutilisation des composants. Dans le cas du système vidéo par exemple, la modification directe des composants constitutifs du filtre vidéo crée une dépendance implicite entre la conception du système vidéo et la conception du filtre vidéo. Cette dépendance va à l'encontre de l'idée même de composants conçus séparément et réutilisables sous la forme de boîtes noires. De plus, un composant ne doit pas pouvoir agir sur les éléments qu'il ne contient pas. Un composant ne peut pas modifier un autre composant de même niveau dans la hiérarchie, puisqu'il ne connaît pas son existence. Le filtre vidéo par exemple, ne doit pas pouvoir adapter le filtre audio car ils sont de même niveau. Les intro-actions doivent rester purement internes.

La portée des intro-actions est donc restreinte à un composant composite et aux composants qu'il contient. Elle se limite aux éléments directement accessibles, c'est-à-dire aux composants de premier niveau. Dans le cas du système vidéo, aucune hypothèse ne doit être posée quant à l'organisation interne des composants disponibles et seuls, les éléments (services) qu'ils exposent sur les ports (ou interfaces) peuvent être pris en compte.

Régie des intro-actions

La portée des intro-actions, limitée à un composant composite, implique donc que celui-ci en est responsable. C'est donc lui qui prend les décisions et déclenche les actions nécessaires. Pour cela, il faut qu'il puisse interagir avec chacun de ses sous-composants. Cette interaction entre un composant composite et ses sous-composants est implicite dans les ADL et les modèles à composants. Dans Fractal par exemple, les interfaces de contrôle ne sont jamais directement connectées mais sont utilisées par le composant composite (dans du code *ad hoc*). Il faut donc aller plus loin, et décrire cette interaction au même titre que les autres, c'est-à-dire par le biais d'un port dédié qui caractérise et contractualise les interactions entre un composant composite et ses sous-composants.

L'idée est de considérer les interfaces extra-fonctionnelles comme des interfaces fonctionnelles. La distinction entre le fonctionnel et l'extra-fonctionnel est d'ailleurs une question de point de vue. Les problèmes liés à des contraintes de temps sont, par exemple, purement fonctionnels pour celui qui implémente les horloges associées, mais purement extra-fonctionnels pour celui qui décrit la logique métier de l'application. Dans le cadre des composants hiérarchiques, le composant composite, lui, gère des problématiques extra-fonctionnelles, accède alors aux services extra-fonctionnels (configuration des sous-composants) de manière standard.

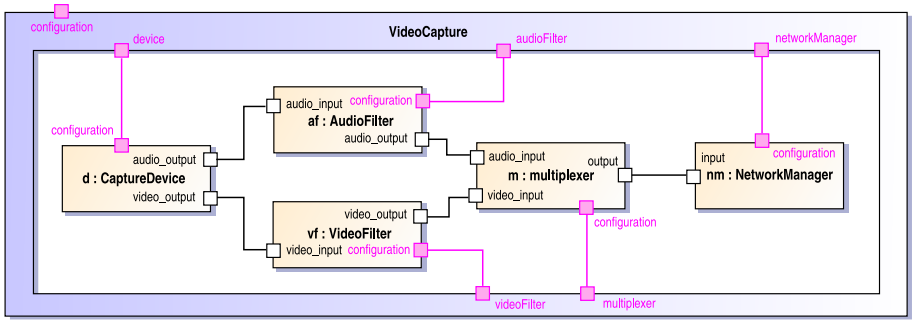

FIG. 3.1 – Modélisation des interactions entre composants composés et composite

La figure 3.1 illustre ce problème de régie de l'adaptation sur le cas de système vidéo. Chaque composant expose sur un port dédié (nommé « configuration ») les interfaces qui caractérisent les interactions qu'il peut avoir avec le composite qui l'englobe. De son coté, le composite englobant dispose de ports dédiés à la configuration de chaque sous-composant possible et également de son propre port de configuration qui lui permet d'être réutilisé et configuré à son tour par un autre composant.

3.1.2 Adaptation architecturale vs. adaptation configurative

On peut distinguer deux types d'intro-actions : les intro-actions architecturales et les intro-actions configuratives. Les intro-actions architecturales modifient l'architecture alors que les intro-actions configuratives modifient la configuration locale des composants.

Intro-actions architecturales

Le terme « intro-action architecturale » désigne une action qui modifie l'agencement des composants et des connecteurs qui les relient. Les ADL dynamiques ainsi que certains modèles à composants supportent ce type d'adaptation à l'aide de langages impératifs simples permettant de créer ou de détruire les composants et les connecteurs. Par leur nature architecturale, et leur aspect « gros grain », ces actions architecturales interviennent de manière ponctuelle dans le cycle de vie du composant. De plus la complexité de leur mise en œuvre (arrêt potentiel de tout ou partie du système, sécurité, etc) limite leur fréquence dans le temps.

Comme nous l'avons évoqué à propos des ADL dynamiques, le support des intro-actions se fait généralement par le biais de procédures impératives basées sur des actions architecturales simples : création et destruction de composants et de connecteurs. Les modèles à composants hiérarchiques définissent deux actions pour modifier la structure d'un composant composite. Les modèles qui intègrent la notion de déploiement disposent en plus d'une action de déplacement, traduisant la mobilité potentielle des composants d'un noeud à un autre sur le réseau. De manière similaire, nous modélisons ces actions architecturales primitives en nous limitant aux modèles à composants hiérarchiques.

- *Start* et *Stop* s'appliquent sur les composants et permettent de les démarrer et de les arrêter
- *Create* et *Destroy* s'appliquent à des composants et permettent de créer et de détruire des instances de composants.
- *Connect* et *Disconnect* s'appliquent aux ports des composants et permettent de connecter et de déconnecter deux ports entre eux.
- *Add* et *Remove* permettent d'ajouter et de retirer une instance de composant d'un composant composite.

Listing 3.1 – Adaptation architecturale utilisée pour contrôler la vitesse d'émission du capteur vidéo

```
reconfiguration deployColorFilter
is
  request videoFilter to do stop
  disconnect videoFilter.input
  disconnect videoFilter.output
  create ColorFilter named newFilter
  self.videoFilter := newFilter
  connect videoFilter.input to device.videoOutput
  connect videoFilter.output to multiplexer.videoInput
  request videoFilter to do start
end
```

Le listing 3.1 présente une procédure de reconfiguration architecturale décrite à l'aide des actions disponibles dans l'outil TANGRAM. Les connections et déconnexions de ports se traduisent directement par une action et l'ajout d'un nouveau composant prend la forme d'une affectation entre sous-composants. L'arrêt et le démarrage des composants sont délégués à des services dont la description est à la charge de l'utilisateur. En effet, si la gestion du cycle de vie est nécessaire pour pouvoir modifier proprement une architecture, elle dépend généralement du domaine d'application. Un autre point important est le problème du transfert d'état. Comme pour la gestion du cycle de vie, la gestion de transfert d'état est laissée à la charge de l'utilisateur car, cela dépend également du domaine d'application. Si ces deux fonctionnalités, la gestion du cycle de vie et la

gestion du transfert d'état pouvaient être modélisées indépendamment du métier, on pourrait alors décrire une procédure de remplacement entre les composants, ce qui simplifierait encore davantage l'écriture de telles procédures d'adaptation architecturales et refléterait directement le principe de substitution à la base de l'approche par composant.

Intro-actions configuratives

Par opposition, le terme « intro-action configurative » désigne une action qui modifie la configuration locale d'un composant. Il s'agit de modifier les paramètres qui régissent le fonctionnement d'un composant, ce qui n'implique aucune modification architecturale. Ce type d'adaptation n'est pas traité par les ADL et les modèles à composants, puisque ils se focalisent sur l'aspect architectural, mais est abordé par certains intergiciels car il s'agit d'une adaptation moins complexe et moins coûteuse. Elle nécessite cependant l'existence d'une notion de configuration dans la plate-forme ou le modèle sous-jacent. Fractal, par exemple, dispose d'interfaces de contrôle (et notamment de l'interface *AttributeController*) qui peuvent être utilisées pour cela. Par leur nature « grains fins », ces adaptations configuratives sont potentiellement continues dans le temps et traduisent un ajustement aussi précis que possible de la configuration d'un composant aux variations de l'environnement. Dans le cas du système de capture vidéo, le taux de compression appliqué aux images capturées est ajusté en fonction des variations de la bande passante. La taille des données transmises est alors ajustée aux capacités du réseau.

Nous proposons de réifier la configuration d'un composant sous la forme de propriétés numériques et des services d'accès associés. Le taux de compression d'un filtre vidéo, par exemple, fait partie de sa configuration locale et est accessible via des services dédiés : *getCompressionRate* and *setCompressionRate*. Cette approche rejoint celle proposée dans Fractal avec la définition de l'interface de contrôle *AttributeController* qui regroupe les services d'accès aux attributs de configuration. La figure 3.2 complète l'architecture proposée pour le système vidéo en réifiant les interfaces de contrôle nécessaires pour agir sur les taux de compression associés aux filtres audio et vidéo. Ces interfaces caractérisent les ports de configuration qui permettent au système vidéo de communiquer avec ses sous-composants.

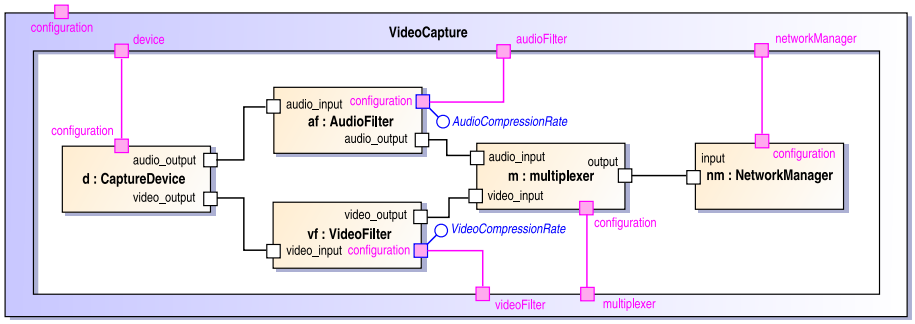

FIG. 3.2 – Réification des intro-actions configuratives sous forme de services dédiés

Au niveau de la modélisation, le modèle initial proposé par la figure 4.3 (cf. page 53) est suffisant puisque les propriétés de configuration sont réifiées par les services d'accès qui leur sont associés. Chaque composant devra donc exposer sur son port de configuration les services per-

mettant d'accéder aux attributs de sa configuration, permettant ainsi au composant qui l'englobe de modifier proprement sa configuration sans violer les notions d'encapsulation et d'isolation. Cette approche permet également d'envisager la contractualisation des services de configuration au même titre que des services fonctionnels standards.

3.2 Observation de données qualifiables

L'observation des données contextuelles pose deux questions principales : il faut quantifier ou mesurer les données et être capable de les exploiter le plus directement possible, c'est-à-dire le plus qualitativement possible.

3.2.1 Observation de données

Avant de pouvoir exploiter les données contextuelles, il faut être capable de les mesurer. Dans le cadre des composants logiciels, elles sont internes ou externes au composant. Les informations contextuelles externes ne sont pas liées au composant lui-même ; c'est le cas lorsqu'elles concernent les ressources disponibles par exemple. Elle seront donc fournies par l'environnement du composant. Elles peuvent également être internes et nécessitent alors un développement *ad hoc*. C'est le cas généralement pour les données liées au métier du système telles que la vitesse d'encodage par exemple qui nécessite le développement d'une sonde dédiée.

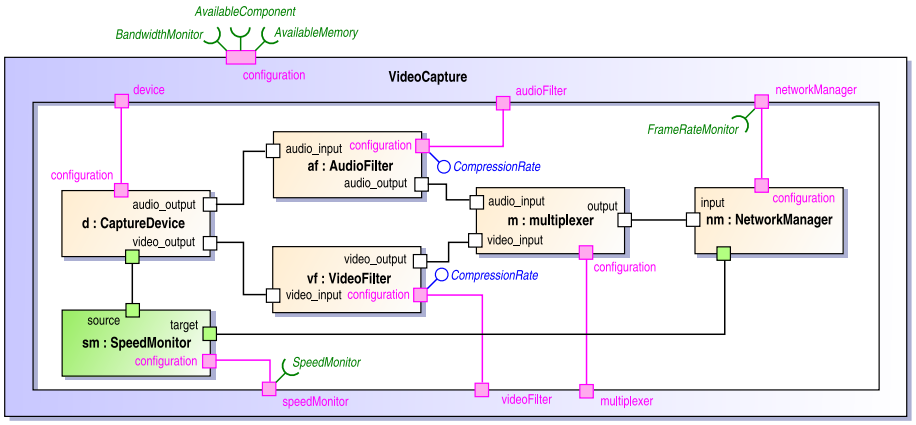

FIG. 3.3 – Intégration des outils de mesure dans l'architecture du système vidéo

La figure 3.3 présente les différents éléments qui sont ajoutés au système vidéo pour mesurer les données contextuelles. Ces données transitent via les ports de configuration qui permettent au système de communiquer avec les composants qui le constituent. Par exemple, le système peut questionner le composant *NetworkManager* pour connaître la qualité actuellement perçue par l'utilisateur puisque que celui-là conserve une connexion continue avec ce dernier. De même, le système requiert de son environnement, via son propre port de configuration, les informations relatives aux ressources disponibles sur la plate-forme : mémoire, bande passante, composants disponibles, etc.

La mesure de données internes, ou métier, se fait par l'ajout d'un composant ou d'un port dédié. Dans la figure 3.3, un composant dédié mesure la vitesse d'encodage du système et la communique au système via son port de configuration. Il est relié au dispositif d'acquisition et au gestionnaire de réseau (*Network Manager*) et peut ainsi calculer le temps mis pour encoder le flux vidéo.

3.2.2 Modélisation de données qualifiables

L'une des principales difficultés lors de l'observation de données est d'en caractériser un état particulier. La plupart des approches existantes (telles que SAFRAN/WildCat) utilise au mieux des seuils pour déclencher des actions différentes en fonction des paliers atteints. Le principal désavantage de cette approche est le calcul de ces seuils qui n'est pas du tout trivial, spécialement lorsque le système est en cours d'élaboration. Cette section illustre l'utilisation des variables linguistiques de la logique floue pour modéliser les données et pouvoir les manipuler de manière qualitative.

Comme nous l'avons évoqué dans la section 2.4.1 page 19, la logique floue permet de définir et de manipuler des données qualifiées. Pour cela elle associe à une variable donnée, une terminologie décrivant les différentes plages de valeurs que peut prendre cette variable. A chaque terme utilisé correspond alors un ensemble flou caractérisé par sa fonction d'appartenance qui en reflète la sémantique. Cet ensemble de termes, associé à la variable qu'il qualifie, forme une *variable linguistique*.

Le principal obstacle à l'utilisation de variables linguistiques pour la modélisation de données contextuelles est lié au calcul des fonctions d'appartenance. La spécification d'une fonction est encore plus compliquée que la spécification d'un seuil précis. Il existe cependant, un certain nombre de fonctions d'appartenance de base qui permettent d'inférer des fonctions par défaut à partir du domaine d'entrée et de la liste des termes associés. Le domaine d'entrée, identifié par un intervalle de valeurs, est alors découpé en tranches correspondant aux différents termes utilisés et une fonction d'appartenance est inférée pour chacune des tranches. Comme le montre la figure 3.4, dans le cas de la bande passante, qui évolue entre 0 ko/s et 10000 ko/s et qui est décrite par les termes faible, médium, et large, les fonctions inférées sont triangulaires et centrées sur le seuil correspondant au terme qui leur est associé.

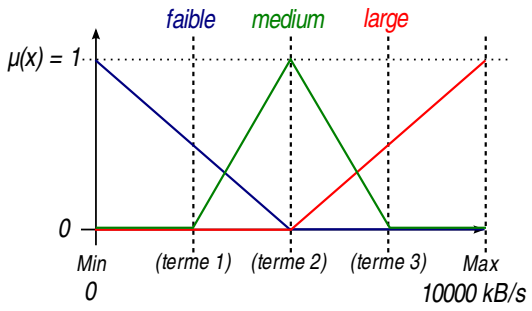

FIG. 3.4 – Génération d'une variable linguistique pour la bande passante

Cette approche permet au concepteur de ne pas avoir à spécifier de seuils difficiles à inférer. Toutefois, le choix des fonctions d'appartenance impacte l'adaptation du système. Des algorithmes

d'optimisation (algorithmes génétiques notamment) peuvent être utilisés pour calculer les seuils et les types de fonction associés (gaussienne, triangulaire, etc).

Il est alors possible de qualifier les données à l'aide de prédicats flous tels que *bandwith is large* par exemple. Ces prédicats peuvent en outre être enrichis à l'aide d'adjectifs comme « très », « peu », ou « extrêmement » qui correspondent à des modificateurs flous. Ces opérateurs unaires très utilisés pour préciser les prédicats flous modifient, par composition de fonctions mathématiques, les fonctions d'appartenance. Les principaux modificateurs sont :

- *Very* défini tel que $\mu_{very}(A) = \lambda x.(\mu_A)^2$. En élevant au carré les valeurs d'appartenance initiales, on aboutit à une fonction d'appartenance plus restreinte : un élément faible à 90% sera donc très faible à 81% et un élément faible à 100% sera également très faible à 100% également. Il s'agit d'accentuer les « pics d'appartenance ».
- *Moderately* défini tel que $\mu_{moderately}(A) = \lambda x.1 - \big(2 \times |\mu_A(x) - 0.5|\big)$ En soustrayant à un, le double de la distance qui sépare l'appartenance de la moyenne, on reflète les endroits où l'appartenance avoisine 0.5.
- *Slightly* défini tel que $\mu_{slightly}(A) = \lambda x.\sqrt{\mu(A)}$ En calculant la racine carrée des valeurs d'appartenance initiales, on accentue les valeurs dont l'appartenance est faible.

La logique floue offre donc un moyen élégant de caractériser de manière qualitative l'état de l'environnement et facilite ainsi la description d'une procédure de décision.

3.3 Prise de décision

3.3.1 Sélection d'intro-actions

De manière générale, les règles d'adaptation sont des règles floues qui mettent en relation l'état de l'environnement décrit de manière qualitative et les intro-actions disponibles au sein du système. Elles prennent la forme connue événement/condition/action où l'événement est une expression qualifiant l'environnement, la condition une garde architecturale permettant de restreindre l'utilisation de la règle, et l'action une intro-action. La principale difficulté dans la description de cette action et de capturer, à la fois le déclenchement ponctuel des intro-actions architecturales et celui, continu, des intro-actions configuratives. En effet, les premières sont de nature discrète et interviennent de manière périodique alors que les secondes sont de nature continue et sont toujours applicables. On définit donc deux types de règles dédiés aux deux types d'intro-action existants, respectivement les règles architecturales et les règles configuratives.

Déclenchement d'intro-actions configuratives

Les intro-actions configuratives mettent finalement en relation des données observables issues de l'environnement et des données modifiables issues de la configuration d'un sous composant. Les règles configuratives sont des règles floues pures où l'antécédent et la conséquence sont des prédicats flous. Traduire l'évolution que doit suivre le taux de compression vidéo en fonction de la bande passante peut s'exprimer par les trois règles suivantes :

- **when** bandwidth is *low* **then** compressionRate is *high*
- **when** bandwidth is *medium* **then** compressionRate is *medium*
- **when** bandwidth is *high* **then** compressionRate is *low*

où chaque règle fait référence aux terminologies définies pour la bande passante (*bandwidth*) et pour le taux de compression (*compressionRate*). Le principe du contrôle flou permet alors de

calculer les valeurs optimales pour la configuration et de répercuter celles-ci via les services de configuration disponibles dans l'architecture.

Déclenchement d'intro-actions architecturales

Les intro-actions architecturales décrivent une modification à apporter à l'agencement des composants du système. Toutefois, cet agencement ne peut pas évoluer dans deux directions simultanément et les règles architecturales vont donc décrire l'évolution de l'utilité des intro-actions architecturales par rapport aux variations de l'environnement. C'est cette utilité qui sera ensuite utilisée pour choisir l'intro-action architecturale à déclencher à un instant précis. Les règles architecturales sont donc des règles floues qui mettent en relation l'évolution de l'environnement et l'utilité d'une intro-action architecturale. Par exemple, l'utilisation du filtre vidéo noir et blanc peut se traduire par les règles suivantes :

- **when** bandwidth is *low* **then utility of** setupBWFilter is *high*
- **when** bandwidth is *medium* **then utility of** setupBWFilter is *medium*
- **when** bandwidth is *high* **then utility of** setupColorFilter is *low*

où chaque règle fait référence aux terminologies définies pour décrire l'environnent et à celle définie (par défaut) pour qualifier une utilité.

Algorithme d'adaptation

L'algorithme utilisé pour appliquer les politiques d'adaptation est basé sur le processus d'évaluation des règles floues présenté dans la section 2.4.1. Appliqué aux différentes règles d'une politique d'adaptation, ce processus permet de calculer à la fois une nouvelle valeur pour chacune des propriétés impactées par les règles mais également l'utilité de chaque action architecturale. Le principe de l'algorithme est donc le suivant :

1. L'ensemble des règles configuratives est évalué, ce qui produit une nouvelle valeur pour chaque propriété impactée par au moins une règle.
2. L'indice d'utilité de chaque action architecturale est calculé à l'aide du même procédé, mais appliqué aux règles de reconfiguration architecturale (l'utilité d'une règle est considérée comme une propriété à part entière dont le domaine, défini par défaut, est une valeur comprise entre 0 et 100).
3. L'action qui a l'indice d'utilité le plus élevé est déclenchée mais elle n'est exécutée que si l'indice dépasse un seuil d'utilité spécifié par l'utilisateur ; ce qui permet de conserver le système dans une certaine stabilité.

Listing 3.2 – Algorithme d'interprétation des politiques d'adaptation décrit en Kermeta

```
operation adaptation(dataRules : Set<Rule>, architecturalRules : Set
    <Rule>) is
do
  var controller : Controller init Controller.new
  var newValues : Table<FuzzyProperty, Value> init Table<
      FuzzyProperty, Value>

  controller.control(dataRules, newValues)
  newValues.each{t:Entry | t.getKey().set(t.getValue())}
```

```
9     var newActionUtilities : Table<ActionUtility, Value> init Table<
          ActionUtility, Value>

11    controller.control(architecturalRules, newActionUtilities);

13    var selectedAction : ActionUtility init
      newValues.select{ e:Entry | newValues.notexists{t:Entry | st.
          getValue() > e.getValue()} }.getKey()
15
      if (maxUtility > UsefulnessBound) then
17       rule.action.execute()
      end
19  end
```

3.3.2 Modélisation de politiques d'adaptation

Comme le montre les travaux présentés dans le chapitre 2, exprimer les règles qui régissent l'adaptation d'un système dans son ensemble n'est pas chose aisée surtout si les propriétés de l'environnement ainsi que les possibilités d'adaptation sont nombreuses. Il est donc important de pouvoir appliquer le principe de séparation des préoccupations (*Separation of Concerns* [36]). L'idée est de briser la complexité en exprimant différentes adaptations ; chacune étant relative à un problème ou à un aspect donné. Par exemple, dans le cas du système vidéo, on va vouloir exprimer d'une part l'adaptation relative à la gestion des filtres vidéo et audio (en fonction de la bande passante notamment) et d'autre part, l'adaptation relative à la sélection de la caméra adéquate (luminosité notamment). Le principe de séparation des préoccupations implique alors de pouvoir composer ces préoccupations, c'est-à-dire de construire par composition l'adaptation qui résulte du traitement simultané de plusieurs préoccupations.

Dans le cadre des systèmes auto-adaptatifs, une préoccupation particulière telle que la luminosité ou la bande passante est appelée une politique d'adaptation. Chaque politique d'adaptation contient donc directement des données observables, des intro-actions et des règles de décision qui mettent en relation les données observables et les intro-actions. Dans le cas du système vidéo, deux politiques sont donc définies, l'une prenant en compte les variations de luminosité en sélectionnant les filtres pertinents, l'autre sélectionnant le choix des caméras.

La figure 3.5 présente un modèle pour les politiques d'adaptation. Une politique est constituée d'un ensemble de propriétés (observables et/ou modifiables) ainsi qu'un ensemble de règles (architecturales ou configuratives). Les règles sont composées de deux expressions floues (l'antécédent et la conséquence) ainsi que d'une expression architecturale qui permet de limiter leur validité. Les règles architecturales se réfèrent à l'utilité d'une intro-action architecturale alors que les règles configuratives se réfèrent directement à une propriété modifiable (de configuration).

3.3.3 Composition de politiques d'adaptation

Comme nous l'avons expliqué précédemment, la description d'une politique d'adaptation générale à un système est délicate à cause de la complexité de l'environnement et des différentes variations possibles de l'architecture. Pour pouvoir briser cette complexité, il faut pouvoir traiter les problèmes séparément, à l'aide de politiques d'adaptations distinctes. Toutefois, il faut alors être capable de composer facilement les politiques d'adaptation. Si l'on considère la composition

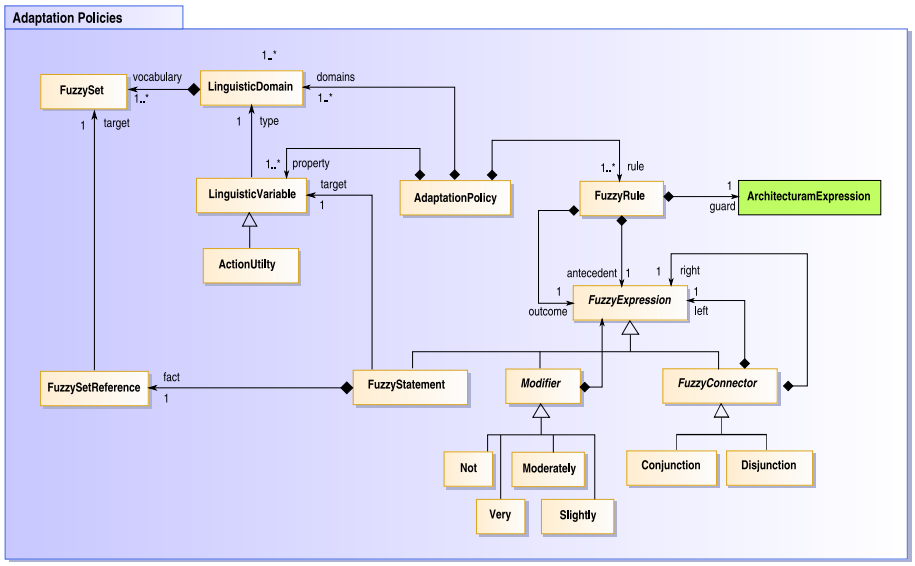

FIG. 3.5 – Modélisation des politiques d'adaptation, des variables linguistiques et des règles de décision

comme l'union des règles et des propriétés qui constituent les politiques d'adaptation, l'ordre d'unification, dans les règles notamment, ne doit pas avoir d'impact sur l'interprétation de la politique résultante. La suite de cette section montre cela.

