MOCAS : un modèle de composants basé états pour l'auto-adaptation

Cyril Ballagny

THÈSE

PRÉSENTÉE À

L'UNIVERSITE DE PAU ET DES PAYS DE L'ADOUR

ECOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS

PAR

Cyril BALLAGNY

POUR OBTENIR LE GRADE DE

DOCTEUR

Spécialité :
INFORMATIQUE

MOCAS : un modèle de composants basé états pour l'auto-adaptation

---

Soutenue publiquement le :
8 mars 2010

Après avis de :

M. CONSEL Charles	Professeur, ENSEIRB
M. HAGIMONT Daniel	Professeur, INPT-ENSEEIHT

Devant la commission d'examen formée de :

M. JEZEQUEL Jean-Marc	Professeur, Université de Rennes 1 (Président)
M. CRNKOVIC Ivica	Professeur, Université de Mälardalen
M. BARBIER Franck	Professeur, Université de Pau et des Pays de l'Adour
M. HAMEURLAIN Nabil	Maître de Conférences, Université de Pau et des Pays de l'Adour

-2010-

THÈSE

PRÉSENTÉE À

L'UNIVERSITE DE PAU ET DES PAYS DE L'ADOUR

ECOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS

PAR

Cyril BALLAGNY

POUR OBTENIR LE GRADE DE

DOCTEUR

Spécialité :
INFORMATIQUE

MOCAS : un modèle de composants basé états pour l'auto-adaptation

---

Soutenue publiquement le :
8 mars 2010

Après avis de :

M. CONSEL Charles	Professeur, ENSEIRB
M. HAGIMONT Daniel	Professeur, INPT-ENSEEIHT

Devant la commission d'examen formée de :

M. JEZEQUEL Jean-Marc	Professeur, Université de Rennes 1 (Président)
M. CRNKOVIC Ivica	Professeur, Université de Mälardalen
M. BARBIER Franck	Professeur, Université de Pau et des Pays de l'Adour
M. HAMEURLAIN Nabil	Maître de Conférences, Université de Pau et des Pays de l'Adour

-2010-

Remerciements

La thèse est une expérience unique qui nécessite de se remettre régulièrement en question. Cette expérience ne saurait être complète sans la stimulation ni le soutien apportés par l'ensemble des personnes rencontrées pendant cette période. Je tiens ici à les remercier pour cela et pour bien plus encore.

Tout d'abord, je remercie Charles Consel et Daniel Hagimont de m'avoir fait l'honneur de rapporter cette thèse. Je remercie d'autant Charles Consel qu'il m'a offert l'opportunité de lui exposer mes travaux avant de les rapporter. Ensuite, je remercie Jean-Marc Jézéquel et Ivica Crnkovic pour avoir accepté de faire partie de mon jury. Especially, I would like to thank Ivica Crnkovic since my talk was in French.

J'adresse tous mes remerciements à Franck Barbier pour la manière dont il a dirigé ma thèse et pour m'avoir appris à être exigeant avec moi-même. Je remercie Nabil Hameurlain pour sa présence et pour les discussions que nous avons eu. Merci à tous les deux pour avoir recherché la qualité et m'avoir « tiré vers le haut ».

Un grand merci à tous les membres du laboratoire et du département d'informatique de l'Uppa : merci particulièrement à Nicolas pour m'avoir aidé, depuis le master, à décrypter le monde universitaire, merci à Laurent pour sa disponibilité et pour m'avoir fait découvrir le vtt, le vrai (celui avec des vélos à plus de 2000€ j'entends !), merci à Régine et à Yvette pour leur réactivité et bien sûr merci à Annig, Eric, Sophie, Bruno...

Une thèse sans thésards ne serait pas une vraie thèse ! Alors merci à mes copains de « galère » : Damien, Eric, Natacha et mes copains de « chambrée » et de « terrain » : les Juliens et Pierre. Courage aux suivants, Youssef et Nour !

Pour finir, je remercie mes parents qui m'ont permis de suivre d'aussi longues études et qui, je l'espère, ne regrettent pas leur investissement ! Enfin, j'adresse un merci particulier, plein d'émotions et de sentiments à Lydia, ma femme, qui a toujours eu foi en moi et m'a supporté, sans que je sache toujours la remercier à sa juste valeur. Ghislain, je ne t'oublie pas : j'espère ne pas t'avoir dégouté des études longues ! Maintenant, je peux regarder vers l'avenir et vers toi, petit être qui grandit...

« Ce que tu veux me dire,
est-ce vrai ? Est-ce bien ? Est-ce utile ?
Sinon je ne veux pas l'entendre. »
Socrate.

Table des matières

Table des figures

Liste des tableaux

Introduction

1 Contexte général

L'informatique est fondamentalement destinée au traitement automatique de l'information. Les systèmes informatiques traitent des données dans un volume et dans un temps qui surpassent les capacités humaines. Dans ce but, les systèmes matériels, les systèmes logiciels, les infrastructures de télécommunication sont interconnectés pour construire et garantir un flot continu de données. La mise en place de ces systèmes de systèmes nécessite beaucoup de temps, de moyens et de personnes issues de domaines différents (informatique, automatique, télécommunication...) [1]. De ce fait, plus les systèmes sont amenés à traiter d'informations, plus leur conception et leur administration se complexifient.

IBM, à travers la voix de Paul Horn [8], n'hésite pas à parler d'une industrie des technologies de l'information en crise face à cette complexité croissante. A tel point que les avantages apportés par de tels systèmes seraient largement amoindris. La réduction de la complexité de leur mise en œuvre doit alors passer par une augmentation de leur propre complexité. IBM ouvre ainsi la voie vers un nouveau type de systèmes qualifiés d'*autonomiques*. De tels systèmes assurent des fonctions annexes – telles que leur déploiement dans un environnement quelconque, la découverte de cet environnement, la mise en relation avec les autres systèmes déjà présents, le maintien de leur fonctionnement – qui dépassent les fonctions « applicatives » pour lesquelles ils ont été conçus. Toutes ces tâches qui normalement incombent aux administrateurs sont assurées par des systèmes qui s'auto-gèrent.

Dans la vision d'IBM, la complexité précédemment gérée par l'administrateur se retrouve donc transférée au niveau du système. Du point de vue humain, elle est transférée au niveau du concepteur du système autonomique. Le concepteur doit donc disposer de moyens facilitant la réalisation d'un tel système. Les méthodes de conception descendantes (par ex. le raffinement et la décomposition modulaire) atteignent leur limite dans ce contexte car elles nécessitent une connaissance globale du système. La complexité des systèmes rend cette connaissance difficile à acquérir. De ce fait, les méthodes ascendantes, en préconisant la construction des systèmes par assemblage de sous-systèmes existants, présentent un atout majeur dans le contexte de l'informatique autonomique.

2 Cadre de la thèse

Les systèmes autonomiques disposent de nombreuses propriétés, qualifiées d'*auto-**[2] telles que l'auto-configuration (par exemple la « conscience » de l'environnement précédemment énoncée) et l'auto-réparation. Une propriété est d'un intérêt tout particulier car primordiale dans la réalisation de propriétés autonomiques de haut-niveau : *l'auto-adaptation*. Un système auto-adaptable, ou *adaptatif*, a la particularité de pouvoir modifier son comportement et sa structure en réponse à des événements internes et externes, dans le but de maximiser la disponibilité et les performances de ses fonctions. L'auto-adaptation pose entre autres le problème de la cohé-

1. 40% des investissements dans les technologies de l'information sont utilisés juste pour l'intégration des différentes technologies utilisées un sein d'un même système [7].

2. La littérature anglo-saxonne qualifie ces propriétés de « self-star » ou « self-* » [9].

rence du système : une modification en un des points de ce dernier peut avoir des répercussions sur tout le système et ainsi en compromettre la stabilité. De nombreux efforts sont encore à faire en génie logiciel afin de proposer des méthodes de conception et de développement des systèmes adaptatifs qui soient simples à mettre en place, garantissent la cohérence du système et satisfassent les besoins des systèmes autonomiques.

Ces dernières années, deux grandes tendances ont émergé au sein de la communauté du génie logiciel afin de diminuer le coût et le temps de réalisation et d'augmenter la qualité des systèmes informatiques :

1. l'utilisation des composants logiciels, avec l'« ingénierie des composants logiciels » (*Component-Based Software Engineering*, CBSE) ;
2. l'utilisation des modèles, avec l'« ingénierie des modèles » (*Model-Driven Engineering*, MDE).

La première est connue pour favoriser la réutilisation [10] grâce à une conception modulaire [3]. La seconde permet la vérification et la validation d'un système dès la phase de conception ; accélère la réalisation du système en exploitant les modèles de conception dans le processus de développement (génération de code, exécution de modèles...) ; enfin, facilite la communication entre les personnes intervenant tout au long de la vie du système (concepteurs, développeurs, testeurs, administrateurs...).

3 Objectifs de la thèse

Dans cette thèse, nous traitons des systèmes adaptatifs pour l'informatique autonomique. Nous proposons de nous appuyer sur les concepts de composant logiciel et de modèle afin de réduire la complexité de conception, de développement et d'administration de ces systèmes. Du point de vue méthodologique, nous nous concentrons particulièrement sur les trois critères suivants :

1. *réutilisation* : capitaliser le travail de conception de ces systèmes en réutilisant les modèles de conception pour leur exécution et leur administration ;
2. *utilisabilité* : proposer une approche pratique de conception et d'administration des systèmes adaptatifs qui soit rapide à prendre en main et à appliquer ;
3. *indépendance* : minimiser le couplage entre les composants pour permettre de développer individuellement chaque composant du système.

Du point de vue conceptuel, nous nous focalisons sur les critères suivants :

1. *transparence* des mécanismes d'adaptation pour le concepteur ;
2. *flexibilité* du processus de contrôle réalisé par un système autonomique ;
3. *cohérence* de l'adaptation au niveau d'un composant logiciel autonomique et du système dans lequel il est déployé.

3. Plus un composant est réutilisé, plus il est confronté à des situations susceptibles de le mettre en défaut. Son concepteur peut donc le corriger/améliorer afin qu'il satisfasse tous ces cas d'utilisation, ce qui contribue alors à augmenter sa qualité.

Nous proposons pour cela le modèle de composants MOCAS (*Model Of Components for Adaptive Systems*). MOCAS permet de concevoir un système adaptatif à base de composants logiciels. Un composant MOCAS est spécifié avec le langage de modélisation UML. Son comportement est décrit par une machine à états UML. Cette même machine est *exécutée* par le composant pour réaliser son comportement. Un composant MOCAS est installé dans un conteneur respectant ce même modèle pour devenir adaptable. Une boucle de contrôle à base de composants MOCAS permet de réaliser des composants adaptatifs et de les doter de propriétés autonomiques.

L'originalité majeure de cette approche réside dans son utilisation des modèles : alors que les approches actuelles exploitent des modèles d'architecture représentant les liaisons entre les composants du système, MOCAS les exploite différemment. Dans MOCAS, les modèles sont utilisés pour la représentation interne et comportementale des composants, tout au long de leur réalisation, et sont le support pour réaliser les propriétés auto-*.

La présente thèse regroupe les contributions théoriques et pratiques suivantes :

- le modèle de composants MOCAS, bâti sur celui d'UML, supportant la mise à jour et l'adaptation du comportement des composants ;
- la spécification d'une boucle de contrôle dont un composant MOCAS est doté pour s'auto-adapter afin de lui conférer des propriétés auto-* ;
- le module d'extension MOCAS4TopCased permettant de générer un composant MOCAS déployable à partir de sa spécification dans la plateforme Topcased [4] ;
- la librairie MOCASEngine [5] permettant d'exécuter les machines à états UML ;
- la plateforme MOCASA de déploiement et d'administration de composants MOCAS.

4 Organisation du document

Ce document s'articule autour de huit chapitres séparés en trois parties :

- **la partie I** présente les concepts généraux nécessaires à la caractérisation d'un système adaptatif à base de composants logiciels :
  - **le chapitre 1** introduit les composants logiciels. Il explique notamment la manière dont les composants peuvent être assemblés afin de réaliser des systèmes. Le langage de modélisation UML est ensuite présenté en tant que langage permettant la spécification de la structure et du comportement des composants. Le concept de composant logiciel est illustré avec le modèle de composants PauWare, qui est une des bases de nos travaux,
  - **le chapitre 2** décrit les systèmes adaptatifs. Il présente l'informatique autonomique qui justifie notre intérêt pour ces systèmes. Après avoir distingué les différents types d'adaptation, l'accent est mis sur l'adaptation dynamique et les problèmes qu'elle engendre,
  - **le chapitre 3** expose les approches pratiques à l'auto-adaptation des systèmes à base de composants. Il s'intéresse à la localisation des supports de l'adaptation dans les systèmes ainsi qu'aux techniques permettant de rendre un système adaptable. Finalement, des systèmes adaptatifs à base de composants sont étudiés afin de mettre en évidence les

4. http ://www.topcased.org/
5. http ://mocasengine.sourceforge.net/

faiblesses des approches actuelles ;

- **la partie II** présente le modèle de composants MOCAS, notre contribution à la conception des systèmes adaptatifs :
  - **le chapitre 4** détaille le modèle de composants MOCAS. Il présente ses liens avec le modèle PauWare et sa formalisation avec le langage UML. La manière dont les composants MOCAS sont composés est notamment expliquée,
  - **le chapitre 5** décrit le conteneur de composants permettant de rendre adaptable un composant MOCAS. Il montre la manière dont les problèmes de l'adaptation dynamique sont gérés,
  - **le chapitre 6** présente les composants MOCAS autonomiques. Il détaille les composants permettant de réaliser une boucle de contrôle. Il expose ensuite des politiques autonomiques permettant de rendre auto-configurable et auto-réparable un composant MOCAS. Enfin, il présente la manière dont les composants d'un système MOCAS sont coordonnés afin d'assurer la cohérence globale du système.
- **la partie III** présente les différentes réalisations ayant permis la validation du modèle de composants MOCAS :
  - **le chapitre 7** expose les outils et la méthode permettant de réaliser des composants MOCAS. Il détaille notamment le moteur d'exécution de machines à états MOCASEngine avec la sémantique UML adoptée. Il présente aussi la plateforme MOCASA permettant de déployer les composants,
  - **le chapitre 8** présente un exemple utilisant MOCAS. Il détaille la conception d'une boîte de vitesse robotisée et comprend une étude quantitative de MOCAS. Il étudie notamment l'impact d'une approche par conteneur et par modèles sur les temps d'exécution et sur la mémoire occupée.

Première partie

Etat de l'art

Chapitre 1

Les composants logiciels et UML

Sommaire

Dès sa naissance en 1968 en tant que discipline de recherche [11], le génie logiciel a eu pour but d'augmenter la qualité des systèmes informatiques tout en réduisant leur coût et leur temps de réalisation. Dans cette optique, différentes méthodes de conception se sont succédées (conception structurée, conception orientée objet...). Néanmoins, l'idée d'appliquer les principes de conception des circuits électroniques, reposant sur la réutilisation de composants élémentaires catalogués, est apparue très tôt [10]. Ce principe s'est formalisé seulement dans les années 90 avec l'avènement du génie logiciel basé composant (*Component-Based Software Engineering*, CBSE) [12].

Nous introduisons dans la section 1.1 les concepts fondamentaux de la conception à base de composants (CBC) : nous y définissons les composants, les services, les principes de composition et les modèles de composants. Nous nous intéressons dans la section 1.2 à la manière de spécifier la structure et le comportement d'un composant avec le langage de modélisation UML (*Unified Modeling Language*) [1]. Nous présentons dans la section 1.3 le modèle de composants PauWare qui repose sur les machines à états UML pour spécifier le comportement des composants.

1.1 Les composants logiciels

La CBC favorise la réutilisation en accentuant la modularité prescrite par la conception orientée objet [13]. La CBC doit permettre d'atteindre trois objectifs majeurs [14] :

- assembler un système à partir de composants réalisés indépendamment les uns des autres ;
- réaliser des composants de manière à ce qu'ils soient réutilisables ;
- maintenir un système en permettant le remplacement et la personnalisation des composants.

Pour atteindre ces objectifs, la conception d'un composant repose sur les principes :

- d'abstraction, à travers la séparation de sa spécification et de son implémentation ;
- de composition, à travers l'explicitation de ses points d'interaction ;
- et d'adaptabilité, à travers sa personnalisation/configuration [15].

Ces principes visent à maximiser la réutilisabilité d'un composant en minimisant les dépendances, en facilitant l'assemblage tout en satisfaisant les besoins d'un maximum d'utilisateurs.

1.1.1 Définition d'un composant logiciel

> *« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »* [12].

Cette définition d'un composant logiciel, proposée par Szyperski [12] à la fin des années 90, fait consensus. Un composant est ainsi décrit comme ayant des points d'interaction clairement définis dans des interfaces, pouvant être composé avec d'autres composants et être mis en production par des personnes ne l'ayant pas conçu. Le point le plus discutable dans la vision de Szyperski est qu'un composant ne doit pas avoir d'état observable [16] : les instances d'un même type de composant ne doivent pas être distinguables. Si cette vision simplifie l'administration des composants (notamment le maintien de la cohérence des composants), elle ne reflète pas les besoins : de ce fait des composants industriels comme les EJB (*Enterprise JavaBeans*) [17] existent en deux types : les composants sans-état (*stateless session beans*) pour lesquels les clients ne peuvent distinguer les instances auxquelles ils s'adressent (rien n'empêche ce type de composants d'être implémenté avec des objets ayant eux un état), et les composants avec-état (*stateful session beans*) pour lesquelles les instances sont spécifiques d'un client.

> *« Reusable software components are self-contained, clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status. »* [18]

Dans la même période, Sametinger [18] met la réutilisabilité au cœur de sa définition. Un composant peut être une simple spécification de fonctionnalités, il n'est pas obligé de fournir une implémentation. Sametinger introduit aussi la notion de *documentation associée à un composant*. Cette documentation permet d'évaluer la pertinence d'un composant pour un contexte particulier, de l'adapter et de l'intégrer dans un nouvel assemblage. Enfin, il précise qu'un composant doit être « traçable » pour en assurer la maintenance.

> *« A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. »* [19]

Plus récemment, Councill et Heineman [19] font intervenir dans leur définition le principe de *conformité à un modèle de composant*[6], que nous développons dans la section 1.1.4. De plus, ils précisent qu'un composant doit être utilisable sans modification, ce qui va à l'encontre de la vision de Sametinger pour qui le composant est personnalisable. Effectivement, dans un monde idéal, un composant n'a pas à être personnalisable. En effet, il devrait être suffisament bien conçu pour convenir à toutes les situations. De façon plus réaliste, le fait que le composant doive être personnalisable peut être vu comme une conséquence du fait que le concepteur n'est pas capable d'envisager toutes les utilisations possibles.

> *« A component is a software element (modular unit) satisfying the following three conditions : (1) it can be used by other software elements, its "clients"; (2) it possesses an official usage description, which is sufficient for a client author to use it ; (3) it is not tied to any fixed set of clients. »* [20]

Enfin, la définition de Meyer [20] confirme la vision d'un composant composable, documenté et indépendant. Oussalah [15] approfondit la notion de documentation en rajoutant qu'un composant doit disposer d'*« un mécanisme d'introspection permettant de connaître et modifier dynamiquement ses caractéristiques »*. Il rappelle aussi l'importance d'avoir des composants adaptables afin de satisfaire des contextes d'exécution particuliers.

De notre point de vue, un composant logiciel n'est réutilisable que s'il est compréhensible par une personne qui ne l'a pas conçu. Pour cela, nous retiendrons les quatre caractéristiques suivantes :

1. un composant est lié à un modèle, servant à borner l'espace des composants possibles ;
2. un composant est lié à une forme de documentation, servant à comprendre son fonctionnement en s'affranchissant des détails d'implémentation ;
3. un composant a des points d'interaction clairement définis, pour savoir comment l'intégrer dans un environnement ;
4. un composant est personnalisable, afin de convenir à des cas d'utilisation variés.

1.1.2 Service d'un composant

En économie, un service se caractérise comme étant un acte proposé par une entité à une autre, sans que cette dernière ne dispose des moyens de production de cet acte [21]. En informatique, la définition d'un service varie suivant le domaine :

– dans les systèmes distribués, l'intergiciel [22] offre de nombreux services pour rendre interopérable des systèmes hétérogènes (des systèmes s'exécutant sur des environnements

6. Szyperski parle dans [12] de *« component world »* alors que dans la deuxième édition du même livre [16], il utilise *« component model »*.

matériels et logiciels différents). Chaque service est standardisé et est donc accédé uniformément par les différents systèmes. Par exemple, l'intergiciel CORBA [23] comporte dix-sept services : un service de gestion de cycle de vie pour créer/supprimer/copier/déplacer des objets, un service d'annuaire pour nommer/retrouver des objets, un service transactionnel pour assurer l'atomicité d'un ensemble d'opérations, un service d'horloge pour synchroniser des systèmes, etc ;

- dans les architectures orientées services [24], un service est un mécanisme offrant à un consommateur des fonctionnalités proposées par un fournisseur. Le fournisseur ne connait pas obligatoirement les futurs consommateurs et le consommateur ne sait pas comment est réalisé le service. Le consommateur peut utiliser le service de manière non prévue par le fournisseur. L'accès au service se fait via une interface déterminée contenant les détails des fonctionnalités. Le consommateur connaît ainsi le comportement du service et peut déterminer si ce dernier est approprié à ses besoins.
- dans les systèmes à base de composants, un service est réalisé par un composant. Un composant est donc un fournisseur de services. Un composant peut requérir d'autres services pour réaliser ceux qu'il fournit. Dans ce cas, il est à la fois fournisseur et consommateur.

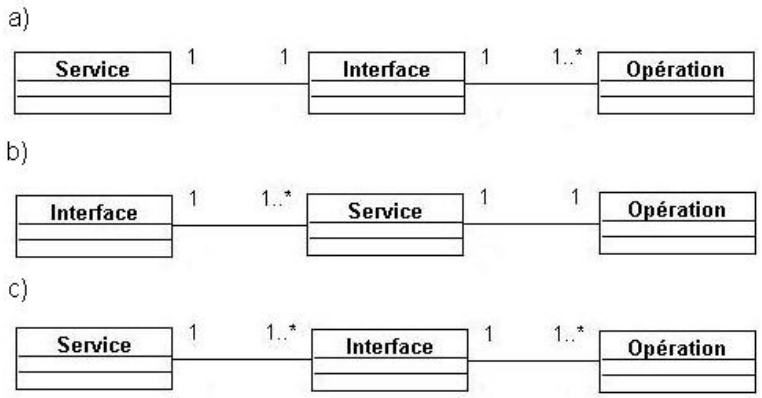

FIGURE 1.1 – Sémantiques de service

La manière dont un service est rattaché à un composant diffère suivant les auteurs (cf. figure 1.1) :

a) un service correspond à une interface exposée par un composant. Cette interface liste les opérations du service. Le service correspond alors à un ensemble cohérent d'opérations réalisées par un composant et exposé dans une interface [12, 25] ;

b) une interface expose des services, chaque service correspond à une opération [26]. La granularité du service est alors faible ;

c) un service a plusieurs interfaces, chaque interface liste les opérations du service. Il est ainsi possible d'accéder de plusieurs manières au service. Ceci est notamment utile lorsque le service évolue : chaque interface correspond alors à une version « historique » du service.

Nous adhérons à la version *a)* qui distingue un service d'une opération en lui permettant d'avoir une granularité plus forte que l'opération. Nous parlerons donc d'« interfaces de ser-

vice »[7].

La typologie service requis/ service fourni caractérise les flux de services entrant dans le composant et sortant du composant mais ne différencie pas l'utilité d'un service. A cet effet, les services d'un composant sont répartis en deux catégories :

- *les services fonctionnels*, propres à un composant et correspondant aux fonctionnalités pour lesquelles il a été créé ;
- *les services non fonctionnels*, communs à un ensemble de composants, correspondant à des problématiques générales de la CBC. Ces services permettent par exemple de gérer les liaisons, le cycle de vie, le comportement du composant. Ils sont souvent propres à un modèle de composants (cf. section 1.1.4).

L'autre point concerne le nombre de services qu'un composant fournit. Si un composant n'offre qu'un service, alors ce composant aura une granularité faible : de ce fait, le système devra comporter de nombreux composants pour assurer toutes ses fonctionnalités et il sera plus difficile à « manipuler ». En revanche, autoriser un composant à offrir plusieurs services permet d'agglomérer les services. Le composant propose alors un point d'accès centralisé à plusieurs services. Cette problématique rejoint celle de la composition que nous détaillons ci-après.

1.1.3 Composition de composants

Un composant, en tant qu'*« unité de composition »*, est destiné à faire partie d'un assemblage. Cet assemblage, à travers les liaisons entre les composants d'un système, caractérise l'architecture de ce système.

1.1.3.1 Typologie de composition

En se limitant à des aspects architecturaux, la composition de composants est possible suivant deux directions [27] :

- la composition horizontale met en relation deux composants de même granularité. Elle permet de résoudre les dépendances d'un composant en faisant correspondre ses interfaces de service requis avec les interfaces de service fourni d'autres composants. Les deux composants sont distincts dans le système ;
- la composition verticale (ou hiérarchique) permet à un composant, dit alors composite, d'encapsuler d'autres composants (simples ou eux-même déjà composites). Ces derniers sont des sous-composants du composite. Le composite est alors de plus forte granularité que ses sous-composants. Le composite peut ainsi proposer un point d'accès unique aux services offerts par ses sous-composants. De plus, il peut résoudre en interne ses dépendances et ainsi cacher au reste du système les services qu'il requiert.

Ainsi, sur la figure 1.2, le composant `Engine` est composé horizontalement avec le composant `Wheel` via l'interface `IWheel` ; le composant `Car` est un composite dont l'interface de service fourni `ICar` est réalisé par le composant `Engine`.

La composition, qu'elle soit horizontale ou verticale, consiste de façon traditionnelle à une composition *structurelle*, c.-à-d. à la mise en correspondance syntaxique d'interfaces de deux

7. « service » est au singulier car une interface expose un service, ce service est soit requis, soit fourni.

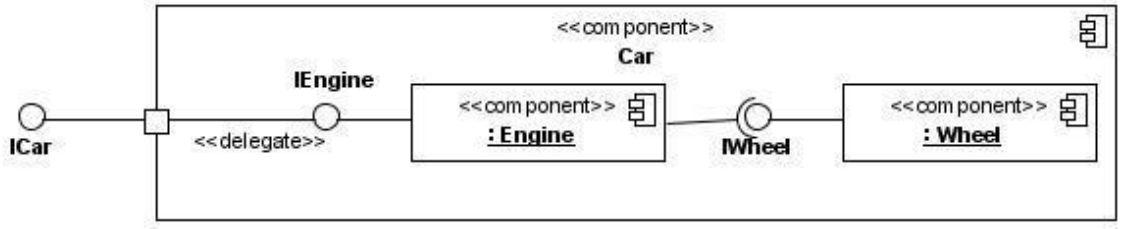

FIGURE 1.2 – Exemple de compositions horizontale et verticale

composants distincts. Ce type de composition n'est pas suffisant car il ne tient pas compte des contraintes comportementales des composants composés. En effet, la composition des comportements individuels de chaque composant doit former un nouveau comportement global cohérent. Les caractéristiques de ce nouveau comportement dépendent des caractéristiques de chacun des comportements combinés et de la manière dont ces comportements sont combinés. La composition structurelle est alors associée à une composition *comportementale*. Cette dernière fait intervenir des contraintes de séquentialité ou de concurrence sur les opérations relatives à un service [28–30]. Par exemple, considérons le système, dérivé de [30], comportant les trois composants suivants :

- le composant `Gestionnaire de stockage`, qui propose un service de stockage avec les opérations `charger` et `sauvegarder` ;
- le composant `Gestionnaire de cryptage`, qui propose un service de cryptage avec les opérations `crypter` et `decrypter` ;
- le composant `Gestionnaire de stockage crypté`, qui propose un service de stockage crypté et qui est composé horizontalement avec les deux composants précédents.

Nous ne voulons autoriser que le stockage de fichiers cryptés. De ce fait, le composant `Stockage crypté` ne doit invoquer l'opération `sauvegarder` qu'après avoir invoqué l'opération `crypter` et invoquer l'opération `décrypter` qu'après avoir invoqué l'opération `charger` (cf. fig. 1.3). La simple mise en correspondance des interfaces ne permet pas d'obtenir ce type de comportement qui introduit une contrainte sur la séquence des opérations. Le composant `Stockage crypté` doit donc implémenter la séquence correcte d'invocations.

1.1.3.2 Moment de la composition

Les composants sont des briques logicielles réutilisables et déployables indépendamment. Dans l'ingénierie des composants logiciels, les méthodes de conception ascendante sont privilégiées : un système est un assemblage de composants préexistants. Si un service nécessaire au système n'est fourni par aucun composant préexistant, un composant doit être conçu spécialement pour l'assurer. Dans ce contexte, la composition intervient à différents moments :

- à la conception d'un composant, ce dernier peut être raffiné en différents sous-composants faisant de lui un composant composite. La composition est alors verticale entre le composite et ses sous-composants. Elle préserve l'unité du composite en masquant la composition interne du composite, de ce fait rien n'empêche ce dernier d'avoir des sous-composants

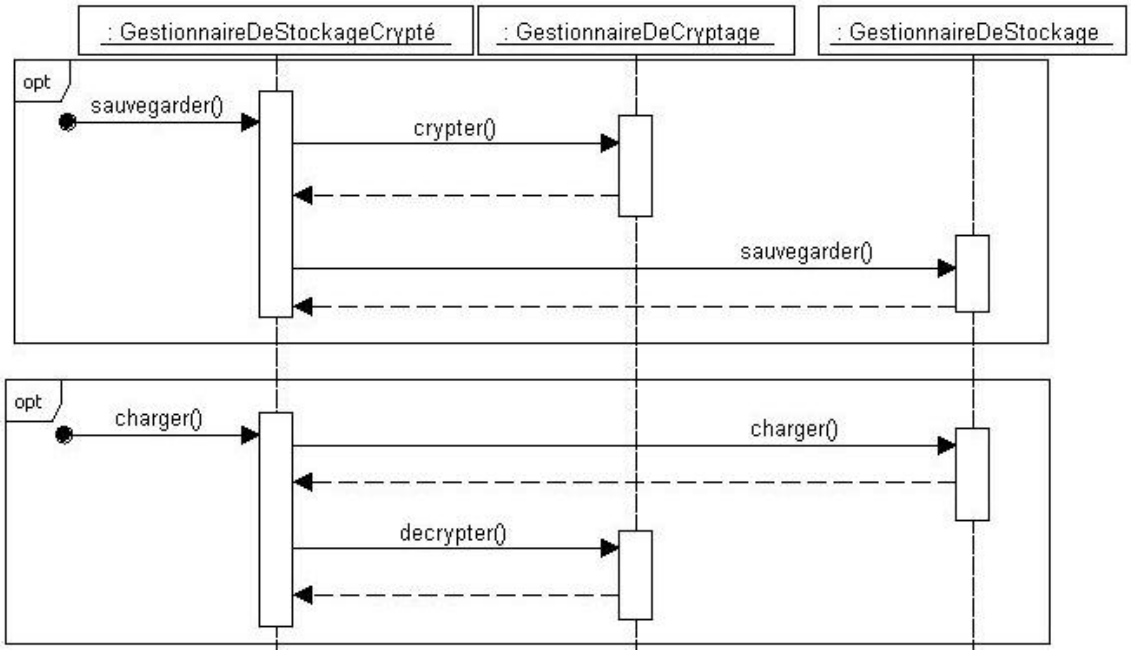

FIGURE 1.3 – Exemple de contrainte de comportement

composés horizontalement entre eux. En revanche, lier des composants horizontalement en dehors de la formation d'un composite empêche le déploiement indépendant des composants, ce qui est contraire à la définition retenue (cf. la définition de Councill et Heineman à la section 1.1.1) ;

- au déploiement, la composition consiste à instancier un assemblage particulier de composants (composition horizontale et verticale) : à partir d'un même jeu de composants, des assemblages différents peuvent être déployés ;
- à l'exécution, la composition consiste à modifier les liaisons existant entre composants : des instances différentes d'un même assemblage vont évoluer de façon indépendante ; dans le cas d'une modification de la composition horizontale, la littérature parle de reconfiguration dynamique (cf. 3.1.1). La modification d'une composition verticale nécessite d'avoir accès à la structure interne du composite. Ceci dépend alors du niveau de visibilité offert par le composant.

1.1.3.3 Visibilité du composant

La composition soulève le problème de la visibilité du composant, c'est-à-dire des informations dont a besoin un consommateur pour utiliser un composant. La littérature distingue trois niveaux de visibilité [12, 15, 31] :

- la visibilité *boîte noire* : le composant n'est compréhensible que par ce qu'il expose dans ses interfaces. L'interface précise la syntaxe pour invoquer les opérations du service. Elle peut aussi préciser la sémantique du service à l'aide de préconditions (c.-à-d. les conditions de

fonctionnement du service), d'invariants (c.-à-d. des conditions vraies à tout instant de la vie du composant) et de postconditions (c.-à-d. le résultat de l'invocation du service) ;
- la visibilité *boîte blanche* : tous les détails d'implémentation du composant sont accessibles, au risque de violer le principe d'encapsulation et de limiter les possibilités de substitution du composant. Avec ce type de visibilité, le consommateur est amené à dépendre de l'implémentation. Il peut altérer l'implémentation afin de personnaliser le composant. Même si l'implémentation n'est pas modifiée, le consommateur va s'appuyer sur des détails peut-être non pertinents pour le fournisseur mais qui peuvent l'être pour le consommateur ;
- la visibilité *boîte grise*, introduite par Buchi et Weck [31], offre un compromis des visibilités précédentes. Elle doit permettre une meilleure compréhension du fonctionnement d'un composant sans rentrer dans les détails d'implémentation. Buchi et Weck arguent qu'il est entre autres nécessaire de connaître la séquence des interactions d'un composant lors de l'invocation des opérations (cf. composition comportementale, section 1.1.3.1).

Si l'implémentation révèle tous les détails d'un composant, nous savons d'expérience que sa compréhension nécessite de nombreuses investigations. De plus, l'accès à l'implémentation ne permet pas de protéger un savoir-faire. De ce fait, d'autres informations sur le composant doivent être exploitées : la structure interne, les séquences d'invocation, la documentation, l'état courant, le modèle, etc.

1.1.4 Définition d'un modèle de composants

Un modèle est une abstraction d'un système autorisant les prédictions ou les inférences [32]. Grâce à son pouvoir d'abstraction, un modèle permet de faire face à la complexité du système qu'il représente et permet ainsi d'en faciliter la compréhension. Pour réduire la complexité, un modèle se focalise sur un nombre réduit d'aspects du système (structure, comportement, interactions...). Un ensemble de modèles permet alors de décrire un système sous différents angles. De plus, un modèle est souvent exprimé à l'aide d'un langage graphique afin de favoriser la communication entre les personnes intervenants sur le système.

Dans le contexte de la CBC, un modèle de composants définit la manière dont un composant doit être construit. Le modèle de composants standardise la structure, les interactions et les principes de composition que les composants doivent respecter pour être conformes au modèle [19]. Le composant étant ainsi standardisé, sa production de masse, systématique, est facilitée. Tous les composants respectant un même modèle de composants ont une « forme » commune et peuvent interopérer.

D'après Lau et Wang [26], un modèle de composants logiciels est caractérisé par :
- une sémantique décrivant les composants (définition des interfaces, services fournis/requis, opérations, etc) ;
- une syntaxe (graphique ou textuelle) servant à décrire la structure du composant, indépendamment de la syntaxe d'implémentation ;
- un principe de composition, détaillant la manière d'assembler les composants.

Actuellement, il n'existe pas de terminologie et de critères standardisés qui permettent d'identifier clairement un composant. Il n'y a pas de modèle unifié de composants, seulement des modèles de composants. Ainsi, un composant est identifiable comme tel seulement par

rapport à son modèle de composants. De ce fait, un composant devrait être systématiquement associé à son modèle afin de répondre au besoin de documentation (cf. section 1.1.1).

1.2 UML et les composants

1.2.1 UML et l'ingénierie des modèles

L'ingénierie des modèles (*Model Driven Engineering*, MDE) promeut l'utilisation des modèles dans le cycle de vie d'un système logiciel. Pour cela, le MDE cherche à rationaliser leur utilisation en définissant, d'une part, les abstractions nécessaires à la description d'un système et, d'autre part, les méthodes exploitant ces abstractions. L'ingénierie des modèles a ainsi pour but :

1. de faciliter la description d'un système avec ses problèmes et ses solutions ;
2. de maintenir la synchronisation d'un système entre sa spécification et son implémentation ;
3. de tester et de valider un système au plus tôt de son cycle de conception.

L'essor du MDE dans la CBC [14, 33–37] est en grande partie dû à l'initiative de l'*Object Management Group* (OMG) avec la standardisation du langage de modélisation graphique UML (*Unified Modeling Language*) [1] et de leur approche MDA (*Model-Driven Architecture*) [38].

UML regroupe un ensemble de modèles pour décrire la structure, le comportement, l'intégration d'un système et pour organiser ces modèles. Leur « forme » est contrainte par le métamodèle d'UML, qui est lui-même un modèle. Un modèle doit ainsi être conforme à son métamodèle pour être valide. Le métamodèle d'UML est extensible grâce à la création de *profils*. Un profil regroupe un ensemble de *stéréotypes* s'appliquant à des éléments du métamodèle d'UML et modifiant ainsi leur interprétation.

Le MDA est une méthode de conception itérative exploitant les modèles UML. Il propose de partir de modèles fonctionnels abstraits d'un système (*Platform Independent Model*, PIM, cf. fig. 1.4) pour obtenir une implantation exécutable du système sur une plate-forme de réalisation spécifique (*Platform Specific Model*, PSM, cf. fig. 1.4). Le passage du PIM au PSM se fait par des étapes successives d'enrichissements et de transformations de modèles (cf. figure 1.4) pour à la fin générer du code spécifique à un environnement d'exécution. Ces étapes de transformation garantissent par construction la correction et la conformité de l'implémentation d'un système avec sa spécification.

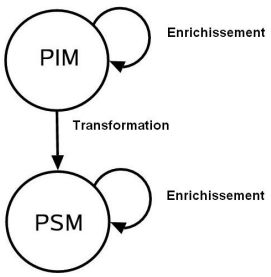

FIGURE 1.4 – Démarche MDA

Dans le but de valider le comportement d'un système dès la phase de conception, les modèles ont vocation à être directement exécutés, plutôt que transformés en du code exécutable. Un « moteur d'exécution » est alors chargé d'interpréter le modèle. Il est ainsi possible d'obtenir rapidement un prototype du système pour vérifier son comportement de façon expérimentale ou formelle. L'exécution de modèles nécessite alors que le modèle dispose d'une sémantique formelle. Malheureusement, celle d'UML est ambiguë et possède de nombreux « points de variation sémantique » [1]. De ce fait, il ne peut exister un unique moteur d'exécution lui étant strictement *conforme*. A cet effet, plusieurs initiatives récentes (ModelWare [39], TopCased [40], Eclipse Model Execution Framework [41]) proposent des architectures génériques pour développer des moteurs d'exécution dans lesquels la sémantique d'exécution est paramétrable.

1.2.2 Le modèle de composants UML

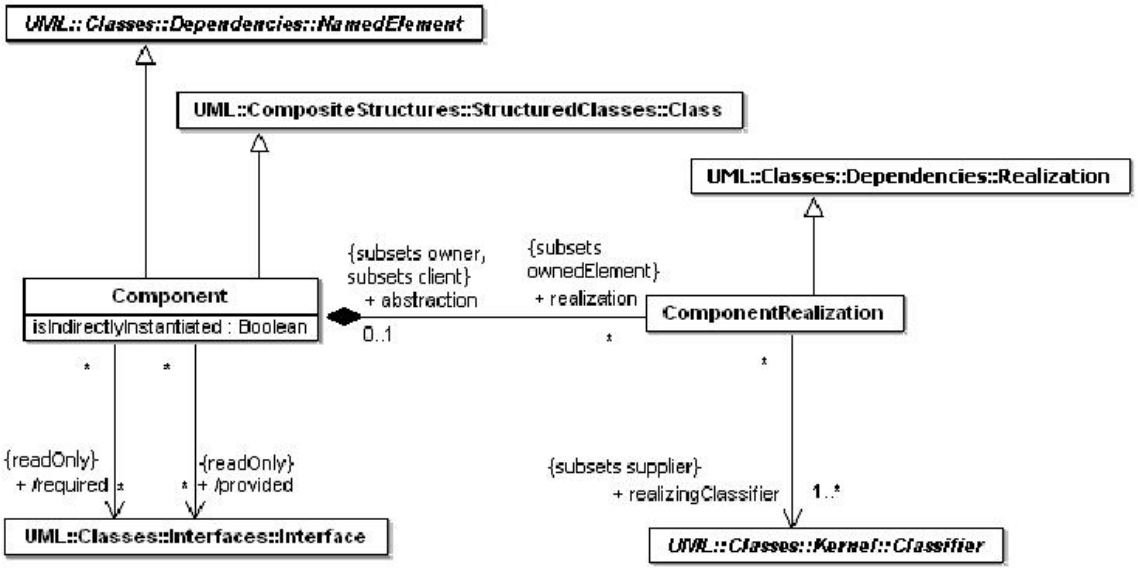

FIGURE 1.5 – Modèle de composants UML [1]

UML dispose dans son métamodèle d'une partie relative à la modélisation des composants logiciels. La figure 1.5 présente cette partie qui décrit le modèle de composants supporté. Ce modèle se veut générique et permet donc de modéliser des composants EJB, CCM (CORBA *Component Model*) et Microsoft .NET [1]. Un composant UML réalise la séparation entre spécification et réalisation des services. Le composant (`Component`) dispose d'un ensemble d'interfaces fournies (association `provided`) et requises (association `required`) pour présenter les services qu'il propose et qu'il requiert. Chacun des classifieurs qui lui est lié (association `realizingClassifier`) propose une manière de réaliser les services.

Les composants sont composés par des connecteurs spécifiques au type de composition. Ces connecteurs relient les interfaces entre elles (cf. fig. 1.2, p. 18) :

- les connecteurs de délégation sont utilisés pour la composition verticale. Un tel connecteur permet de déléguer un service d'un composant à un autre. Lorsque le connecteur relie une interface de service fourni d'un composite à celle d'un de ses sous-composants, cela signifie que le service est réalisé par le sous-composant. Lorsque le connecteur relie une interface de service requis d'un sous-composant à celle du composite, cela signifie que le service requis du sous-composant devient un service requis du composite ;
- les connecteurs d'assemblage sont utilisés pour la composition horizontale. Ils servent à relier un consommateur de service à un fournisseur. Le connecteur d'assemblage relie l'interface de service requis d'un composant avec l'interface de service fourni du composant en charge de la réalisation du service.

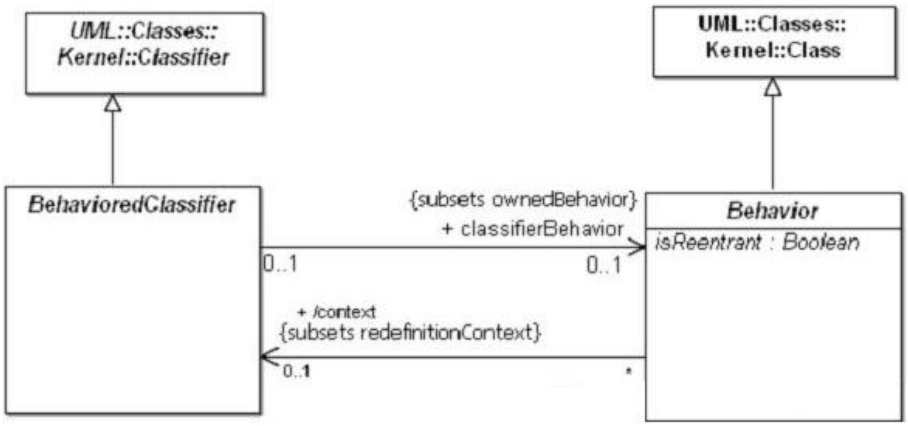

FIGURE 1.6 – Classifieur avec comportement [1]

UML autorise le classifieur qui réalise un composant à avoir un comportement (cf. fig. 1.6, association `classifierBehavior`). Nous nous intéressons ci-après à la spécification de ce comportement à l'aide de machines à états.

1.2.3 Les machines à états UML

Une machine à états UML permet de décrire le comportement d'un classifieur UML, qui peut lui-même servir à la réalisation des services d'un composant. Ces machines à états reposent sur le formalisme des *Statecharts* de Harel [42] qui sont eux-mêmes une évolution des automates à états finis [43].