Soit Σ une structure telle que $\Sigma = \langle P, V, T, K \rangle$ où l'on considère les éléments suivants :

- P est un ensemble de propriétés
- V est une fonction partielle qui associe une valeur à chaque propriété telle que :

$$V = \{v : P \nrightarrow R\}$$

- T est l'ensemble des topologies possibles
- K est l'ensemble des intro-actions architecturales qui conservent la valeur des propriétés associées à chaque composant. Plus formellement on obtient :

$$\begin{aligned} K = \{k : P, V, T, K \rightarrow P, V, T, K \mid \\ \forall\, (t, v) \in T \times V, \forall\, p \in dom(v), \\ \text{let } (t', v') = k(t, v) \\ \text{in } v(p) = v'(p) \end{aligned}$$

Pour pouvoir définir plus formellement l'algorithme, considérons les éléments suivants :

- A comme l'ensemble des politiques d'adaptation où chaque politique d'adaptation est un ensemble de règles d'adaptation tel que :

$$A = \{\, \langle R_P, R_A \rangle \,\}$$

où R_P est l'ensemble des règles configuratives R_A est l'ensemble des règles architecturales. On peut donc définir l'opérateur de composition entre deux politiques d'adaptation comme

une union des ensembles de propriétés (en admettant un renommage) et une union des ensembles de règles. Formellement, on obtient :

$$\begin{aligned} &\forall\,(f,g) \in A^2 \\ f \otimes g \quad &= \quad \langle R_{P_F} \cup R_{P_G}, R_{A_F} \cup R_{A_G} \rangle \end{aligned}$$

– U est une fonction qui associe une utilité aux intro-actions architecturales

$$U = \{\, u : K \to \mathbb{R} \,\}$$

– $control$ est une fonction qui représente l'algorithme de contrôle flou. Elle calcule une nouvelle valeur pour chaque propriété de l'architecture qu'elle a reçue en paramètre et calcule également l'utilité de chaque intro-action architecturale.

$$control : P, V, T, K, A \to V, U$$

– $select$ est une fonction qui sélectionne l'intro-action architecturale la plus utile. Elle retourne l'intro-action la plus utile s'il existe un maximum unique parmi les utilités associées aux intro-actions. Si plusieurs intro-actions ont la même utilité, alors elle utilise une fonction σ (commutative) pour sélectionner l'une des intro-actions ayant une valeur d'utilité maximale.

$$\begin{aligned} select \quad &: \quad U \to K \\ select(u) \quad &= \quad \begin{cases} k \in \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) & \Big| \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| = 1 \\ \sigma\Big(\max\limits_{k_i \in dom(u)} \big(u(k_i)\big)\Big) & \Big| \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| > 1 \end{cases} \end{aligned}$$

où σ est une fonction commutative qui sélectionne un élément parmi n. Elle rend alors la fonction $select$ commutative également.

A l'aide de ces éléments notre algorithme peut se modéliser comme suit :

$$\begin{aligned} apply \quad &: \quad P, V, T, K, A \to P, V, T, K \\ apply(p, v, t, k, a) \quad &= \quad \text{let } (v', u) \text{ be } control(p, v, t, k, a) \\ &\quad\;\; \text{in} \\ &\qquad\quad \text{let } k_0 \text{ be } select(u) \\ &\qquad\qquad \text{in } k_0(p, v', t, k) \end{aligned}$$

On peut alors démontrer que l'ordre parmi les règles n'a pas d'importance et que l'opérateur de composition est bien commutatif. Formellement, cela se traduit par l'équation suivante :

$$\begin{aligned} &\forall (p, v, t, k) \in (P \times V \times T \times K), \forall\,(f, g) \in A^2 \\ &\quad apply(p, v, t, k, \langle f, g \rangle) = apply(p, v, t, k, \langle g, f \rangle) \end{aligned}$$

Considérons les trois cas suivants :

– Si $(f, g) \in (R_A)^2$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v', \{u_f, u_g\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(\{u_f, u_g\}) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v', \{u_g, u_f\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(\{u_g, u_f\}) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

Dans ces deux cas précis, k_0 prend la même valeur puisque l'opération $select$ est commutative.

– Si $(f, g) \in (R_P \times R_A)$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(u_g) \\ &\qquad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(u_g) \\ &\qquad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

Dans ces deux cas, l'opération $select$ est utilisée sur des ensembles à un seul élément et k_0 prend donc la valeur de cet unique élément.

– Si $(f, g) \in (R_P)^2$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(\varnothing) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

$$
\begin{aligned}
apply(p, v, t, k, \langle g, f \rangle) \quad = \quad & \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \\
& \text{in} \\
& \qquad \text{let } k_0 \text{ be } select(\varnothing) \\
& \qquad\qquad \text{in } k_0(p, v', t, k)
\end{aligned}
$$

Nous considérons ici que f et g ne portent pas sur les mêmes propriétés. Dans ce cas particulier, la fonction $control$ produirait une seule valeur pour cette propriété (une tendance moyenne) et l'ordre n'a donc pas non plus d'influence ici.

3.4 Conclusion

Ce chapitre a présenté comment modéliser les trois aspects d'un système auto-adaptatif : observation, décision, action. L'architecture du système est modélisée sous la forme d'un assemblage de composants. Cela facilite la description d'intro-actions architecturales qui vont venir modifier cet assemblage. Les intro-actions, sont réifiées sous la forme de services de configuration accessibles via des ports de configuration dédiés qui explicitent les interactions entre le système et ses parties. Toutes les données contextuelles sont ensuite réifiées sous la forme de capteurs dédiés, soit de manière *ad hoc* lorsque ces données sont internes, soit sous la forme de fonctionnalités requises vis-à-vis de l'environnement. Pour pouvoir être exploitées facilement, ces données contextuelles sont associées à des variables linguistiques qui permettent de les décrire de manière qualitative en utilisant une terminologie adéquate. Finalement, la prise de décision se traduit par la conception de *politiques d'adaptation* définissant le déclenchement des intro-actions architecturales et configuratives sous la forme de règles floues. Ces politiques d'adaptation sont composables ce qui permet d'appliquer aisément le principe de séparation des préoccupations et ainsi de réduire la complexité inhérente à l'adaptation.

Les travaux présentés dans ce chapitre ont été publiés dans les ateliers M-ADAPT'07 (dans la conférence ECOOP) [17] et sous la forme d'un article court à la conférence ASE 2008 [18].

Chapitre 4

Validation *a priori* par simulation

Ce chapitre présente un métamodèle utilisé pour capturer et simuler les architectures logicielles. La première partie décrit les différents concepts choisis pour représenter les architectures tandis que la seconde partie présente la sémantique choisie et identifie les points de variations qui vont permettre de refléter une plate-forme d'exécution.

4.1 TanGraM : Modélisation exécutable d'architectures logicielles

L'architecture logicielle est définie comme une activité qui vise à concevoir la structure d'un système logiciel, c'est-à-dire son organisation en un assemblage de composants clairement identifiés. Toutefois, cette définition issue d'un parallèle avec le génie civil, réduit l'architecture logicielle à un problème purement structurel. Cependant, avant d'introduire plus en détails les moyens mis en œuvre pour capturer et simuler ces architectures, il convient de justifier cette démarche. Pourquoi est-ce nécessaire de modéliser ces architectures, quels en sont les bénéfices ? Plus important encore, pourquoi se donner la peine de valider ces modèles, puisqu'ils ne sont finalement que des abstractions d'une réalité excessivement complexe ?

Pourquoi modéliser l'architecture ? L'architecture logicielle, est tout d'abord un moyen de briser la complexité inhérente du système en le divisant en éléments indépendants et réutilisables. Le système vidéo, par exemple, est organisé comme un encodeur reliant une source de données à un périphérique d'émission. Si les choix architecturaux semblent inopérants quant à la finalité du système, ils sont pourtant la clé de voûte de l'ensemble du processus (futur) de développement : l'organisation des équipes de conception, de développement, de test en sera impactée, de même que les futures évolutions du système. Modéliser l'architecture, c'est donc construire une abstraction intelligible de l'ensemble du système qui permet de raisonner, de comprendre et d'anticiper les conséquences des choix architecturaux.

Les modèles architecturaux et les modèles en général, sont ensuite des éléments clés du processus de développement. La crise du logiciel en particulier a montré que la complexité du logiciel est telle qu'il est impossible de ne pas documenter ou de ne pas communiquer sur les décisions et les stratégies adoptées. Les modèles permettent de capturer et de capitaliser, à la fois sur des connaissances techniques et sur des connaissances historiques. La modélisation au sens large systématise et synthétise le processus de développement.

Enfin, pour être réellement pertinents, les modèles doivent être productifs, c'est-à-dire suffisamment concis et formalisés pour être exploitables automatiquement. L'ingénierie des modèles repose sur ce principe et propose de raffiner, de compléter, ou de composer les modèles par transformation. Dans le cadre de l'auto-adaptation, un modèle productif est un modèle directement exploitable par une plate-forme d'exécution pour piloter l'adaptation du système.

Ainsi l'utilisation de modèles d'architecture, productifs de surcroît, est un point crucial pour le développement de systèmes auto-adaptatifs.

Pourquoi valider ces modèles ? La capacité d'un système à s'auto-adapter repose sur son bon découpage en éléments clairement identifiés comme interchangeables sur la plate-forme d'exécution. Ces éléments, les composants logiciels, sont directement issus des premières étapes de la conception. Modifier les règles qui régissent l'adaptation peut donc nécessiter de repenser ce découpage de l'architecture en composants et donc mener à la rétro-conception complète du système. Pour éviter ce type de retour arrière dans le processus, il convient de valider le plus tôt possible les choix architecturaux relatifs à l'auto-adaptation, c'est à dire de valider les modèles d'architecture vis-à-vis des exigences en matière d'adaptation.

De plus, de par leur capacité à observer un environnement complexe, la validation et la vérification d'un système auto-adaptatif sont des tâches difficiles et fastidieuses. Confronter le système à des scénarios réels nécessite de reproduire à la fois les aspects fonctionnels et les aspects extra-fonctionnels de ces scénarios et donc de reproduire potentiellement tout ou partie de l'environnement. Une validation sur les modèles permet de reproduire la logique de l'environnement et de mettre en évidence des dysfonctionnements logiques dans l'auto-adaptation.

Enfin, dans le cas des modèles productifs, utilisés directement pour piloter le système final, une erreur dans le modèle initial se reportera directement sur la plate-forme. Un système d'aide au diagnostic par exemple, produira des diagnostics erronés si sa base de connaissances est incomplète ou incohérente. La validation de modèles productifs est donc inévitable et rejoint le problème de validation des bases de connaissances dans les systèmes experts.

Ainsi, l'influence des modèles productifs sur la conception, le développement, et l'exécution de système auto-adaptatif rend incontournable leur validation pendant le cycle de développement.

Principe général La suite de ce chapitre présente comment modéliser et valider *a priori* les politiques d'adaptations à l'aide de l'outil TANGRAM. L'idée est d'utiliser les méthodes de méta-modélisation pour capturer à la fois la structure et le comportement des architectures, mais également les politiques qui en contrôlent l'adaptation. En associant une sémantique opérationnelle aux modèles d'architecture et d'adaptation, il est possible de simuler, lors de la conception, les politiques d'adaptation pour en observer le comportement et ainsi détecter d'éventuels problèmes d'incohérence ou d'incomplétude. Comme le montre la figure 4.1, l'objectif est de pouvoir, ensuite utiliser les modèles d'adaptation directement sur une plate-forme d'exécution pour piloter l'application réelle, les politiques d'adaptation étant alors des modèles productifs.

La figure 4.1 synthétise le principe et l'objectif de l'outil TANGRAM. Il s'agit de permettre au concepteur de système adaptatif de tester les politiques d'adaptation qu'il conçoit avant de les déployer directement sur le système réel pour qu'elles en pilotent l'adaptation. Lors de la conception, la validation confronte les politiques d'adaptation à des scénarios de test. Ces mêmes politiques sont ensuite déployées sur le système, comme paramètres du déploiement et de la mise en production.

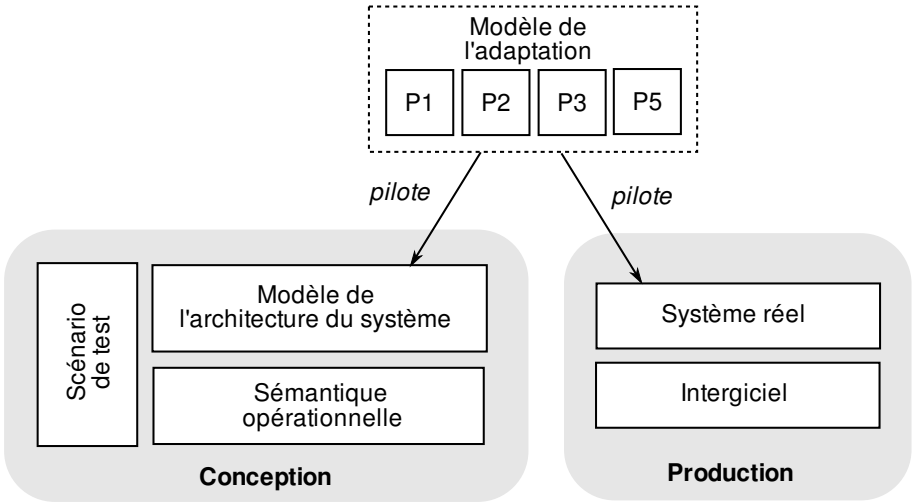

FIG. 4.1 – Principe général de l'outil TANGRAM

4.2 Modélisation des architectures à base de composants

Si l'architecture logicielle doit permettre de briser la complexité d'un système, elle doit avant tout donner un aperçu du système dans sa largeur. Il ne s'agit pas d'être exhaustif, mais plutôt de connecter les différents points de vue possibles : principalement les aspects structuraux, comportementaux et la description des données. Toute la difficulté est donc de trouver l'équilibre entre une modélisation trop riche, difficilement utilisable, mais facilement exploitable et une modélisation plus pauvre, facilement utilisable mais difficilement exploitable.

Parmi les nombreuses définitions du concept de composant proposées dans la littérature, nous avons retenu les propriétés suivantes qui nous semblent essentielles :

1. Un composant est *autonome*, c'est-à-dire qu'il peut déclencher des traitements internes, mais également faire appel à son environnement sans forcément répondre à un stimulus externe.
2. Un composant est *isolé* et n'interagit avec son environnement que par le biais des points de connexion qui l'y relient. Toutes les informations qui entrent ou sortent du composant le font sur les points de connexions appropriés.
3. Un composant peut faire la distinction entre plusieurs stimuli identiques provenant de son environnement. Cette capacité de *discernement* est essentielle car elle permet de différencier deux éléments de l'environnement qui sollicitent le même service.

Le modèle que nous présentons dans la suite de cette section est construit sur ces trois propriétés. Nous ne prétendons pas quelles forment une définition du concept de composant mais simplement qu'elles ont guidé notre approche.

Dans le modèle présenté ici, un composant est décrit selon quatre axes différents comme le montre le tétraèdre présenté par la figure 4.2.

– Les aspects structuraux définissent les frontières entre le composant et son environnement et déterminent les interactions possibles.

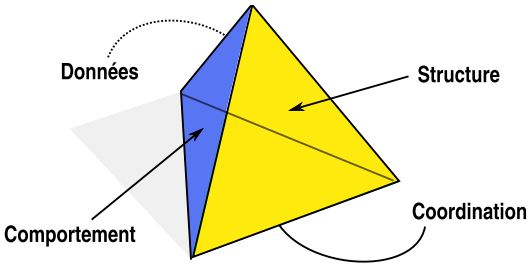

FIG. 4.2 – Les quatre facettes d'un composant

- Les données décrivent la structure des données qui sont manipulées ou stockées par le composant.
- Le comportement décrit les traitements que le composant peut appliquer sur les données et l'utilisation qu'il a des services qu'il requiert de son environnement.
- La coordination décrit les règles qui régissent le déclenchement de traitements en fonction des stimuli reçus de l'environnement.

Ce tétraèdre ne prétend pas que l'architecture logicielle doit être restreinte à ces seuls quatre aspects, mais simplement qu'ils forment une abstraction exécutable minimale de l'architecture logicielle. Si l'on considère qu'un système logiciel se présente comme une sphère composée d'une infinité de facettes (ou d'aspects), alors, le tétraèdre est bien une abstraction de la sphère.

4.2.1 Modélisation des aspects structuraux

En dépit des nombreux ADL existant dans la littérature, un consensus semble se cristalliser autour de la notation proposée dans UML2.0. Tous représentent plus ou moins des concepts similaires : composants, ports, connecteurs. Seule la sémantique associée à ce noyau de concepts évolue d'une approche à l'autre. Nous proposons dans cette section, une variante de la palette de concepts proposée par UML2.0.

Comme le montre la figure 4.3, les concepts que nous avons retenus sont les suivants :

Component représente la notion de composant de manière abstraite.

Interface représente un ensemble de services identifié par un nom. Ces ensembles peuvent être requis ou fournis par un composant aux travers de ses ports.

Service représente un *service* ou une fonctionnalité identifiée par un nom et l'ensemble de ses paramètres d'entrée ou de sortie.

Port représente un point de connexion (identifié par un nom) entre un composant et son environnement. Chaque port est typé par un ensemble d'interfaces requises ou fournies regroupées au sein d'un objet *PortType*. Ils peuvent être simples, multiples ou optionnels selon la multiplicité qui leur est associée.

```
context Component
  inv: self.port->unique(name)
  inv: self.portType->unique(name)
  inv: self.port->forall( p | self.portType->include(p.type))
```

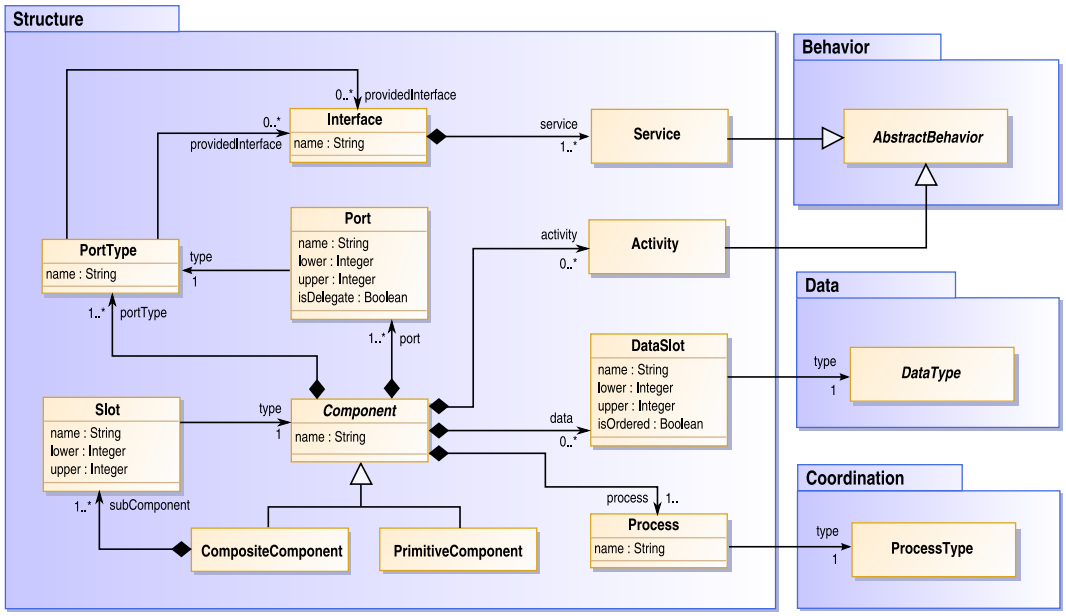

FIG. 4.3 – Paquetage *Structure*, définit l'ensemble des concepts utiles pour décrire la structure des collaborations entre composants

PortType représente le type d'un port, c'est-à-dire les interfaces requises ou fournies par le port. Les types de port ne sont visibles qu'à l'intérieur d'un composant.

```
context PortType:
inv: requiredInterface->intersection(providedInterface)->empty
```

PrimitiveComponent représente les composants de base d'un système, c'est-à-dire les composants qui ne sont pas constitués d'autres composants.

```
context PrimitiveComponent
inv: self.port->forall( p | not p.isDelegated() }
```

CompositeComponent représente les composants complexes qui réalisent une collaboration entre plusieurs autres composants.

```
context CompositeComponent
inv: self.subComponent->unique(name)
```

Slot représente les sous-composants possibles au sein d'un composant composite. Ces emplacements sont simples ou multiples en fonction de la multiplicité qui leur est associée.

Par rapport à la notation UML telle qu'elle est présentée dans la norme (2.0), nous avons principalement ajouté quelques restrictions.

– UML autorise la définition de multiplicités sur les extrémités des connecteurs. Cela permet de spécifier qu'un port est connecté à un ou plusieurs ports d'un autre composant. Notre modèle interdit cela en ne définissant que des connecteurs simples, c'est-à-dire des connecteurs qui relient un port à un autre.

– UML définit la notion de composant comme une extension du concept de classe alors que nous séparons fortement ces deux concepts, évitant ainsi de devoir définir l'héritage entre composants, les associations entre composants, etc.
– La notion de connecteur n'existe pas dans notre approche, car elle y est purement dynamique. Les ports peuvent être dynamiquement connectés et déconnectés les uns des autres.
– Un composant UML peut contenir un grand nombre de concepts de manière plus ou moins vague puisque qu'un composant est également une sous-classe de *Package*. Dans notre modèle, un composant peut contenir quatre types d'éléments : des sous- composants, des données, des processus et des activités.

La syntaxe proposée pour la définition d'une interface est présentée dans l'exemple 4.1. On y définit une interface *FrameReader* et son service *processFrame* qui recoit un paramètre en entrée nommé *frame*.

Listing 4.1 – Déclaration des interfaces utilisées

```
interface AudioFrameReader
is
   service processAudioFrame
      in frame : Frame
end
```

Une fois que les interfaces sont définies, la définition d'un port peut y faire référence. L'exemple 4.2 présente la définition du filtre audio. Le composant *AudioFilter* est donc équipé de deux ports nommés respectivement *input* et *output*. Ces ports sont typés par les interfaces qu'ils fournissent ou requièrent. Un troisième port, le port de configuration (défini grâce au mot-clé *configuration*) définit les interfaces requises et fournies pour la configuration du filtre audio. De manière plus générale, la définition d'un composant reprend les quatre vues possibles : structure, données, comportement, et coordination.

Listing 4.2 – Définition d'un composant primitif

```
component AudioFilter
is
  -- Ports du composant
  configuration
    provide CompressionRate

  port input : InputPort
    provide AudioFrameReader

  port output : OutputPort
    require CompressedFrameReader

  -- Donnnees possedees par le composant
  data inputBuffer : AudioFrame[0..*] {ordered}
  data outputBuffer : AudioFrame[0..*] {ordered}

  -- Activites du composant
  creation activity make
    in compressionRate : Real
  do ... end

```

```
    activity filter
23    in frame : AudioFrame
    do ... end
25
    -- Coordination du composant
27  coordination
    is ... end
29 end
```

La définition d'un composant composite est sensiblement la même si ce n'est qu'elle peut inclure la définition de sous-composants, des ports délégués, ainsi que des politiques d'adaptation. Les sous-composants sont déclarés suivant le modèle : « *slot* nom : type multiplicité ».

4.2.2 Modélisation des données

Un autre aspect de l'architecture concerne les données qui y sont manipulées. Leur modélisation influence directement la description des traitements qui les manipulent : plus la description des données est détaillée, plus les traitements à décrire seront compliqués. Dans le cas du système vidéo par exemple, les données manipulées sont extrêmement complexes, mais leur structure interne réelle n'intéresse pas l'architecte logiciel qui se concentre sur la logique de l'application. Une abstraction des différents types de données tels que les flux vidéo et audio est suffisante. L'objectif de la description des données est de capturer les propriétés qui peuvent avoir un impact sur le fonctionnement global de l'application au niveau fonctionnel comme au niveau extra-fonctionnel. Le taux d'échantillonnage utilisé pour encoder le flux audio influe sur les performances du filtre audio, et cette information doit donc être présente. La figure 4.4 présente les données utilisées et stockées par le composant *Multiplexer* sous la forme d'un diagramme de structures composites.

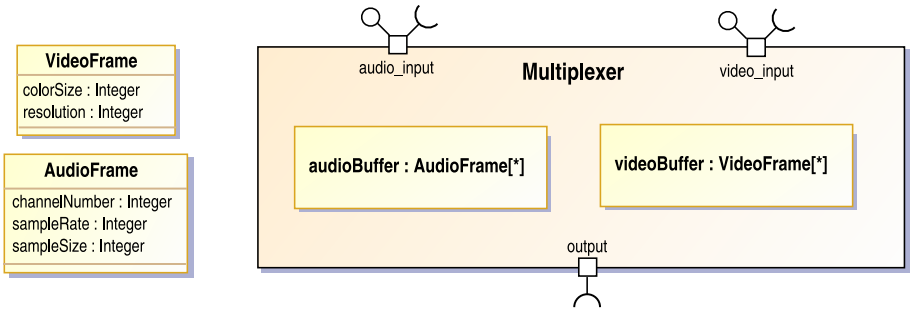

FIG. 4.4 – Exemple de données contenues par un composant, ici le composant *Multiplexer*

Nous proposons de modéliser les données à l'aide d'un modèle à objets inspiré des standards que sont EMoF [84] et ou ECore [13]. Le concept de classe a fait ses preuves et est maintenant largement utilisé pour modéliser les données structurées. Le modèle que nous proposons n'apporte aucune contribution, mais est intégré proprement au reste de notre modèle. La figure 4.5 (cf. page 56) décrit les concepts utilisés pour décrire les données. Une classe contient des propriétés typées et des opérations. Les opérations ont été ajoutées ici dans le but de faciliter la description des traitements utilitaires associés à un type de données particulier. Ces opérations ne sont pas du tout dédiées à la description de traitements métiers.

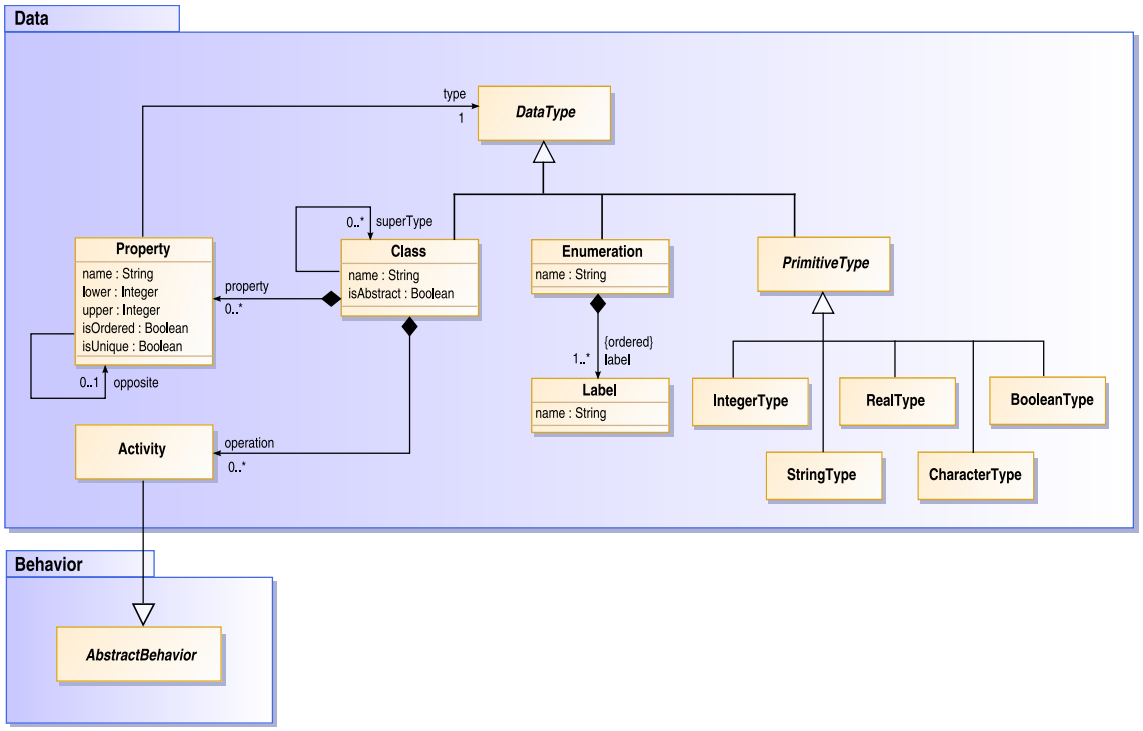

FIG. 4.5 – Paquetage *Data*, définit l'ensemble des concepts utiles pour décrire les données manipulées dans une architecture

Enumeration représente des types énumérés définis par un ensemble ordonné d'étiquettes (classe *Label*).

```
context Enumeration
inv: self.label->isUnique(name)
```

Class représente l'équivalent de la notion de classe d'UML. Il s'agit d'un concept (au sens propre du terme) décrit à l'aide des propriétés qui le caractérisent. Un classe peut « hériter » des propriétés d'autres classes par le biais d'une relation d'héritage (association *superType*).

```
context Class
-- Calcul la cloture transitive des super classes
def: superTypeClosure(acc:Set(Class)):Set(Class) =
  if (not acc->includes(self))
  then
     self.superType->iterate(c:Class, acc2:Set(Class)=Set{} |
          c.superTypeClosure(acc->union(Set{self})))
     )
  endif
inv: self.superType->forall(c:Class|not c.superTypeClosure(Set{})->
    includes(self))
inv:
self.property->collect(name)->union(self.operation->collect(name))
    ->forall(n1,n2| not n1 = n2)
```

Property représente les propriétés associées à une classe. Identifiées par leur nom, elles sont simples, optionnelles, ou multiples en fonction de la multiplicité qui leur est associée. Les attributs *isOrdered* et *isUnique* indiquent respectivement l'existence d'un ordre, ou l'absence de doublons parmi les éléments.

```
context Property
inv: self.opposite->opposite = self
```

Operation représente les opérations qui sont associées aux classes.

PrimitiveType représente une abstraction des types de base communs : entier, booléen, réel, caractère, et chaîne de caractères.

Comparé à UML, ce modèle des données est une version simplifiée du diagramme de classes. Les principales différences sont l'absence de propriétés dérivées ainsi que de contraintes sur les associations.

Pour prototyper les données qui sont manipulées dans l'architecture, TANGRAM permet de définir des classes et des énumérations. L'exemple 4.3, issu du système vidéo, présente la définition d'un type structuré de données. Ce type structuré modélise les trames compressées utilisées par le multiplexeur. On peut noter la définition des propriétés qui le composent ainsi que la présence d'une opération de construction (préfixée par le mot-clé *creation*).

Listing 4.3 – Exemple de prototypage des données

```
1 data CompressedFrame extends Frame
2 is
     property lossRate : Integer
4    property internalFrame : Frame

6    creation operation make
```

```
        in lossRate : Integer
        in frame : Frame
    do
        self.lossRate := lossRate
        self.internalFrame := frame
    end
end
```

4.2.3 Modélisation des traitements

La troisième facette de l'architecture logicielle est dédiée au comportement des composants. Nous proposons d'utiliser le concept d'activité d'UML pour représenter les comportements métiers. La figure 4.6 présente l'activité de synchronisation du composant *synchronizer* sous la forme d'un diagramme d'activité. Il s'agit ici de synchroniser les éléments présents dans les tampons audio et vidéo, tant qu'aucun d'entre eux n'est vide. On fait donc appel à la fois aux données stockées dans le composant (*read audio buffer*) et aux services requis (*play audio data*).

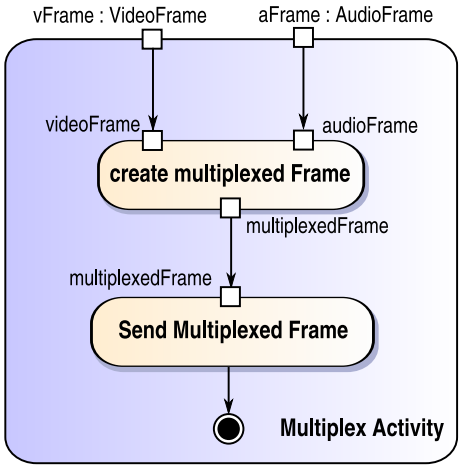

FIG. 4.6 – L'activité métier du composant *Synchronizer*

Comme on peut le constater, rien n'est spécifié quant au déclenchement de cette activité, car c'est la coordination qui va préciser son déclenchement. Pour décrire les activités, nous proposons de nous restreindre à un petit langage d'actions permettant d'exprimer les concepts de base que sont : l'affectation, l'itération, l'appel à un service requis, l'appel à une autre activité, etc. La figure 4.7 présente le modèle utilisé pour capturer ces actions.

Activity représente la notion d'activité. Chaque activité est identifiée par un nom et par sa signature, c'est-à-dire l'ensemble de ses paramètres. La corps d'une activité est composé d'un ensemble d'instructions appelées « *Statement* ».

```
context Activity
inv: self.parameter->unique(name)
```

Parameter représente les paramètres (d'entrée ou de sortie) potentiels d'une activité ou d'un comportement. Ils sont identifiés par leur nom et peuvent être simples, multiples, ou option-

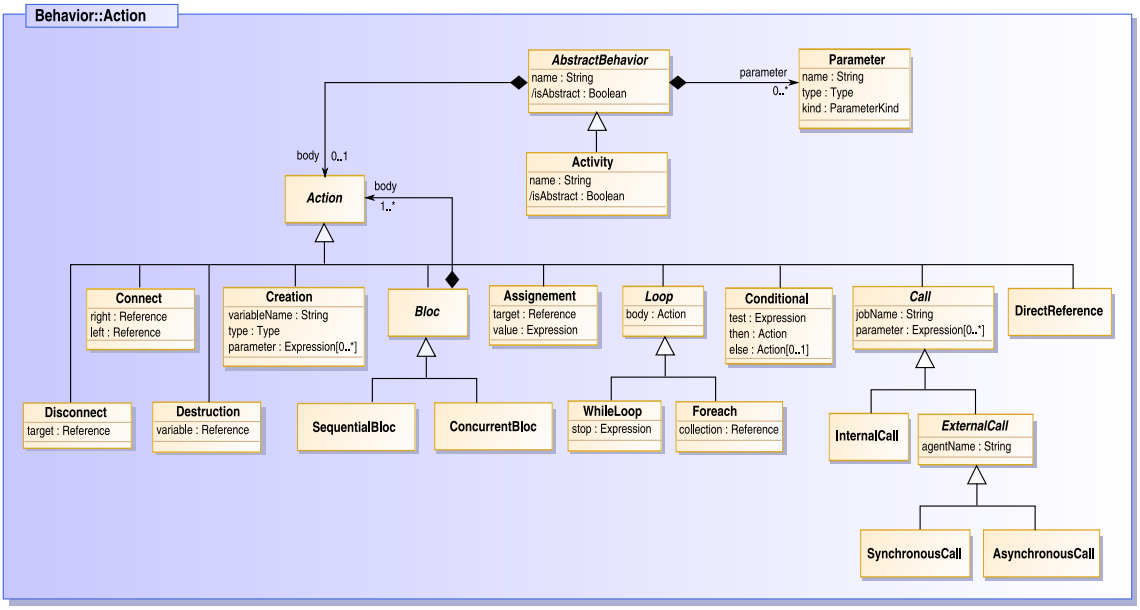

FIG. 4.7 – Paquetage *Activity*, définit l'ensemble des concepts utiles pour décrire le comportement des composants

nels en fonction de la multiplicité qui leur est attachée. Les attributs *isUnique* et *isOrdered* indiquent respectivement la présence d'un ordre ou l'absence de doublons parmi les éléments.

Assignement représente l'affectation d'une valeur. Elle peut concerner des propriétés de base (entier, réel, etc) mais également des ports, des composants, ou des processus comme nous le verrons par la suite.

Bloc représente la notion de bloc d'instructions. Un bloc est donc une instruction composite qui limite la portée des variables qui y sont définies. Deux types de bloc sont possibles : des blocs séquentiels où les instructions sont ordonnées et des blocs concurrents où les instructions sont exécutées en parallèle.

Conditional représente l'alternative entre deux actions selon la valeur d'une expression.

Loop représente l'itération jusqu'à la satisfaction d'une condition. Deux itérations sont possibles : l'itération jusqu'à la satisfaction d'une condition ou l'itération sur les éléments d'une collection (*ForeachLoop*).

Call Plusieurs types d'appels sont possibles. Une activité peut faire appel à une autre activité et il s'agit alors d'un appel interne. Elle peut également faire appel aux services requis sur les ports du composant, il s'agit alors d'appels externes. Les appels externes asynchrones ne sont pas bloquants, alors que les appels synchrones stoppent l'exécution de l'activité jusqu'à la réception d'une réponse.

Creation/Destruction Il s'agit de créer ou de détruire les éléments du modèle, à savoir, composant, données, port, et processus.

Connection/Disconnection Appliquée à deux ports, cette action construit un connecteur qui les relient. Appliquée à deux propriétés de données, cette action construit l'association entre ces deux objets.

Le modèle que nous proposons ici diffère nettement des activités présentées dans UML. En effet, notre modèle est plus proche d'un langage d'action alors que les diagrammes d'activités d'UML sont orientés flot de données.

La description d'expressions qui permettent de naviguer au sein de l'architecture est également nécessaire. Ces expressions définissent principalement les opérations arithmétiques et logiques sur les types de base, les appels vers des opérations sur les données (*helpers*), et les références vers les différents éléments de l'architecture. La langage des expressions possibles est présenté par la figure 4.8.

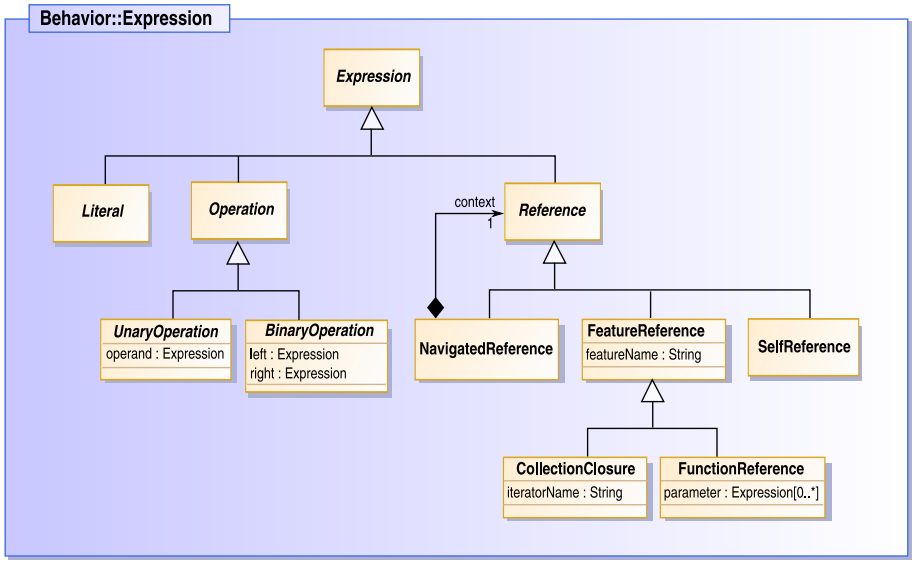

FIG. 4.8 – Paquetage *Expression*, définit le langage des expressions possibles pour naviguer dans l'architecture

Literal représente les différents types de valeurs littérales possibles, à savoir, les entiers, les réels, les booléens, les caractères et les chaînes de caractères (*String*). Ces types ne figurent pas sur la figure 4.8 par mesure de concision.

Operation représente les différentes opérations arithmétiques et logiques que l'on peut faire sur les valeurs littérales. Ce sont : l'addition, la soustraction, la multiplication et la division, pour les opérations arithmétiques et la conjonction, la disjonction, la négation, ainsi que le tests de comparaison pour les opérations logiques. Sur la figure 4.8, ces opérations sont séparées en deux catégories selon qu'elles possèdent un ou deux opérandes mais ne sont pas non plus représentées par mesure de concision.

Référence représente une référence vers un ou plusieurs éléments de l'architecture. Les références directes désignent un élément directement accessible depuis l'élément où elles sont évaluées (*FeatureReference*). Elles peuvent être également « naviguées » (*NaviguatedReference*) si

elles sont relatives à un autre élément de l'architecture qu'il faudra alors évaluer au préalable.

L'exemple suivant décrit l'activité de multiplexage et introduit la syntaxe associée à la description des activités. Elle intègre la plupart des constructions habituelles dans les langages impératifs, l'itération, la conditionnelle, et les blocs concurrents. On peut remarquer également la syntaxe des appels asynchrones (ligne 9) préfixés par le mot-clé *notify*. La syntaxe des appels synchrones suit le principe suivant : « *request* port *to do* service(paramètres) ». L'exemple 4.4 représente, dans le détail, l'activité représentée par la figure 4.6. La syntaxe proposée permet de capturer les constructions associées au diagramme d'activités, activités composites, fork, join, etc. On peut noter, cependant, que rien n'est dit par rapport au déclenchement de cette activité : comme nous le verrons par la suite, les processus de coordination en ont la charge.

Listing 4.4 – Description de l'activité principale du multiplexeur

```
1  activity multiplex
2    in aFrame : AudioFrame
     in vFrame : VideoFrame
4  do
     create MultiplexedFrame named frame
6    frame.internal.add(vFrame)
     frame.internal.add(aFrame)
8
     notify output of handle(frame)
10 end
```

4.2.4 Description des coordinations

Les règles qui régissent les réactions d'un composant aux stimuli externes viennent clore la description d'un composant. Comme nous l'avons vu précédemment, un composant fournit des services à son environnement : lorsque ces services sont sollicités, le composant déclenche les traitements internes appropriés. Nous proposons de représenter la coordination à l'aide de processus à la manière des diagrammes d'états d'UML2. Un composant définit donc un ou plusieurs processus qui réagissent aux événements (aux appels de services) qui surviennent sur ses ports. Les processus reflètent par leurs transitions les règles de coordination de la forme : événement, garde, action.

Les événements correspondent à des appels à un service sur un port. Cependant, l'existence de ports multiples (avec une multiplicité $[x..*]$) complexifie l'expression des événements. Deux cas sont possibles. D'une part, on peut vouloir réagir à la réception simultanée d'un même message sur plusieurs instances d'un port. Cela correspond à la conjonction ($\wedge$) des réceptions du message sur les instances du port. D'autre part, on peut vouloir réagir à la réception potentielle d'un même message sur plusieurs instances d'un port. Cela correspond à la disjonction ($\vee$) des réceptions d'un même message sur différentes instances d'un port.

Les gardes associées aux règles de coordination sont des expressions qui portent sur la structure et les données de l'architecture. Elles sont naturellement décrites à l'aide du langage d'expressions présenté dans la section précédente.

Les actions correspondent aux réactions à adopter lorsqu'un événement survient. Les actions possibles correspondent aux actions décrites précédemment. Généralement il s'agit d'exécuter une activité interne.

L'utilisation de tels processus permet de renforcer la séparation entre la spécification des services (les interfaces exposées sur les ports) et leur implémentation. Un composant peut alors proposer différentes implémentations pour un même service, soit en fonction du port sur lequel le service est sollicité, soit en fonction de l'état des processus qui coordonnent le composant.

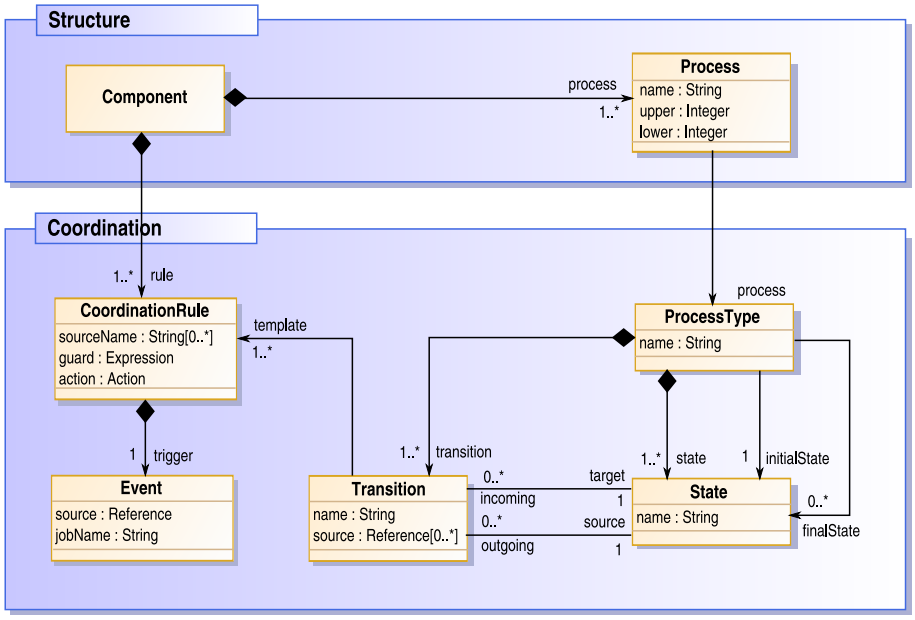

FIG. 4.9 – Paquetage *Coordination*, définit le langage des expressions possibles pour naviguer dans l'architecture

CoordinationRule représente une règle de coordination. Chaque règle est identifiée au sein d'un composant par son nom. Une règle décrit la réaction attendue du système lors de l'occurrence d'un événement. Les règles de coordination sont paramétrées, ce qui les rend réutilisables pour plusieurs ports ou pour des groupes de ports.

```
context Component:
  inv: self.process.type.transition.template->forall(cr |
  self.rule->includes(cr)
  inv: self.process.type->forall(pt | self.processType>includes(pt))
```

Event représente l'appel d'un service sur un ou plusieurs ports.

ProcessType représente un type de processus de coordination. Chaque type de processus de coordination peut être instancié plusieurs fois à l'exécution. Chaque type de processus est constitué d'un ensemble d'états reliés par des transitions qui font référence à une règle de coordination.

```
context ProcessType
  inv: self.state->includes(self.initialState)
  inv: self.finalState->forall(s | self.state->includes(s))
  inv: self.transition->forall(t | self.state->includes(t.target)
  and self.state.include(t.source)
  inv: self.state->forall(s | s.outgoing->forall(t |
```

```
self.transition->includes(t)) and s.incoming->forall(t | self.
    transition.includes(t)))
```

Une syntaxe concrète pour la coordination

La syntaxe permet également de décrire le comportement des processus qui régissent la coordination interne à un composant. On définit dans un premier temps les règles de coordination qui mettent en relation les événements observables (appels de service) avec les activités qu'ils déclenchent. Dans l'exemple 4.5 extrait du composant *Multiplexer*, une règle nommée *multiplex* contrôle le déclenchement de l'activité du même nom (présentée précédemment dans l'exemple 4.4). Cette règle fait usage d'une conjonction d'événements et stipule que lorsque le service *processFrame* est appelé sur le port *audioInput* et que (*ie.* simultanément) le service *processFrame* est appelé sur le port *videoInput*, et que les paramètres de ces deux appels sont relatifs à une même *frame*, alors on déclenche le multiplexage.

Les règles permettent ensuite de définir le comportement de processus qui coordonnent le comportement d'un composant. Dans l'exemple 4.5, un processus nommé *handler* applique la règle *multiplex* de manière répétée. Le comportement d'un processus est décrit, ici, via une expression régulière portant sur les règles de coordination. Ces processus sont créés lors de la construction du composant, de la même manière que les autres objets et peuvent également être modifiés en cours de fonctionnement.

Listing 4.5 – Coordination du composant *Multiplexer*

```
1  component Multiplexer
   is
3
     creation activity make
5    do
       create Handler named newHandler
7      self.handler := newHandler
     end
9
     coordination
11   is
       rule multiplex
13     is
         when  processFrame(aFrame) called by audioInput
15           and processFrame(vFrame) called by videoInput
         if aFrame.id = vFrame.id
17       then run multiplex(aFrame, vFrame)
       end
19
       process handler : Handler do multiplex*
21   end
   end
```

Utilité d'une coordination séparée

Il peut sembler lourd et redondant de séparer les aspects liés à la coordination du comportement même du composant. En réalité, la définition et l'utilisation de ports multiples impliquent de

pouvoir décider de la coordination entre les appels et le déclenchement des activités. Prenons le cas d'un composant n'ayant qu'un seul port $p1$ offrant un service quelconque. Si ce port est simple, la coordination n'est pas réellement utile puisque que le composant n'accepte qu'une seule connexion à un instant donné. Par contre, si ce port est multiple, le composant peut potentiellement devoir traiter plusieurs connexions simultanément. Deux principaux cas sont possibles, mais ne relèvent pas de la sémantique du modèle, mais bien du domaine de l'utilisateur.

La première solution consiste à gérer les appels séquentiellement. Dans ce cas, on ne va définir qu'un seul processus pour traiter tous les appels entrants, quelle que soit la connexion utilisée. Cela revient à décrire la coordination proposée par le listing 4.6. Un seul processus s'occupe de toutes les connexions existantes. Dans cet exemple, la règle *work* utilisée pour décrire la coordination utilise le mot-clé *any* pour spécifier que le service $s1$ peut survenir sur n'importe quelle instance du port *user* (celui-ci a été déclaré multiple à la ligne 3). Il s'agit d'une disjonction portant sur l'ensemble des instances du port existant lors de l'évaluation de la règle. (La conjonction aurait utilisé le mot-clé *all* et aurait eu un sens complètement différent).

Listing 4.6 – Coordination séquentielle

```
1  component SequentialServer
2  is
     port user : UserPort [0..*]
4      provide UserService

6    creation activity make
     do
8      create Handler named newHandler
       self.handler := newHandler
10   end

12   activity s1
     do ... end
14
     coordination
16   is
       rule work
18     is
         when s1 called by any user
20       then run s1
       end
22
       process handler : Handler do work*
24   end
   end
```

L'autre solution consiste à traiter en parallèle les requêtes des utilisateurs en utilisant par exemple, un processus par utilisateur. Cette solution pose deux problèmes :

1. D'une part, il faut être capable de créer des nouveaux ports et donc de nouveaux processus. Les créations d'instances de ports et de processus sont donc intrinsèquement liées par la gestion choisie pour les ports multiples. De plus, si la création d'un port doit être effectuée par le composant lui même (lui seul connaît les processus à mettre en place), la création d'un nouveau port doit, elle, être initiée par le composant composite englobant. Un nouveau port va modifier la configuration architecturale de la collaboration environnante, seul le

composant englobant en est responsable. La création d'instances de ports multiples sera donc un service offert au composant englobant via le port de configuration.

2. D'autre part, il faut être capable d'associer un processus à une instance de port particulière, ici à un port *user* particulier. Il faut donc pouvoir spécifier les instances de port qu'un processus est susceptible de contrôler.