1.2.3.1 Structure des machines à états

Une machine à états est une structure arborescente (cf. l'exemple de la figure 1.7, partie gauche) ayant les caractéristiques suivantes :

- la racine est la machine elle-même ;
- les nœuds des niveaux impairs sont des régions ;
- les nœuds des niveaux pairs sont des états ;

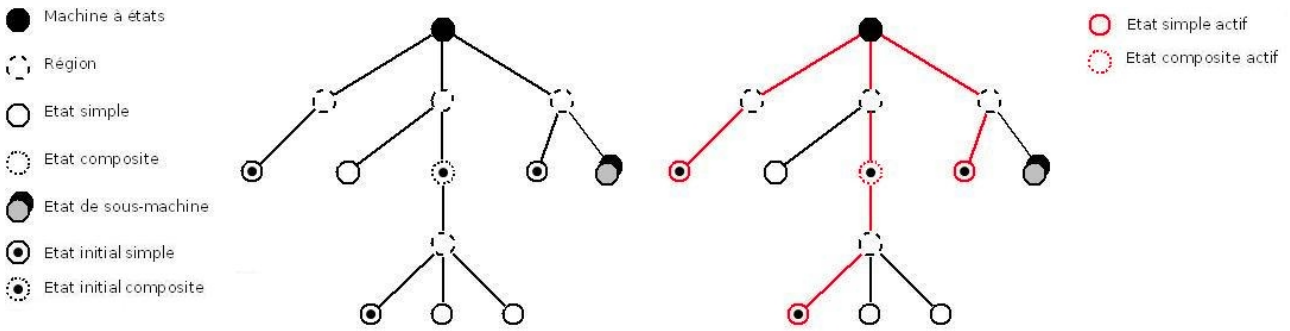

FIGURE 1.7 – Exemple de structure d'une machine à états

- les feuilles sont des états : la machine à états possède au moins une région et une région possède au moins un état ;
- un des états d'une région est l'état initial de cette région ;
- une région peut posséder un état final, distinct de l'état initial de cette région ;
- un état final est toujours une feuille.

La terminologie suivante s'applique alors :

- un état qui possède au moins une région est dit *composite* ;
- un état qui ne possède pas de région est dit *simple*, une feuille est donc un état simple ;
- un état qui référence la racine d'une autre machine à états est dit *de sous-machine*, ses fils sont alors les régions de la machine référencée (sur la figure 1.7, les sous-arbres de l'état de sous-machine ont été omis) ;
- deux états sont dits *orthogonaux* lorsque leur ancêtre commun est un état ;
- deux états sont dits *exclusifs* lorsque leur ancêtre commun est une région.

Deux états exclusifs peuvent être connectés par une *transition* : un des états est l'état-*source* de la transition, l'autre est l'état-*cible*[8]. Une transition est étiquetée suivant la syntaxe [<*déclencheur*>][<*garde*>]["/"<*effet*>] pour laquelle :

- le *déclencheur* spécifie l'événement (cf. section ci-après) auquel la transition est sensible. Une transition qui n'a pas de déclencheur est une *transition d'achèvement* ;
- la *garde* est une condition booléenne ;
- l'*effet* est une suite d'actions (par ex. invocation d'opérations, envoi de signal...) réalisées lorsque la transition est franchie.

Par ailleurs, un état peut posséder :

- des transitions dites *« internes »*, qui sont des transitions n'ayant pas d'état-cible et, par conséquent, n'entraînant pas de changement d'états ;
- une *action d'entrée* et une *action de sortie* ;
- une *activité*, qui, à la différence d'une action, s'exécute en un temps non négligeable ;
- un *invariant*, qui est une condition booléenne.

8. Une transition pour laquelle l'état-cible est aussi l'état-source est une auto-transition.

1.2.3.2 Fonctionnement des machines à états

Une instance de machine à états possède une *configuration d'états actifs*. La configuration initiale d'états actifs est l'arbre de la machine à états pour lequel chaque région ne possède que le fils étant l'état initial de cette région (cf. fig. 1.7, partie droite). Les feuilles de la configuration d'états actifs sont alors des états simples, orthogonaux entre eux.

Une fois que la configuration initiale a été construite, la machine est prête à réagir à des événements. Ces événements sont de plusieurs types :

- *les événements de signal* : une machine à états communique de façon asynchrone avec une autre machine en lui envoyant un signal. La réception du signal constitue un événement pour la machine destinataire. La machine émettrice n'attend pas que le signal soit traité pour poursuivre son propre fonctionnement. La machine émettrice peut être son propre destinataire. C'est le type d'événement utilisé dans les *Statecharts* ;
- *les événements d'appel* : une opération est appelée sur le classifieur dont la machine décrit le comportement. L'invocation de l'opération constitue un événement pour la machine du destinataire. L'appelant attend que la machine ait traité l'événement pour poursuivre son fonctionnement [44] ;
- *les événements de changement* : le déclencheur d'une transition est exprimé avec une condition booléenne (indépendamment de celle correspondant à la garde de la transition). Le fait que cette condition devienne vraie, lorsque l'état-source de la transition associée est actif, constitue un événement pour la machine à états. Pour des raisons de performance, ce type d'événement est à utiliser avec prudence. En effet, il nécessite souvent une attente active monopolisant les ressources du système ;
- *les événements temporels* : *absolus*, lorsqu'un moment particulier vient de passer, ou *relatifs*, lorsqu'une certaine durée vient de s'écouler. La durée écoulée est alors relative au moment de l'activation de l'état qui est la source de la transition spécifiant l'événement comme déclencheur ;
- *des événements d'achèvement* : un événement d'achèvement est généré :
  - lorsqu'un état simple est activé, s'il ne possède ni activité ni action d'entrée,
  - après que l'action d'entrée et l'activité d'un état simple se soient toutes les deux terminées,
  - lorsque l'activité d'un état composite se termine et que les états finaux de ses régions sont actifs.

  Cet événement est prioritaire par rapport aux autres et permet de déclencher une transition d'achèvement (cf. section 1.2.3.3).

La machine à états traite les occurrences d'événement suivant le mode *run-to-completion*, c.-à-d. qu'elle ne considère qu'une occurrence à la fois. Lorsqu'un événement est reçu, il est mis à la fin de la file d'attente des événements de la machine à états (sauf pour les événements d'achèvement qui sont traités prioritairement). Cette dernière prélève l'événement en tête de file et regarde s'il constitue un déclencheur pour une transition. Cette dernière est franchissable si sa garde est vraie et que son état-source est actif. Le franchissement s'effectue toujours selon l'ordre suivant : l'activité de l'état-source est arrêtée si elle n'est pas terminée, l'action de sortie

de l'état-source, puis l'effet de la transition, ensuite l'action d'entrée et enfin l'activité de l'état-cible sont exécutés. Le franchissement d'une transition interne ne provoque pas de changement d'état. Seul l'effet de la transition est alors exécuté.

Deux transitions sont franchissables en même temps seulement si leurs états-sources sont orthogonaux. Deux transitions franchissables dont les états-sources sont exclusifs sont soumis à un mécanisme de priorité : la transition dont l'état-source est un sous-état de l'état-source de l'autre transition est la transition qui sera effectivement franchie. Si les deux états-sources sont identiques, alors la transition dont le déclencheur est le plus spécialisé (dans le cas d'une hiérarchie de signaux) sera effectivement franchie. Si le choix ne peut être fait entre les deux transitions, la machine est indéterministe et par conséquent mal formée. Notons que le mécanisme de priorité rend possible la redéfinition d'une transition dans un sous-état, masquant ainsi celle contenue dans le super-état.

1.2.3.3 Exemple de machine à états

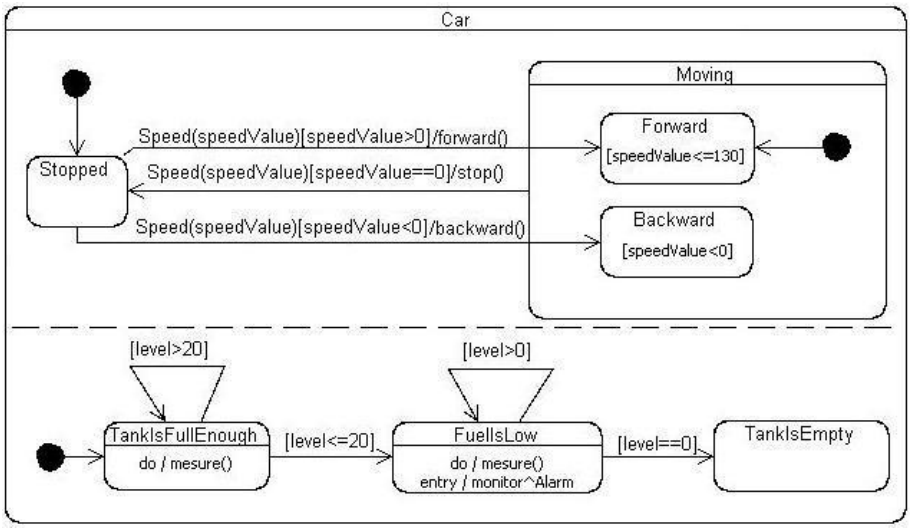

FIGURE 1.8 – Exemple de machine à états

La figure 1.8 présente la syntaxe graphique utilisée pour les machines à états avec l'exemple de la machine à états `Car`. Cette dernière décrit le comportement d'une voiture. Elle comprend deux régions (matérialisées par le trait discontinu les séparant) dont les états `Stopped` et `TankIsFullEnough` sont les états initiaux (désignés par le disque noir et la flèche qui en sort). L'état `Moving` est un état composite comportant une région qui contient deux sous-états simples. Chaque transition est matérialisée par une flèche orientée de l'état-source vers l'état-cible.

L'état `Stopped` est exclusif par rapport à l'état `Moving` et orthogonal par rapport à l'état `TankIsFullEnough`. L'état `Backward` a pour invariant `[speedValue<0]` qui précise que la vitesse de la voiture est négative tant que le véhicule se déplace en marche arrière. Lorsque les états

`Backward` et `TankIsFullEnough` sont actifs, la configuration active comprend les états `Moving`, `Backward` et `TankIsFullEnough`.

L'état `TankIsFullEnough` exécute l'activité `mesure()` (désignée par `do/`), qui mesure le niveau du réservoir de la voiture, tant qu'il est actif et que la mesure n'est pas terminée. Lorsque cette dernière se termine, la transition d'achèvement dont la garde est vraie est franchie. Si le niveau du réservoir devient inférieur ou égal à 20%, l'état `FuelIsLow` est activé. Le signal `Alarm`, indiquant que le niveau du réservoir est bas, est émis à l'entrée dans cet état (désigné par `entry/`). Ce signal est envoyé à la machine à états `monitor` (l'envoi de signal s'exprime avec la notation `destinaire^signal` qui correspond à l'opérateur OCL *hasSent*). L'activité `mesure()` est ensuite exécutée dans cet état.

1.3 Le modèle de composants PauWare

PauWare[9] [2] est un modèle de composants issu de l'ingénierie des modèles. Un composant PauWare a la particularité d'associer à sa spécification structurelle une spécification comportementale à base de machines à états. Il propose des principes de composition horizontale et de composition verticale basés sur le comportement des composants, plutôt que sur la structure comme le font les modèles de composants EJB [17], CCM [45] et Fractal [46].

1.3.1 Principes

Un composant PauWare utilise une machine à états UML pour décrire son comportement et le réaliser. La machine à états est alors embarquée dans le composant et est exécutée par un moteur d'exécution respectant la sémantique d'UML[10]. Le composant fournit un ensemble de services fonctionnels listés dans son interface de services fournis. Chaque service fourni correspond à un signal étiquetant une transition dans la machine à états du composant. Les actions correspondent à l'invocation d'opérations dans la classe d'implémentation et à des requêtes de services. Les services requis correspondent à des signaux envoyés à d'autres composants. Un composant PauWare fournit aussi un ensemble de services non fonctionnels listés dans son interface de configuration. Ces services permettent par exemple d'agir directement sur la machine à états en forçant un état particulier. Cette structure simple permet d'intégrer les principes de PauWare à d'autres modèles, comme cela a été fait pour les EJB dans [47].

La figure 1.9 présente en exemple les spécifications structurelle et comportementale d'un composant PauWare nommé `PauWare Component`. L'interface `PauWare component functional interface` expose les services fournis du composant (`go()`, `request_b()`, etc.). L'interface `PauWare component configuration interface` propose le service `reset()` pour réinitialiser le composant en activant la configuration initiale de sa machine à états. La classe `PauWare component implementation class` liste les opérations invoquées (`a()`, `w()`, etc.) par les actions de la machine à états. Le comportement représenté par la machine à états est exécuté par

9. http ://www.pauware.com/

10. Comme précisé à la section 1.2.1, ce respect est limité par les points de variation sémantique et par l'ambiguïté d'UML.

le moteur `Statechart_monitor` intégré au composant.

A son initialisation, le composant est dans l'état `Idle`. Seule l'invocation du service `go()` provoque une réaction de la part du composant. La machine à états transite en conséquence vers l'état `Busy` et active ses états initiaux (`S11`, `S22`, `S31` et `S32`). L'action `w()` est ainsi exécutée à l'entrée dans l'état `S11`. Le service requis `request_h()` est appelé à l'entrée dans l'état `S22`. A partir de cette configuration active, seule l'invocation de l'un des services `request_b()`, `request_c()`, `request_d()` et `request_e()` provoque une réaction de la part du composant.

1.3.2 Composition des composants PauWare

Les composants PauWare supportent une forme de composition horizontale en communiquant de manière asynchrone par envoi de signaux. La figure 1.10 montre les machine à états d'un composant `Car` et d'un composant `GearBox`. Lorsque le service `shift()` est invoqué sur le composant `Car`, ce dernier passe de l'état `Stopped` à `Moving` et invoque le service `up()` du composant `GearBox`. Celui-ci déclenche en réponse l'action interne `moveUp()`.

Les composants PauWare supportent de plus la composition verticale en composant les machines à états via les états de sous-machine [48]. Un état de sous-machine orthogonal au comportemement du composite est alors rajouté pour chaque sous-composant. Cet état est destiné à accueillir la machine à états du sous-composant. Ainsi, lorsque le composite reçoit un signal, il est diffusé dans les machines à états des sous-composants.

Sur la figure 1.11, le composant `Car` est maintenant composé verticalement avec le composant `GearBox` et avec un composant `Tank Monitor`. Lorsque le service `shift()` est invoqué, le composant `Car` transite toujours de l'état `Stopped` à `Moving` mais cette fois-ci s'envoie à lui-même (`self^`) le signal `up()`. Ce signal est alors diffusé dans sa machine à états qui atteint directement celle du composant `GearBox` et aussi celle du composant `Tank Monitor`. Le signal `up()` ne correspondant à aucun service du composant `Tank Monitor`, il n'a pas de conséquence sur ce dernier.

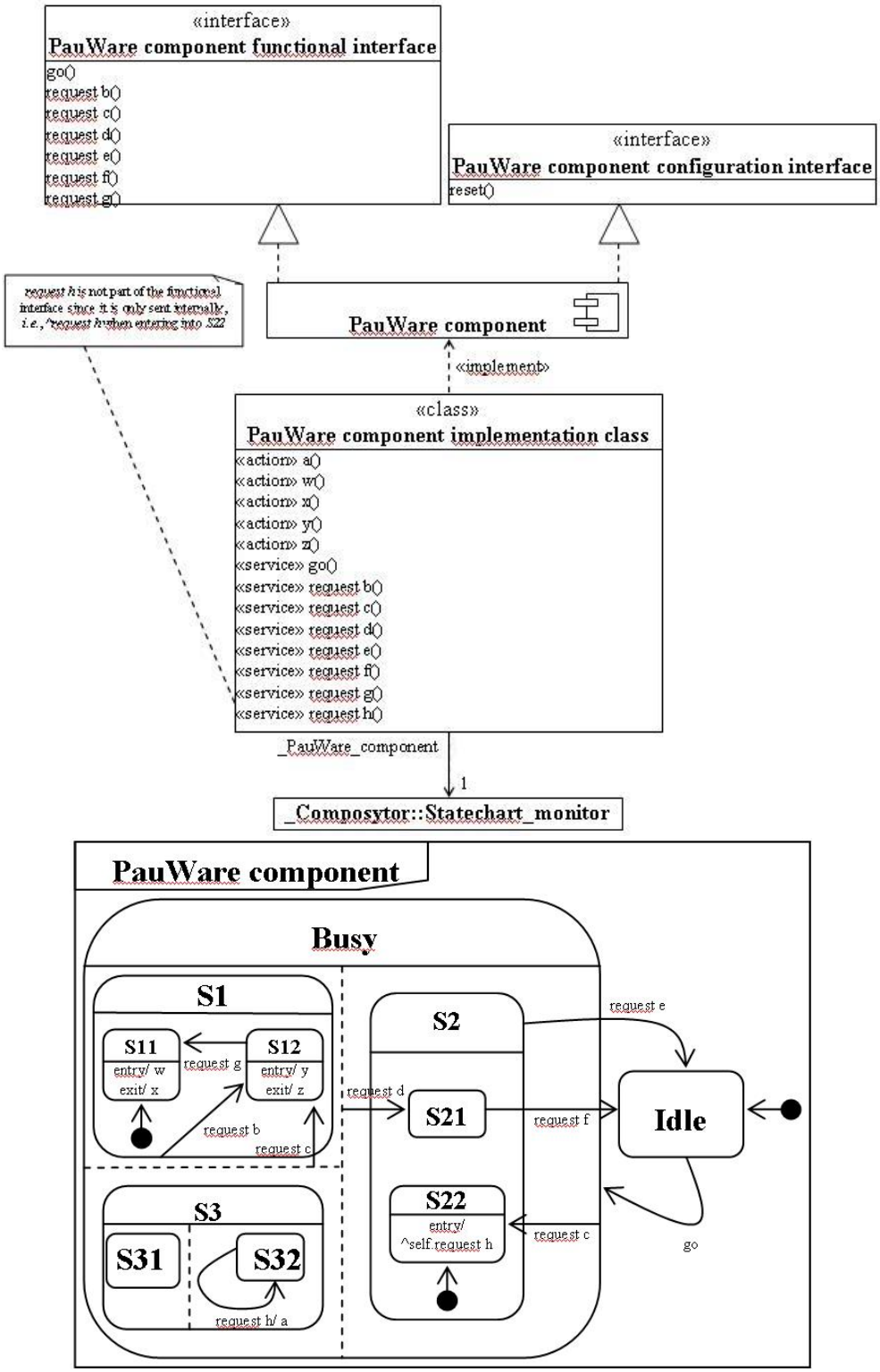

FIGURE 1.9 – Spécification du composant `PauWare Component` [2]

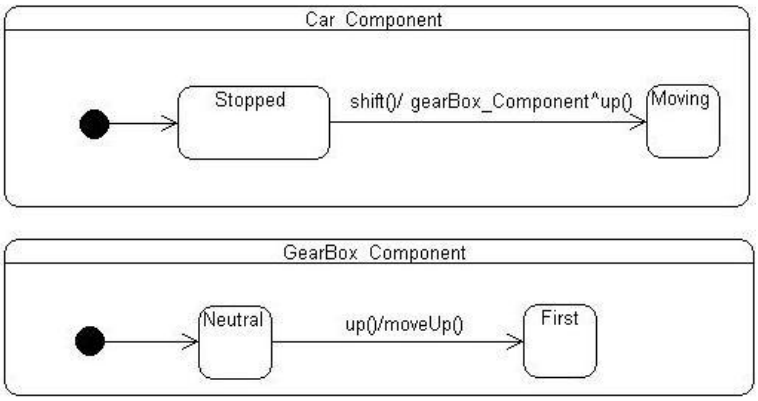

FIGURE 1.10 – Composition horizontale dans PauWare

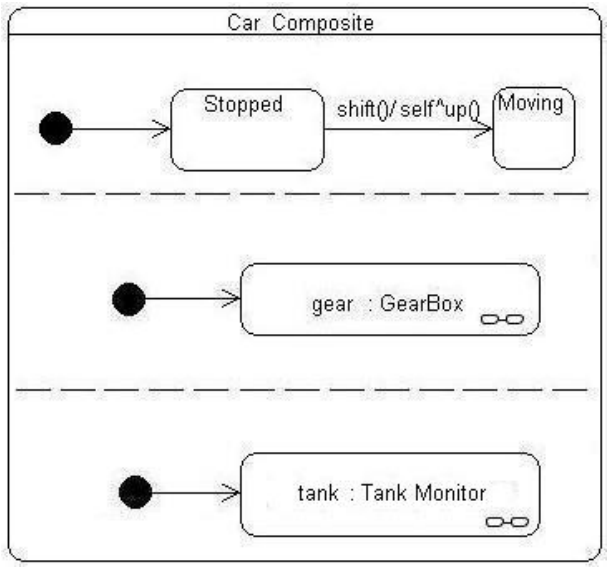

FIGURE 1.11 – Composition verticale dans PauWare

1.4 Synthèse

Un composant est conçu pour la réutilisation. Il est destiné à être composé avec d'autres composants afin de constituer un système. Pour cela, il nécessite d'avoir des points d'interaction clairement définis. Afin de faciliter son utilisation, il doit être associé à une documentation permettant de comprendre son fonctionnement sans accéder à l'implémentation. Il est nécessairement associé à un modèle de composants permettant de caractériser et de borner l'espace auquel il appartient.

Un langage de modélisation comme UML spécifie un modèle de composants générique afin de concevoir des composants. Il propose un ensemble de notation pour spécifier la structure et le comportement de ces derniers. Le comportement d'un composant peut notamment être spécifié avec des machines à états. Le modèle de composants PauWare utilise ces dernières comme support à la composition.

Un concepteur de composants ne peut pas envisager toutes les situations dans lesquelles sera utilisé un composant. Ce dernier doit de ce fait être personnalisable, ou encore *adaptable* par celui qui va l'utiliser. Afin de limiter les interventions humaines, un composant doit idéalement être *adaptatif*, c.-à-d. capable de s'adapter lui-même.

Les systèmes, indépendamment du fait qu'ils soient ou qu'ils ne soient pas conçus avec des composants, ont eux-même besoin d'être adaptés. Nous introduisons dans le chapitre suivant les raisons qui motivent le besoin de construire des systèmes adaptables et adaptatifs dans le contexte de l'informatique autonomique et nous détaillons les spécificités liées à ces systèmes.

Chapitre 2

Les systèmes logiciels adaptatifs

Sommaire

La complexité atteinte par les systèmes informatiques actuels entrave leur intégration, leur évolution et leur gestion. Ces activités normalement dévolues à des intégrateurs, développeurs ou administrateurs systèmes tendent à être assurées par les systèmes eux-mêmes. Cette vision de systèmes informatiques qui s'autogèrent a été développée en 2001 par IBM lorsque Paul Horn a introduit l'informatique autonomique [8]. Le terme « autonomique » a été choisi par analogie avec le système nerveux autonome (*autonomic nervous system*) qui assure entre autres, la régulation de notre fréquence cardiaque, de notre glycémie, de notre respiration, « sans conscience ni effort » de notre part. Les *systèmes autonomiques* sont donc des systèmes capables d'autogestion (*self-management*) sur lesquelles les interventions humaines sont limitées afin d'accroître la réactivité et la disponibilité.

La section 2.1 introduit l'informatique autonomique : elle présente les propriétés qui lui sont liées, elle définit le concept de « boucle de contrôle » et elle sensibilise au besoin de coordina-

tion nécessaire au maintien de la cohérence d'un système autonomique. La section 2.2 définit les systèmes adaptatifs, montre qu'ils sont une base nécessaire à la réalisation des systèmes autonomiques et justifie le besoin de dynamisme et d'ouverture de ces systèmes. La section 2.3 présente les problèmes liés au déroulement d'un processus d'adaptation en s'intéressant au moment où l'adaptation doit être appliquée, à la gestion de la cohérence du système adapté ainsi qu'à la continuité de l'exécution du système après son adaptation.