L'exemple 4.7 présente la syntaxe TANGRAM utilisée pour décrire cette solution concurrente. La création d'un nouveau port est réifiée par un service *createNewUser* exposé sur le port de configuration par l'interface *UserManagement*. Un processus nommé *confHandler* est alors dédié à la configuration et, ici, à la création des nouveaux utilisateurs. Ce processus déclenche l'activité *createUser* qui crée à la fois une instance de port et le processus *UserManager* qui lui est associé. La ligne 41 de l'exemple 4.7 montre que les processus *UserManager* peuvent être paramétrés par un port particulier, port qui est utilisé ensuite dans la description du comportement ainsi que dans les règles associées.

Listing 4.7 – Coordination séquentielle

```
1  component ParalellServer
   is
3    configuration
       provide UserManagement
5
     port user : UserPort [0..*]
7      provide UserService

9    creation activity make
     do
11     create ConfHandler named newHandler
       self.confHandler := newHandler
13   end

15   activity createUser
       out result : UserPort
17   is
       create UserPort named result
19     self.user.add(result)
       create UserManager(result) named newManager
21     self.userManager.add(newManager)
     end
23
     activity s1
25   do ... end

27   coordination
     is
29     rule work(p : UserPort)
       is
31       when s1 called by p
         then run s1
33     end

35     rule newUser
       is
37       when createNewUser called by configuration
```

```
      then run newConfiguration
39  end

41  process userManager(p : UserPort) : UserHandler[0..*] do work(p)
        *
    process confHandler : ConfHandler do newUser*
43  end
end
```

Ce simple exemple montre la complexité induite par l'utilisation des ports multiples. Cependant, ils sont nécessaires pour décrire les architectures auto-adaptables, et permettent de gérer proprement les architectures complexes à base de composants.

4.3 Sémantique et simulation d'architectures auto-adaptatives

Le métamodèle utilisé pour représenter les architectures logicielles est suffisamment détaillé pour envisager d'en simuler les instances. Cependant, lors de la conception d'une architecture logicielle, l'architecte n'a que peu de connaissances de la plate-forme d'exécution. Il est alors important que la sémantique associée au métamodèle puisse être paramétrée pour en refléter les propriétés.

En effet, l'expérience d'UML a montré qu'il n'existait pas de sémantique unique pour l'architecture logicielle et que de nombreux aspects dépendent de la plate-forme sur laquelle le système est censé s'exécuter. Pour pallier ce problème, UML a introduit la notion de point de variation sémantique et est devenu alors une sorte de « patron » sémantique que l'utilisateur doit compléter. Ce sont précisément ces points de variations qui vont refléter les caractéristiques les plus importantes de la plate-forme.

Cette section décrit les éléments communs de la sémantique que nous proposons pour les architectures logicielles et identifie clairement les différents points de variations qui permettent de refléter le comportement de la future plate-forme d'exécution. La sémantique choisie s'appuie sur une représentation interne des composants, appelée domaine sémantique, qui permet de décrire la sémantique des communications, celle des processus et finalement celle des composants.

Pour peupler le domaine sémantique, il faut instancier les éléments déclarés par le concepteur avant de pouvoir les simuler. Par exemple, lorsque l'architecte définit le composant composite correspondant au lecteur vidéo, il définit en réalité un type de collaboration composée d'un démultiplexeur, de deux décodeurs, et d'un composant chargé de la synchronisation. A l'exécution, plusieurs instances de cette collaboration peuvent donc coexister.

La figure 4.10 présente les éléments instanciés lors de la simulation d'une architecture. Ces éléments sont les suivants :

Element représente une abstraction des objets présents à l'exécution, que ces objets soient des composants, des ports ou des messages. Chaque élément est une instance d'un type créé par l'utilisateur.

```
context Element:
inv: self.slot->forall(s| self.type.feature->include(s.definition))
```

Slot Chaque *Feature* défini dans un type trouve un équivalent dans un *Slot* qui contient directement la ou les valeurs associées à cette *Feature* pour un élément donné.

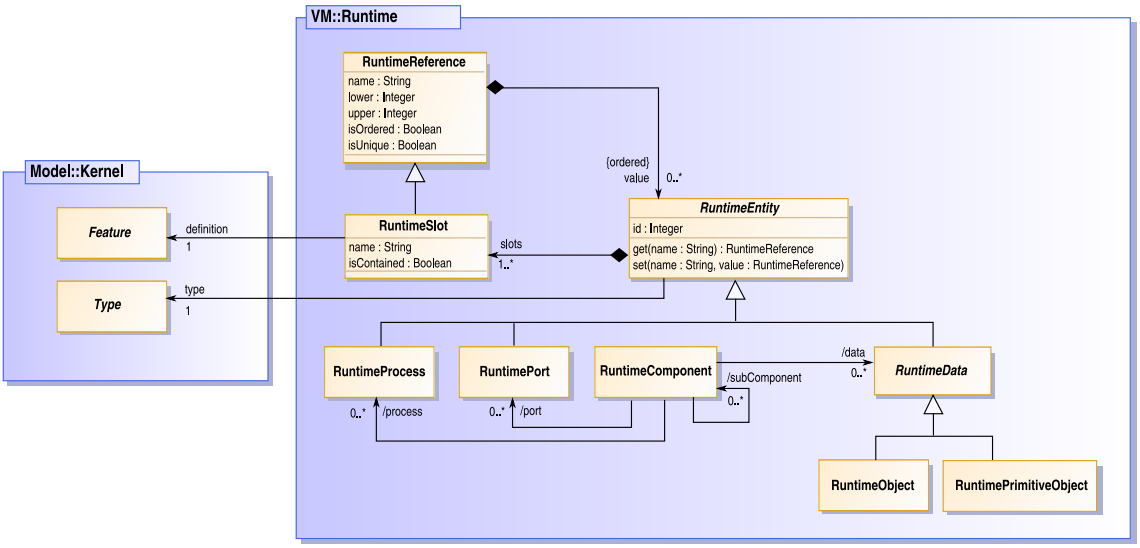

FIG. 4.10 – Modèle général des objets lors de la simulation

Component représente les éléments qui sont des instances de *Component*. Ils contiennent (sous forme de propriétés dérivées) un ensemble de ports, un ensemble de processus et un ensemble de données.

```
context Element:
inv: self.type.oclIsKindOf(Structure::Component)
```

Port représente la notion de port, c'est-à-dire de point de connexion entre un composant et son environnement. Chaque port regroupe un ensemble de messages entrants et un ensemble de messages sortants.

DataElement Les objets de données sont représentés par des objets à part qui pourront être stockés, échangés ou modifiés par les composants. A chaque propriété définie dans une classe de données correspond un *slot* dans l'objet instance. Ce slot peut être une valeur simple ou une collection en fonction de la multiplicité associée à la propriété de la classe.

```
context Element:
inv: self.type.oclIsKindOf(Data::DataType)
```

RunningProcess représente les processus qui coordonnent le fonctionnement d'un composant. L'état courant de chaque processus fait directement référence à l'un des états définis par l'utilisateur.

```
context RunningProcess:
inv: self.slot->forall(s| self.type.feature->include(s.definition))
```

4.3.1 Simulation des communications

La représentation interne des composants lors de l'exécution est proche des éléments déclarés par l'architecte. Les composants, ainsi que leurs ports et les processus qu'ils contiennent sont directement réifiés.

Les ports sont instanciés différemment en fonction de leur multiplicité. Les ports simples (multiplicité $[1..1]$) sont réifiés directement. Les ports multiples, eux, sont représentés par une collection ordonnée de ports simples. Un port p déclaré avec une multiplicité $[x..*]$ est donc représenté par une collection de ports où les ajouts et les suppressions sont possibles. Les ports optionnels (multiplicité $[0..1]$) sont également représentés à l'aide d'une collection. L'utilisation de collections permet de modifier dynamiquement l'architecture.

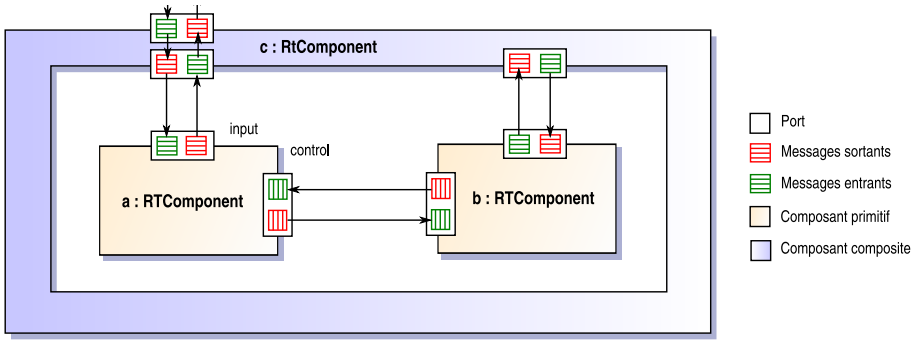

FIG. 4.11 – Structure des composants lors de la simulation

Chaque port est composé de deux « boîtes à lettres » qui contiennent respectivement les messages entrants et sortants du composant. Chaque boîte à lettres est connectée à une autre boîte à lettres formant ainsi une chaîne capable de transporter les messages au travers de l'architecture. La figure 4.11 présente un exemple abstrait d'architecture logicielle où un composant C englobe une collaboration entre deux autres composants appelés A et B. Ces deux composants sont reliés par l'intermédiaire de deux ports dont les boîtes à lettres sont connectées. Ainsi les messages sortant de A par le port p_1 sont redirigés vers le port p_2 du composant b et vice versa. Les ports délégués, c'est à dire les ports qui traversent un composant composite, sont réifiés par deux ports. L'un de ces ports est placé sur la face externe du composant composite et l'autre est placé sur sa face interne. Leurs boîtes à lettres sont reliées pour permettre aux messages de traverser le composant composite et ainsi d'arriver jusqu'aux composants internes de la collaboration.

La connexion qui existe entre deux boîtes à lettres est unidirectionnelle et l'acheminement des messages est à la charge de l'émetteur. Il s'agit donc d'une communication de type *push* par opposition à une communication de type *pool* où la transmission des messages est à la charge du destinataire. Ce choix n'influence pas le comportement du système et n'est donc pas considéré ici comme un point de variation sémantique.

4.3.2 Simulation des processus

Les processus déclarés au sein d'un composant sont instanciés sous la forme de *RuntimeProcess* (cf. figure 4.10). Les processus simples, avec une multiplicité $[1..1]$ sont directement réifiés, alors que les processus potentiellement multiples sont stockés à l'aide d'une collection. Lors d'une simulation, chaque processus est défini par son état courant, c'est-à-dire par une référence vers l'un des états possibles de son type.

La présence d'un message dans une boîte à lettres entrante correspond à un appel de service. Pour mémoire, puisque les événements traduisent les appels de services sur les instances de port,

la présence d'un message entrant dans un port déclenche l'événement associé. Dans [21], la sémantique associée aux processus est décrite de manière impérative par la procédure *step* présentée par le listing 4.8. Cette procédure, conçue pour des processus n'ayant qu'une seule boîte à lettres entrante, ne s'applique pas directement aux processus utilisés ici, mais de manière informelle, cette procédure est la suivante :

1. En fonction d'un critère de priorité, on sélectionne, parmi les événements survenus, les événements les plus pertinents (par ordre chronologique, par priorité, etc ...)
2. De manière arbitraire, on départage ces événements en en choisissant un particulier.
3. Cet événement est utilisé pour calculer l'ensemble des transitions sortantes de l'état courant qui sont potentiellement déclenchables.
4. De manière arbitraire ou non, on sélectionne une transition particulière que l'on exécute.

Listing 4.8 – Sémantique opérationnelle pour des processus n'ayant qu'une boîte à lettres entrante

```
operation step() : Void is
do
  var es : OrderedSet<Event>  init eventPool.select{s | ... }
  var e : Event init es.one
  var ts : OrderedSet<Transition> init outgoing.select{ t | t.
      isTriggeredBy(e)}
  var t : Transition init ts.one
  t.fire()
  eventPool.remove(e)
end
```

Deux solutions sont possibles pour adapter cette procédure à des processus ayant plusieurs boîtes à lettres entrantes. La première consiste à garder le même schéma, et à sélectionner dans chaque boîte à lettres, un nouveau message, puis, à sélectionner les transitions sensibilisées par l'ensemble des couples (port, message) ainsi obtenus. On peut alors choisir une transition et la déclencher pour faire ainsi progresser le processus. Cette solution pourtant simple, a le désavantage d'amener à une sémantique très « stricte » et parfois contre-intuitive. Considérons par exemple la situation suivante, où les ports p_1 ont reçu les messages suivants :

- $p_1 = [m_1 : call(s_1);\ m_2 : call(s_2)]$
- $p_2 = [m_3 : call(s_1);\ m_4 : call(s_2)]$

Que ce soit avec une sémantique FIFO ou LIFO pour l'accès aux messages, l'expression « s_1 called by p_1 and s_2 called by p_2 » sera évaluée à *faux* et ne déclenchera aucune transition. En effet, avec un accès FIFO, (m_1, m_3) seront sélectionnés mais ils ne vérifient pas l'expression en question. De même, avec une sémantique LIFO, (m_2, m_4) seront sélectionnés, mais ils ne vérifient pas non plus l'expression en question. Pourtant, les couples (m_1, m_4) et (m_2, m_3) satisfont l'événement de l'expression, ce qui peut être contre-intuitif pour l'utilisateur, lorsqu'il utilise les conjonctions d'événements pour décrire des synchronisations entre les ports.

L'autre sémantique, plus souple, consiste à évaluer d'abord tous les événements associés aux transitions sortant de l'état courant puis de choisir une transition parmi celles dont l'événement est survenu. Cette évaluation des événements repose sur un accès libre aux boîtes à lettres. Ainsi, la conjonction est vraie, s'il existe un message se référant aux services cités sur tous les ports en question, alors qu'une disjonction sera vraie s'il existe au moins un port (parmi ceux cités) qui contienne un message se référant aux services cités. L'accès FIFO ou LIFO étant alors utilisé pour

sélectionner le bon couple de messages quand plusieurs sont possibles. Dans l'exemple précédent, deux couples sont possibles (m_1, m_4) et (m_2, m_3) : le premier correspond à un accès en FIFO alors que le second correspond à un accès en LIFO.

Listing 4.9 – Sémantique opérationnelle pour des processus généraux

```
operation step() : Void is
do
  var ee : EventEvaluator init EventEvaluator.new
  var ts : OrderedSet init outgoing.select{ t | ee.isTriggered(t) }
  var t : Transition init ts.one
  t.fire()
end
```

Le listing 4.9 propose une autre version de la procédure *step* reflétant cette seconde solution. On peut noter que le calcul des transitions tirables se fait à l'aide d'un évaluateur d'événements. L'accès aux boîtes à lettres n'est plus directement explicité mais reste nécessaire pour déterminer quels sont les messages à prendre en compte (notamment s'ils contiennent des paramètres).

4.3.3 Simulation des composants

L'utilisation d'une description architecturale hiérarchique permet de simuler l'ensemble de l'architecture en simulant le comportement du composant englobant. Basée sur l'algorithme de simulation des processus, la simulation des architectures logicielles suit le principe suivant :

1. On simule d'abord la progression des messages entrant dans le composant. Il s'agit de faire progresser les messages d'une boîte à lettres sortante vers un boîte à lettres entrante.
2. On simule les processus internes du composant qui vont produire de nouveaux messages sortant du composant.
3. Si le composant est composite,
   (a) On fait ensuite progresser les messages sortant sur les ports internes car ils sont à destination des sous-composants et sont nécessaires à la simulation des sous-composants.
   (b) On simule les composants internes.
   (c) On simule les processus internes au composant.
4. On fait progresser les messages sortant du composant.

On voit également apparaître deux nouveaux points de variation sémantiques liés respectivement au choix des sous-composants et au choix des processus. Le premier, qui détermine l'ordre de simulation des sous-composants au sein d'un composant composite, peut être arbitraire ou basé sur un système de priorité. Le second, qui détermine l'ordre de simulation des processus au sein d'un composant, se décline de la même façon.

4.3.4 Le simulateur TANGRAM

L'outil TANGRAM inclut un moteur de simulation qui fait évoluer une architecture en fonction d'un scénario de test, lui même représenté par un ou plusieurs composants refermant l'architecture logicielle.

Deux principales versions du simulateur ont été développées. La première, purement orientée-modèle, est une implémentation en Kermeta du métamodèle présenté dans le chapitre 4. Dans cette première version, les données sont modélisées directement en Kermeta et les expressions (ou les actions) qui les concernent sont donc évaluées à la volée par l'interpréteur Kermeta. Cependant, le chargement dynamique d'un interpréteur Kermeta réduit grandement les performances de la simulation. De plus, l'utilisation de Kermeta comme DSL pour représenter les données, tient plus de l'astuce que d'une nécessité réelle.

Par ailleurs, l'intégration de politiques d'adaptations et notamment de leur sémantique floue dégrade encore plus les performances pour deux raisons. D'une part, l'intégration d'une sémantique floue nécessite des calculs numériques et donc de nombreux accès aux librairies mathématiques de Java. D'autre part, le choix d'une sémantique continue pour les politiques d'adaptations, c'est-à-dire d'une sémantique où l'adaptation est continuellement réévaluée, nécessite d'effectuer plus d'étapes de simulation pour obtenir des résultats exploitables (notamment à cause des communications, principalement asynchrones) ce qui dégrade également les performances.

Pour pallier ce problème de performance, une version « pure Java » a été développée. Elle intègre son propre moteur de logique floue qui est utilisé également dans le contrôleur Fractal TANGRAM dont nous parlerons par la suite. Cette implémentation en Java conserve toutefois les principes chers à l'ingénierie des modèles, à savoir, une séparation forte entre les domaines syntaxiques et sémantiques ainsi que des transformations permettant de raffiner les modèles. Les principaux éléments du moteur de simulation TANGRAM sont donc :

- Un analyseur syntaxique qui permet de construire un modèle à partir d'une description textuelle. Il s'agit ici d'instancier les éléments décrits à l'aide de la syntaxe présentée dans la section précédente.
- Une machine virtuelle permettant d'exécuter un modèle d'architecture et de récupérer les données mesurées lors de la simulation de ses composants.
- Une console permettant de lancer les simulations, mais aussi de mettre au point les reconfigurations architecturales.

4.4 Validation *a priori* d'architectures auto-adaptables

La seule simulation des architectures n'est pas suffisante pour permettre de mettre en place un processus de validation *a priori* des architectures auto-adaptables. En effet, si la simulation peut détecter des erreurs ou des incohérences introduites par les intro-actions, il est primordial de pouvoir vérifier que les politiques d'adaptation remplissent bien leur rôle. Par exemple, dans le cas du système vidéo, il est essentiel de pouvoir s'assurer que le déploiement d'un filtre vidéo noir et blanc réduit bien le volume de données transmis sur le réseau. En d'autres termes, puisque ce filtre noir et blanc doit être déployé en fonction de la bande passante, il faut pouvoir faire varier la bande passante et mesurer la taille des données produites par le système. Ainsi on est en mesure de vérifier que la logique de l'adaptation est respectée.

4.4.1 Critère de validation d'une architecture auto-adaptable

L'objectif implicite d'une politique d'adaptation peut généralement se ramener à un problème de stabilisation. Il s'agit de maintenir une propriété donnée dans une enveloppe précise en fonction des variations de l'environnement. Dans le cas du système vidéo par exemple, la sélection d'un

filtre vidéo permet de réduire le volume de données transmis au client, qui, de son point de vue, ne cherche qu'à stabiliser la vitesse d'émission des images. Ainsi, si les images sont petites, le temps d'émission sera plus court et la vitesse sera stabilisée par conséquent.

Pour pouvoir valider une politique d'adaptation, il faut donc observer la propriété à stabiliser ce qui nécessite l'existence de sondes dédiées. Par exemple, pour mesurer la vitesse d'émission des images sur le réseau, il faut disposer d'une sonde dédiée qui peut être portée par le composant *NetworkManager*. Puisque cette sonde est réifiée dans l'architecture, elle peut donc être utilisée lors de la simulation pour mesurer les variations de vitesse d'émission. Du point de vue architectural, il n'est pas choquant d'embarquer dans le système des moyens de vérification et cela peut permettre de mettre en place des contrats pour les politiques d'adaptation.

Cependant, le test d'une politique d'adaptation n'est pas un test booléen comme le test traditionnel, mais plutôt une valeur de satisfaction. Comment caractériser en effet le caractère stable d'une propriété dans le temps. Faut-il tenir compte des pics d'instabilité brefs et occasionnels ? Faut-il accepter un dépassement très faible ? C'est là un problème difficile qui n'est pas abordé dans ce document. Nous proposons une validation semi-automatique où le concepteur des politiques d'adaptation est le seul à pouvoir jouer le rôle de l'oracle. En observant l'évolution d'une propriété lors d'un scénario donné, il décide par lui-même du caractère correct ou incorrect du résultat. Une solution possible est d'utiliser la logique floue pour décrire de manière qualitative la qualité attendue pour un scénario donné. Cette idée n'a pas été encore explorée et est décrite plus en détails dans les perspectives (cf. chapitre 8, section 8.4 page 107).

4.4.2 Simulation des effets de bords extra-fonctionnels

L'un des problèmes liés à la simulation est de réussir à prendre en compte les différents effets de bords qui existent entre les propriétés fonctionnelles ou extra-fonctionnelles d'une architecture. Comme évoqué précédemment à propos du système vidéo, la sélection d'un filtre vidéo influe sur la vitesse d'émission des images. Il existe effectivement dans l'absolu une relation implicite entre la taille d'une image et le temps nécessaire à son émission sur le réseau et c'est cette relation qui guide le concepteur lorsqu'il choisit les règles d'adaptation. Cependant le modèle proposé pour le système vidéo ne reflète pas cette relation implicite et il est donc impossible de mesurer la vitesse d'émission du *NetworkManager*.

Pour prendre en compte cette relation, nous proposons de la réifier en utilisant le même langage que celui utilisé pour le reste de l'architecture. En effet, chaque propriété extra-fonctionnelle a potentiellement des effets de bords sur les autres et l'utilisation de la réification permet de ne pas se restreindre à un sous-ensemble de propriétés extra-fonctionnelles donné. Dans le cas du système vidéo et de la vitesse d'émission, il faut que le composant *NetworkManager* qui envoie les données sur le réseau, prenne en compte à la fois la taille des données et la bande passante disponible à un moment donné.

La réification de ces effets de bord implicites à l'aide du langage architectural permet de les intégrer plus facilement dans l'architecture. L'intégration se fait à la fois au niveau structurel et au niveau comportemental.

Intégration structurelle

La première étape consiste donc à réifier les éléments nécessaires pour prendre en compte les effets de bord. Dans le cas du système vidéo, il s'agit de connecter le composant *NetworkManager*

à un composant *Network*, placé à l'extérieur du système. Ce composant réifie le réseau et fournit donc la bande passante nécessaire à l'émission des images. Le composant *SpeedMonitor* réifie la sonde qui va mesurer la vitesse d'émission des images. La figure 4.12 résume les modifications architecturales apportées au système vidéo.

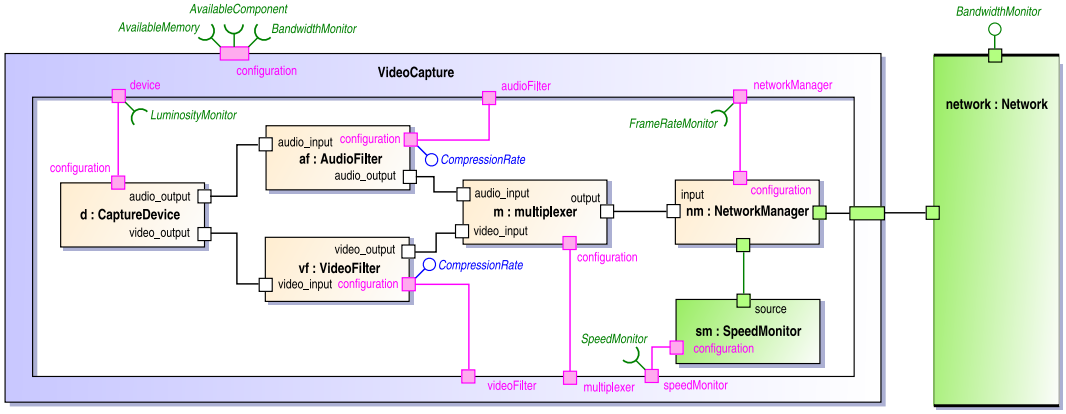

FIG. 4.12 – Intégration structurelle des effets de bords dans l'architecture du lecteur vidéo

Intégration comportementale

Il faut également modifier et/ou décrire le comportement des composants qui ont été ajoutés dans le système. Dans le cas du lecteur vidéo, le composant *SpeedMonitor* sera notifié à chaque émission d'une image et pourra donc calculer la vitesse d'émission d'une image. Pour cela, le composant *NetworkManager* doit tout d'abord requérir suffisamment de bande passante de la part du réseau pour pouvoir envoyer les informations dans leur totalité. Pour pouvoir faire cela, il faut donc nécessairement que les images soient dotées d'une taille ce qui implique de modifier également les données modélisées.

Les données sont donc modélisées comme suit. Trois types d'information (représentés par des classes) circulent au sein du système vidéo. Le périphérique de capture produit à la fois des trames vidéo et des trames audio qui sont transmises aux filtres respectifs. Ces deux filtres produisent des trames compressées qui sont transmises au multiplexeur pour être assemblées et en faire des trames « multiplexées » qui seront, ensuite, transmises sur le réseau par le *NetworkManager*.

4.4.3 Exemple de validation

Cette section présente plus en détails un exemple de simulation exécutée sur le cas du système vidéo. Cet exemple très simple a pour but de montrer l'intérêt de vérifier *a priori* les politiques d'adaptation, car en prévoir les effets est une chose délicate, même sur des systèmes simples. Le premier scénario de simulation que nous proposons décrit une situation où la bande passante s'écroule : d'une valeur *high*, elle passe à une valeur *low*. Le comportement du système sans aucune adaptation est présenté par la figure 4.13.