2.1 L'informatique autonomique

2.1.1 Les propriétés *auto-**

Lors de la présentation de l'informatique autonomique, IBM a mis en avant quatre propriétés fondamentales des systèmes autonomiques [8, 49], *les self-*CHOP (*self-configuration, self-healing, self-optimization, self-protection*) [50]. Ces propriétés ont pour but de maintenir les services assurés par le système :

- *l'auto-configuration* : un système autonomique assure ses services quel que soit l'environnement dans lequel il évolue et quelle que soit l'évolution de cet environnement ;
- *l'auto-réparation* : un système autonomique minimise l'impact de ses dysfonctionnements sur les services qu'il fournit ;
- *l'auto-optimisation* : un système autonomique assure ses services avec la meilleure qualité possible à chaque instant ;
- *l'auto-protection* : un système autonomique s'immunise face aux attaques menées contre lui.

Ces propriétés constituent la base historique de l'informatique autonomique. Elles ont depuis été largement déclinées et raffinées (l'auto-organisation, l'auto-récupération, l'auto-assemblage, l'auto-simulation...) [51–53]. Par ailleurs, elles sont associées à quatre caractéristiques nécessaires pour les réaliser [8] :

- *la conscience de soi* : un système autonomique ne peut gérer que ce qu'il connaît de lui-même ;
- *la conscience de son environnement* : un système autonomique est déployé au milieu d'un environnement qu'il partage avec d'autres systèmes. Il doit connaître cet environnement et sa place dans cet environnement pour mieux l'appréhender ;
- *l'ouverture* : un système autonomique est ouvert sur les autres systèmes présents dans l'environnement. Il est capable de communiquer avec eux ;
- *l'anticipation* : un système autonomique se prépare à traiter des événements avant leur occurrence afin de mieux y répondre.

2.1.2 La boucle de contrôle

Dans un système classique, c.-à-d. non-autonomique, un administrateur est chargé de surveiller le système, d'en analyser les besoins et de choisir les actions à réaliser pour maintenir les services assurés par ce système. Dans un système autonomique, l'approche utilisée en génie logiciel pour réaliser les propriétés autonomiques consiste à doter le système d'une boucle de

contrôle [50, 51, 54, 55]. La procédure précédemment effectuée par l'administrateur est ainsi automatisée. Ce dernier devient alors le superviseur de la boucle dont le comportement est guidé par des politiques de gestion qu'il établit.

Dans son architecture de référence pour les systèmes autonomiques [3], IBM présente des éléments distincts pour réaliser les différentes fonctions de la boucle. Cette dernière comprend (cf. figure 2.1) :

- un ensemble de capteurs (alias sondes, détecteurs...) attachés au système à surveiller. Les sondes peuvent vérifier les valeurs des paramètres fonctionnels (taux de perte de paquets sur un réseau [56]), la levée d'exceptions [57], la violation d'invariants [58], l'introduction de nouveaux composants [36], les ressources (taux d'utilisation du processeur, de la mémoire), etc. ;
- un surveillant qui collecte les informations auprès des capteurs, les aggrège, les filtre ;
- un analyseur (alias évaluateur, décideur...) qui corrèle les informations reçues du surveillant et modélise les phénomènes observés ;
- un planificateur qui, en fonction des éléments rapportés par l'analyseur, produit la liste des actions à réaliser ;
- un exécuteur qui réalise le plan des actions précédemment établi ;
- un ensemble d'effecteurs (alias actionneurs...) pour agir sur le système ;
- une base de connaissances qui regroupe toutes les informations nécesaires au contrôle (modèles mathématiques, historique des événements, plans réalisés, connections...) ;
- une politique qui guide la prise de décision de l'analyseur. Trois types se distinguent [59] :
  - une politique à base d'actions, la plus simple, est un ensemble de règles *événement-condition-actions* où la condition se rapporte à l'état courant du système sur lequel la règle s'applique. L'action permet d'atteindre un nouvel état ;
  - une politique à base de buts dans laquelle, à partir d'un état courant, un ensemble d'états-cibles est spécifié. Le système est chargé de déterminer les actions qui permettent de les atteindre ;
  - une politique à base de fonctions d'utilité dans laquelle les états-cibles sont ordonnés par ordre de satisfaction. La fonction d'utilité calcule alors à partir de l'état courant du système un indice de satisfaction pour chaque état cible.

L'exemple de la figure 2.2, inspiré de [55], réalise une boucle contrôlant la température d'une pièce. Un capteur mesure périodiquement la température de la pièce et envoie la valeur à un thermostat. Le thermostat comporte une politique d'actions :

- si la température est supérieure à 20 °C, le thermostat commande la mise en marche de la climatisation afin de refroidir la pièce ;
- si la température est inférieure à 18 °C, la climatisation est arrêtée.

2.1.3 Coordination des boucles de contrôle

Une boucle de contrôle peut exister à plusieurs niveaux dans un système autonomique :

- à un niveau global : le système dispose d'une unique boucle de contrôle. Le contrôle est ainsi centralisé et nécessite une connaissance globale du système pour l'appréhender. Cette approche permet d'avoir un interlocuteur unique et connu pour orchestrer le sys-

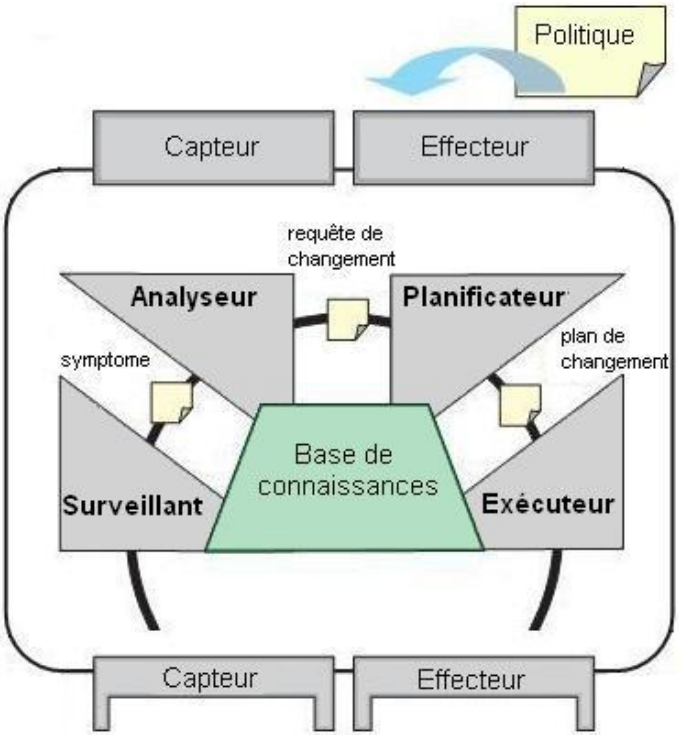

FIGURE 2.1 – La boucle de contrôle selon IBM [3]

tème. Néanmoins, elle est fragile car elle présente un goulot d'étranglement lorsque le système se complexifie (problème de passage à l'échelle) et elle manque de flexibilité car les informations sont concentrées en un point ;

– à un niveau local : chaque entité du système dispose de sa propre boucle de contrôle. Le contrôle est ainsi décentralisé et la boucle agit sur le système en fonction de ses propres connaissances. Cette approche a l'avantage de supporter le passage à l'échelle et d'être flexible grâce à la répartition des informations et du processus de contrôle. Néanmoins, elle est complexe à mettre en place car elle implique de nombreuses interactions entre les boucles pour maintenir la cohérence du système [60].

La présence de multiples boucles de contrôle à différents niveaux réalisent mieux la vision de l'informatique autonomique [50]. L'enjeu est donc de définir leurs relations afin d'avoir un système au comportement déterministe [55], c.-à-d. un système dont le comportement reste prévisible quelle que soit l'évolution de son environnement, de ses entités et de leurs interactions. Les boucles de contrôle peuvent se coordonner de deux manières différentes [55] :

– soit de manière indirecte, lorsqu'une boucle modifie des paramètres de l'environnement sur lesquels les autres s'appuient. Par exemple, sur la figure 2.3, une boucle de contrôle réalisée par une personne est ajoutée à celle existant dans l'exemple de la figure 2.2. Les deux boucles coexistent de manière indépendante et contrôlent le même paramètre, c.-à-d. la température de la pièce, mais de manière différente : la boucle `Personne` augmente la température en remettant du bois dans la cheminée alors que la boucle `Thermostat` diminue

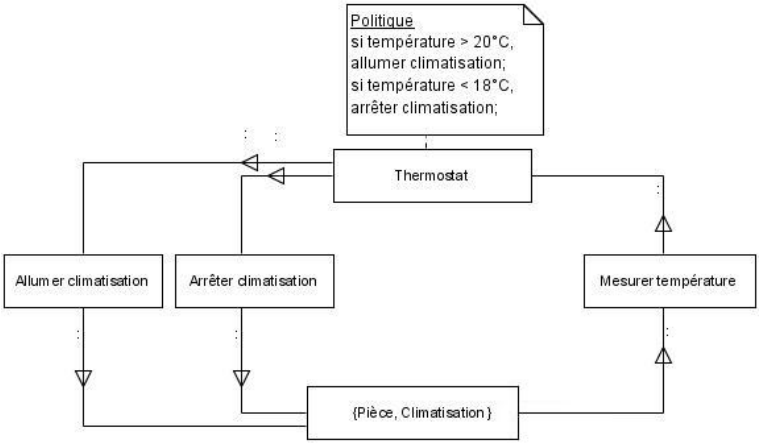

FIGURE 2.2 – Exemple de boucle de contrôle

la température en allumant la climatisation ;
– soit de manière directe, lorsqu'une boucle agit, en fonction de ses propres sensations et objectifs, sur la politique réalisée par une autre. Par exemple, sur la figure 2.4, la boucle `Personne` agit, en fonction de la température qu'elle ressent, directement sur la boucle `Thermostat` en modifiant la température cible définie dans la politique.

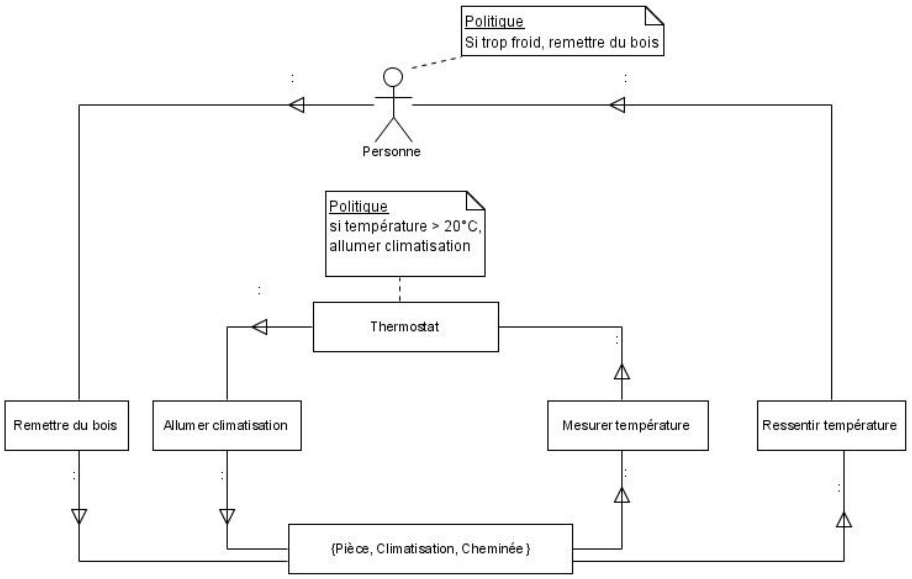

FIGURE 2.3 – Relation indirecte entre deux boucles de contrôle

La coordination indirecte des boucles, du fait de leur indépendance, ne permet pas de garantir la convergence des politiques pour réaliser les objectifs de chaque boucle. Des comportements émergents résultant des actions de chaque boucle peuvent alors être observés. Les politiques

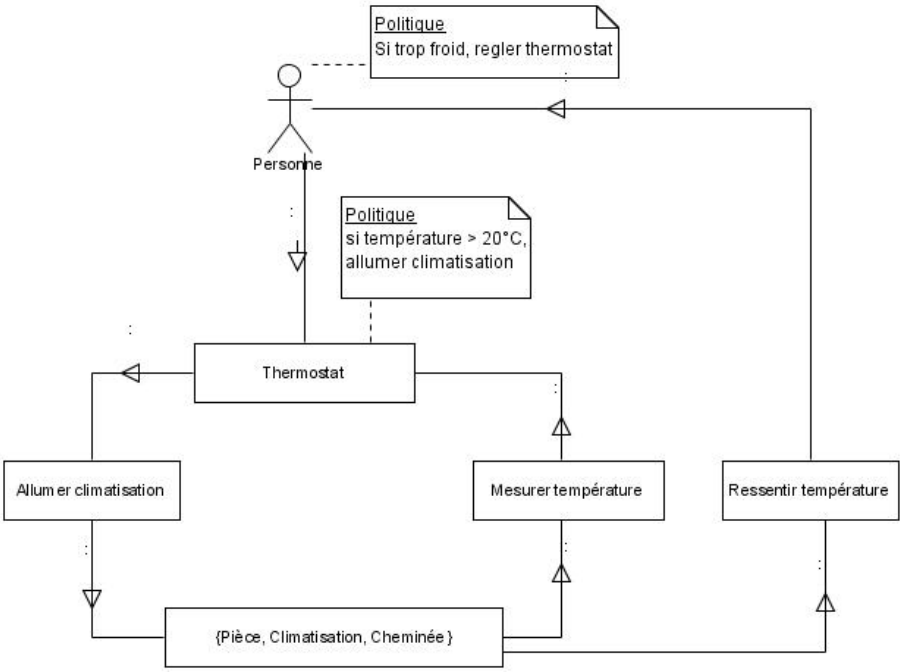

FIGURE 2.4 – Relation directe entre deux boucles de contrôle

peuvent très bien induire une mise en concurrence des boucles (cf. fig. 2.3, le thermostat cherche à réduire la température en dessous de 20 °C alors que la personne peut chercher à atteindre 25 °C) ou bien elles peuvent être complémentaires (lorque la personne souhaite 18 °C).

En revanche, la coordination directe des boucles permet de contrôler explicitement leurs politiques. Il est possible d'établir des relations de type maître-esclave (cf. fig. 2.3) afin d'assurer la convergence des politiques. Néanmoins, la coordination se complexifie exponentiellement avec le nombre de boucles impliquées et le nombre de messages nécessaires pour arriver, si cela est possible, à une décision satisfaisant chaque politique.

Par exemple, Sterritt et al. [61] proposent un bus de communication auquel chaque composant autonomique est rattaché. Chacun émet régulièrement une « pulsation » signifiant qu'il est toujours apte à assurer ses fonctions. Brooks [62] propose une architecture dite « de subsomption » dans laquelle les boucles de contrôle sont hiérarchisées suivant un niveau de priorité. Lorsqu'une boucle de contrôle de niveau n doit agir sur le système, elle inhibe celles des niveaux inférieurs.

L'architecture du projet SELFMAN [55] n'est pas sans rappeler l'architecture de subsomption décrite ci-dessus. Elle repose aussi sur des boucles de contrôle hiérarchisées, à la différence qu'une boucle de niveau n influence la politique réalisée par les boucles des niveaux inférieures (cf. fig. 2.3). De même, Dai et al. [63] proposent un contrôle hiérarchique : chaque niveau surveille le niveau qui lui est inférieur. De plus, les contrôleurs d'un même niveau peuvent se surveiller entre eux. Le contrôle hiérarchique offre un bon équilibre entre la lourdeur d'un contrôle décentralisé

et la rigidité d'un contrôle centralisé [63].

2.1.4 Synthèse

Le tableau 2.1 résume les propriétés et caractéristiques de l'informatique autonomique. Il les illustre par des exemples d'actions et pointe vers des travaux les concernant. La majorité de ces travaux se sont jusqu'alors concentrés sur l'auto-configuration et l'auto-optimisation de systèmes spécifiques (serveurs web, bases de données, centres de calcul) [7, 49]. Certains proposent des techniques pour doter un système d'une propriété autonomique particulière. La boucle de contrôle est alors définie de manière *ad hoc* par rapport au système qui doit être contrôlé.

Bien que des bibliothèques de capteurs pour surveiller l'environnement d'un système (surveillance du taux d'utilisation du processeur, de la présence de connexion au réseau, de la charge de la mémoire) existent [64], tous les éléments de la boucle ne sont pas généralisables. En effet, les capteurs surveillant le système lui-même, ainsi que les effecteurs agissant sur le système, restent propres au système contrôlé car ils réalisent l'interface entre le système et la boucle de contrôle. De même, les politiques liant les informations rapportées par les capteurs et les actions que les effecteurs devront déclencher dans le système, sont aussi spécifiques au système. Les autres éléments sont généralisables au sein d'un « cadriciel autonomique » tant au niveau de leur structure que de leurs fonctionnalités. Il devient alors intéressant d'utiliser les systèmes à base de composants afin de segmenter le contrôle des systèmes autonomiques. Néanmoins, de tels cadriciels proposant une architecture générique aux systèmes autonomiques et unifiant les propriétés autonomiques autour d'un système unique prenant en compte toutes les propriétés autonomiques sont rares [65].

2.2 Les systèmes adaptatifs

L'informatique autonomique est une discipline émergente qui cherche ses limites : si son but général est clair, les moyens d'y parvenir sont eux encore à définir. Dans ce contexte, les systèmes adaptatifs constituent une base intéressante pour la réalisation des systèmes autonomiques.

Dans la chronologie établie par IBM, la construction de systèmes adaptatifs est l'étape à atteindre avant celle des systèmes autonomiques [54]. Cette étape doit permettre la « fermeture » de la boucle de contrôle : l'administrateur humain n'est plus que le superviseur du contrôle automatisé des ressources du système.

2.2.1 Définition d'un système adaptatif

> *« Self-adaptive software modifies its own behavior in response to changes in its operating environment. »* [84]

> *« Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible. »* [85]

Propriétés et Caractéristiques	Résultats	Exemples	Références
Auto-configuration	être prêt à assurer ses services	installation de composants, résolution de dépendances, découverte de services	projet DySCAS [66], Unity [67], protocoles réseaux [68], DACAR [69]
Auto-réparation	ne pas défaillir	mise hors-ligne, remplacement, redémarrage de composants défaillants, limitation de services, fonctionnement alternatifs, mise à jour, introduction de redondance	Autonomia [70], KX [71], ArcOS [57], Tune [72], acCAT [73]
Auto-optimisation	donner le meilleur de soi-même	ajustement de paramètres, équilibrage de charge, répartition de ressources	Unity [74], projet Kendra [75], TRAP/J [56], Medium de communication [36], PLASMA [76]
Auto-protection	s'immuniser	détection d'attaques, élimination des virus, mise à jour, cryptage, fermeture de canaux de communication	définition [77], projet ADS [78], Autonomia [79], DARPA Self-Regenerative Systems [80]
Conscience de soi	se connaître	connaître sa structure, son état, son comportement, ses interactions, ses ressources	Mowbray et Bronstein [81]
Conscience de son environnement	connaître les autres	consultation d'annuaires, présentation de ses services	Schilit et al. [82]
Ouverture	interagir	standardisation d'interfaces et de protocoles, *open source*	Services Web [83]
Anticipation	être proactif	déploiement de serveurs en prévision d'un pic de charge, surveillance accrue de matériels en prévision de pannes	Autonomia [79], Prévisions probabilistes [63]

TABLE 2.1 – Résumé des propriétés et caractéristiques autonomiques

A travers leurs définitions, Oreizy et al. [84] et Laddaga [85] se rejoignent sur le principe des systèmes adaptatifs : ce sont des systèmes qui modifient leur comportement. Néanmoins, en considèrant les raisons de l'adaptation, ils adoptent des points de vue différents : Oreizy et al. se concentrent sur l'extérieur du système en s'intéressant aux changements de l'environnement alors que Laddaga considère l'intérieur du système en s'intéressant au fonctionnement de ce

dernier.

La définition de Laddaga [85] est plus proche de l'informatique autonomique car elle contient les intentions exprimées avec les propriétés d'auto-réparation et d'auto-optimisation ainsi que la conscience de soi nécessaire à l'évaluation du comportement. L'auto-adaptation est ainsi un moyen de réaliser un objectif essentiel exprimé par les propriétés autonomiques.

Pour notre part, nous limiterons la définition d'un système adaptatif à « un système capable de se modifier par lui-même ». Un système adaptatif est donc un système *auto-adaptable*. Nous le distinguons du système adaptable qui est « un système qui peut être modifié ». Ainsi, nous n'anticipons pas dans la définition les raisons de l'adaptation. Nous étudions ces dernières ci-après.

2.2.2 Les raisons de l'adaptation

La réalisation d'un système informatique est constituée de nombreuses itérations au sein des phases de conception, de développement, de tests ou bien encore de déploiement. Au cours de ces phases, les modifications apportées à un système le sont principalement pour l'une des quatre raisons suivantes [86, 87] :

- *correction* : les tests révèlent que le comportement du système n'est pas identique à celui décrit dans sa spécification. Dans le contexte de l'informatique autonomique, il s'agit de l'auto-réparation ;
- *changement d'environnement* : l'environnement dans lequel le système doit être déployé ne correspond pas à celui pour lequel il a été créé ; les systèmes avec lesquels il interagit ont été modifiés, il doit évoluer en conséquence. Dans le contexte de l'informatique autonomique, il s'agit de l'auto-configuration ;
- *perfectionnement* : le comportement observé du système déployé révèle une qualité de services décevante. Les performances doivent être améliorées. Dans le contexte de l'informatique autonomique, il s'agit de l'auto-optimisation ;
- *extension* : l'utilisateur du système requiert de nouvelles fonctionnalités. Elles doivent être intégrées dans le système sans le dégrader. Il n'y a pas d'équivalent dans les propriétés autonomique car elles ont été définies dans le cadre de systèmes fermés (cf. section 2.2.4).

Ces raisons sont corrélées à une divergence de l'implémentation et de la spécification du système [88]. La spécification, à travers les descriptions du comportement, des fonctionnalités, de l'environnement de déploiement, des performances attendues, des conditions de fonctionnement (par ex. un capteur de température qui ne fonctionne qu'entre -10 °C et +40 °C) du système, est le document de référence qui permet de mesurer cette divergence.

Ainsi, lorsque le système sort des limites décrites dans sa spécification, une adaptation est nécessaire pour qu'il revienne dans ses limites ou bien pour qu'il fonctionne suivant un mode tolérant ces nouvelles limites. Dans ce sens, des techniques de « différenciation » ont ainsi été proposées : Egyed [89] simule le modèle d'un système en parallèle de l'exécution du système lui-même. Les entrées du système exécuté servent aussi d'entrées à la simulation. Lorsque les sorties de la simulation et de l'exécution sont différentes, une action d'adaptation est entreprise.

2.2.3 Une adaptation dynamique

Au-delà des raisons y conduisant, l'adaptation se différencie en trois types suivant la phase pendant laquelle elle est réalisée [4, 90] :

- *l'adaptation statique* : l'adaptation porte sur le système en cours de conception et de développement. Elle correspond aux activités traditionnellement liées à la maintenance [86]. Son support peut être le modèle de conception, la documentation ou bien encore le code source. Son application nécessite alors l'arrêt du système et donc une rupture des services fournis. Toutes les futures instances présenteront le caractère adapté ;
- *l'adaptation semi-dynamique* : l'adaptation a lieu au cours du déploiement du système en fonction de l'environnement cible. Elle correspond aux activités réalisées par un administrateur système. Son support est principalement le script de déploiement du système. Les paramètres sont propres à chaque environnement de déploiement. Cette adaptation semi-dynamique doit disparaître puisqu'elle est soumise à la propriété d'auto-configuration du système ;
- *l'adaptation dynamique* : l'adaptation a lieu alors que le système est en cours d'exécution. Son support est donc une instance du système. De ce fait, chaque instance évolue indépendamment et ne présente donc pas obligatoirement le caractère adapté. Cette adaptation a pour but de maintenir la continuité des services assurés par le système. Elle implique de devoir garantir la cohérence du système pendant et après l'adaptation.

Le fait que les systèmes autonomiques et adaptatifs aient « conscience d'eux-mêmes » implique qu'ils soient « vivants ». Se situer dans un contexte dynamique est donc implicite lorsque l'on parle de ces systèmes. Par conséquent, dans les sections suivantes, lorsque le terme adaptation est mentionné, il n'est plus fait référence qu'à l'adaptation dynamique, sauf mention contraire.

2.2.4 Le support des adaptations non anticipées

Un système adaptatif se caractérise par un ensemble de comportements et par un ensemble d'adaptations qui permettent de transiter d'un comportement à un autre [91]. Un système est dit fermé lorsque ces ensembles sont bornés par les comportements et les adaptations définis lors de la phase de conception [84]. Or, parmis les raisons pour lesquelles l'adaptation est requise, toutes ne peuvent être anticipées dès cette phase. Afin de pouvoir pallier ultérieurement ce manque d'exhaustivité, le système doit supporter, de manière dynamique, des « adaptations non anticipées » [92].

Le système doit alors être ouvert afin de pallier les défauts de la conception initiale. Dès lors, de nouveaux comportements mais aussi de nouvelles politiques et de nouvelles entités réalisant les boucles de contrôle vont pouvoir être introduits dans le système. Ces éléments peuvent être ajoutés par un administrateur système ou bien par un autre système logiciel. L'ouverture n'est malheureusement pas supportée par tous les types de système : particulièrement les systèmes embarqués dont l'occupation mémoire est strictement bornée et les systèmes enfouis pour lesquels les nouvelles entités sont difficilement délivrables.

2.3 Gestion de l'adaptation

L'adaptation d'un système doit faire face à trois problèmes majeurs. Ebraet et al. [92] les décrivent comme étant liés à l'activité des entités adaptées, au transfert d'état nécessaire à la continuité du fonctionnement et à l'incertitude des effets de l'adaptation. Chacun de ces problèmes correspond à une étape dans la réalisation de l'adaptation. Nous les présentons ici suivant l'enchaînement de ces étapes.

2.3.1 Le moment de l'adaptation

L'adaptation dynamique s'applique lorsque le système est en cours d'exécution. Etant donné que l'adaptation est une opération de modification, elle est une opération risquée qui peut porter atteinte à l'intégrité du système. L'adaptation doit donc avoir lieu à un moment maîtrisé dans l'exécution du système. Ce moment doit minimiser l'impact sur le système afin de maintenir ses fonctionnalités et sa réactivité.

Dans les approches *ad hoc* à l'adaptation (Polylith [93], Zang et Cheng [94]), le concepteur précise lui-même les points d'application de l'adaptation. Lorsque ces points sont atteints au cours de l'exécution, le système sait qu'il est dans un état lui permettant d'être adapté. Cette approche comporte des risques d'erreurs et peut manquer d'exhaustivité car elle dépend fortement de la capacité du concepteur à appréhender le système. De plus, elle n'est pas transparente pour le concepteur qui doit connaître les mécanismes d'adaptation et manque de réactivité étant donné que le système doit attendre d'atteindre ces points pour être adapté.

Afin de systématiser les moments de l'adaptation, Kramer et Magee [95] introduisent la condition de « quiescence ». Ils considèrent un système dans lequel les entités interagissent deux à deux suivant un protocole de communication. Dans ce contexte, un protocole est une suite cohérente de messages échangés entre deux entités : une entité est l'initiateur du protocole alors que l'autre est le participant (cf. fig. 2.5, le protocole entre les entités X et Y). Une entité participant à un protocole peut initier en conséquence un protocole avec une troisième entité (cf. fig. 2.5, le protocole entre les entités Y et Z). Lorsque l'adaptation d'une entité est requise, le système attend que cette entité soit dans un état quiescent. Un état quiescent est un état dans lequel l'entité est isolée du reste du système : elle n'est pas engagée dans un protocole qu'elle a initié ou dans lequel elle participe. Ainsi, sur la figure 2.5, l'entité Z ne peut être adaptée qu'avant l'envoi du message 1 par l'entité X ou bien qu'après la réception du message 8 marquant la fin du protocole.

Vandewoude et al. [96] jugent la quiescence comme étant une condition trop stricte. En effet, la quiescence d'une entité implique que toutes les entités nécessitant directement ou indirectement la participation de cette entité, aient terminé leur protocole (c.-à-d. que tous les messages aient été échangés). L'entité qui doit être adaptée doit de plus prévenir les autres entités de s'interrompre afin de ne pas initier de protocole l'impliquant, ce qui augmente d'autant le couplage entre les entités. Pour pallier cela, les auteurs définissent la condition de « tranquillité ». Celle-ci impose la visibilité boîte noire aux entités du système. Ainsi, lorsqu'une entité initie un protocole avec une autre entité, elle n'est pas au courant des protocoles que cette dernière initiera en conséquence. Par exemple, sur la figure 2.5, l'entité X ne sait pas que l'entité Y a

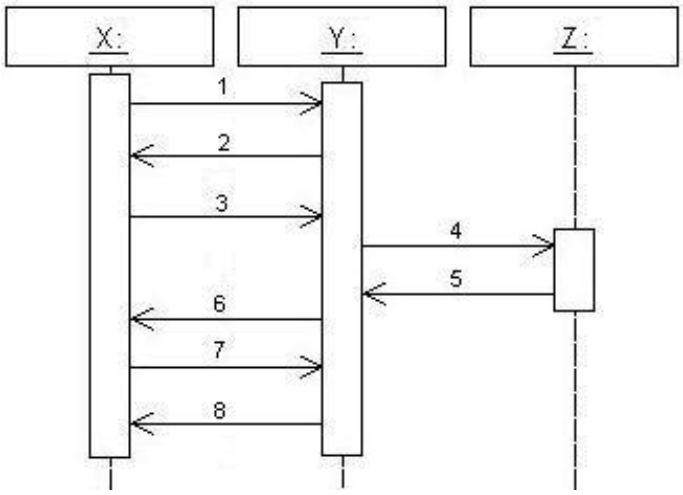

FIGURE 2.5 – Le protocole entre Y : et Z : intervient dans le protocole entre X : et Y :

besoin de communiquer avec l'entité Z pour réaliser le protocole entre X et Y. De ce fait, l'entité Z peut être adaptée avant la réception du message 4 et après l'envoi du message 5.

Néanmoins, la tranquillité ne peut pas être atteinte lorsqu'une entité participe à deux protocoles simultanément. Ceci implique la *famine* du processus d'adaptation qui ne peut accéder à l'entité qu'il doit modifier. Les auteurs préconisent dans ce cas-là l'utilisation de la condition de quiescence.

De Palma et al. [97] proposent une autre approche en considérant que l'adaptation est toujours possible. Ils utilisent pour cela un système transactionnel : une entité n'est alors impliquée que dans un protocole à la fois. Un protocole initié peut être annulé, si besoin est, en annulant tous les effets des messages échangés jusqu'à ce moment-là. Une adaptation est aussi effectuée de manière transactionnelle : ainsi, si une entité est impliquée dans un protocole au moment de l'adaptation, le protocole est annulé pour laisser la priorité à l'adaptation. Le protocole doit alors être recommencée après l'adaptation.

Les différentes approches décrites ont l'inconvénient de ne pas être transparentes pour le système : elles nécessitent la connaissance des protocoles en cours ainsi que les entités impliquées. De plus, elles peuvent conduire à une famine de l'adaptation ou des protocoles qui pourraient ainsi ne jamais être réalisés.

2.3.2 Le transfert d'état

L'adaptation d'une entité d'un système ne doit pas rompre la continuité des services assurés. Lorsque cette adaptation consiste au remplacement d'une entité par une autre (ou par une autre version d'elle-même), la nouvelle entité doit être dans un état qui permet cette continuité. L'opération d'adaptation doit donc assurer le « transfert d'état » entre les deux entités. Ce transfert revêt deux aspects :

1. le transfert de l'ensemble des valeurs des variables de l'entité ;

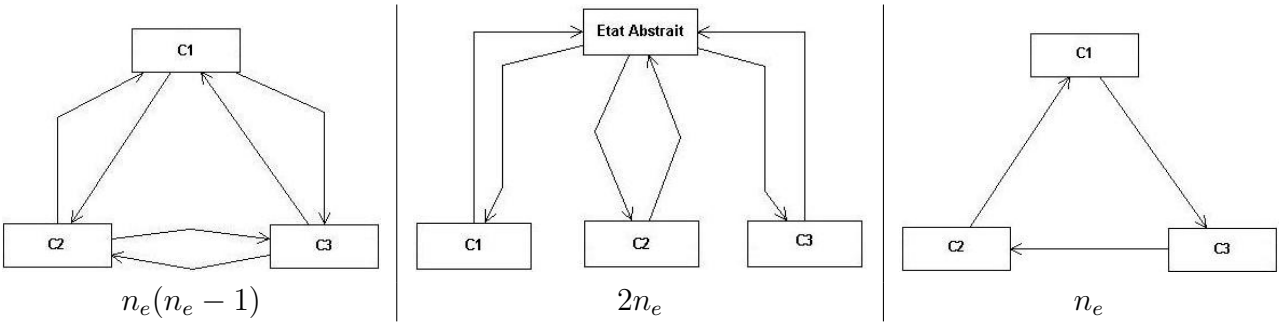

FIGURE 2.6 – Nombre d'opérations de transfert d'état entre trois composants

2. le transfert du point d'exécution atteint par le système.

Le transfert des valeurs est généralement réalisé par leur externalisation depuis l'ancienne entité suivie de leur injection dans la nouvelle entité à son instanciation (par ex. K-Component [98], SOFA/DCUP [99]). L'ancienne entité doit être inactive afin que ces valeurs n'évoluent pas au cours de l'adaptation et ainsi maintenir sa synchronisation entre l'ancienne et la nouvelle entité.

Les structures de données caractérisant les variables de la nouvelle entité ne sont pas obligatoirement identiques à celles de l'ancienne. Une opération additionnelle de conversion des anciennes valeurs pour qu'elles correspondent aux nouvelles structures de données est alors nécessaire. L'enjeu est de minimiser le nombre n_{op} d'opérations de conversion à écrire en fonction du nombre n_e d'entités se substituant les unes aux autres. L'approche générale consiste à écrire une opération pour chaque substitution tel que $n_{op} = n_e(n_e - 1)$. Par exemple, sur la figure 2.6, partie gauche, si trois composants (c_1,c_2,c_3) peuvent être utilisés indifféremment, il sera nécessaire d'écrire six opérations de conversion (c_1 vers c_2, c_1 vers c_3, c_2 vers c_1...).

Une alternative pour éviter ce problème d'explosion combinatoire consiste en l'abstraction des structures de données. Chaque entité réalise ces structures abstraites de la façon qui lui convient. Une seule opération de conversion est écrite par entité de sorte que $n_{op} = 2n_e$ (cf. figure 2.6, partie centrale). Cette opération est fonction des structures abstraites et non de l'implémentation de l'entité qui doit être remplacée. Par exemple, un composant implémentera une collection de données avec un vecteur tandis qu'un autre préfèrera une liste chainée [93].

Vandewoude et Berbers [100] proposent une approche hybride en utilisant une conversion explicite des valeurs d'une variable d'une entité à une autre. La complexité est réduite en établissant une chaine de conversion entre plusieurs entités : par transitivité, si c_2 sait convertir les valeurs de c_1 et c_3 sait convertir celles de c_2 alors c_3 sait convertir celles de c_1 ; de telle manière que $n_{op} = n_e$ (cf. figure 2.6, partie droite). Grondin et al. [101] approfondissent cette approche en établissant un graphe de transitivité pour relier les fonctions de conversion. Un nœud correspond à une variable et un arc à une fonction de conversion. Un arc a plusieurs nœuds sources et un nœud cible lorsqu'une fonction requiert les valeurs de plusieurs variables pour en calculer une autre. De façon identique, un arc a un nœud source et plusieurs nœuds cibles lorsque la valeur source doit être répartie sur plusieurs autres.

Dans toutes ces approches, le concepteur de l'entité a la charge de produire les fonctions de conversion entre les anciennes structures de données et les nouvelles. Même si certains outils sont développés pour l'assister, le manque d'information sémantique sur les variables obligent à toujours se référer au concepteur [102].

L'autre aspect concerne le transfert du point d'exécution. L'approche souvent adoptée consiste à n'autoriser l'adaptation que lorsque l'entité est inactive (comme lorsque une entité est quiescente [95, 101]). De ce fait, l'entité n'exécute aucune fonction : son remplacement ne necessite alors pas la désignation d'un point de reprise. Dans l'approche d'Hofmeister [93], le concepteur spécifie un point d'adaptation au sein de l'exécution d'une méthode d'un objet. Le code est alors instrumenté à la compilation afin de sauvegarder les variables de la pile d'exécution. Le nouvel objet reconstruit alors la pile d'exécution en ignorant les instructions modifiant les données. Cette approche nécessite une forte instrumentation du code (sur l'exemple donné dans [93], 19 lignes de code correspondent au code fonctionnel, 26 au code de l'adaptation).

2.3.3 La cohérence du système

L'adaptation correspond à la modification d'un système alors que ce dernier est en cours d'exécution. Cette adaptation n'a pas forcément été envisagée au moment de la conception du système et peut être introduite par la suite dans le système s'il est ouvert. Chaque point d'ouverture doit alors être soumis à un contrôle afin de préserver la cohérence du système. Moazami [103] définit un système comme étant cohérent avant et après son adaptation si :

1. le système respecte ses contraintes structurelles (cardinalité des liaisons, correspondance des interfaces requise et fournie...) ;
2. les entités du système sont dans des états mutuels consistants (deux entités sont dans des états mutuels consistants si leurs interactions font progresser ces états) ;
3. les invariants du système (conditions booléennes sur les propriétés du système) sont vrais.

Par exemple, Zhang et al. [104] garantissent la condition 1 dans des systèmes à base de composants. Les contraintes portent sur les relations de dépendance et sur les communications entre les composants du système. Une relation de dépendance peut préciser quels sont les composants qui sont couplés dans leur fonctionnement soit physiquement (les services requis par un composant doivent être obligatoirement fournis par un certain type de composant), soit logiquement (l'utilisation d'un composant dans le système n'a de sens qu'en utilisant un autre précis, par exemple un décodeur et un encodeur), ou bien préciser le nombre d'instances requis pour un type de composant.

L'approche classique pour assurer la condition 2 consiste à établir un système transactionnel qui garantit la quiescence lors de l'adaptation [58, 95, 97], comme décrit à la section précédente. Dans l'approche de De Palma et al. [97], le système est cohérent après une adaptation si l'adaptation ne modifie pas les résultats des services fournis par le système. Ils distinguent la gestion de la cohérence du système au niveau local, c.-à-d. au sein d'un composant, et au niveau global, c.-à-d. pour tous les composants impliqués dans la réalisation d'un service. Au niveau local, les sources d'incohérence relevées sont liées à des problèmes :

1. de référence : un composant n'est plus accessible car sa référence n'est plus valide ;
2. d'états : un composant n'est pas capable de poursuivre son fonctionnement et/ou donne des résultats faux ;
3. de canaux de communication : les messages en transit n'arrivent pas à leur destinataire.

Au niveau global, la source d'incohérence correspond à la cassure du lien « cause-effet » lors des interactions entre composants, c.-à-d. lorsque l'invocation d'un service fourni par un composant requiert que ce dernier invoque un service fourni par un autre composant, et ainsi de suite.

Kulkarni et Biyani [88] s'intéressent à la condition 3 en considérant aussi des systèmes à base de composants. L'état d'un composant est caractérisé par l'ensemble des valeurs de ses attributs. L'état est modifié après l'exécution d'une action. Un invariant global est défini en relation avec les attributs des composants du système. Le processus d'adaptation est une suite d'opérations atomiques. Chaque opération enlève ou rajoute des actions au système. Chaque opération est associée à un invariant en relation avec les attributs avant adaptation et ceux après adaptation. A l'issue de chaque opération, l'invariant associé doit être valide, c.-à-d. que les actions présentes dans le système doivent préserver l'invariant. Ainsi, le processus d'adaptation permet de passer par une suite de transitions d'un système valide à un autre.

En outre, Sibertin-Blanc et al. [105] considèrent le respect de contraintes comportementales. Ces dernières concernent les interactions entre les composants d'un système. Les interactions sont décrites sous la forme d'un protocole – une suite cohérente de messages échangés entre plusieurs composants – à l'aide d'un réseau de Petri. Chaque transition du réseau est étiquetée par une réception ou un envoi d'un message associé à un service d'un composant. Dans leur système, ces interactions sont gérées par un composant dédié, le modérateur, qui a un rôle d'intermédiaire dans les communications entre les autres composants, c.-à-d. les *participants*. L'adaptation consiste alors à ajouter une transition au réseau et à en inhiber d'autres. L'adaptation est alors transparente : pour un participant si ce dernier accepte la séquence de messages envoyée par le modérateur ; pour le modérateur, s'il accepte la séquence de messages envoyée par les participants. Les séquences acceptées par le modérateur avant l'adaptation sont les mêmes après l'adaptation. Les séquences envoyées par le modérateur après l'adaptation sont un sous-ensemble des séquences acceptées par les participants. Leur approche empêche ainsi le changement de l'interface des composants. Néanmoins, la vérification de leur condition de transparence est soumise au problème d'explosion d'états car elle nécessite d'inspecter tous les réseaux des participants [106].

Aucune de ces approches ne tient compte de l'évolution de la spécification du système. Or, nous avons vu dans la section 2.2.2 que l'adaptation est requise lors d'un changement dans la spécification. La validation peut donc se faire par rapport à cette nouvelle spécification plutôt que par rapport à l'ancien système, ce dernier n'étant plus valide par rapport à la nouvelle spécification (cf. figure 2.7).

2.4 Synthèse

Afin que les administrateurs soient libérés de tâches complexes et répétitives, les systèmes informatiques sont dotés de capacité d'auto-gestion. Cette idée est dévelopée dans le cadre de

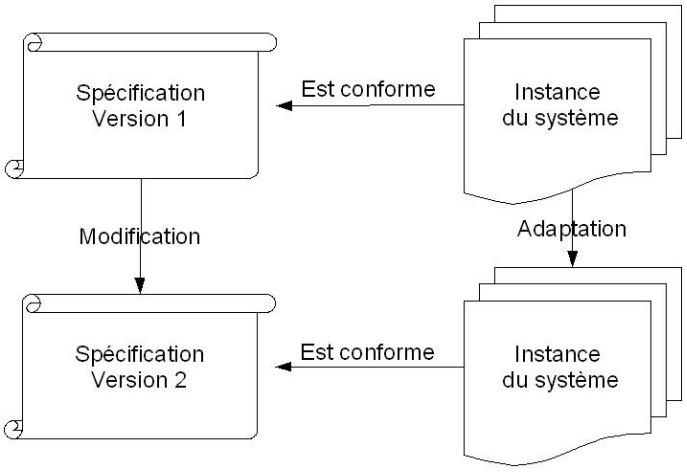

FIGURE 2.7 – Conformité du système après adaptation

l'« informatique autonomique ». Les systèmes autonomiques sont capables de s'auto-réparer, de s'auto-configurer, de s'auto-optimiser et de s'auto-protéger.

Ces quatres propriétés fondamentales impliquent que le système ait conscience de lui-même et qu'il soit capable de se modifier alors qu'il est en cours d'exécution. Ces deux caractéristiques sont celles d'un système adaptatif, un système capable de s'auto-adapter en réponse à des événements qui lui sont internes ou externes. Un système adaptatif est ouvert afin de supporter des adaptations non anticipées, notamment celles impliquées par l'évolution de ses spécifications.

Un système adaptatif opère dans un contexte dynamique : il doit supporter sa modification tout en minimisant l'indisponibilité de ses services. Il doit de plus garantir que les modifications appliquées ne le rendent pas incohérent. Il doit enfin permettre la continuité de ses services après son adaptation.

La réduction de l'« écart » entre la spécification et l'implémentation permet de minimiser les risques de divergence du système. Dans ce contexte, une technique comme l'exécution de modèles (cf. section 1.2.1) est intéressante car le système exécute directement les modèles issus de la spécification. Ainsi, il n'y a plus de raisons de mesurer la divergence puisque les problèmes décelés à l'exécution résultent dans ce cas directement d'une mauvaise spécification.

Dans le chapitre suivant, nous étudions les caractéristiques des systèmes adaptatifs à base de composants logiciels.

Chapitre 3

Auto-adaptation des systèmes à base de composants

Sommaire

En plus de leur vocation à être réutilisés, les composants logiciels présentent un atout pour structurer des systèmes de façon modulaire. Cette modularité combinée à des points d'interaction clairement définis est une base à la réalisation de systèmes flexibles. De ce fait, les composants logiciels sont actuellement le support privilégié pour réaliser des systèmes adaptatifs.

Nous présentons dans la section 3.1 les différents points d'adaptation possibles dans un système à base de composants ainsi que l'endroit où est localisé le support de l'adaptation. Nous développons dans la section 3.2 les techniques rendant possible l'adaptation de ces systèmes. Nous exposons dans la section 3.1 les différents lieux dans lesquels peuvent être localisés les mécanismes d'adaptation. Nous détaillons dans la section 3.3 des systèmes adaptatifs à base de composants que nous estimons représentatifs des approches actuelles. Nous terminons en mettant en évidence les faiblesses des approches actuelles dans la construction des systèmes adaptatifs à vocation autonomique.