La figure 4.13 présente sous la forme d'un graphique les différents éléments mesurés pendant la simulation. La bande passante est présentée en pointillés rouges, la vitesse du *NetworkManager*

en bleu, et les taux de compression associés aux filtres vidéo et audio sont représentés en vert et en orange. Comme le montrent ces premiers résultats, face à une chute de la bande passante, le système se comporte de manière assez intuitive et la vitesse d'émission du *NetworkManager* chute également.

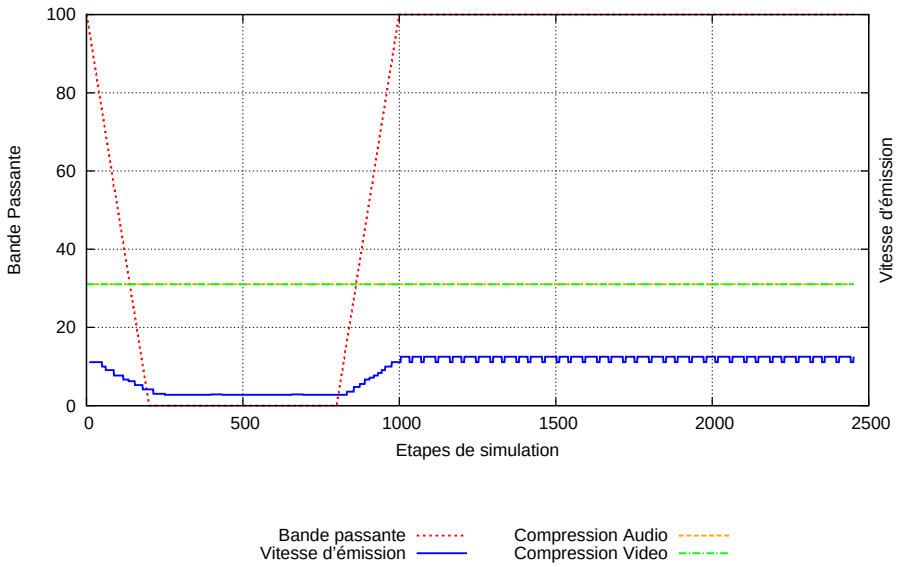

FIG. 4.13 – Comportement d'un système vidéo classique, sans adaptation, face à une chute de la bande passante

Dans cette simulation, on vérifie le comportement d'une politique d'adaptation censée contrôler la vitesse d'émission des informations en fonction d'une bande passante qui fluctue. L'architecture utilisée est donc celle présentée précédemment par la figure 4.12. Pour contrôler la vitesse d'émission, le concepteur propose la politique d'adaptation présentée par les extraits de code 4.10, 4.11 et 4.12. Son objectif est de déployer le filtre couleur lorsque la bande passante est bonne et d'augmenter les différents taux de compression lorsque qu'elle diminue. Cet exemple est présenté à l'aide de la syntaxe textuelle utilisée dans l'outil TANGRAM.

Listing 4.10 – Adaptations architecturales utilisées pour contrôler la vitesse d'émission du capteur vidéo

```
1  policy SpeedControl
   is
3    reconfiguration deployColorFilter
     is
5      videoFilter.stop
       disconnect videoFilter.input
7      disconnect videoFilter.output
       create ColorFilter named newFilter
9      videoFilter := newFilter
       connect videoFilter.input to device.video_output
11     connect videoFilter.output to multiplexer.video_input
```

```
    videoFilter.start
13  end

15  reconfiguration deployBlackAndWhiteFilter
    is
17    videoFilter.stop
      disconnect videoFilter.input
19    disconnect videoFilter.output
      create BalckAndWhiteFilter named newFilter
21    videoFilter := newFilter
      connect videoFilter.input to device.video_output
23    connect videoFilter.output to multiplexer.video_input
      videoFilter.start
25  end
```

Le listing 4.10 présente les deux intro-actions architecturales utilisées pour déployer respectivement le filtre couleur et le filtre noir et blanc. Il s'agit à chaque fois de déconnecter le filtre vidéo actuellement en place, en prenant soin de l'arrêter auparavant et de le remplacer par le filtre pertinent.

Listing 4.11 – Propriétés observées et modifiées pour contrôler la vitesse d'émission du capteur vidéo

```
1   -- Proprietes observables/modifiables
    domain Compression : Real
3     evolves in [5, 30] as 'weak' 'medium' 'strong'

5   property audioCompression : Compression
      actuator is setCompression on audioFilter
7
    property audioCompression : Compression
9     actuator is setCompression on videoFilter

11  property bandwidth : Integer
      evolves in [75000, 200000] as 'low' 'medium' 'high'
13    sensor is getBandwidth on configuration
```

Le listing 4.11 décrit les différentes propriétés observées et modifiées dans le contexte. Il s'agit de la bande passante accessible via le port de configuration du système vidéo et des deux taux de compression, accessibles via les ports de configuration des filtres vidéo et audio. Chaque propriété est associée explicitement ou implicitement à un domaine linguistique défini par un intervalle de valeur et par un ensemble de termes adéquats.

Listing 4.12 – Une politique d'adaptation pour contrôler la vitesse d'émission du capteur vidéo

```
1   -- Regles configuratives
    when bandwidth is 'low'
3     then audioCompression is 'medium' or moderatly 'strong'

5   when bandwidth is 'medium'
      then audioCompression is 'medium'
7
    when bandwidth is 'high'
9     then audioCompression is 'weak'
```

```
11    when bandwidth is 'low'
        then videoCompression is very 'strong'
13
      when bandwidth is 'medium'
15      then videoCompression is 'medium'

17    when bandwidth is 'high
        then videoCompression is very 'weak'
19
      -- Regles architecturales
21    when bandwidth is 'low'
        then utility of deployColorFilter is 'low'
23
      when bandwidth is 'medium'
25      then utility of deployColorFilter is 'low'

27    when bandwidth is 'high'
        then utility of deployColorFilter is 'high'
29
      when bandwidth is 'low'
31      then utility of deployBlackAndWhiteFilter is 'high'

33    when bandwidth is 'medium'
        then utility of deployBlackAndWhiteFilter is 'low'
35
      when bandwidth is 'high'
37      then utility of deployBlackAndWhiteFilter is 'low'
    end
```

Finalement, le listing 4.12 présente les règles d'adaptation configuratives et architecturales utilisées à la fois pour configurer les différents taux de compression et pour déployer, si besoin est, le filtre couleur. L'idée générale exprimée par les règles configuratives est de compresser plus l'image que le son de manière à garder des informations sonores exploitables même si, au pire, les informations visuelles sont très dégradées voire inexistantes. Les résultats de la simulation du système vidéo avec cette politique d'adaptation déployée dans le système vidéo sont présentés par la figure 4.14. On peut noter tout d'abord que le système a bien réagi en réajustant de manière continue, les taux de compression associés aux filtres audio et vidéo. De plus l'effet sur la vitesse du *NetworkManager* est correcte puisqu'elle se rétablit au fur et à mesure de simulation.

Un point intéressant dans les résultats présentés par la figure 4.14 est le délai entre la mise à jour des taux de compression et le retour à la vitesse d'émission initiale. Ce délai est dû au caractère asynchrone des communications entre les différents composants. L'architecture du système vidéo est en fait un *pipeline* de traitements opérant sur un flux de données et même si le système ajuste les taux de compression lorsqu'il détecte une variation de la bande passante, une partie des trames faiblement compressées a déjà été transmise au *NetworkManager*. L'impact de la mise à jour des taux de compression n'est donc visible que lorsque toutes les trames plus faiblement compressées ont été transmises et ce, avec une bande passante faible. Il y a en effet une différence entre les vitesses d'impact des propriétés extra-fonctionnelles : si la bande passante influe directement, comme nous l'avons vu grâce aux résultats de la première simulation (cf. figure 4.13), les taux de compression ont un impact bien moins dynamique. Cette différence d'impact est encore plus flagrante lors de variations ponctuelles dans la bande passante comme le montre la figure 4.15.

Dans le cas de la figure 4.15, le système vidéo auto-adaptatif est confronté à une chute ponctuelle

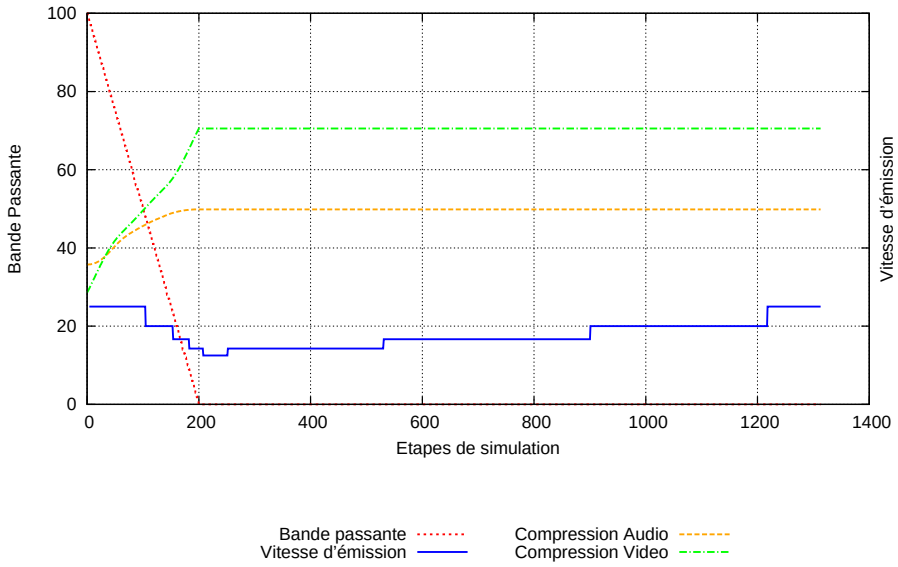

FIG. 4.14 – Comportement d'un système vidéo auto-adaptatif face à une chute de la bande passante

de bande-passante. Comme dans le cas précédent le système réagit exactement comme spécifié dans la politique d'adaptation *SpeedControl*. Cependant, dû à la latence induite par les communications asynchrones entre les composants du système (les informations sont mises en tampon par chaque composant qui travaille indépendamment des autres), l'effet de l'adaptation se fait ressentir alors que la bande passante est revenue à son état initial. Une partie des données a été sur-compressée inutilement, ce qui augmente bien la vitesse d'émission, mais à « contre temps ».

Cette relation complexe et implicite entre la structure du système et la performance des politiques d'adaptation montre combien il est difficile d'anticiper le comportement et l'efficacité d'une politique d'adaptation donnée, dans un environnement donné. Cette relation justifie à elle seule la vérification *a priori* de systèmes auto-adaptatifs et particulièrement des politiques d'adaptation. En effet, on voit bien ici que, puisque la structure du système a une influence sur les performances des politiques d'adaptation, une vérification *a posteriori* entraînerait une modification de l'architecture même du système ce qui reviendrait, dans la plupart des cas, à reprendre la majeur partie de la conception et impliquerait un coût clairement non négligeable.

4.5 Synthèse et discussion

Nous avons présenté dans ce chapitre un métamodèle qui permet de capturer les quatre facettes principales d'une architecture logicielle, à savoir : la structure, les données, le comportement, et la coordination. Ce métamodèle n'apporte pas de concepts originaux mais définit un langage d'architecture clos et suffisamment détaillé pour envisager la simulation des modèles. D'autre part, ce modèle respecte les propriétés que nous avons énoncées en début de chapitre, à savoir : l'isolation, le discernement et l'autonomie. Un composant est isolé de son environnement et ne communique

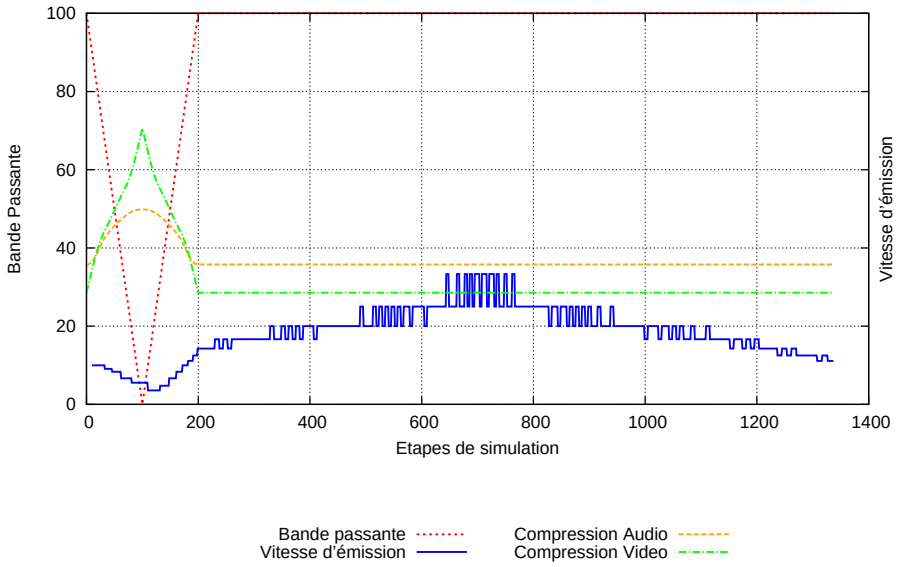

FIG. 4.15 – Comportement d'un système vidéo auto-adaptatif face à une chute ponctuelle de la bande passante

avec ce dernier que par les ports qui les relient. Toutes les informations qu'ils échangent transitent ainsi par ces ports. La présence des ports et de processus chargés de la coordination renforce la capacité de discernement des composants qui peuvent alors exposer un même service sur plusieurs ports, en offrant des implémentations différentes. Pour ce qui est de l'autonomie, les composants ne sont pas réellement autonomes puisque leur comportement est assujetti à la réception de stimuli provenant de l'environnement, mais ils sont capables de prendre des décisions basées sur des politiques d'adaptation floues.

Finalement, nous avons proposé une sémantique permettant de simuler le comportement des composants. Cette sémantique présente quatre points de variation qui permettent de lui faire refléter une plate-forme d'exécution particulière. Ces points de variation sont : la sélection d'un message, la sélection d'une transition, la sélection d'un processus et la sélection d'un sous-composant. Ces quatre critères de sélection permettent de paramétrer, dans une certaine mesure, le comportement de la simulation.

Plus important encore, comme l'a montré l'exemple du système vidéo, il s'avère que la vérification *a priori* est plus que nécessaire dans le cas des systèmes auto-adaptatifs car la complexité qui lie leur performance, leur structure, et l'environnement rend quasiment impossible la prédiction de leur comportement dans une situation donnée. L'outil que nous proposons, est construit sur les techniques récentes en matière de méta-modélisation exécutable et fait un premier pas vers le développement de systèmes auto-adaptatifs fiables.

Toutefois, pour être effective et utilisable, la simulation de modèles nécessite de réifier en détails les relations ou les effets de bords entre les différentes propriétés fonctionnelles et extra-fonctionnelles du système. La pertinence des vérifications *a priori* repose donc clairement sur

la pertinence de la description de ces même effets de bords. De plus, la réification conduit parfois à l'ajout de composants qui n'ont pas d'avenir en dehors de la simulation. Pour évaluer la consommation mémoire du système, il faut, en effet, réifier le concept de mémoire et décrire la consommation des différents composants. Si cela permet en effet de mesurer la consommation de mémoire, les ports et les composants ajoutés pour cela ne sont alors pas exploitables en dehors de la simulation.

Les travaux présentés dans ce chapitre, et plus précisément les aspects autour de la validation *a priori* de modèles ont été publié à ERTS 2008 [20].

Chapitre 5

Intégration dans la plate-forme Fractal

Ce chapitre décrit l'implémentation d'un mécanisme d'exécution pour les politiques d'adaptation qualitatives telles qu'elles ont été introduites dans les chapitres précédents. Il s'agit d'une extension de la plate-forme Julia (implémentation de référence de Fractal) qui prend la forme d'un contrôleur dédié à l'adaptation.

5.1 Conception d'un contrôleur Fractal dédié

La plate-forme Fractal (cf. section 2.5.3 page 31) offre la possibilité d'exécuter des applications à base de composants. Elle est intègre un modèle de composant où les composants exposent des interfaces (au sens Java) à leur environnement.

5.1.1 Extension de Fractal à l'aide de contrôleurs

Les composants Fractal ont la particularité de définir des interfaces de contrôle, c'est-à-dire des interfaces dédiées à la gestion de problèmatiques extra-fonctionnelles, telles que la configuration, le cycle de vie, la qualité de service. Un certain nombre d'interfaces existent par défaut dans la plate-forme Julia, telles que l'interface *AttributeController* pour la configuration ou l'interface *LifeCycleController* pour le cycle de vie. Ces interfaces de contrôle vont de paire avec l'idée qu'un composant Fractal composite n'est qu'une membrane dédiée à la gestion de problèmatiques extra-fonctionnelles. Dans ce sens, une membrane Fractal se rapproche de la notion conteneur (*container* en anglais) que l'on trouve sur d'autres plates-formes comme J2EE par exemple. La plate-forme Julia offre également un mécanisme de *Mixin* qui permet de combiner simplement différents contrôleurs Fractal au sein d'une même membrane. La construction d'une membrane peut donc se faire simplement en déclarant les contrôleurs nécessaires et en laissant le soin au système de mixin de constuire les objets Java associés (dans le cas de Julia).

Dans le cadre des systèmes auto-adaptatifs, les problèmes liés à l'adaptation sont extra-fonctionnels puisqu'ils ne modifient pas la fonction première du système. Le système vidéo peut parfaitement être envisagé sans aucune capacité d'auto-adaptation : ses performances dans des environnements dégradés ne seront simplement pas optimales. De plus, c'est le composant composite qui a la responsabilité du bon déroulement de l'adaptation. Ce mécanisme de contrôleur est donc parfaitement adapté au besoin de l'auto-adaptation. L'extension de Fractal qui est proposée ici, prend donc naturellement la forme d'un contrôleur dédié à l'auto-adaptation dans les composants

composites et peut donc être composée avec d'autres problématiques. Vue sous la forme d'une interface de contrôle, ce contrôleur permet :

- de définir dynamiquement les politiques d'adaptation relatives à un composant composite.
- de contrôler (arrêter et démarrer) le processus d'adaptation : observation, décision, action.
- de configurer l'adaptation, notamment le seuil d'utilité et la fréquence du cycle d'adaptation.

5.1.2 Principe du contrôleur d'adaptation

Le principe général de fonctionnement du contrôleur d'adaptation est décrit par la figure 5.1. Le contrôleur est construit au dessus de la plate-forme Fractal Julia, munie de son extension FScript qui permet d'exécuter dynamiquement des adaptations architecturales. Ce contrôleur réalise les trois étapes du cycle d'adaptation de la manière suivante :

Observation Le contrôleur d'adaptation utilise les interfaces *AttributeController* définies de manière standard dans Fractal pour accéder à l'environnement. Le contrôle suppose donc que toutes les informations nécessaires sont disponibles sur ce type d'interface. Une propriété nommée *bandwidth* par exemple devra donc être disponible, en lecture, sous la forme d'un service nommé *getBandwidth*. Sur la figure 5.1 le contrôleur extrait les informations directement de la plate-forme Fractal (ici, l'application et la plate-forme sont confondues). Toutes les observations passent donc par le biais de « *Getters* ».

Décision Le contrôleur utilise ensuite les informations mesurées pour effectuer les adaptations nécessaires. Les règles de décision sont décrites de manière textuelle sous la forme de politiques d'adaptation. Ces politiques sont chargées dans le contrôleur depuis des fichiers lors de l'initialisation du contrôleur. Leur évaluation, via l'utilisation d'un moteur de logique floue interne, lui permet de déclencher les adaptations nécessaires. Dans la figure 5.1 trois politiques d'adaptation sont chargées dans le contrôleur.

Action Les intro-actions configuratives utilisent également les interfaces *AttributeController* fournies par les composants de l'architecture sous-jacente. Le contrôleur les utilise alors en écriture via des « *Setters* » de la forme *SetCompressionRate* pour un taux de compression par exemple. La figure 5.1 montre cet accès en écriture à la configuration des composants. Le déclenchement des actions architecturales se fait via l'interpréteur FScript. Le contrôleur demande alors l'exécution d'une procédure d'adaptation architecturale décrite sous la forme d'une action FScript.

Avant son exécution, le contrôleur d'adaptation doit être configuré. La configuration du contrôleur définit plusieurs paramètres dont la fréquence d'adaptation, c'est à dire le temps écoulé entre deux adaptations (cycle observation, décision, action). Elle définit également le seuil d'utilité au dessus duquel le contrôleur déclenche les actions d'adaptation architecturale.

5.1.3 Spécialisation des politiques d'adaptation pour Fractal

Le contrôleur proposé ici peut exécuter des politiques d'adaptation décrites dans des fichiers séparés. La syntaxe associée diffère légèrement de celle utilisée dans le moteur de simulation car elle fait référence à des éléments architecturaux existants sur la plate-forme Fractal à l'aide d'expressions FPath. Les actions de reconfigurations architecturales font directement référence à des fichiers externes contenant les actions FScript correspondantes. Par ailleurs, les gardes associées à chaque règle d'adaptation sont décrites à l'aide du langage FPath. Les extraits de fichiers suivants

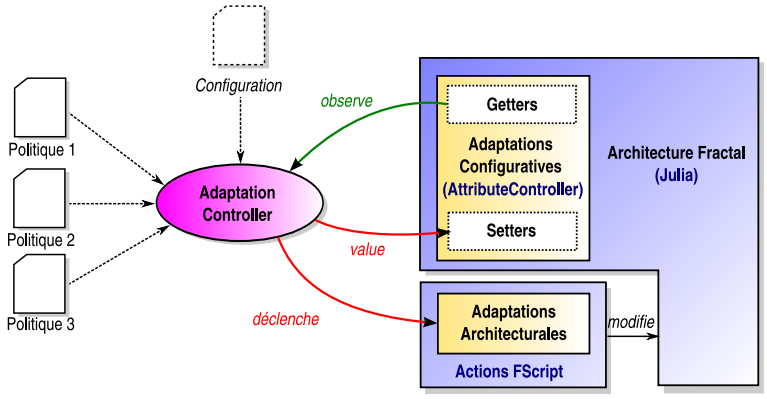

FIG. 5.1 – Principe général du contrôleur d'adaptation pour Fractal

décrivent une politique d'adaptation permettant de sélectionner le bon filtre vidéo dans le cas du système vidéo.

Intro-actions architecturales Les actions d'adaptation architecturales sont décrites dans des fichiers séparés. Comme l'a montrée la section 2.5.3, les actions FScript sont décrites de manière impérative à l'aide d'un langage d'actions dédié. Chaque politique d'adaptation renvoie donc directement aux actions FScript qu'elle est susceptible de déclencher. L'exemple 5.1 est extrait du système vidéo et fait donc référence à deux fichiers FScript décrivant respectivement les procédures pour déployer le filtre noir et blanc et pour déployer le filtre couleurs.

Listing 5.1 – Référence à des actions FScript existantes

```
1 policy SelectVideoFilter
2 is
        reconfiguration deployBWFilter is "deployBWFilter.fscript"
4       reconfiguration deployColorFilter is "deployColorFilter.fscript"
```

Variables linguistiques Chaque politique d'adaptation décrit également les variables linguistiques qu'elle est susceptible d'utiliser. Ces variables peuvent représenter soit des valeurs observées dans l'environnement, soit des valeurs associées à la configuration locale des composants. Pour cela, chaque variable est associée à son domaine d'entrée ainsi qu'à une terminologie associée. Également extrait du système vidéo, l'exemple 5.2 présente les variables linguistiques associées à la bande passante (donnée contextuelle) et au taux de compression (donnée de configuration).

Listing 5.2 – Description des variables linguistiques

```
1       property bandwidth : Real
2               evolves in [0, 10000] as 'low' 'medium' 'high'
                sensor is getBandwidth on networkManager
4
        property compressionRate : Real
6               evolves in [0, 1] as 'light' 'standard' 'strong'
                sensor is getCompressionRate on filter
8               actuator is getCompressionRate on filter
```

Règles d'adaptation Finalement, les règles d'adaptation sont décrites telles qu'elles ont été présentées dans le chapitre 3. Les règles architecturales influencent l'utilité des actions architecturales précédemment associées aux fichiers FScript et les règles configuratives influencent les variables qui concernent la configuration des composants locaux. L'exemple 5.3 présente un exemple de règle architecturale et un exemple de règle configurative.

Listing 5.3 – Description des règles d'adaptation

```
1         when bandwidth is 'low' and luninosity is 'medium'
2                 if size($context/child::filter) > 0
                  then utility of removeCache is 'high'
4
          when bandwidth is 'medium'
6                 if size($context/child::filter) == 0
                  then compressionRate is 'standard'
8
end policy
```

5.1.4 Conception du contrôleur d'adaptation

De manière plus concrète, le contrôleur offre une interface relativement simple, représentée en haut à gauche de la figure 5.2. La méthode *loadPolicy* permet de charger une politique d'adaptation. Le contrôleur d'adaptation implémente également l'opérateur de composition présenté dans le chapitre 3 et il compose les différentes politiques qu'il charge successivement. Les opérations *startAdaptation* et *stopAdaptation* respectivement démarre et arrête le processus d'adaptation alors que les deux opérations restantes permettent de configurer la fréquence de l'adaptation ainsi que le seuil d'utilité utilisé dans l'algorithme d'adaptation présenté par le listing 3.2 page 42.

Lorsque l'opération *startAdaptation* est appelée, le contrôleur instancie un autre processus chargé de déclencher les adaptations de manière périodique en appelant une opération encapsulant l'algorithme de contrôle (algorithme 3.2) présenté page 42. Le contrôleur inclut également un analyseur syntaxique permettant de charger en mémoire les politiques décrites textuellement dans des fichiers externes. Chargées en mémoire, ces différentes politiques d'adaptations représentent la base de connaissances du système de décision intégré dans le contrôleur.