3.1 Localisation de l'adaptation

3.1.1 Localisation des points d'application

L'adaptation d'un système à base de composants est possible à deux niveaux [93,97,107,108] :

- *au niveau structurel* : l'adaptation modifie l'architecture du système, c.-à-d. les composants et les liens qui les unissent. Ce type d'adaptation porte souvent le nom de « reconfiguration dynamique ». Elle se divise en deux catégories :
  - *l'adaptation structurelle logique* : l'adaptation s'applique au niveau des liaisons des composants. Un composant est supprimé/ajouté/remplacé dans le système. Le remplacement d'un composant nécessite la déconnexion des composants utilisant l'ancien composant et leur reconnexion au nouveau. Le nouveau composant doit assurer la continuité du fonctionnement ;
  - *l'adaptation structurelle physique* : l'adaptation s'applique sur un système distribué. Un composant est déplacé d'un environnement d'exécution à un autre. La mobilité d'un composant repose sur le même principe que le remplacement et nécessite aussi d'agir sur les liaisons du composant déplacé ;
- *au niveau comportemental* : l'adaptation porte sur le composant lui-même. Elle concerne :
  - *son implémentation* : les algorithmes ou les structures de données employés pour la réalisation d'un service sont remplacés ;
  - *ses interfaces* : des services sont ajoutés/supprimés/modifiés, la signature d'un service ou la séquence des services est modifiée.
  - *ses attributs* : les valeurs des attributs du composant sont modifiées (ce type d'adaptation porte le nom de *paramétrisation*).

L'adaptation comportementale est conditionnée par le niveau de visibilité des composants (cf. section 1.1.3.3) car elle nécessite l'accès à des éléments qui leur sont internes (c.-à-d. non exposés dans les interfaces des composants). En contrepartie, elle nécessite des changements moins lourds dans le système car elle n'affecte qu'un composant à la fois. De ce fait, elle tend à prendre effet plus rapidement et donc à accroître la réactivité du système face au besoin d'adaptation [107, 108]. Elle est particulièrement indiquée pour l'auto-optimisation et dans le cadre des systèmes embarqués.

La figure 3.1 illustre un scénario d'adaptation structurelle :

a) le composant A requiert des services réalisés par le composant B ;

b) le composant C est instancié et introduit dans le système dans le but de remplacer B ;

c) A est déconnecté de B puis reconnecté à C ;

d) B est retiré du système.

La figure 3.2 illustre un scénario d'adaptation comportementale :

a) le composant A requiert des services réalisés par le composant B ayant une implémentation $B1$;

b) la nouvelle implémentation $B2$ est instanciée et introduite dans le système dans le but de remplacer $B1$;

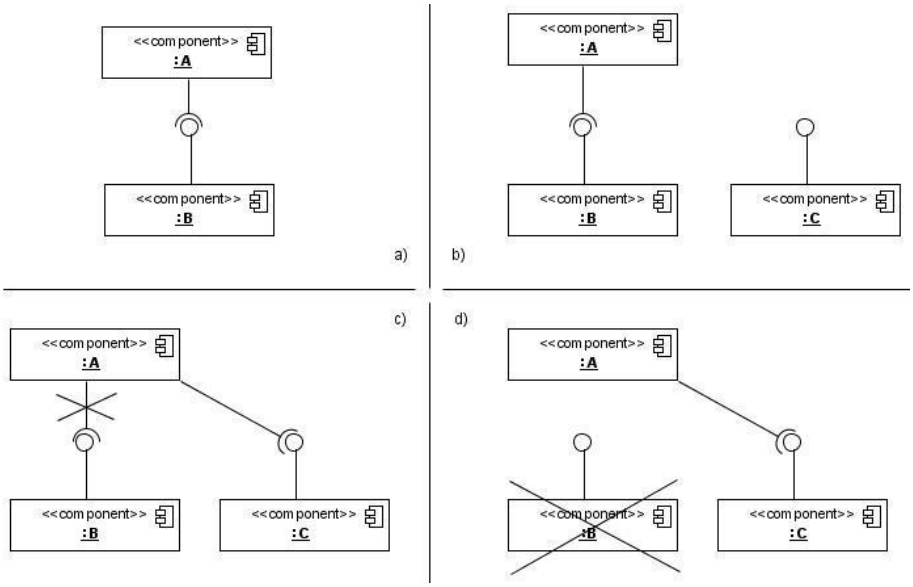

FIGURE 3.1 – Scénario d'adaptation structurelle

c) $B2$ est substituée à $B1$;

d) $B1$ est retirée du système.

3.1.2 Localisation du support de l'adaptation

Dans le respect du principe de séparation des préoccupations et afin d'en faciliter la modification, le code non-fonctionnel, c.-à-d. celui relatif à l'adaptation, ne doit pas être « mélangé » au code fonctionnel du système. Pour cela, trois endroits sont privilégiés dans les systèmes à base de composants pour accueillir le support de ce code non-fonctionnel.

3.1.2.1 Dans une infrastructure de composants

Le déploiement de systèmes sur des sites locaux et/ou distants nécessite de coordonner un ensemble d'opérations telles que l'envoi du code des composants sur les sites distants, l'instanciation, la paramétrisation, la liaison des instances. Ces opérations sont alors centralisées dans une infrastructure. L'infrastructure, de par la connaissance globale qu'elle possède du système, est un endroit naturel où implémenter la logique d'adaptation. Elle est alors à même de manipuler l'architecture et de modifier le comportement des composants.

Ainsi, Ayed et al. [109] accompagnent le modèle de composants CCM d'une infrastructure de déploiement ayant des capacités d'auto-configuration. L'infrastructure est ainsi capable de déployer une architecture particulière en fonction de l'environnement d'exécution. Parlavantzas et al. [110,111] proposent OpenORB, un intergiciel adaptatif conçu pour l'auto-optimisation de l'utilisation des ressources d'un système. OpenORB maintient une représentation de l'architec-

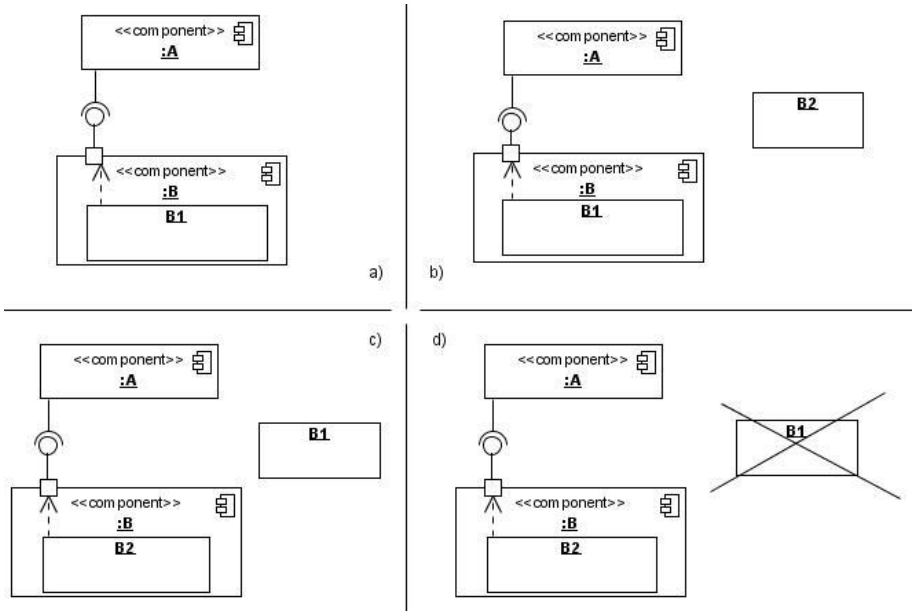

FIGURE 3.2 – Scénario d'adaptation comportementale

ture et des ressources du système déployé. Il permet par exemple de modifier l'algorithme de compression utilisé en fonction de la bande passante du réseau.

Localiser les mécanismes d'adaptation dans une infrastructure permet de contrôler l'adaptation de toute une architecture de manière centralisée. Le maintien de la cohérence du système est ainsi facilité. Néanmoins, les composants restent dépendants de l'infrastructure dans laquelle ils évoluent pour réaliser leur adaptation.

3.1.2.2 Dans un composant dédié

Un système à base de composants est un système modulaire dans lequel chaque composant réalise des services. Le système comprend un ensemble de composants respectant le même modèle de composants. Si les composants ont plutôt vocation à réaliser des services fonctionnels, rien n'empêche que certains soient dédiés à des aspects non-fonctionnels. De ce fait, un intergiciel comme CORBA a été implémenté à l'aide de composants [11] [112].

Certains composants sont ainsi dédiés à des services non-fonctionnels liés à l'adaptation. Par exemple, Phung-Khac et al. [36] s'intéressent aux systèmes distribués à base de composants. Un composant servant d'intermédiaire dans les communications entre les composants répartis sur plusieurs sites s'adaptent pour changer les algorithmes de répartition des données. De même, Sibertin et al. [105] rendent adaptable un modérateur, un composant implémentant un protocole de communication entre plusieurs composants. L'adaptation consiste alors à changer la manière dont les messages sont échangés entre les composants.

11. Les services non-fonctionnels du point de vue d'une application sont alors fonctionnels du point de vue de l'intergiciel.

PLASMA [76] est un modèle de composants basé sur le modèle Fractal et dédié à l'auto-adaptation d'applications multimédia. Le modèle comprend trois types de composants : les composants *Media*, les composants *Monitoring* et les composants *Reconfiguration*. Un composant *Reconfiguration* peut déclencher des opérations d'adaptation sur les trois types de composants en fonction d'événements envoyés par les composants *Monitoring*. Ces opérations modifient la valeur des attributs des composants. Ces attributs correspondent à des paramètres ou bien à des références vers d'autres composants. Un ADL (*Architecture Description Language*, cf. section 3.2.1) permet de spécifier les liaisons entre ces composants et de décrire les politiques d'adaptation des composants *Reconfiguration*.

Les approches rencontrées n'offrent pas de mécanismes génériques pour l'adaptation. En effet, les composants dédiés se concentrent sur l'adaptation de services particuliers. Le composant est alors réutilisable seulement dans les systèmes répondant aux mêmes préoccupations. Rien n'empêche d'imaginer un système qui comporte des *composants d'adaptation* responsables chacun de l'adaptation d'un sous-ensemble de composants. Néanmoins, l'approche par composant dédié ne permet pas d'envisager des composants fonctionnels « autonomes » car ces derniers ne disposent d'aucune capacité d'adaptation par eux-mêmes.

3.1.2.3 Dans un conteneur de composants

La notion de conteneur est surtout répandue dans les modèles de composants technologiques comme EJB, CCM et Fractal. Lorsqu'un composant est déployé, il est encapsulé dans un conteneur générique ou propre au type du composant. Le conteneur est ainsi à l'interface du composant et de l'infrastructure. Il est le support pour des services non-fonctionnels (sécurité, transaction, nommage...) qui s'appliquent au niveau local du composant plutôt qu'au système entier.

Le modèle de composants SOFA/DCUP [99] exploite les conteneurs pour rendre ses composants adaptables. Un composant SOFA/DCUP est séparé en une partie permanente et une partie remplaçable. Ses services sont répartis entre les services fonctionnels propres à chaque composant et les services de contrôle identiques entre les différents composants. Les services de contrôle assurent la quiescence du composant et le transfert d'état. Le conteneur assure la transparence et la cohérence de l'adaptation du point de vue de l'extérieur. Lors de l'adaptation, toutes les opérations du composant à adapter sont arrêtées, puis les valeurs de ses attributs sont externalisées. Le nouveau composant est ensuite instancié et les valeurs sont injectées.

Localiser les mécanismes d'adaptation dans un conteneur offre la possibilité de créer des conteneurs personnalisés en fonction du type de composant. De plus, l'adaptation est décentralisée en restant locale au composant. Chaque composant dispose ainsi de façon autonome de ses propres mécanismes d'adaptation. En contrepartie, le maintien de la cohérence du système dans le cadre d'adaptations globales, comme une reconfiguration d'architecture, est plus compliqué car il nécessite de nombreuses interactions pour coordonner les composants impliqués. De plus, la multiplication des conteneurs dans le système peut avoir un impact non négligeable sur la mémoire occupée par le système.

3.2 Techniques d'adaptation

L'adaptation d'un système nécessite d'y opérer des modifications. Ces modifications sont généralement permises par l'introduction d'un niveau d'indirection dans le code du système. Le point où il est introduit permet alors d'intercepter le flot d'exécution et ainsi de le rediriger [108, 113]. McKinley et al. [113] recensent de nombreuses techniques préparant un système à son adaptation. Nous présentons ici les plus courantes.

3.2.1 Adaptation par reconfiguration

Sous l'appellation « adaptation par reconfiguration », nous regroupons les techniques permettant de modifier l'architecture du système à l'exécution. Ces techniques reposent sur la réification de cette architecture, c.-à-d. la création d'entités décrivant l'architecture et permettant de la manipuler.

Les langages de description d'architecture (*Architecture Description Language*, ADL) [114] sont privilégiés pour ce type d'adaptation. A l'origine, ils servent à spécifier l'architecture d'un système, c.-à-d. les composants du système et les liaisons entre ces composants. Les opérations supportées par l'ADL dépendent du modèle de composants utilisé. Elles permettent au moins de déclarer et de connecter les composants entre eux ; si le modèle est hiérarchique, elles permettent aussi d'ajouter un composant à un composite.

Darwin [115] est un des ADL les plus connus. Un script Darwin permet de déclarer des instances de composants et les connexions entre l'interface de service requis d'une instance et l'interface de service fourni d'une autre. Le composant est ensuite instancié lorsque le service requis est invoqué. Le script permet aussi d'associer une interface requise d'une instance de composant à un type de composant. Chaque invocation de ce service crée alors une instance. Dans ce cas, l'interface requise de cette instance est associée à une interface fournie en conformité avec la façon dont la connexion est spécifiée dans le script. Le dynamisme fourni est minimal puisque limité à ce mécanisme d'instanciation « sur commande ». L'architecture est constante : les interfaces sont connectées dès la conception, et toute liaison est définitive. Seul le nombre d'instances des composants remplissant les interfaces requises évolue durant l'exécution du système.

Sur le même principe, Egyed et Wile [116] laissent une instance de composant directement instancier les composants dont elle a besoin. Le comportement d'un composant est décrit avec une machine à états. La machine à états déclenche la création d'un composant dont le type est contenu dans un attribut du composant. En modifiant la valeur de cet attribut, il est possible de faire instancier de nouveaux types de composants, ainsi que des versions corrigées du même type. La modification ne peut être appliquée qu'à de nouvelles instances et non sur les instances déjà créées. Leur approche permet aussi de remplacer des composants en déconnectant le consommateur d'un fournisseur obsolète puis en le reconnectant à un nouveau.

Les ADLs, utilisés au départ pour le déploiement du système, ont évolué vers une forme dynamique pour exprimer les reconfigurations (cf figure 3.1) [117]. Ces ADL dynamiques ont des capacités réflexives permettant « d'interroger » le modèle de l'architecture à propos des composants présents et à propos des connexions existants entre les composants. Des opérations

ont ainsi été ajoutées pour supprimer et déconnecter les composants.

ArchStudio [118] est un ADL permettant la reconfiguration de systèmes à base de composants C2 [119]. Un composant C2 possède un port d'entrée et un port de sortie. Les composants sont connectés en faisant correspondre leur port d'entrée et leur port de sortie. Ils communiquent de manière asynchrone en s'envoyant des messages, qui transitent alors par les ports. ArchStudio permet par exemple d'ajouter un composant au système, puis de connecter le port d'entrée de ce dernier au port de sortie d'un composant appartenant déjà à l'architecture et enfin de démarrer le composant nouvellement ajouté.

Les langages de description d'architecture sont utilisés pour des adaptations globales, c.-à-d. au niveau de l'architecture du système. Ils nécessitent donc une connaissance globale du système obligeant une adaptation centralisée. Les mécanismes les utilisant sont alors localisés dans une infrastructure de composants. Ces approches ne s'intéressent pas aux modifications de faible granularité, comme la modification de l'implémentation d'un composant. Ce type de modification passe alors systématiquement par la déconnexion d'un composant et la connexion de celui le remplaçant.

3.2.2 Patrons de conception pour l'adaptation

Les patrons de conception sont des réponses architecturales à des problèmes récurrents en conception orientée objet. Le Gang des Quatre [120] en a référencé plus de vingt.

Le patron *stratégie* (cf. figure 3.3) est intéressant car il permet de choisir le comportement d'une classe à l'exécution. Typiquement, il existe une classe `Contexte` qui détient des données sur lesquelles est appliqué un algorithme. L'algorithme est déclaré dans une interface `Strategie` et réalisé de différentes manières dans des classes contrètes (`StrategieConcreteA, StrategieConcreteB...`). La classe concrète est choisie à un moment de l'exécution du système et injectée dans le contexte à son instanciation. Ainsi, le modèle de composants auto-adaptatifs ACEEL [121, 122] utilise ce patron pour modifier l'implémentation de ses composants.

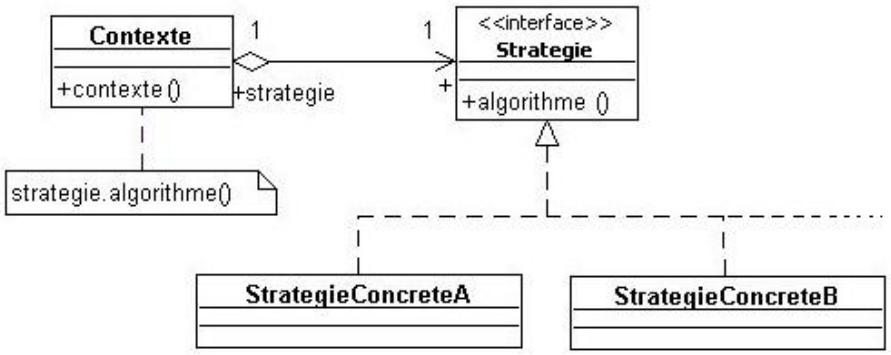

FIGURE 3.3 – Patron de conception Stratégie

Le patron de conception *adaptateur* (*Adapter/ Wrapper*) (cf. fig. 3.4) est aussi utile dans le cadre de l'adaptation car il permet de faire correspondre l'interface de service fourni d'un composant à celle de services requis d'un autre composant. Dans ce cas, l'adaptation consiste à une

mise en conformité de la syntaxe d'appel des opérations du service en faisant correspondre leur nom et leurs paramètres. Ainsi, sur la figure 3.4, l'implémentation de l'opération `requete()` de l'interface `InterfaceRequise` invoque l'opération `requetespecifique()` de l'interface `InterfaceFournie` (cf. note sur la figure 3.4).

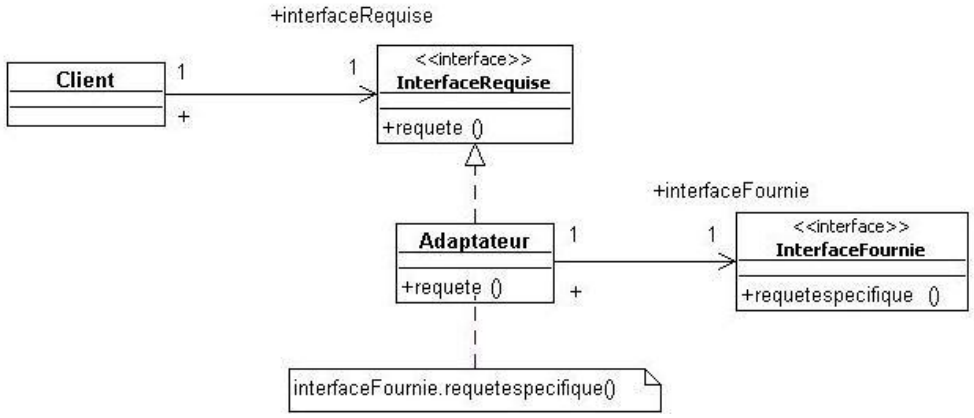

FIGURE 3.4 – Patron de conception Adaptateur

Kniesel [90] utilise ce patron pour remplacer l'implémentation des services des composants. Il propose notamment une approche récursive dans laquelle un adaptateur peut encapsuler un autre adaptateur, et ainsi de suite jusqu'au composant. L'opération implémentée dans l'adaptateur de plus haut niveau peut ainsi masquer la même opération implémentée dans un adaptateur de niveau inférieur ou bien peut rediriger ses appels vers ces adaptateurs.

Néanmoins, Boinot et al. [123] montrent les limites des patrons de conception en soulignant la surcharge qu'amène leur utilisation dans l'implémentation d'un système auto-adaptable.

3.2.3 Adaptation par modification de code

Au niveau le plus bas, l'adaptation d'un composant s'opère directement en modifiant le code exécuté.

La programmation orientée aspect (*Aspect-Oriented Programming*, AOP) [124] est une technique de programmation, indépendante de tout langage, qui permet de factoriser le code répondant à des problématiques transversales – les *aspects* – au sein d'un système. L'AOP permet ainsi de découpler les préoccupations fonctionnelles d'un composant des préoccupations non-fonctionnelles, comme la gestion des transactions et des exceptions. Les aspects déclarent des points de jonction dans les composants dans lesquels ils sont alors tissés (*weaving*). Si le tissage était initialement fait à la compilation, des techniques permettent maintenant un tissage à l'exécution [125,126]. L'adaptation, en tant que préoccupation commune à tous les composants d'un système, est ainsi une bonne canditate pour être gérée sous forme d'aspects [123,127–129].

Par ailleurs, les langages réflexifs sont aussi intéressants dans le cadre de l'adaptation. La réflexion implique à la fois l'introspection, c.-à-d. la découverte de la structure et du comportement d'un programme, et l'intercession, c.-à-d. la modification de la structure et du comporte-

ment d'un programme. L'introspection suppose d'avoir accès à une représentation de soi-même, conférant au programme une forme de « conscience de lui-même ». Par exemple, tout objet Java pointe vers sa représentation sous forme de classe et permet d'en inspecter le contenu. Il est ainsi possible de connaître les méthodes et les attributs disponibles. Ceci suppose qu'il y ait un métamodèle de classe clairement défini. L'intercession est la possibilité d'altérer la structure de la classe. Java ne le permet pas à l'inverse de langages comme Smalltalk [130], Python [131] ou encore Ruby [132]. Avec ces langages, il est possible d'écrire des méthodes qui rajoutent des méthodes à d'autres classes.

Les techniques de modification de code sont très puissantes dans la mesure où elles permettent potentiellement de modifier toutes les parties d'un système, et, par conséquent, sont aussi très intrusives. De ce fait, la vérification de la cohérence du système peut devenir très complexe si les points de modification ne sont pas connus à l'avance.

3.3 Etude de systèmes adaptatifs à base de composants

Nous présentons dans cette section des systèmes adaptatifs à base de composants. Chaque approche est différente et se projette dans le contexte de l'informatique autonomique (cf. tab. 3.4, page 65).

3.3.1 MaDcAr

3.3.1.1 Présentation

MaDcAr [4] est *« un modèle de moteurs d'assemblage »* permettant l'adaptation structurelle logique et l'adaptation de l'implémentation de systèmes à base de composants. Un moteur d'assemblage permet, à partir d'un ensemble de composants et d'une description de système, d'assembler un sous-ensemple de composants safisfaisant la description de ce système. Un moteur est constitué (cf. fig. 3.5) :

- d'un gestionnaire de contexte, qui comporte des capteurs pour sonder l'environnement de déploiement du système et le fonctionnement des composants ;
- d'une description d'application, qui spécifie des architectures valides pour le système considéré et un « réseau de transfert d'état » (cf. section 2.3.2) pour assurer le transfert d'état des composants ;
- d'une politique d'assemblage, à base de fonctions d'utilité, qui relie un état de l'environnement à une architecture.

MaDcAr se veut indépendant du modèle de composants utilisé pour l'implémentation. Pour cela, les architectures valides sont spécifiées avec des rôles – des composants dont l'implémentation n'est pas connue – connectés par leur interfaces fournis et requises. Chaque rôle porte des attributs fonctionnels et des contraintes techniques (par ex. des contraintes sur la mémoire, sur la bande passante...). Le modèle pour l'implémentation des composants est au choix du concepteur. Le réseau de transfert d'état établit la correspondance entre les attributs de chaque rôle dans les différentes architectures spécifiées.

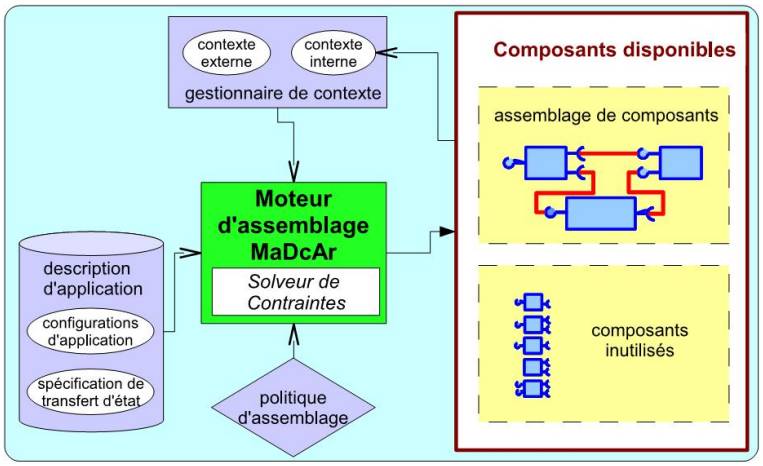

FIGURE 3.5 – Structure de MADCAR [4]

A partir des informations envoyées par le gestionnaire de contexte, le moteur choisit, en fonction de la politique d'assemblage, l'architecture à déployer parmi celles contenues dans la description d'application. Le moteur détermine ensuite, pour chaque rôle de l'architecture choisie et parmi les composants disponibles, ceux pouvant remplir les rôles. Un composant doit satisfaire les interfaces du rôle auquel il est affecté, doit disposer des mêmes attributs et doit satisfaire les contraintes de ce rôle. Le déploiement effectif consiste alors à connecter les composants choisis suivant les liaisons de l'architecture abstraite (c.-à-d. à base de rôles).

Une adaptation structurelle survient lorsque les conditions environnementales changent : une nouvelle architecture abstraite est alors choisie, puis de nouveaux composants sont déterminés pour la réaliser. Ensuite, les valeurs des attributs des composants de la nouvelle architecture sont calculées à partir du réseau de transfert d'état et des valeurs des attributs des composants de l'ancienne architecture.

3.3.1.2 Discussion

Le modèle est intéressant pour la manière dont il gère le transfert d'état : le réseau de transfert d'état relient des attributs à des fonctions de conversion. Ainsi, plusieurs attributs vont permettre de calculer une valeur d'un nouvel attribut, qui de manière transitive, va permettre le calcul de nouvelles valeurs.

De plus, la politique à base de fonctions d'utilité permet de décrire des « situations contextuelles » à partir de paramètres environnementaux et de déterminer l'intérêt de chaque architecture dans cette situation.

Néanmoins, l'approche comporte de sérieuses lacunes :

- l'adaptation structurelle, comme l'ajout d'un rôle, nécessite le redéploiement complet du système avec une nouvelle architecture intégrant le nouveau rôle. MADCAR oblige à un arrêt de l'application pendant cela et ne permet pas la reprise du point d'exécution ;
- l'adaptation comportementale, qui consiste au remplacement du composant réalisant un

rôle, est effectuée suivant une adaptation structurelle dans l'architecture de composants « concrets », suivant le principe de quiescence de Kramer et Magee (cf. section 2.3.1) ;
– des fonctions de correspondance entre l'architecture à base de rôles et celles à base de composants doivent être implémentées (fonctions pour manipuler l'architecture, pour éditer les propriétés, pour arrêter les composants...).

L'adaptation structurelle de MaDcAr ne correspond pas à notre définition de l'« adaptation dynamique » car elle nécessite l'arrêt global du système là où seul l'arrêt des composants affectés suffirait. De plus, lorsque deux architectures abstraites ont des composants concrets en commun, ces composants sont détruits pour être réinstanciés et réinitialisés avec les précédentes valeurs de leurs attributs. De plus, l'adaptation de l'implémentation se retrouve soumise aux mêmes contraintes que l'adaptation structurelle puisqu'elle correspond aussi à la connexion/-déconnexion de composants. L'approche est donc très lourde en termes de temps d'adaptation et nuit à la réactivité du système. Elle est de plus limitée à l'auto-configuration et à l'auto-optimisation du système déployé.

Avantages	Inconvénients
+ réseau de transfert d'état	- lourdeur des deux niveaux architecturaux
+ politique à base de fonctions d'utilité	- adaptation comportementale via une adaptation structurelle

Table 3.1 – Avantages et inconvénients principaux de MaDcAr

3.3.2 K-Component

3.3.2.1 Présentation

K-Component [5] est un modèle architectural pour construire des systèmes distribués auto-adaptables à base de composants logiciels. Il permet de coordonner les adaptations des composants de manière décentralisée et de réaliser l'auto-optimisation et l'auto-réparation du système. Un composant supporté par *K-Component* comprend une interface de service fourni et, facultativement, des interfaces de services requis. Il a différents états et expose des actions d'adaptation (cf. fig. 3.6). Des connecteurs relient les interfaces requises aux interfaces fournies.

Un *K-Component* correspond à un nœud d'un système distribué. Chaque *K-Component* dispose de sa propre boucle de contrôle comprenant un modèle de l'architecture locale de composants, un gestionnaire de configuration et un gestionnaire d'événements (cf. fig. 3.7). Le gestionnaire d'événements permet aux *K-Components* de communiquer par envoi d'événements, indépendamment des communications entre les composants qu'ils hébergent. Le modèle de l'architecture est un graphe des liaisons entre les composants au sein du *K-Component* ainsi que les liaisons des composants vers des composants appartenant à d'autres *K-Components*. L'architecture est découverte après le déploiement du système grâce à un ensemble d'actions d'introspection.

```
component <name> {
   provides <Interface>;
   [uses <Interface> <uname>;]*
   [state <sname>;]*
   [action <aname>;]*
};
```

FIGURE 3.6 – Le modèle de composants supporté par le modèle *K-Component* [5]

Le gestionnaire de configuration détient un ensemble de « contrats d'adaptation » qui s'exécutent en parallèle des composants. Un contrat implémente une politique d'adaptation à base d'actions. Il décrit les conditions de déclenchement des actions à partir des états des composants et des connecteurs du nœud auquel il appartient et à partir des événements envoyés par les autres nœuds. Les actions correspondent à celles implémentées par les composants et aux actions de reconfiguration de l'architecture. Elles permettent notamment de remplacer un composant dans l'architecture en agissant sur les connecteurs. Le gestionnaire s'assure alors de la quiescence d'un composant. Il vérifie que le composant n'effectue pas d'opérations et qu'aucune communication est en cours avec d'autres composants. Pendant l'adaptation, les connecteurs bloquent les communications. L'adaptation est validée par le système de typage qui s'assure qu'une interface requise est connectée à la bonne interface fournie.

Un contrat d'adaptation fait aussi appel à l'apprentissage par renforcement (*Reinforcement Learning*) [133] afin d'améliorer sa politique d'adaptation. Pour cela, la politique s'appuie sur un automate dont les états sont un sous-ensemble de ceux des composants. Les transitions sont étiquetées par les actions d'adaptation des composants. Chaque action renvoie une valeur scalaire après son exécution. Cette valeur indique le montant d'une « récompense » permettant de déterminer l'intérêt de l'action réalisée en fonction du contexte. L'augmentation du montant associé à une action fait que cette action sera privilégiée lors des prochaines adaptations survenant dans un contexte similaire. Une variante collaborative de l'apprentissage par renforcement permet de coordonner les gestionnaires des différents nœuds du système afin de réaliser une auto-optimisation globale.

3.3.2.2 Discussion

L'approche est intéressante car elle concerne des composants distribués et des adaptations décentralisées. La réification des connecteurs permet de détecter une déconnexion des composants autorisant ainsi l'auto-réparation de ces connexions. Ceci est un net avantage dans les environnements distribués. La logique adaptative est clairement séparée de la logique applicative. De plus la politique d'adaptation, en utilisant l'apprentissage par renforcement, évolue en améliorant les décisions prises.

En revanche, le fait que la boucle de contrôle ne soit pas explicite la rend statique : elle n'autorise pas l'introduction de nouveaux capteurs et effecteurs. De plus, les capteurs ne surveillent que les attributs du composant.

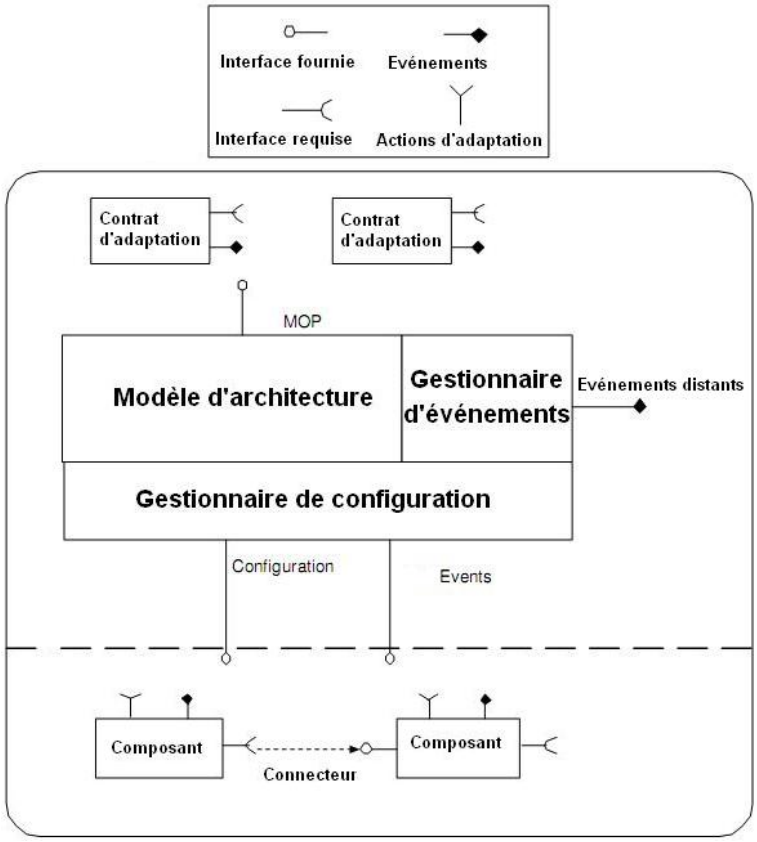

FIGURE 3.7 – Structure d'un *K-Component*

Le modèle *K-Component* n'instrumente que les adaptations structurelles du système même s'il permet de déclencher des actions d'adaptation propres à un composant. En effet, la spécification et l'implémentation de ces actions restent à la charge du concepteur et du développeur du composant sur lequel elles s'appliquent. Le fait que le modèle de l'architecture du système soit découvert après le déploiement empêche d'envisager l'auto-configuration du système (puisqu'il doit déjà être configuré pour activer la logique adaptative).

Par ailleurs, une politique d'auto-réparation se limite à la reconnexion de composants dont les dépendances ont été rompues à cause de problèmes dans l'infrastructure de communication. La politique d'apprentissage par renforcement devient optimale au fur et à mesure des adaptations, ce qui signifie que le système doit tolérer des adaptations hasardeuses à son initialisation. Enfin, une politique étant locale à un nœud et déclenchant des actions sur des composants distants, un processus circulaire d'adaptation peut se mettre en place : une action déclenchée sur un composant distant déclenche une action, directement ou indirectement, sur le composant qui a initié l'action depuis un autre nœud. L'approche ne précise pas comment un tel processus se termine.

Finalement, la gestion de la cohérence est minimale dans le processus d'adaptation. La notion d'état dans le composant est confuse : un état peut correspondre à une propriété scalaire comme

Avantages	Inconvénients
+ adaptation décentralisée	- se limite à l'adaptation structurelle
+ politique d'apprentissage par renforcement	- transfert d'état

TABLE 3.2 – Avantages et inconvénients principaux de *K-Component*

à un état activé/désactivé au sens d'une machine à états. De ce fait, elle est sous-exploitée dans la conception du composant. L'implémentation du transfert d'état reste donc à la charge du développeur.

3.3.3 SAFRAN

3.3.3.1 Description

SAFRAN [134] est une approche générique pour construire des systèmes adaptatifs à base de composants. Il supporte l'adaptation structurelle logique et l'adaptation comportementale via la paramétrisation et la redirection des invocations de services. Il repose sur le modèle de composants *Fractal* [46] dont il utilise les mécanismes de réflexion.

Un composant *Fractal* dispose d'une *membrane* et d'un *contenu* (cf. fig. 3.8, partie gauche). La membrane expose des interfaces listant des opérations. Une interface est de type *serveur* si elle fournit des opérations, de type *client* si elle requiert des opérations. La membrane expose aussi des interfaces de contrôle pour, par exemple, agir sur le cycle de vie du composant ou sur ses attributs, pour introspecter le composant et aussi pour intercepter les invocations d'opérations. *Fractal* supporte la composition verticale : le contenu du composant peut encapsuler d'autres composants (cf. fig. 3.8, partie droite). Il supporte aussi le partage d'un composant entre plusieurs composites.

SAFRAN étend *Fractal* en ajoutant de nouvelles interfaces de contrôle aux composants :

- une interface permettant l'ajout d'un « méta-composant » destiné à intercepter les invocations de services et à les rediriger. Ce méta-composant est lui-même un composant *Fractal* qui peut donc être composite et qui peut aussi disposer d'un méta-composant ;
- une interface permettant d'attacher un ensemble de contraintes architecturales à un composant ;
- une interface permettant d'attacher à chaque composant une politique d'adaptation. Cette politique ne peut modifier que le composant auquel elle est attachée. Etant donné que ce composant peut très bien être un composite disposant de sous-composants partagés, la politique peut donc en fait affecter tous les composants du système.

Un composant SAFRAN est d'abord conçu comme un composant *Fractal*, c.-à-d. en ne se focalisant que sur sa logique fonctionnelle. Il est ensuite déclaré adaptatif si nécessaire au moment de son déploiement. Le composant supporte l'ajout de politiques d'adaptation à l'exécution. De plus, le caractère ouvert d'un système basé sur SAFRAN permet de déployer de nouvelles sondes et d'introduire de nouveaux composants dans le système.

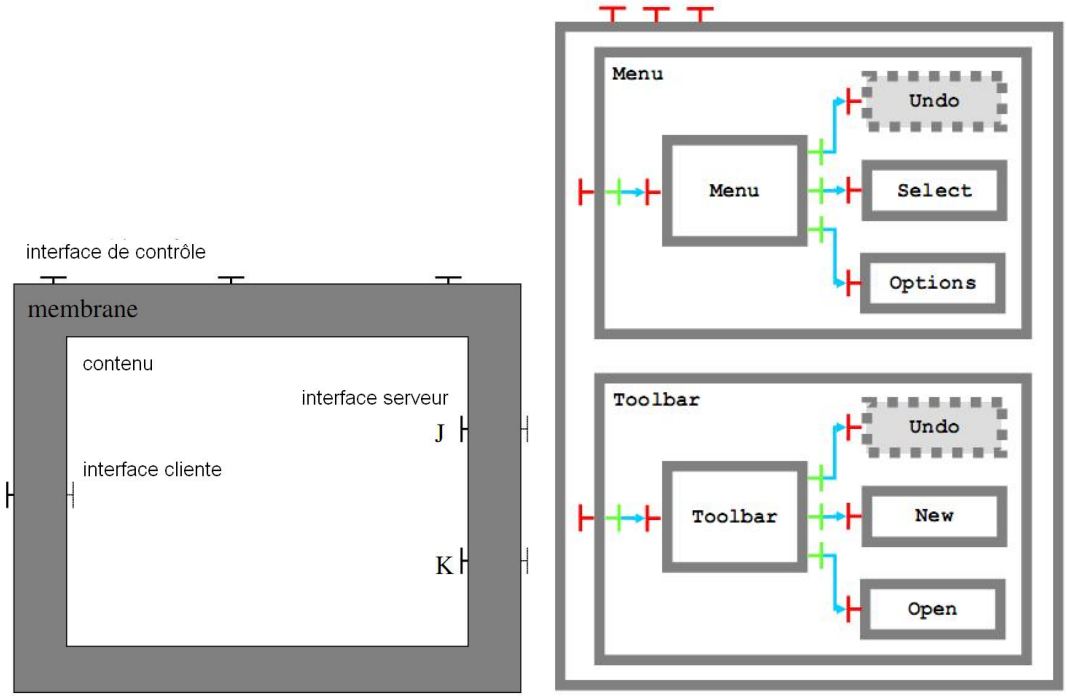

FIGURE 3.8 – Exemple de composants Fractal [6]

Les politiques dans SAFRAN sont à base de règles *événement-condition-action*. Les événements utilisent les informations envoyées par des capteurs indépendants du système (ces capteurs sont fournis par la bibliothèque *WildCat*). Une condition est spécifiée avec le langage dédié *FPath*, qui permet entre autres de naviguer dans l'architecture du système. Les actions sont spécifiées avec le langage *FScript*, permettant de manipuler les composants (modification des valeurs d'attributs, ajout de méta-composants...) et l'architecture (connexion/déconnexion de composants, ajout de sous-composants...).

SAFRAN garantit une adaptation en s'assurant de l'atomicité et de la consistance de l'adaptation : si une des actions échoue ou si les contraintes architecturales ne sont pas validées après la réalisation de l'adaptation, l'architecture précédente est restaurée. De plus, SAFRAN arrête un composant avant d'en remanier les connexions. Les messages arrivant alors au composant sont sauvegardés pour lui être délivrés à son redémarrage. Enfin, SAFRAN garantit qu'il n'arrive qu'une seule adaptation à la fois.

3.3.3.2 Discussion

SAFRAN est un modèle riche de par les langages dédiés à chaque aspect de l'auto-adaptation (politiques d'adaptation, actions de reconfiguration et contraintes architecturales). Par exemple, le langage de définition des politiques autorise une règle à attendre l'occurrence simultanée d'événements, des séquences particulières d'événements, des temps d'attente ou bien encore des absences d'événements pendant un temps donné. Néanmoins, si la définition d'un langage

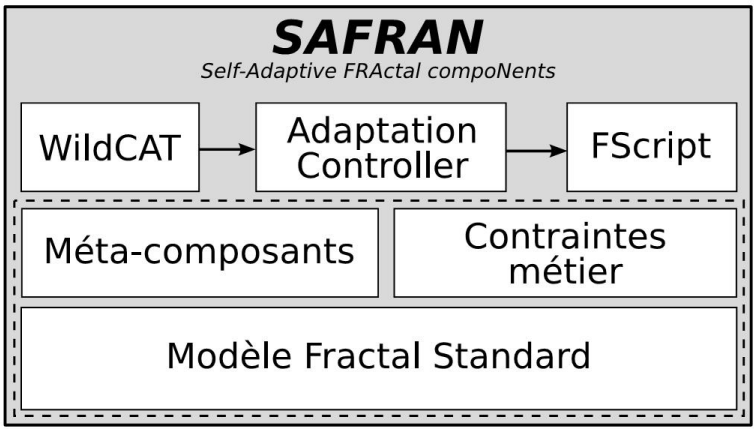

FIGURE 3.9 – Structure de SAFRAN

permet de mieux cibler chaque aspect, la multiplication des langages nuit à la prise en main à cause du temps nécessaire à la maîtrise de chacun de ces langages. De plus, si le langage *FPath* est nécessaire à SAFRAN pour exprimer les contraintes architecturales du système, il est redondant avec un langage plus générique comme OCL.

SAFRAN tient compte de la cohérence des adaptations grâce à un mode transactionnel. L'annulation de l'adaptation consiste alors à exécuter les opérations inverses de celles effectuées. Ceci nécessite donc que pour chaque opération $op \in Op$, il existe une opération op^{-1} telle que $op \circ op^{-1} = id$. De ce fait, les adaptations possibles sont limitées par la combinaison des opérations contenues dans l'ensemble Op. De plus, exécuter op^{-1} est une nouvelle source possible d'erreurs et entraîne un temps de latence dépendant du nombre d'opérations à effectuer.

Par ailleurs, SAFRAN souffre de problèmes inhérents à *Fractal*. Bien que les adaptations soient locales à un composant, il n'y a pas d'exécution parallèle des adaptations du fait de la partageabilité des composants. De plus, *Fractal* ne permet pas l'adaptation comportementale de façon native. De ce fait, le méta-composant défini pour chaque composant doit le permettre. Ce méta-composant est un autre composant *Fractal* réinterprétant les opérations de services. Il rajoute de la complexité et de la confusion en introduisant un niveau de « composition » ambigu. Enfin, les contraintes architecturales ne sont présentes qu'à titre informatif : leur violation ne peut pas être exploitée car *« les* APIs *de Fractal ne sont pas conçues pour notifier ce genre d'erreur »* [134].

Avantages	Inconvénients
+ richesse des langages	- trois langages différents
+ gestion de la cohérence	- pas de transfert d'état
	- notion de méta-composant

TABLE 3.3 – Avantages et inconvénients principaux de SAFRAN

Modèles	Système	Raisons				Adaptation					Autonomie				Gestion				Technique	Localisation
		Correction	Environnement	Perfectionnement	Extension	Logique	Physique	Implémentation	Interface	Attributs	Auto-configuration	Auto-réparation	Auto-optimisation	Auto-protection	Quiescence	Transfert des valeurs	Transfert du point d'exécution	Cohérence		
MaDcAr [4]	Fermé		✓	✓		✓		✓		✓	✓		✓		✓	✓			Réflexion	Infrastructure
K-Component [5]	Fermé			✓		✓						✓	✓		✓			✓	Réflexion	Infrastructure
SAFRAN [134]	Ouvert		✓			✓		✓		✓			✓		?			✓	Réflexion/AOP	Infrastructure / Conteneur

✓ : la caractéristique est supportée.
? : la caractéristique est indéterminée.