5.2 Génération de code pour Fractal

Pour rendre opérationnelle l'utilisation des politiques d'adaptation définies lors de la conception, il est nécessaire de les transformer pour qu'elles soient exploitables par le contrôleur Fractal. Le modèle structurel proposé dans TANGRAM et le modèle Fractal se distinguent en plusieurs points :

- Le modèle Fractal est un modèle purement structurel alors que le modèle TANGRAM explicite également le comportement et les données manipulées par les composants.
- Les composants TANGRAM exposent des ports complexes, c'est-à-dire des ports qui requièrent et fournissent des interfaces à leur environnement. Les composants Fractal, eux, requièrent ou fournissent directement des interfaces, sans les regrouper sur des ports explicitement nommés : ce sont les interfaces qui sont nommées en Fractal.

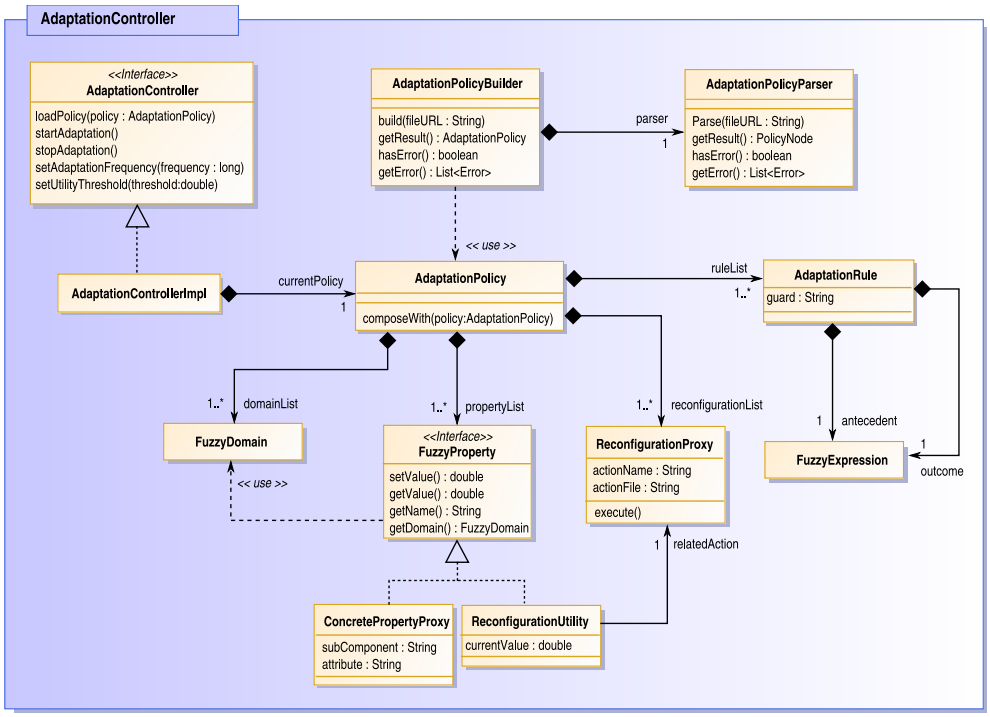

FIG. 5.2 – Conception détaillée du contrôleur d'adaptation pour Fractal

– Dans TANGRAM, les composants composites définissent précisément les sous-composants qu'ils sont susceptibles de contenir. Ces emplacements, nommés *slot*, sont fortement typés, et contraignent de manière statique l'ajout de nouveaux sous-composants. A l'inverse, un composant composite Fractal est un espace où l'ajout de nouveaux sous-composants est libre de toute contrainte.

5.2.1 Un patron pour les composants

Pour implémenter un composant TANGRAM dans la plate-forme Fractal, il est nécessaire d'appliquer un patron spécifique pour réifier dans le modèle Fractal les ports complexes utilisés dans TANGRAM. Ce patron permet, entre autres, d'offrir plusieurs implémentations différentes à un même service, ce qui n'est pas, *a priori*, évident dans le monde Fractal.

La partie gauche de la figure 5.3 présente un composant qui fournit deux implémentations différentes pour le service *service1* selon qu'il est appelé depuis le port *user* ou *admin* ; la partie droite présente l'implémentation que nous proposons pour ce composant. L'idée de base pour implémenter ce type de composant est de réifier chaque port : les classes *AdminPort* et *UserPort* représentent donc respectivement les ports *admin* et *user*. Chacune de ces classes implémente directement les interfaces fournies par le port et se rapporte aux interfaces requises par le port. Ainsi, la classe *AdminPort* implémente l'interface *Interface1* et fait référence à l'interface *Interface2*.

Les deux implémentations du service *service1* sont réifiées par les deux opérations *adminService1* et *userService1*. Ces deux opérations sont respectivement appelées par les opérations *Service1* des classes *UserPort* et *AdminPort*. On peut noter également que la définition d'un port

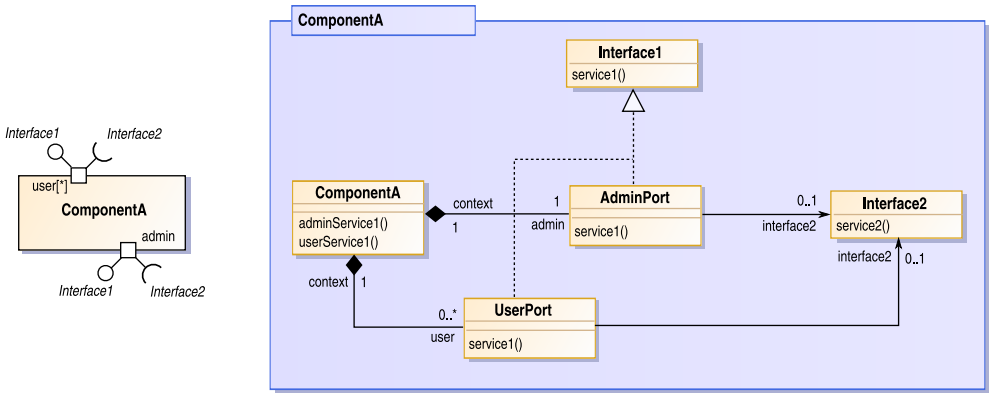

FIG. 5.3 – Un patron pour supporter les ports multiples et complexes dans Fractal

multiple se traduit par une multiplicté [0..*] entre la classe *ComponentA* et la classe *UserPort*.

Ce patron peut facilement être utilisé dans le monde Fractal en ajoutant les interfaces dédiées (*ContentController*, *BindingController*, etc) de Fractal sur la classe *ComponentA*. Il peut également être combiné à d'autres patrons tels que les patrons *Etat* et *Commande* pour implémenter les automates qui reflètent le comportement du composant. Ce patron permet également d'envisager une implémentation directe d'un composant en code natif, indépendamment d'une quelconque plate-forme d'exécution.

Le cas des emplacements utilisés dans les composants composites TANGRAM peut être géré en utilisant des règles de nommage associées aux composants Fractal. En effet, au sein d'un composite Fractal, tous les sous-composants sont accessibles par un nom. De manière pragmatique, ce nom peut être utilisé pour refléter la structure des emplacements, mais cela n'est pas suffisant pour prendre en compte les contraintes de typage statique induites par l'utilisation d'emplacements fortement typés. La transformation de ces contraintes de typage n'est pas supportée par la transformation actuelle.

5.2.2 De TANGRAM vers FScript/FPath

Le modèle TANGRAM inclut directement les constructions nécessaires pour exprimer les politiques d'adaptations : actions architecturales, propriétés et règles. Dans le monde Fractal, les constructions disponibles sont différentes : il est donc nécessaire de transformer les éléments de TANGRAM en éléments du monde Fractal.

Transformation des actions architecturales TANGRAM ne différencie pas les activités métiers des activités qui incluent des intro-actions architecturales. On peut donc aisément écrire des procédures architecturales complexes qui manipulent des données métiers ou des appels à des services externes ou internes du composant. La conversion de ce type de reconfiguration architecturale en action FScript, est délicate puisque FScript n'inclut qu'un nombre limité de primitives dédiées aux manipulations architecturales.

Transformation des expressions architecturales Les règles définies dans les politiques d'adaptations de TANGRAM sont gardées par des expressions booléennes. Ces expressions peuvent faire également appel à des données métiers qui ne sont pas accessibles dans le modèle Fractal, purement structurel. En effet, le langage FPath, utilisé par FScript, n'adresse que des éléments architecturaux et ne peut donc pas capturer une expression portant sur les données.

La transformation actuelle ne traite donc qu'un sous-ensemble des constructions disponibles dans TANGRAM. De plus, la transformation des expressions TANGRAM en expressions FPath n'est pas triviale puisqu'il faut prendre en compte la réification de ports multiples et des emplacements à l'aide du patron présenté dans la section précédente. L'implémentation n'est pour l'instant que partielle et ne prend pas en compte la gestion d'emplacements multiples notamment.

5.3 Synthèse

Dans ce chapitre nous avons présenté l'architecture d'une extension de la plate-forme Fractal permettant d'interpréter les politiques d'adaptation que nous avons définies dans les chapitres précédents. Ces politiques ne sont pas strictement représentées dans le même formalisme que celui utilisé lors de la validation *a priori*, car certains aspects sont spécifiques à la plate-forme Fractal. Une transformation *ad hoc* entre le monde TANGRAM et le monde Fractal permet d'assurer la réutilisation du modèle à l'exécution. Cette extension de la plate-forme Fractal a été présentée lors de la conférence LMO 2008 [19].

Chapitre 6

Etude de cas

Pour valider les idées présentées dans les chapitres précédents, nous les avons appliquées à la conception d'un serveur HTTP. L'objectif est de concevoir un serveur HTTP capable de modifier son architecture en fonction du nombre de requêtes qu'il reçoit et en fonction de leur dispersion (par rapport au nombre de pages dont il dispose). Ce chapitre décrit la modélisation du serveur HTTP et de deux politiques d'adaptation. Il s'agit bien ici simplement de montrer la validité de l'approche quant à la conception de composants autonomes.

6.1 Architecture du serveur CHEROKEE

Le serveur HTTP conçu dans cette étude de cas est une variation de l'exemple *Comanche*[1] utilisé dans plusieurs communications [33] autour de la plate-forme Fractal. Les requêtes HTTP sont lues sur le réseau par le composant *RequestReceiver* qui les transmet au composant *RequestHandler*. Pour traiter une requête, ce dernier peut, soit consulter le cache (composant *CacheHandler*), soit la transmettre au composant *RequestDispatcher* qui interroge alors un groupe de serveurs de fichiers pour résoudre la requête. La figure 6.1 présente l'architecture proposée pour le serveur HTTP *Cherokee*.

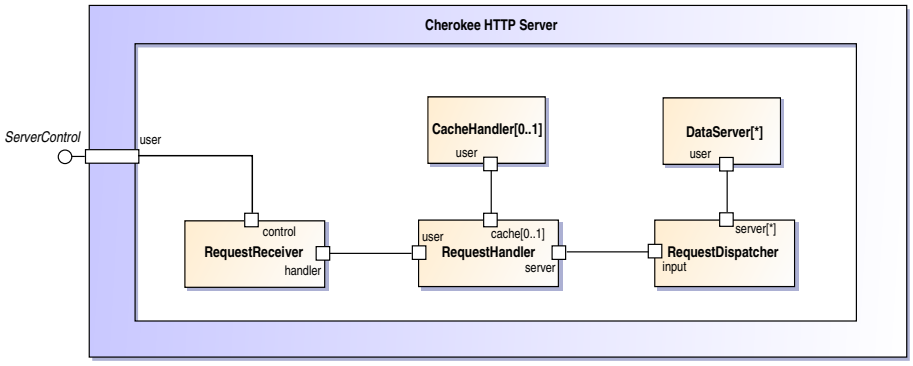

FIG. 6.1 – Architecture de base du serveur HTTP Cherokee

[1] Accessible dans le tutorial Fractal, à l'adresse : `http://fractal.objectweb.org/tutorial/index.html`

Cette architecture est dotée d'un cache (*CacheHandler*) et d'un contrôleur de charge (composant *RequestDispatcher*) pour pouvoir maîtriser le temps de réponse et le garder aussi court que possible. Pour pouvoir supporter d'éventuelles montées en charge, les exigences suivantes ont été définies :

1. Le composant *CacheHandler* ne doit être déployé que si le nombre de requêtes HTTP similaires (l'indice de dispersion) est élevé.
2. La quantité de mémoire allouée pour le fonctionnement du composant *CacheHandler* doit évoluer en fonction de la charge globale du serveur.
3. La durée de validité des informations du cache doit évoluer également en fonction de la charge globale du serveur (nombre de requêtes par unité de temps).
4. Le nombre de serveurs de données déployés doit être corrélé avec la charge globale du serveur

Plusieurs caractéristiques de ces exigences doivent être soulignées. D'une part, ces exigences peuvent être classifiées en deux catégories : les règles locales et les règles architecturales. Les règles locales concernent la configuration d'un composant particulier. C'est le cas de l'exigence 2 qui ne concerne que la configuration du cache. Les règles architecturales, par opposition, concernent la configuration de la collaboration entre les composants. Les exigences 1 et 4 en sont de bons exemples puisque qu'elles nécessitent de modifier l'agencement des composants et des connecteurs (appelés *bindings* dans la terminologie Fractal).

D'autre part, ces exigences sont décrites de manière qualitative, c'est-à-dire à l'aide d'un vocabulaire spécifique à chaque type de propriétés. Dans la première exigence par exemple, il est question d'un indice de dispersion « élevé », ce qui est purement qualitatif car aucune valeur précise n'est fournie pour quantifier le terme « élevé ».

6.2 Intégration de propriétés extra-fonctionnelles

En plus des différentes propriétés extra-fonctionnelles évoquées dans la section précédente (charge moyenne et dispersion des requêtes, taille du cache, etc), le principal objectif de qualité qui est visé est la minimisation du temps de réponse. Il est donc essentiel de disposer d'une sonde dédiée au temps de réponse, qui va nous permettre de mesurer l'influence logique des politiques d'adaptations que nous allons mettre en place.

Pour cela, une sonde dédiée est ajoutée à l'architecture. Elle mesure le temps qui s'écoule entre la réception d'une requête HTTP et l'envoi de la réponse correspondante. Cette sonde, est construite sur le même modèle que la sonde utilisée pour mesurer le nombre d'images produites par seconde dans l'exemple du système vidéo *frame rate*. Elle est insérée dans l'architecture et est connectée au *RequestReceiver* qui signale la réception et le traitement des requêtes. La figure 6.2 présente les modifications ainsi apportées au serveur HTTP.

Des sondes correspondant aux autres propriétés extra-fonctionnelles évoquées dans les règles d'adaptations, ont également été intégrées dans l'architecture comme le montre la figure 6.2. Les propriétés du cache sont maintenant accessibles sur le port de configuration du *CacheHandler*. La taille du cache est implémentée en fixant un nombre de pages maximales et la durée de validité des ces informations est calculée en fonction d'une horloge. Le composant Cherokee peut donc accéder à ces valeurs.

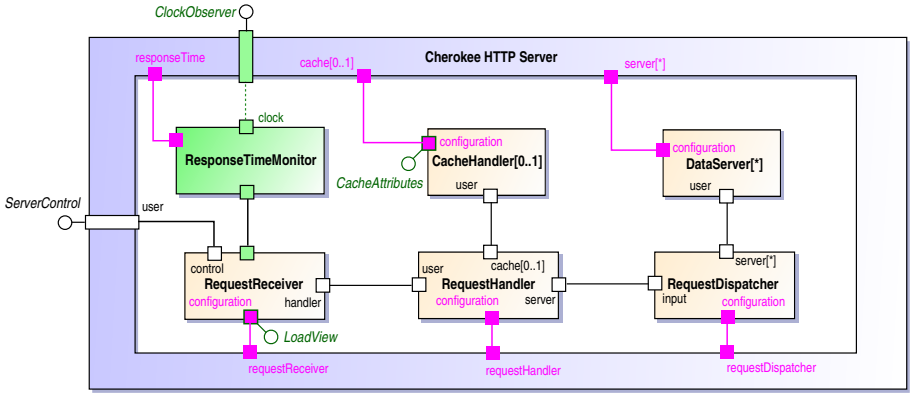

FIG. 6.2 – Intégration de sondes extra-fonctionnelles dans le serveur HTTP Cherokee

Les propriétés relatives à la charge du serveur sont, elles, fournies par le composant *RequestReceiver* qui calcule la charge moyenne du serveur ainsi qu'un indice de dispersion des requêtes calculé de la manière suivante :

$$d = \frac{\sum_{i=1}^{p} r_i / r_n}{p}$$

où la dispersion des requêtes d est calculée en fonction du nombre de pages p disponibles sur le serveur, du nombre de requêtes r_i visant la i-eme page, et du nombre total de requêtes, r_n.

6.3 Vers un serveur Cherokee dynamique

Les exigences présentées au début de ce chapitre impliquent de déployer le cache en fonction de la charge du serveur, mais également de déployer plus ou moins de serveurs de données. L'architecture est manifestement dynamique et la première étape pour obtenir un système autonome est de concevoir les règles qui régissent ce dynamisme.

On identifie donc quatre reconfigurations architecturales, c'est-à-dire des actions qui vont impacter l'état de l'architecture en terme de composants et de connecteurs. Parmi ces actions, deux concernent la gestion du cache et deux concernent le déploiement de nouveaux serveurs de données.

Les deux reconfigurations architecturales nécessaires sont l'ajout et la suppression du composant cache. Ces deux procédures sont exprimées de manière impérative dans l'architecture, sous la responsabilité du composite, c'est-à-dire du serveur HTTP.

```
1 reconfiguration addCache
  do
3         create CacheHandler named newCache
          self.cache := newCache
5         connect self.cache.user to self.requestHandler.cache
          request cache to do start()
7 end
```

```
reconfiguration removeCache
do
        request cache to do stop()
        disconnect self.cache.user from self.requestHandler.cache
        self.cache.remove(self.cache.first)
end
```

On procède de la même façon pour le déploiement de nouveaux serveurs de données. Deux actions architecturales sont définies : l'une pour l'ajout d'un serveur de données et l'autre pour sa suppression.

```
reconfiguration addDataServer
do
        var port : Port
        request requestDispatcher to do createNewPort(port)
        create DataServer named newServer
        self.server.add(newServer)
        connect port to newServer.user
end

reconfiguration removeDataServer
do
        var serverToRemove : DataServer init self.server.one
        disconnect serverToRemove.user
        self.server.remove(serverToRemove)
end
```

Ces quatre reconfigurations architecturales vont permettre de modifier dynamiquement l'architecture du serveur web.

6.4 Modélisation des politiques d'adaptation

6.4.1 Gestion du cache

Comme nous l'avons vu précédemment, la gestion du cache nécessite d'observer l'évolution de deux propriétés, à savoir, la charge moyenne du serveur et la dispersion des requêtes. Pour cela, on définit la terminologie associée à ces deux propriétés appelées *load* et *requestDensity*.

On définit également les propriétés que l'on veut influencer, c'est-à-dire la taille du cache et la durée de validité des informations dans le cache. On associe à chacune une terminologie qui permet de connaître de manière qualitative son état. La durée de validité, dénommée par *validityDuration* évolue alors entre 30s et 2min, depuis des durées dites « courtes (*short*) » vers des durées dites « longues (*long*) ». L'exemple suivant présente la syntaxe utilisée pour décrire le comportement de ces éléments.

```
  property load : Real
    evolves in [0, 100] as 'low' 'medium' 'high'
    sensor is cacheHandler.getAverageLoad

  property requestDensity : Real
    evolves in [0, 100] as 'low' 'medium' 'high'
    sensor is getAverageRequestDeviation
```

```
  property size : MemorySize
    evolves in [0, 25] as 'small' 'medium' 'large'
    sensor is getSizeMemory on cache
    actuator is setSizeMemory on cache

  property validityDuration : Duration
    evolves in [30000, 120000] as 'short' 'normal' 'long'
    sensor is getDataValidityDuration on cache
    actuator is setDataValidityDuration on cache
```

On définit ensuite les règles qui vont gouverner le déclenchement des actions architecturales précédemment définies. Dans le cas du cache, on ne veut déclencher son déploiement que lorsque la densité des requêtes augmente, c'est-à-dire si elle est *medium* ou *high*. De plus, le cache doit être retiré lorsque la densité des requêtes est faible. De manière assez naturelle on définit donc des règles du type : « quand la densité des requêtes est haute, alors l'utilité de déployer le cache est grande ».

```
    # Adding the cache
    when requestDensity is 'high' or 'medium'
        if not cacheHandler.isEmpty
        then utility of addCache is 'high'

    when requestDensity is 'low'
        if not cacheHandler.isEmpty
        then utility of addCache is 'low'

    # Removing the cache
    when requestDensity is 'high'
        if cacheHandler.isEmpty
        then utility of removeCache is 'low'

    when requestDensity is 'medium'
        if cacheHandler.isEmpty
        then utility of removeCache is 'medium'

    when requestDensity is 'low'
        if cacheHandler.isEmpty
        then utility of removeCache is 'high'
```

On définit également les règles qui régissent la configuration locale du cache, c'est-à-dire qui dictent l'évolution de sa taille et de la durée de validité des informations qu'il contient. Pour mémoire, on veut que la taille du cache augmente en fonction de la densité des requêtes et que la durée de validité augmente en fonction de la charge du serveur.

```
    # Update the size of the cache
    when requestDensity is 'low'
        if not cacheHandler.isEmpty
        then size is 'small'

    when requestDensity is 'medium'
        if not cacheHandler.isEmpty
        then size is 'medium'

    when requestDensity is 'high'
```

```
11          if not cacheHandler.isEmpty
            then size is 'large'
13
        # Update the data validity duration
15      when load is 'low'
            if not cacheHandler.isEmpty
17          then validityDuration is 'short'

19      when load is 'medium'
            if not cacheHandler.isEmpty
21          then validityDuration is 'normal'

23      when load is 'high'
            if not cacheHandler.isEmpty
25          then validityDuration is 'long'

27  end policy
```

6.4.2 Gestion des serveurs de fichier

La politique d'adaptation qui gère le déploiement de nouveaux serveurs de données est légèrement plus simple puisque qu'elle n'implique aucune reconfiguration locale. Il s'agit ici de déployer des nouveaux serveurs de données lorsque la charge augmente. La seconde politique d'adaptation est donc la suivante :

```
1   policy FileServerManagement
    is
3      property load : Real
          evolves in [0, 400] as 'low' 'medium' 'high'
5         sensor is requestReceiver.getAverageLoad

7      # Add Data Server
       when load is 'high'
9         if self.dataServer.size() <= 10
          then utility of addFileServer is 'high'
11
       when load is 'medium'
13        if self.dataServer.size() <= 10
          then utility of addFileServer is 'low'
15
       when load is 'low'
17        if self.dataServer.size() <= 10
          then utility of addFileServer is 'low'
19
       # Remove Data Server
21     when load is 'low'
         self.dataServer.size() > 1
23       then utility of removeFileServer is 'high'

25     when load is 'medium'
         if self.dataServer.size() > 1
27       then utility of removeFileServer is 'medium'

29     when load is 'high'
```

```
      if self.dataServer.size() > 1
31     then utility of removeFileServer is 'low'

33 end policy
```

6.5 Validation *a priori*

Les résultats obtenus lors de la simulation sont présentés par la figure 6.3. Les deux graphiques présentent respectivement l'évolution des propriétés impactées par les deux politiques d'adaptation, à savoir, celles relatives à la gestion du cache et au nombre de serveurs de données déployés. Pour plus de lisibilité et de concision, les valeurs mesurées ont été ramenées à un pourcentage (dans l'intervalle où elles sont définies) en ordonnée. Les étapes de simulation (temps discret) sont présentées en abscisse. Cinq propriétés ont été mesurées :

- La charge du serveur, c'est-à-dire, le nombre de requêtes reçues par le serveur (exprimé en nombre de requêtes par seconde).
- La dispersion des requêtes HTTP. Il s'agit d'un indice mesurant (indépendamment de la charge) la dispersion des requêtes.
- La taille du cache, c'est-à-dire, le nombre maximal de pages que le cache peut contenir.
- La durée de validité des pages contenues dans le cache. Il s'agit d'une durée en millisecondes pendant laquelle une page peut rester dans le cache.
- Le nombre de serveurs de données déployés.

La simulation présentée ici décrit le comportement adaptatif de notre serveur HTTP dans un contexte où la charge du serveur (en req/s) augmente jusqu'à une valeur élevée, puis décroît jusqu'à une valeur dite faible. Dans le même temps, la dispersion des requêtes évolue plusieurs fois entre des valeurs dites « faible » et « élevée ». La simulation montre ainsi les différentes combinaisons possibles entre la charge du serveur et la dispersion des requêtes.

Pour ce qui concerne la gestion des propriétés relatives à la gestion du cache, on peut noter que le composant cache n'est activé que lorsque la charge et la dispersion sont élevées. De plus, la taille du cache évolue correctement en fonction de ces deux paramètres. Par exemple, entre les étapes 100 et 300 de la simulation, la taille du cache augmente en fonction de la dispersion des requêtes. Il en est de même pour la durée de validité des informations qui évolue en fonction de la charge du serveur.

Le comportement de la seconde politique d'adaptation montre que l'ajout de serveurs de données est bien corrélé à l'évolution de la charge du serveur.

6.6 Implémentation dans la plate-forme Fractal

A partir des spécifications décrites dans la section précédente, le système Cherokee a été développé pour la plate-forme Fractal à l'aide du contrôleur présenté dans le chapitre 5. Le serveur Cherokee dispose donc d'une interface de contrôle dédiée qui permet de contrôler l'adaptation. L'exemple de code suivant montre l'injection du contrôleur TANGRAM dans la structure de l'application. Il est ajouté directement dans la description de l'ADL Fractal (voir la ligne 11). C'est le fichier de configuration de la plate-forme Julia, fourni avec le contrôleur TANGRAM qui décrit comment sont composés les différents contrôleurs.

```
<?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/basic.dtd">

<definition name="fr.irisa.triskell.cherokee.Cherokee">
        <interface name="server" role="server" signature="fr.irisa.triskell.
            cherokee.requestReceiver.RequestReceiver"/>

        <component name="receiver" definition="fr.irisa.triskell.cherokee.
            requestReceiver.RequestReceiver"/>
        <component name="handler" definition="fr.irisa.triskell.cherokee.
            requestHandler.RequestHandler"/>
        <component name="dispatcher" definition="fr.irisa.triskell.cherokee.
            requestDispatcher.RequestDispatcher"/>
        <component name="cache" definition="fr.irisa.triskell.cherokee.
            cacheHandler.CacheHandler"/>
        <component name="server1" definition="fr.irisa.triskell.cherokee.
            fileServer.FileServer"/>

        <binding client="this.server" server="receiver.server"/>
        <binding client="receiver.handler" server="handler.handler"/>
        <binding client="handler.cache" server="cache.cache"/>
        <binding client="handler.dispatcher" server="dispatcher.dispatcher"/>
        <binding client="dispatcher.server1" server="server1.server"/>

        <controller desc="tangramComposite"/>
</definition>
```

Pour montrer encore une fois l'intérêt de la simulation *a priori*, le comportement du serveur Cherokee a été mesuré dans un environnement qui reflète autant que possible le scénario utilisé lors de la simulation. Dans ce cas précis, construire le scénario réel qui correspond au scénario utilisé lors de la simulation n'est pas une chose simple car la dispersion des requêtes et la charge du serveur ne sont pas indépendantes. Les faire évoluer de façon quasi linéaire a nécessité quelques petits ajustements. Le comportement de l'implémentation dans Fractal/Julia du serveur Cherokee est présenté par la figure 6.4.

Une première constatation est la durée du test qui avoisine les 1000 secondes, c'est à dire les 15 minutes. Comme élément de comparaison, la durée de la simulation est de l'ordre d'une trentaine de secondes. L'autre constatation est que, pour effectuer ce test de charge du serveur HTTP, il faut disposer d'une machine dédiée, car le comportement du serveur dépend énormément des ressources que lui alloue le système. Ce test a donc été effectué sur un serveur dédié (pamplemousse.irisa.fr, en donnant à l'application serveur Cherokee une priorité maximale). De plus l'application cliente s'exécute sur ce même serveur ce qui évite les problèmes liés au trafic réseau. La simulation a donc cet autre avantage de pouvoir être effectuée à moindre coût et d'abstraire ainsi un certain nombre de problèmes techniques non triviaux liés à la complexité du contexte.

Le premier graphique montre combien il peut être difficile de reproduire en réalité des tests pourtant simples *a priori*. Ici, il est très délicat d'obtenir une variation linéaire de la dispersion des requêtes. Dans le cas présent, la variation linéaire de la charge (fréquence des appels) implique une variation non linéaire de leur dispersion et il a donc fallu ajuster la dispersion des appels (de manière non linéaire) pour aboutir à un résultat approximativement linéaire.

Dans ce contexte, le second graphique de la figure 6.4 présente le comportement du serveur HTTP, et notamment la taille du cache et la durée de validité des informations qui y sont déposées. La variation de ces deux propriétés n'est naturellement pas aussi finement contrôlée que lors de la simulation mais ces deux propriétés suivent la tendance générale décrite par les résultats de la simulation (cf. figure 6.3 page 98). On peut également noter que les adaptations architecturales ont été exécutées puisque le cache n'est déployé que quatre fois comme c'est le cas lors de la simulation.

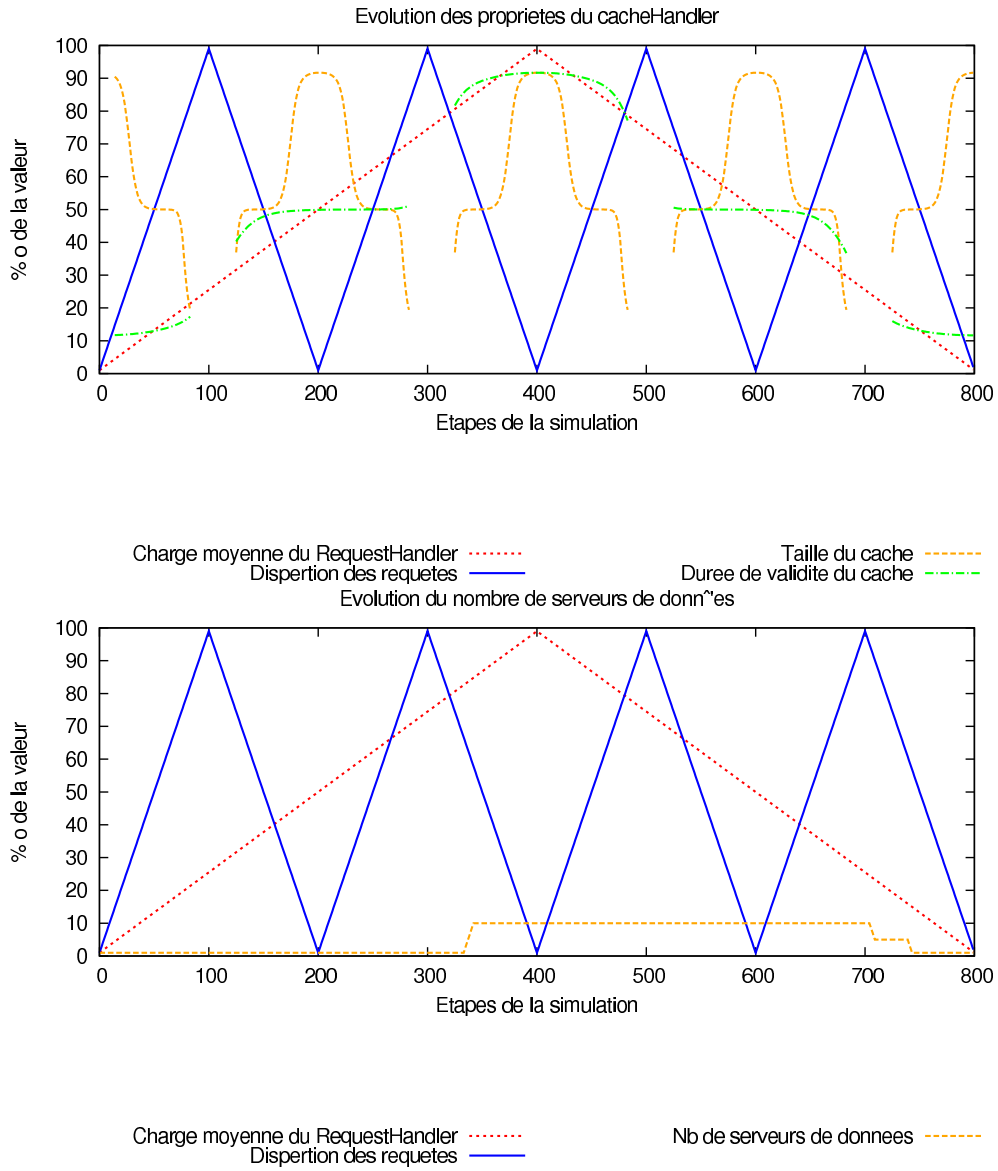

FIG. 6.3 – Évolution des configurations locales des composants impliqués dans la réalisation du serveur web

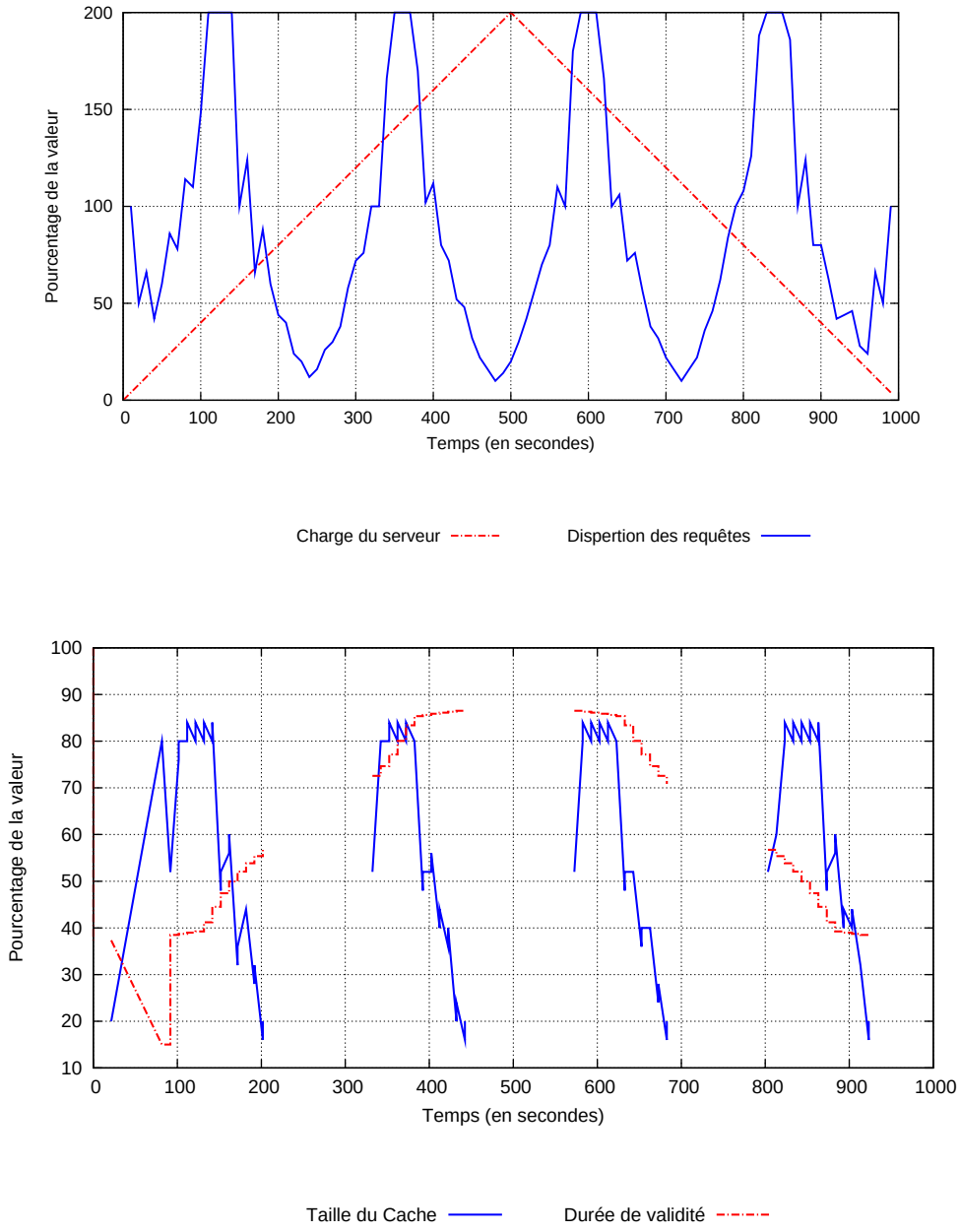

FIG. 6.4 – Comportement du serveur Cherokee implémenté à l'aide du contrôleur Tangram4Fractal

Chapitre 7

Conclusion

L'avènement des systèmes mobiles et ubiquitaires pousse petit à petit le génie logiciel dans ses derniers retranchements : la complexité des systèmes modernes ne cesse de croître, intégrant des problématiques de distribution, d'hétérogénéité, de configuration, de maintenance. Ainsi, la configuration des systèmes complexes est devenue une tâche ardue dédiée à des équipes de spécialistes. Pour faire face à cette crise rampante, les systèmes se dotent de la capacité de s'auto-configurer ou de s'auto-adapter à leur environnement, solutionnant alors les problèmes coûteux de configuration et de maintenance.

Toutefois, la conception de tels systèmes « intelligents » et autonomes n'est naturellement pas une chose aisée puisqu'elle intègre des savoir-faire issus de différents domaines : réseau et télécommunication, génie logiciel, intelligence artificielle, etc. Même si ces systèmes massivement ubiquitaires ne sont pas encore une réalité tangible, leur conception est, de fait, un problème crucial qui intéresse ainsi différentes branches du génie logiciel : systèmes auto-adaptatifs, systèmes sensibles au contexte, systèmes experts, intergiciels, etc. Ces systèmes ont la particularité d'être dotés d'une faculté d'adaptation qui leur permet de s'adapter, sans commandes extérieures, aux variations de leur environnement. Ces variations peuvent être de différentes natures et incluent par exemple les actions de l'utilisateur, la disponibilité des ressources matérielles, la géo-localisation, ou les paramètres extérieurs (température, hydrométrie, champ magnétique, luminosité, etc). Le principe d'adaptation utilisé pour cela, repris de la théorie du contrôle, se résume en trois étapes clés : observation, décision, action.

La littérature a naturellement déjà abordé chacune des ces trois étapes, que ce soit quant à la conception, la vérification ou l'implémentation des techniques associées. Les systèmes sensibles au contexte intègrent généralement des moyens d'observation sophistiqués permettant de récupérer et d'agréger des données issues de milieux différents. Les systèmes experts, issus de l'intelligence artificielle, ont apporté des solutions au problème de la décision et du raisonnement automatique. Finalement les plates-formes à composants et plus généralement les intergiciels disposent d'outils et de techniques permettant de reconfigurer voire d'adapter les architectures dynamiquement. Cependant, comme le montre l'étude présentée dans le chapitre 2, la littérature ne propose pas d'outils dédiés à la réalisation de systèmes auto-adaptatifs, qui prennent en compte à la fois leur conception, leur réalisation et leur implémentation dans une plate-forme donnée.

Les travaux présentés dans ce document ont donc abordé ces trois questions. Le chapitre 3 a montré tout d'abord comment utiliser les outils de la logique floue pour décrire de manière qualitative et non quantitative l'environnement et ainsi pouvoir décrire des règles d'adaptation de

haut-niveau. Ces règles ont la propriété d'être composables et permettent ainsi d'appliquer plus aisément aux systèmes auto-adaptatifs, le principe de séparation des préoccupations, première technique pour briser leur complexité. La décision au sein du système est alors guidée par des *politiques d'adaptation* regroupant des règles configuratives et intro-actives. La conception d'un système auto-adaptatif est ensuite capturée dans un modèle, dont la sémantique, opérationnelle, permet d'envisager une première vérification *a priori*, via la simulation. Si la vérification *a priori* ne remplace pas les techniques de vérification classiques telles que le test par exemple, elle permet de raisonner sur la conception et de mettre en évidence des erreurs ou des incompatibilités entre les choix architecturaux et les exigences liées à l'auto-adaptation. Elle permet ainsi d'éviter de coûteux retours arrière dans le processus de développement. Ces retours arrière, sont d'autant plus coûteux qu'ils remettent généralement en cause l'architecture même du système. Finalement, le chapitre 5 a présenté une extension de la plate-forme Fractal permettant d'embarquer dans une application à composants ces politiques d'adaptation qualitatives.

En terme de développement, les idées présentées dans ce document ont été intégrées dans l'outil TANGRAM qui propose un simulateur de modèles permettant d'animer et tester les modèles issus de la conception. Une extension de TANGRAM permet d'intégrer les politiques d'adaptations dans une application Fractal grâce à un contrôleur dédié. Ce contrôleur prend en charge l'observation du contexte et la procédure de décision, mais délègue la réalisation des adaptations architecturales à l'outil FScript intégré dans Fractal.

De nombreux points restent à creuser et l'approche proposée ici n'est qu'un premier pas vers le développement de systèmes auto-adaptatifs fiables. D'autres pistes peuvent également être explorées telles que l'utilisation de contrats, à la fois pour fiabiliser les systèmes auto-adaptatifs mais également comme éléments déclencheurs des adaptations architecturales. Toutefois, les systèmes adaptatifs n'en sont qu'à leur premier stade. Les techniques présentées dans ce document permettent d'anticiper automatiquement les variations architecturales (et configuratives) d'un système en fonction des évolutions de l'environnement. Il faut donc persévérer et essayer d'anticiper ces évolutions de l'environnement et concevoir des systèmes réellement autonomes.

Chapitre 8

Perspectives

L'approche présentée dans les chapitres précédents jette les bases du développement de systèmes adaptatifs en proposant une technique de modélisation, une technique de validation *a priori*, et une plate-forme d'exécution pour les systèmes auto-adaptatifs. Cependant de nombreuses questions restent en suspens et de nombreux points peuvent encore être améliorés.

8.1 Devenir des contrats dans les architectures auto-adaptatives

L'approche décrite dans les chapitres précédents prend en compte deux types d'adaptation : les adaptations architecturales et les adaptations configuratives. Ces deux types d'adaptations diffèrent notamment par leur interprétation, ponctuelle pour les adaptations architecturales, et continue pour les adaptations configuratives. L'algorithme d'adaptation proposé dans le chapitre 3 utilise un seuil d'utilité pour gérer du même coup ces deux types d'adaptations possibles : les adaptations configuratives sont continuellement exécutées alors que les adaptations architecturales ne le sont que si leur utilité dépasse ce seuil en question, ce qui les rend intrinsèquement ponctuelles.

D'autres approches sont également possibles et notamment l'utilisation de contrats. Comme nous l'avons évoqué dans le chapitre 2, ils permettent d'embarquer de la vérification pendant l'exécution du système. Plus généralement, dans les approches à composants, ils ont été généralisés comme un moyen de contraindre et de certifier les interactions entre les composants. Les contrats spécifient donc à l'exécution, les droits et les devoirs de chaque participant lors d'une interaction entre deux composants.

La figure 8.1 présente la contractualisation des architectures telles que nous les avons modélisées. Chaque connection entre deux composants est régie par un contrat (en rouge sur la figure). Du point de vue d'un composant composite, les contrats peuvent être externes ou internes selon qu'ils régissent une interaction à l'intérieur ou à l'extérieur du composite. Par exemple, sur la figure 8.1 les contrats 1, 2, et 3 sont internes alors que les contrats 4 et 5 sont externes.

L'idée généralement associée aux contrats et plus particulièrement aux contrats de qualité de service, est que leur violation déclenche un traitement spécifique, de type gestion d'erreur. Dans une architecture auto-adaptable, la nature ponctuelle des « violations » de contrat est un moyen élégant de déclencher les adaptations architecturales, ponctuelles également. La figure 8.2 présente un processus de déclenchement des adaptations architecturales et configuratives basé sur les contrats.

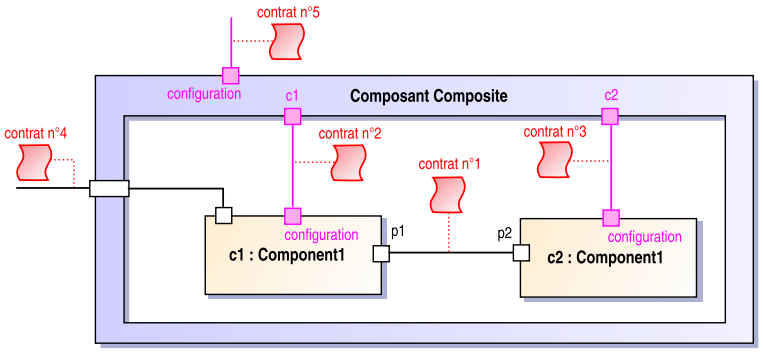

FIG. 8.1 – Intégration de contrats dans les architectures

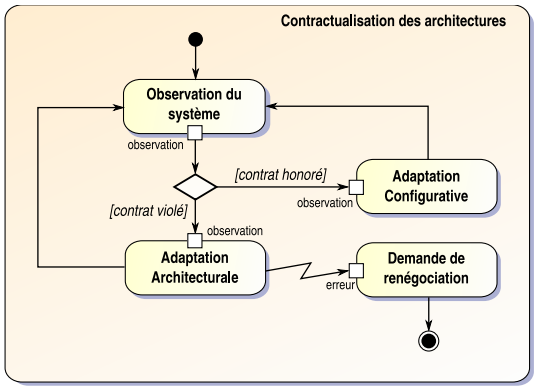

FIG. 8.2 – Déclenchement des adaptations architecturales et configuratives à l'aide de contrat

Dans ce processus, on considère que les adaptations configuratives, qui ne sont valables que dans un sous-ensemble d'architectures donné, permettent d'honorer au mieux un contrat externe. En d'autres termes, un contrat externe définit l'enveloppe dans laquelle le système doit évoluer et les adaptations configuratives maintiennent au mieux le système dans cette enveloppe. Lorsque le contrat est violé, il faut trouver une autre architecture interne qui permette de rester dans cette enveloppe. Si une telle configuration existe, alors elle est déployée, ce qui implique de renégocier tous les contrats internes. Si aucune adaptation architecturale ne permet de rester dans l'enveloppe spécifiée par le contrat externe, alors, le composant composite demande, via son port de configuration, une renégociation du ou des contrats externes.

8.2 Vers un langage de données structurées qualifiables

La modélisation des données contextuelles présentée dans le chapitre 3 n'utilise que des données simples de type « clé-valeur ». Or, comme le montre le chapitre 2, d'autres types de données peuvent être utilisés, notamment des données structurées. Parmi les approches les plus utilisées, les ontologies, issues de l'Intelligence Artificielle permettent de raisonner sur les données, mais ne sont toujours pas bien standardisées (à l'exception du langage OWL). Par opposition, les modèles

orientés objet, mieux standardisés par le biais de l'OMG, ne permettent pas de raisonner aisément sur les données. La combinaison de la logique floue et des modèles de données orientés objet doit permettre de combiner modélisation objet et raisonnement automatique.

Cependant, plusieurs obstacles limitent la généralisation d'une interprétation floue d'un modèle objet. Les modèles objets reposent généralement sur six types de données de base : les booléens, les entiers, les réels, les caractères, les chaînes de caractères et les énumérations. Les types numériques et plus particulièrement les entiers et les réels se prêtent particulièrement bien à la définition de variables linguistiques. Cependant, les booléens et les énumérations nécessitent de définir des variables discrètes. Les cas des caractères et des chaînes de caractères sont plus délicats, ces deux types de données ne se prêtent pas réellement à une qualification. Les caractères forment un espace discret, fermé et totalement ordonné qui peut être utilisé comme domaine d'entrée de fonction d'appartenance, mais cette approche n'apporte pas grand chose en terme de qualification. La relation d'ordre entre les caractères ne semble pas non plus pertinente pour qualifier les chaînes de caractères.

Les modèles objets sont pour la plupart basés sur la notion de classe, définie par un ensemble de propriétés référençant soit les types de bases soit d'autres classes. Il est important de pouvoir définir plusieurs variables linguistiques qui vont qualifier les objets instances d'une classe et ainsi définir une ou plusieurs terminologies par classes. Il faut pouvoir dire des instances d'une classe *Personne* qu'elles sont « jeunes » ou « vieilles » sans forcément adresser directement la propriété « âge » qui leur est associée. La gestion des collections doit également être adaptée. L'utilisation conjointe des opérations sur les collections (sélection, union, itération ...) et des prédicats flous n'est pas nécessairement triviale. Le sens de l'expression *component.select*$\{c \mid c.responseTime\ is\ fast\}$ n'est pas forcément très intuitif : l'interprétation floue de base construit l'ensemble de tous les composants ayant un temps de réponse court (*fast*), même si ce temps de réponse n'est « court » qu'à 0.01%.

La logique floue introduit également un certain nombre de quantificateurs supplémentaires permettant de manipuler les collections. Ils permettent de construire des prédicats flous plus complexes tels que « environ 5 composants répondent rapidement ». En plus du quantificateur *about* x (environ), on trouve également *most of* (la plupart), *few of* (quelque), etc. Si ces nouveaux quantificateurs permettent une description plus fine du contexte, il faut également pouvoir exprimer des contraintes strictes, typiques de la logique classique, telles que « tous les composants ». Cependant, la définition d'un langage d'interprétation du contexte basé sur la logique floue doit permettre de raisonner sur un modèle de données orienté-objet.

La définition d'un langage pertinent pour la description du contexte repose donc en grande partie sur un équilibre entre qualification (logique floue) et quantification (logique du premier ordre).

8.3 Inférence d'intro-action

L'algorithme 3.2 proposé dans le chapitre 3 souffre de deux principaux défauts qui l'empêchent d'être réellement utilisable dans des cas complexes où le nombre d'architectures possibles est important. D'une part, la description de l'architecture proposée au début de ce chapitre ne capture pas les variations possibles sur l'architecture. Rien n'indique par exemple que deux types de caméra sont disponibles, l'une plus sensible à la luminosité que l'autre. D'autre part, le fait de devoir expliciter les procédures d'adaptation architecturale limite la capacité d'adaptation. En effet, par

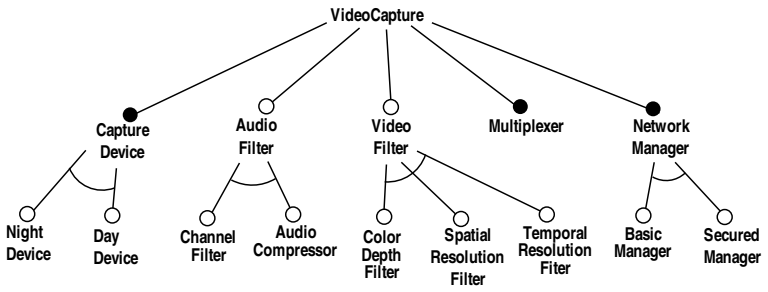

FIG. 8.3 – Diagramme de *Feature* représentant l'ensemble des variations choisies pour le système vidéo

mesure de simplicité, le concepteur sélectionne les architectures les plus utiles et décrit comment passer de l'une à l'autre. S'il a à sa disposition de nombreux composants (et donc de nombreuses variantes), il n'exploite évidemment pas toutes les architectures possibles.

Le diagramme de *feature* offre un moyen élégant de capturer les différents points de variations d'une architecture ainsi que les différentes variantes qui leur sont attachées. De plus, il permet également d'ajouter des contraintes pour restreindre le nombre de combinaison de variantes. Il est intéressant de noter que la représentation nécessaire, ici, un diagramme de *feature* n'existe pas dans UML. Appliqué au système vidéo, le diagramme de *feature* pourrait être celui de la figure 8.3

Comme nous l'avons expliqué, l'approche que nous avons suivie n'exploite généralement pas toutes les variantes possibles. Par exemple dans le cas du système vidéo, le moteur d'adaptation choisit l'adaptation architecturale la plus utile, mais si aucune ne fait allusion au gestionnaire de réseau sécurisé, celui-ci ne sera donc jamais déployé. L'adéquation de l'architecture vis-à-vis de l'environnement dépend donc directement de ces procédures d'adaptation architecturale. Cependant, si le concepteur a à sa disposition de nombreux filtres vidéos (résolution spatiale, résolution temporelle, profondeur des couleurs, etc), le nombre d'architectures possibles devient trop important et il est alors difficile de décrire toutes les intro-actions architecturales. Le système doit donc être capable d'inférer les intro-actions architecturales en fonction de l'architecture actuelle et de l'environnement.

L'objectif est de pouvoir calculer la combinaison optimale des variantes de l'architecture en fonction de l'environnement. A partir de cette architecture « idéale » on peut dériver, par différence la procédure d'adaptation architecturale associée et ainsi adopter l'architecture optimale. Toutefois, le calcul d'une architecture optimale nécessite de connaître les objectifs à atteindre ainsi que l'influence de chaque variante sur ces mêmes objectifs. Dans le cas du système vidéo par exemple, si l'un des objectifs est de minimiser le temps de réponse, il faut connaître l'impact de chaque variante (différents filtres vidéo et audio, gestionnaire de réseau, etc) pour pouvoir construire l'architecture. La sélection d'une architecture peut se faire à l'aide d'un algorithme de résolution de contraintes type CSOP[1]. Le problème est double, il s'agit d'une part de sélectionner les variantes architecturales possibles et d'autre part de choisir une « topologie » possible, c'est-à-dire les connexions possibles entre les composants sélectionnés.

La logique floue offre là aussi un moyen d'éviter de quantifier de manière plus ou moins

[1]Constraint Satisfaction Optimisation Problem

arbitraire les coûts et les objectifs nécessaires au système. De plus la logique floue nous offre un moyen simple de décrire la relation complexe entre l'environnement et les objectifs. Dans ce cas, l'utilisateur va décrire de manière floue une fonction d'utilité associée à chaque objectif. Par exemple, dans le cas du temps de réponse, les règles suivantes (et les terminologies associées) peuvent être utilisées. De manière similaire, on peut décrire les coûts associés à chaque variante. Ici chaque règle met potentiellement en relation, l'état de l'environnement avec une influence sur un objectif.

- **when** responseTime is *short* **then** utility is *high*
- **when** responseTime is *medium* **then** utility is *medium*
- **when** responseTime is *long* **then** utility is *low*

Cependant, pour pouvoir choisir entre plusieurs topologies identiques, c'est-à-dire des topologies qui ne diffèrent que par la manière dont sont inter-connectées les variantes, il faut également exprimer l'influence des topologies sur les objectifs. Il faut par exemple prendre en compte des distances entre plusieurs composants ce qui permet également d'éviter d'inférer des topologies correctes, mais non pertinentes. Une solution est de pouvoir expliciter la relation qui lie la place d'un composant au sein d'une topologie et son coût par rapport à un objectif. Il faut pouvoir exprimer que le multiplexeur est plus utile près du gestionnaire de réseau que près du périphérique d'acquisition.

L'approche décrite dans ce paragraphe est très proche de celle proposée dans l'outil MADAM (cf. chapitre 2) mais va plus loin car elle permet de choisir la topologie la plus pertinente, là où MADAM ne fait que sélectionner des variantes pour un « cablage » pré-établi.

8.4 Description de scénarios de tests

L'une des principales limites de l'approche proposée dans TANGRAM est la description des scénarios de tests qui doivent piloter la simulation. Dans l'outil, les scénarios sont décrits manuellement. Dans le cadre d'une architecture auto-adaptative, un scénario de test doit décrire à la fois un scénario fonctionnel et extra-fonctionnel. Dans le cas du système vidéo, l'aspect fonctionnel peut être par exemple, « le système capture une scène pendant dix minutes » alors que le scénario extra fonctionnel sera : « la luminosité s'écroule, la bande passante reste constante, mais la mémoire diminue progressivement ».

Cet exemple relativement basique met en évidence deux propriétés principales des scénarios de tests pour les systèmes adaptatifs. D'une part, ils mêlent des informations discrètes (appels de service) et des informations continues (évolutions de propriétés extra-fonctionnelles). D'autre part, ils font appel à des notions décorellées, tels que la mémoire, la luminosité, qui correspondent principalement aux informations contextuelles. Un système de description de scénarios pour les architectures auto-adaptables doit donc pouvoir :

- Décrire et composer des scénarios fonctionnels, c'est-à-dire décrire d'une part des séquences d'appels vers les différents ports du composant en cours de validation. Dans l'exemple pris précédemment, filmer une scène de 10 minutes peut se ramener à une séquence de données envoyée par la ou les caméras. Ces séquences d'appels doivent être composables en parallèle et en séquence, à la manière d'un diagramme d'activité pour décrire des cas de tests composites.
- Décrire et composer des scénarios extra-fonctionnels, c'est-à-dire décrire l'évolution d'une propriété de manière relative. Par exemple, dans le cas du système vidéo, il faudrait pouvoir

exprimer les différents états de la bande passante au cours du temps.
– Composer un scénario fonctionnel et un scénario extra-fonctionnel, c'est-à-dire construire à partir de scénarios simples, fonctionnels et extra-fonctionnels, un scénario de test complet qui décrive un cas de test fonctionnel dans un cas de test environnemental.

L'aspect composition est particulièrement important car, pris séparément, les cas de tests ne sont pas nécessairement très nombreux, mais c'est la multiplication des propriétés antagonistes qui rend complexe la vérification des systèmes adaptatifs.

De manière assez intuitive, puisque TANGRAM travaille sur une architecture à base de composants, la description de scénarios fonctionnels revient à définir des composants qui vont fermer l'architecture en interagissant avec les ports extérieurs du composant en cours de vérification. La composition de scénarios fonctionnels est alors obtenue par le support de la concurrence dans TANGRAM. En outre, la description de scénarios extra-fonctionnels peut se faire sous la forme d'une suite de valeurs linguistiques associées à une propriété donnée. Cela capture de manière assez naturelle la phrase « La bande passante est excellente, puis correcte, puis mauvaise ». Il faudrait toutefois pouvoir aller plus loin en décrivant la dynamique de l'évolution, c'est-à-dire, évolution plus ou moins rapide, pics, évolution non linéaire, amortissement, etc ....

La composition de scénarios fonctionnels et extra-fonctionnels est délicate et nécessite d'évaluer *a priori* le nombre d'étapes de simulation nécessaires pour dérouler un test fonctionnel donné. Avec cette information, on peut alors calculer les variations à appliquer sur les propriétés extra-fonctionnelles. Il est toutefois impossible de connaître à l'avance le temps nécessaire (en nombre d'étape de simulation) pour un scénario donné (les scénarios peuvent contenir des itérations, des conditionnelles, etc) et la combinaison automatique de scénarios fonctionnels et extra-fonctionnels reste un problème dur.

Il faut ensuite pouvoir qualifier la qualité attendue vis-à-vis d'un scénario donné. En effet, l'évaluation de l'exécution d'une politique d'adaptation n'est pas un résultat booléen mais une évolution dans le temps en fonction des changements survenus dans l'environnement. Un temps de réponse, par exemple, peut être acceptable, malgré la présence de pics ou de ralentissements ponctuels. Une solution possible est d'utiliser également la logique floue pour décrire de manière qualitative les performances attendues, en définissant la sémantique de termes tels que « ponctuel » ou « récurrent ». Il s'agit de qualifier à la fois la distance par rapport à une performance attendue, mais également de qualifier l'apparition d'une performance donnée au cours du temps.

8.5 Vers un processus de développement

Finalement, l'idéal serait de disposer d'un processus de développement semi-automatisé, qui permettrait de construire des systèmes auto-adaptatifs à partir d'une architecture traditionnelle. L'objectif est d'injecter dans une architecture logicielle les capacités d'observation, de raisonnement et d'intro-action mais également de prendre en compte la validation et la génération de code pour une plate-forme donnée.

D'après ce qui a été présenté précédemment, le cycle de développement pourrait être le suivant :

1. Conception d'une architecture primaire de manière traditionnelle, c'est-à-dire sans prendre en compte les variations de l'environnement. Il s'agit principalement de modéliser les différents composants du système « de base ».

2. Identifier les objectifs associés à l'adaptation du système ainsi que les informations contextuelles à prendre en compte lors de l'adaptation. Chacun de ces éléments devant être réifié par une sonde dédiée permettant une mesure effective dans l'architecture. L'injection de sondes peut prendre plusieurs formes : tissage d'aspects, réification, etc.
3. Identifier les points de variations possibles ainsi que les variantes possibles en terme d'architecture. Pour chaque variante, il faut définir son impact (positif ou négatif) sur les objectifs de qualité associés au système.
4. Décrire et injecter dans le système les mécanismes de décision qui permettent de rendre l'architecture auto-adaptable.
5. Valider *a priori* le fonctionnement de l'architecture en le confrontant à des scénarios pré-établis.
6. Générer tout ou partie de l'application finale, en fonction de la plate-forme choisie. Les outils par l'ingénierie des modèles doivent permettre de concrétiser cette étape.

Ceci ne représente qu'une partie du cycle de développement qui intègre nécessairement des phases de vérification *a posteriori*, des phases de maintenance et d'évolution. Par rapport aux différentes générations de systèmes adaptatifs présentées au début du chapitre 2, il s'agit ici de concevoir des systèmes de seconde génération, c'est-à-dire des systèmes capables d'anticiper les architectures nécessaires en fonction des variations de l'environnement.

Il est ainsi évident que de nombreux problèmes restent d'actualité quant à la conception de systèmes auto-adaptatifs. L'approche proposée et concrétisée dans l'outil TANGRAM n'est qu'une première étape vers un cycle de développement dédié aux systèmes auto-adaptatifs fiables.

Bibliographie

[1] V. Akman and M. Surav. The Use of Situation Theory in Context Modeling. *Computational Intelligence*, 13(3) :427–438, 1997.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Fundamental Approaches to Software Engineering : First International Conference, FASE'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28-April 4, 1998 : Proceedings*. Springer, 1998.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

[4] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4) :263–277, 2007.

[5] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, nov 2005.

[6] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *Software Architecture : 2nd European Workshop, EWSA 2005, Pisa, Italy, June 13-14, 2005 : Proceedings*. Springer, 2005.

[7] D.E. Bell, H. Raiffa, and A. Tversky. *Decision Making : Descriptive, Normative, and Prescriptive Interactions*. Cambridge University Press, 1988.

[8] A. Beugnard, J.M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[9] B.W. Boehm. Verifying and validating software requirements and design specifications. *Software, IEEE*, 1(1) :75–88, Jan. 1984.

[10] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36(4) :19–26, 2007.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. An Open Component Model and Its Support in Java. In *Component-based Software Engineering : 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004 : Proceedings*. Springer, 2004.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. 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.

[13] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, 2003.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture : a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[15] E. Cecchet, H. Elmeleegy, O. Layaïda, and V. Quéma. Implementing Probes for J2EE Cluster Monitoring. In *OOPSLA Workshop on Component and Middleware Proformance, Vancouver, October*, 2004.

[16] C. Chaudet and F. Oquendo. π-SPACE : a formal architecture description language based onprocess algebra for evolving software systems. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 245–248, 2000.

[17] Franck Chauvel and Olivier Barais. Modelling adaptation policies for self-adaptive component architectures. In Gordon Blair, Nelly Bencomo, and Robert France, editors, *1st Workshop on Model-driven Software Adaptation M-ADAPT'07 at ECOOP 2007*, pages 61–68, Berlin, Germany, July 2007. Short paper.

[18] Franck Chauvel, Olivier Barais, Isabelle Borne, and Jean-Marc Jézéquel. Composition of qualitative adaptation policies. In *Automated Software Engineering Conference (ASE 2008)*, 2008. Short paper, to be published.

[19] Franck Chauvel, Olivier Barais, Noel Plouzeau, Isabelle Borne, and Jean-Marc Jézéquel. Expression qualitative de politiques d'adaptation pour fractal. In *Langage Modèles et Objets LMO'08*, Montréal, Quebec, March 2008.

[20] Franck Chauvel, Isabelle Borne, Jean-Marc Jézéquel, and Olivier Barais. A model-driven process for self-adaptive software. In *4th European Congress ERTS Embedded Real-Time Software*, Toulouse, France, jan 2008.

[21] Franck Chauvel and Jean-Marc Jézéquel. Code generation from uml models with semantic variation points. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[22] D. Chefrour and F. Andre. Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL. *L' Objet(Paris)*, 9(1-2) :77–90, 2003.

[23] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03) :197–207, 2004.

[24] M. Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany, November 12-16, 2001 : Proceedings*. Springer, 2001.

[25] P. Collet, A. Ozanne, and N. Rivierre. On contracting different behavioral properties in component-based systems. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1798–1799. ACM Press New York, NY, USA, 2006.

[26] P. Collet and R. Rousseau. ConFract, un système pour contractualiser des composants logiciels hiérarchiques. *L' Objet(Paris)*, 11(1-2) :223–238, 2005.

[27] D. Conan, R. Rouvoy, and L. Seinturier. Scalable Processing of Context Information with COSMOS. In *Lectures Notes in Computer Science*, volume 4531, page 210. Springer, 2007.

[28] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2) :109–126, 2002.

[29] Ole-Johan Dahl. *SIMULA 67 common base language*. Norwegian Computing Center, 1968.

[30] EM Dashofy, A. van der Hoek, and RN Taylor. A highly-extensible, XML-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112, 2001.

[31] P.C. David and T. Ledoux. Towards a Framework for Self-adaptive Component-Based Applications. In *Distributed Applications and Interoperable Systems : 4th Ifip Wg6. 1 International Conference, Dais 2003, Paris, France, November 17-21, 2003, Proceedings*. Springer, 2003.

[32] P.C. David and T. Ledoux. WildCAT : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM Press New York, NY, USA, 2005.

[33] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[34] Linda DeMichiel and Michael Keith. Enterprise javabeans version 3.0 : Ejb core contracts and requirements. Technical Report JSR 220, Sun Microsystems, May 2006.

[35] A.K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1) :4–7, 2001.

[36] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing : A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[37] J. Dowling and V. Cahill. The K-Component Architecture Meta-model for Self-Adaptive Software. In *Metalevel Architectures and Separation of Crosscutting Concerns : Third International Conference, Reflection 2001, Kyoto, Japan, September 25-28, 2001 : Proceedings*. Springer, 2001.

[38] P.H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) Standard : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*, 2003.

[39] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2) :62–70, 2006.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[41] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow : architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004.

[42] David Garlan, Robert T. Monroe, and David Wile. *Foundations of component-based systems*, chapter Acme : architectural description of component-based systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[43] K. Geihs. Middleware challenges ahead. *Computer*, 34(6) :24–31, 2001.

[44] G. Grondin, N. Bouraqadi, and L. Vercouter. MaDcAr : an Abstract Model for Dynamic and Automatic (Re-) Assembling of Component-Based Applications. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006*, volume 4063, pages 360–367, 2006.

[45] T. Gu, X.H. Wang, H.K. Pung, and D.Q. Zhang. An Ontology-based Context Model in Intelligent Environments. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, volume 2004, 2004.

[46] T. Gu, X.H. Wang, H.K. Pung, and D.Q. Zhang. An Ontology-based Context Model in Intelligent Environments. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, volume 2004, 2004.

[47] M. Gupta, D. Katre, S. Shah, and R. Iyer. *Architecting and Building Enterprise Solutions with COM+ and. NET*. Prentice Hall of India, 2005.

[48] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Pervasive Computing : First International Conference, Pervasive 2002, Zurich, Switzerland, August 26-28, 2002 : Proceedings*. Springer, 2002.

[49] G. Huang, H. Mei, and F.Q. Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13(2) :257–281, 2006.

[50] J. Ivers, N. Sinha, and K. Wallnau. A Basis for Composition Language CL. Technical report, Carnegie Mellon University, 2002.

[51] P. Jackson. *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[52] J.M. Jézéquel and B. Meyer. Design by contract : the lessons of Ariane. *Computer*, 30(1) :129–130, 1997.

[53] M. Kaenampornpan and E. O'Neill. Modelling Context : An Activity Theory Approach. In *Ambient Intelligence : Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, November 8-11, 2004 : Proceedings*. Springer, 2004.

[54] J.O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering*, pages 15–22. ACM Press New York, NY, USA, 2005.

[55] JO Kephart and WE Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12, 2004.

[56] Shekhar H. Kirani, Imran A. Zualkernan, and Wei-Tek Tsai. Evaluation of expert system testing methods. *Commun. ACM*, 37(11) :71–81, 1994.

[57] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. Butler, L. Tran, et al. Composite Capability/Preference Profiles (CC/PP) : Structure and Vocabularies. W3c working draft, W3C, 2003.

[58] P. KOGUT, S. CRANEFIELD, L. HART, M. DUTRA, K. BACLAWSKI, M. KOKAR, and J. SMITH. UML for ontology development. *The Knowledge Engineering Review*, 17(01) :61–64, 2002.

[59] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L.C. Magalhaes, and R.H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Middleware 2000 : Ifip/Acm International Conference on Distributed Systems Platforms and Open Distributed Processing New York, NY, USA, April 4-7, 2000 Proceedings*. Springer, 2000.

[60] G. Kraetzschmar, H. Utz, S. Sablatnog, S. Enderle, and G. Palm. Miro-Middleware for Cooperative Robotics. In *Robocup 2001 : Robot Soccer World Cup V*. Springer, 2002.

[61] C. Landauer. Correctness principles for rule-based expert systems. *Expert Systems with Applications*, 1(3) :291–316, 1990.

[62] O. Lassila, R.R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, World Wide Web Consortium W3C, 1999.

[63] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. In *Proceedings of the 31th Euromicro conference*, volume 00, pages 88–95, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[64] M. Leclercq, V. Quéma, and J.B. Stefani. DREAM : a component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 250–255. ACM Press New York, NY, USA, 2004.

[65] B. Li and K. Nahrstedt. QualProbes : Middleware QoS Profiling Services for Configuring Adaptive Applications. In *Middleware 2000 : Ifip/Acm International Conference on Distributed Systems Platforms and Open Distributed Processing New York, NY, USA, April 4-7, 2000 Proceedings*. Springer, 2000.

[66] DC Luckham, JJ Kenney, LM Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *Software Engineering, IEEE Transactions on*, 21(4) :336–354, 1995.

[67] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14. ACM Press New York, NY, USA, 1996.

[68] J. Malenfant, M.T. Segarra, and F. Andre. Dynamic adaptability : the MoleNE experiment. In *Proceedings of Reflection*, pages 110–117. Springer, 2001.

[69] S. Mary. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.

[70] J. McCarthy. Notes on formalizing context. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 555–560, 1993.

[71] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1) :2–57, 2002.

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

[73] H. Mei, F. Chen, Q. Wang, and Y. Feng. ABC/ADL : An ADL Supporting Component Composition. In *Formal Methods and Software Engineering : 4th International Conference on Formal Engineering Methods, Icfem 2002, Shanghai, China, October 21-25, 2002 : Proceedings*. Springer, 2002.

[74] B. Meyer. *Eiffel : the language*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[75] R.T. Monroe. *Capturing Software Architecture Design Expertise with Armani*. School of Computer Science, Carnegie Mellon University, 2000.

[76] RS Moreira, S. Blair, and E. Carrapatoso. Supporting adaptable distributed systems with FORMAware. In *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pages 320–325, 2004.

[77] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML'2005*, LNCS, Jamaica, 2005. Springer.

[78] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D. Pecora. Checking an expert systems knowledge base for consistency and completeness. In *Proceedings of 9th IJCAI*, pages 374–378, 1985.

[79] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D. Pecora. Knowledge-Base Verification. *AI Magazine*, 8(2) :69–75, 1987.

[80] O. Nierstrasz, G. Arevalo, S. Ducasse, R. Wuyts, A.P. Black, P.O. Mailer, C. Zeidler, T. Genssler, and R. van den Bom. A Component Model for Field Devices. In *Component Deployment : IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002 : Proceedings*. Springer, 2002.

[81] R.M. O'Keefe and D.E. O'Leary. Expert system verification and validation : a survey and tutorial. *Artificial Intelligence Review*, 7(1) :3–42, 1993.

[82] D.E. O'Leary. Methods of Validating Expert Systems. *Interfaces*, 18(6) :72–79, 1988.

[83] OMG. Corba component model specification. formal/06-04-01, Object Management Group, April 2006.

[84] OMG. Meta object facility (mof) core specification. Available Specification ptc/04-10-15, OMG, October 2006.

[85] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. OMG Available Specification formal/2007-11-04, Object Management Group, Nov 2007.

[86] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. OMG Available Specification (ptc/03-08-02), Object Management Group, Nov 2007.

[87] F. Oquendo. π-ADL : an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3) :1–14, 2004.

[88] P. Oreizy. Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 54–57, 1998.

[89] D. Pearce. *The Induction of Fault Diagnosis Systems from Qualitative Models*. Turing Institute, 1988.

[90] F. Plásil, D. Bálek, and R. Janecek. Sofa/dcup : Architecture for component trading and dynamic updating. In *CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[91] J. Polakovic, A.E. Ozcan, and J.B. Stefani. Building Reconfigurable Component-Based OS with THINK. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 178–185. IEEE Computer Society Washington, DC, USA, 2006.

[92] D. Preuveneers, J. Van den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, and K. De Bosschere. Towards an Extensible Context Ontology for Ambient Intelligence. In *Ambient Intelligence : Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, November 8-11, 2004 : Proceedings*. Springer, 2004.

[93] P.G. Raverdy, HLV Gong, and R. Lea. DART : A Reflective Middleware for Adaptive Applications. In *Proceedings of the Workshop on Reflective Programming in C++ and Java at the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 37–40, 1998.

[94] J. Rushby. Quality Measures and Assurance for AI (Artificial Intelligence) Software. Technical report, NASA Langley Technical Report Server, 1988.

[95] CB Saab, X. Bonnaire, and B. Folliot. A flexible monitoring platform to build cluster management services. In *Cluster Computing, 2000. Proceedings. IEEE International Conference on*, pages 258–265, 2000.

[96] C.B. Saab, X. Bonnaire, and B. Folliot. PHOENIX : A Self Adaptable Monitoring Platform for Cluster Management. *Cluster Computing*, 5(1) :75–85, 2002.

[97] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit : aiding the development of context-enabled applications. In *CHI '99 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM.

[98] M. Samulowitz, F. Michahelles, and C. Linnhoff-Popien. CAPEUS : An Architecture for Context-Aware Selection and Execution of Services. In *Proceedings of the IFIP TC6/WG6. 1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 23–40. Kluwer, BV Deventer, The Netherlands, The Netherlands, 2001.

[99] MT Segarra and F Andre. A framework for dynamic adaptation in wireless environments. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 336–347, 2000.

[100] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063, pages 139–153. Springer, 2006.

[101] Q.Z. Sheng and B. Benatallah. ContextUML : A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services. In *Proceedings of the International Conference on Mobile Business (ICMB'05)*, pages 206–212, 2005.

[102] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, pages 34–41, 2004.

[103] T. Strang, C. Linnhoff-Popien, and K. Frank. CoOL : A Context Ontology Language to Enable Contextual Interoperability. In *Distributed Applications and Interoperable Systems : 4th Ifip Wg6. 1 International Conference, Dais 2003, Paris, France, November 17-21, 2003, Proceedings*. Springer, 2003.

[104] M. Sugeno and T. Yasukawa. A fuzzy-logic-based approach to qualitative modeling. *IEEE Transactions on Fuzzy Systems*, 1(1) :7–31, 1993.

[105] M. Suwa, A.C. Scott, and E.H. Shortliffe. Completeness and consistency in a rule-based system. *AI Magazine*, 3 :16–21, 1982.

[106] C. Szyperski. *Component Software : Beyond Object-oriented Programming*. Addison-Wesley Professional, 1998.

[107] Wei-Tek Tsai, Rama Vishnuvajjala, and Du Zhang. Verification and validation of knowledge-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1) :202–212, 1999.

[108] TH Tse and SS Yau. Testing context-sensitive middleware-based software applications. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 1, pages 458–466, 2004.

[109] A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236) :433–460, 1950.

[110] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro-middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4) :493–497, 2002.

[111] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[112] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. CORTEX : Towards Supporting Autonomous and Cooperating Sentient Entities. In *Proceedings of European Wireless*, pages 595–601, 2002.

[113] TOC View. Reconfigurable context-sensitive middleware for pervasive computing. *Pervasive Computing, IEEE*, 1(3) :33–40, 2002.

[114] Z. Wang, S. Elbaum, and D. Rosenblum. Automated Generation of Context-Aware Tests. In *Proceedings of the 29th International Conference on Software Engineering*, pages 406–415. IEEE Computer Society Washington, DC, USA, 2007.

[115] Chang Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 30 of *SIGSOFT Softw. Eng. Notes*, pages 336–345, New York, NY, USA, 2005. ACM.

[116] Chang Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, pages 292–301, New York, NY, USA, 2006. ACM.

[117] Chang Xu, S. C. Cheung, and W. K. Chan. Goal-directed context validation for adaptive ubiquitous systems. In *SEAMS '07 : Proceedings of the 2007 International Workshop on*

Software Engineering for Adaptive and Self-Managing Systems, page 17, Washington, DC, USA, 2007. IEEE Computer Society.

[118] S.S. Yau and F. Karim. An Adaptive Middleware for Context-Sensitive Communications for Real-Time Applications in Ubiquitous Computing Environments. *Real-Time Systems*, 26(1) :29–61, 2004.

[119] LA Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30(3) :407–428, 1975.

[120] Z.Q. Zhou, DH Huang, TH Tse, Z. Yang, H. Huang, and TY Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 20–22, 2004.

Table des figures