TABLE 3.4 – Caractéristiques des systèmes auto-adaptatifs étudiés

3.4 Synthèse

Les approches actuelles se focalisent sur l'adaptation structurelle logique via le remaniement des entités réifiant l'architecture d'un système. Elles ne considèrent l'adaptation comportementale qu'en permettant la modification de la valeur des attributs des composants. Aucune approche ne s'intéresse aux protocoles des composants, c.-à-d. aux séquences des opérations de services, ni ne permet l'extension des services fournis par un composant. Ainsi, le changement d'une implémentation nécessite le remplacement d'un composant entier. Ce remplacement possède des inconvénients qui nuisent à la transparence de l'adaptation :

- du point de vue des consommateurs des services du composant remplacé : ce dernier doit être arrêté et les consommateurs doivent être mis en attente ;
- du point de vue du développeur : ce dernier doit implémenter des opérations de transfert d'état ;
- du point de vue du système : son temps de réaction est augmenté par les nombreuses étapes (déconnexion, instanciation, reconnexion, transfert d'état...) nécessaires au remplacement d'un composant, alors que le remplacement d'une implémentation, en appliquant le patron de conception *stratégie*, ne nécessite qu'un remplacement de référence.

Par ailleurs, la logique d'adaptation est centralisée dans une infrastructure pour les composants et s'appuie sur une connaissance globale du système. Si cette solution facilite la gestion de la cohérence globale du système, elle ne favorise ni la concurrence des adaptations ni l'autonomie des composants, qui restent alors dépendants de l'infrastructure dans laquelle ils sont déployés. Même les propositions de décentralisation comme celle de *K-Component* pour les systèmes distribués, continuent à s'appuyer sur l'infrastructure et sur l'architecture globale du nœud.

Enfin, aucune approche n'est tournée vers l'utilisabilité des systèmes adaptatifs. Le cas de SAFRAN en est symptomatique : un langage textuel est nécessaire pour chaque aspect de l'adaptation (gestion des capteurs, déclaration de la politique, déclaration des remaniements d'architecture). Le manque d'uniformité de l'approche nuit ainsi à la prise en main du système. L'administrateur de ce dernier a alors tendance à être écarté du processus de contrôle alors qu'il doit rester capable de superviser le système. Il doit pouvoir délivrer, en plus des politiques d'adaptation, des mises à jour aux composants et pouvoir introduire de nouveaux composants dans le système.

En conclusion, il n'existe pas à notre connaissance d'approche uniforme à la réalisation des systèmes adaptatifs à base de composants, considérant les évolutions de la spécification d'un système, satisfaisant les besoins généraux de l'informatique autonomique et permettant à un administrateur de garder la main sur le système. La partie suivante présente notre contribution à la réalisation des systèmes adaptatifs afin d'améliorer ces différents points.

Deuxième partie

MOCAS : un modèle de composants basé états pour l'auto-adaptation

Introduction

Le modèle de composants MOCAS permet de construire des systèmes auto-adaptables à base de composants logiciels. MOCAS a vocation à faciliter l'intégration des propriétés autonomiques au sein de tels systèmes. Pour cela, MOCAS est :

- *générique* : tous les types d'application sont susceptibles de concerner MOCAS ;
- *uniforme* : l'auto-adaptation repose sur les capacités initiales du modèle de composants MOCAS ;
- *réflexif* : la structure et le comportement d'un composant MOCAS sont découverts et modifiés à l'exécution.

Un système logiciel MOCAS est ouvert afin d'être enrichi à son exécution par de nouvelles politiques d'adaptation et par de nouveaux composants intervenant dans l'adaptation. De plus, un système MOCAS supporte la livraison de mise à jour à ses composants.

Nous avons conçu MOCAS en trois étapes distinctes :

1. lors de la première, nous avons défini le modèle afin qu'il soit préparé pour l'adaptation. Ce modèle trouve son origine dans le modèle de composants PauWare et le modèle de composants UML. Il autorise la conception de composants fonctionnels génériques ;
2. lors de la seconde, nous avons défini les mécanismes d'adaptation des composants. Nous avons pour cela spécifier un conteneur de composants. Ce conteneur respecte lui-même le modèle MOCAS. Il utilise la réflexivité pour contrôler et réaliser le processus d'adaptation du composant qu'il contient ;
3. lors de la troisième, nous avons d'abord construit une boucle de contrôle à l'aide de composants MOCAS afin de rendre un composant auto-adaptable. Les mécanismes d'adaptation permettent notamment de changer dynamiquement et de façon cohérente la politique qu'un composant embarque. Nous avons ensuite défini des politiques d'auto-configuration et d'auto-réparation locales à chaque composant puis nous avons établi un protocole d'interaction pour coordonner une adaptation impliquant plusieurs composants.

Nous présentons respectivement les résultats de ces étapes au chapitre 4, au chapitre 5 et au chapitre 6.

Chapitre 4

Le modèle de composants MOCAS

Sommaire

Les approches pour construire des systèmes adaptatifs se sont jusqu'alors concentrées sur la réification de l'architecture de ces systèmes. Si cette réification donne conscience à un système des composants qui le constituent, elle ne suffit pas à couvrir toutes les formes d'adaptation dont ce système a besoin. Afin de couvrir l'adaptation comportementale jusqu'alors marginalisée, mais aussi de minimiser le temps d'indisponibilité d'un système pendant son adaptation et de rendre transparent les mécanismes d'adaptation pour un concepteur de composants, nous proposons le modèle de composants MOCAS. Pour cela, MOCAS centre l'adaptation sur le composant logiciel et le comportement plutôt que sur l'architecture (la structure) des systèmes. MOCAS s'appuie largement sur l'ingénierie des modèles et la métamodélisation. Il utilise le langage UML, standard de facto largement outillé et utilisé dans l'industrie. Il étend les principes de conception établis avec le modèle de composants PauWare (conception basée états). Alors que ce dernier n'a pas de métamodèle explicite, MOCAS respecte le modèle de composants UML et est formalisé avec les éléments natifs du langage.

Nous exposons à la section 4.1 la structure des composants MOCAS : nous montrons comment elle est instanciée avec les méta-élements UML, puis nous introduisons la terminologie propre à MOCAS en l'incorporant dans un profil. Nous détaillons à la section 4.2 les principes

d'interaction et de composition de MOCAS nécessaires à un « modèle de composants ». Enfin, nous illustrons notre approche à la section 4.3 avec la spécification d'un premier composant MOCAS, le composant `Car`.

4.1 Structure d'un composant MOCAS

Le modèle de composants MOCAS résulte de la spécification du modèle PauWare avec le modèle de composants UML, augmenté d'une séparation distincte entre les attributs, le comportement et la réalisation du composant et d'un mode de communication asynchrone. Il a été conçu afin de faciliter l'adaptation de ses composants en considérant dès le départ les éléments devant être adaptables.

4.1.1 Métamodèle UML

UML est un langage très riche dont le métamodèle dispose de nombreuses associations vers des éléments abstraits. Un concepteur est alors libre de choisir dans ce métamodèle les éléments concrets qui les réalisent. Nous y avons recherché les associations et éléments existants permettant de respecter le principe de spécification du comportement des composants avec des machines à états tout en limitant le couplage, d'une part, entre le comportement et l'implémentation et d'autre part, entre les composants eux-mêmes. La figure 4.1 présente le métamodèle de MOCAS à partir des éléments et des associations existants dans UML.

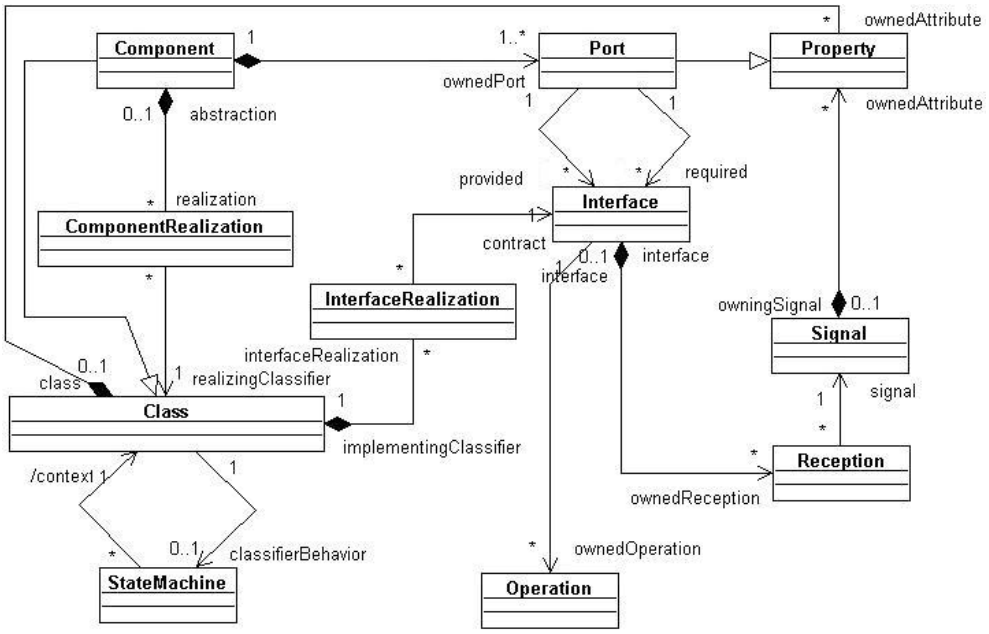

FIGURE 4.1 – Métamodèle de MOCAS

Un composant MOCAS est ainsi un composant UML (`Component`) dont le classifieur de réalisation (association `realizingClassifier`) est une classe (`Class`) qui a son comportement (association `classifierBehavior`) exprimé avec une machine à états (`StateMachine`). La classe de réalisation du composant implémente une interface (association `contract`) listant des opérations (association `ownedOperation`) correspondant à des actions internes au composant (`ownedOperation`). Ce dernier possède des attributs (association `ownedAttribute`) en rapport avec ses fonctionnalités et des ports (association `ownedPort`) pour communiquer.

La communication entre les composants MOCAS se fait par échange de signaux. Un signal dispose d'attributs (association `ownedAttribute`) et est spécialisable par d'autres signaux. Les signaux sont listés dans une interface (association `ownedReception`) associée au port (associations `required` ou `provided`) par lequel ces signaux transitent.

MOCAS introduit ici une distinction sémantique par rapport à UML. UML considère que les signaux listés dans une interface de service requis sont des signaux que le composant émet vers l'environnement et que les signaux listés dans une interface de service fourni sont des signaux que le composant reçoit de l'environnement. Dans MOCAS, un signal d'une interface de service requis peut très bien être un signal que le composant reçoit de l'environnement. De même, un signal d'une interface de service fourni peut très bien être un signal que le composant émet vers l'environnement. De ce fait, nous distinguons les ports d'entrée, sur lesquels un composant MOCAS reçoit des signaux, des ports de sortie, par lesquels un composant MOCAS émet des signaux. Un port supporte des communications bidirectionnelles et peut donc être à la fois d'entrée et de sortie.

Lorsqu'un composant MOCAS fournit (respectivement requiert) un service par un port, ce dernier n'est relié à des interfaces que par l'association `provided` (respectivement `required`). Le service est caractérisé par les signaux listés dans les interfaces associées au port. Chacune de ces interfaces peut lister un ensemble de signaux entrant ou sortant par le port. Un composant MOCAS possède au moins un port par lequel il fournit un service. Rien n'empêche que ce port soit un port de sortie [12] (p. ex. un service d'horloge qui émet un signal `Top` toutes les x secondes).

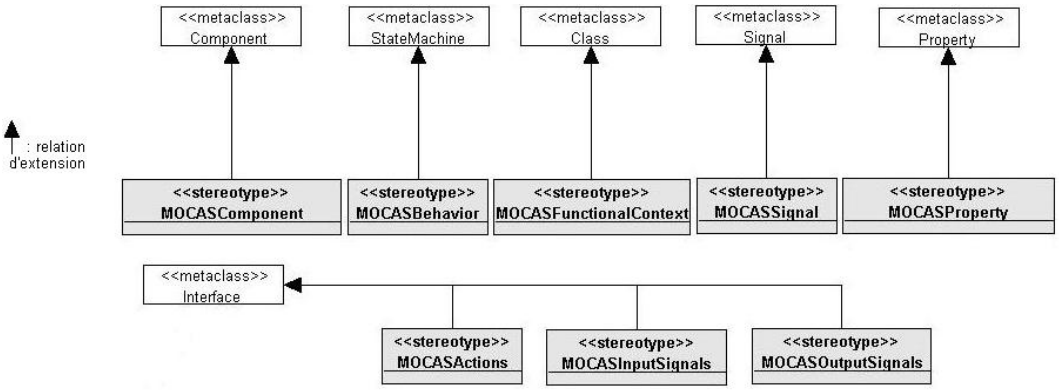

FIGURE 4.2 – Profil de MOCAS

12. Par conséquent, rien n'empêche un port d'entrée de requérir un service.

La figure 4.2 présente le profil UML de MOCAS. Ce profil reprend les métaclasses UML et les spécialisent avec des stéréotypes propres à la terminologie de MOCAS :

- un composant MOCAS est un composant UML stéréotypé `MOCASComponent` ;
- le comportement du composant est une machine à états stéréotypée `MOCASBehavior` ;
- la classe de réalisation est appelé le contexte fonctionnel du composant et est stéréotypée `MOCASFunctionalContext` ;
- l'interface d'actions internes est stéréotypée `MOCASActions` ;
- un port d'entrée est lié à une interface stéréotypée `MOCASInputSignals` ;
- un port de sortie est lié à une interface stéréotypée `MOCASOutputSignals` ;
- un signal destiné à un composant MOCAS est stéréotypé `MOCASSignal` ;
- un attribut du composant est stéréotypé `MOCASProperty`.

4.1.2 Les attributs du composant

Un composant MOCAS possède des attributs caractéristiques des services qu'il assure (par ex. la durée de l'intervalle entre deux `Top` pour un service d'horloge). Un attribut est une propriété UML. Il est décrit par un nom, un type et une visibilité. La visibilité de l'attribut conditionne les possibilités qu'a un autre composant de lire ou de modifier la valeur de cet attribut (cf. tab. 4.1). Ainsi, un composant ne pourra consulter et modifier la valeur d'un attribut que si ce dernier a une visibilité de type lecture/écriture. Les valeurs des attributs accessibles en lecture sont véhiculées entre les composants par les signaux transmis.

Lecture/Ecriture	Lecture	Ecriture	Aucun accès

TABLE 4.1 – Visibilité des attributs MOCAS

Un port est une propriété UML (cf. lien d'héritage entre `Port` et `Property` sur la figure 4.1) et est aussi considéré comme un attribut du composant. Son type correspond à un composant MOCAS. Son nom correspond au service fourni ou requis par ce port. Un port de sortie visible en écriture permet la liaison de ce port, un port de sortie visible en lecture permet l'émission d'un signal par ce port.

Nous utilisons les actions qu'UML fournit (cf. figure 4.3) pour gérer les valeurs d'une « caractéristique structurelle » (*StructuralFeature*) dont hérite une propriété UML :

- l'action `AddStructuralFeatureValueAction(object, structuralFeature, value)` ajoute la valeur `value` à la caractéristique structurelle `structuralFeature` détenue par l'objet `object` ;
- l'action `RemoveStructuralFeatureValueAction(object, structuralFeature, value)` retire la valeur `value` de la caractéristique structurelle `structuralFeature` détenue par l'objet `object` ;
- l'action `ReadStructuralFeatureValueAction(object, structuralFeature)` lit la valeur de la caractéristique structurelle `structuralFeature` détenue par l'objet `object` ;

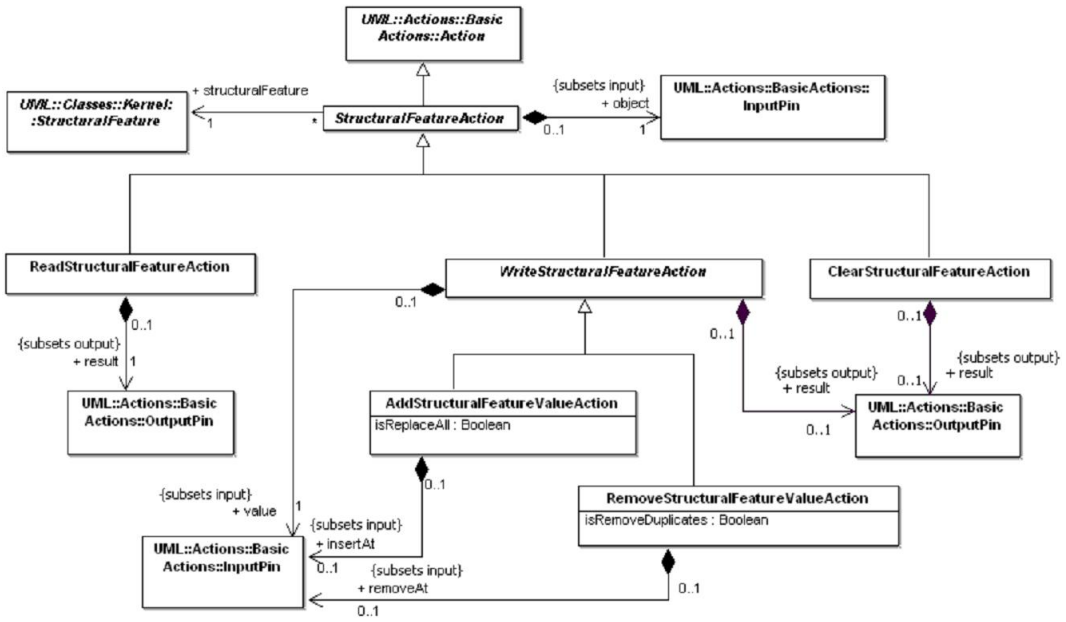

FIGURE 4.3 – Actions UML pour gérer les propriétés [1]

– l'action `ClearStructuralFeatureValueAction(object, structuralFeature)` [13] efface la valeur de la caractéristique structurelle `structuralFeature` détenue par l'objet `object`.

4.1.3 Le comportement du composant

A l'instar de PauWare, le comportement d'un composant MOCAS est décrit avec une machine à états. La machine à états définit le *protocole* du composant en contraignant l'interprétation de la séquence des signaux qu'il reçoit. A la réception d'un signal, la machine déclenche des actions internes (cf. fig. 4.4, `CallOperationAction`), émet des signaux sur les ports de sortie du composant vers les composants qui lui sont liés (cf. fig. 4.4, `SendSignalAction` et `BroadcastSignalAction`) ou crée des objets (cf. fig. 4.4, `CreateObjectAction`). Elle utilise les attributs pour définir ses contraintes, c.-à-d. les gardes de ses transitions et les invariants de ses états.

Le concepteur du composant a la responsabilité de spécifier des états significatifs du point de vue des services du composant. Un état peut représenter la valeur d'un attribut discrétisé ou bien une étape dans le protocole.

Par ailleurs, une machine à états d'un composant MOCAS répond au moins aux quatre signaux suivant :

– `GetPropertyValue` qui permet à un composant de consulter, si la visibilité l'y autorise, la valeur d'un attribut d'un autre composant ;

13. Par souci de concision, ces actions seront respectivement notées `Add`, `Remove`, `Read`, `Clear` sur les transitions des machines à états.

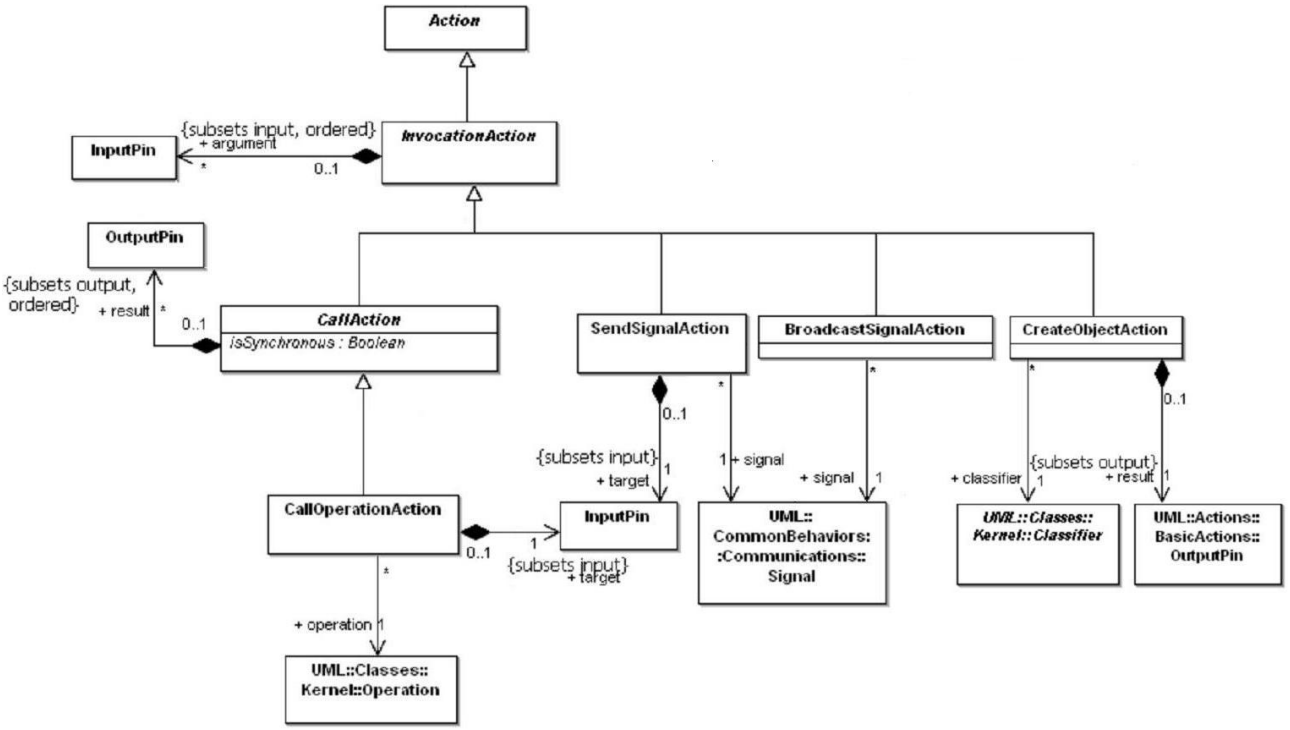

FIGURE 4.4 – Actions UML pour gérer les opérations, les signaux et les objets [1]

- `ReturnPropertyValue` qui permet au composant à qui la valeur a été demandée de répondre ;
- `SetPropertyValue` qui permet de modifier, si la visibilité l'autorise, la valeur d'un attribut d'un composant ;
- `UnsetPropertyValue` [14] qui permet de supprimer, si la visibilité l'autorise, la valeur d'un attribut d'un composant.

Ces signaux héritent du signal abstrait `PropertyValue(receiver, attribute, value, reply_to)` où `receiver` est le composant détenant l'attribut, `attribute` est l'attribut concerné, `value` est la valeur de l'attribut en cas d'écriture, de suppression ou de réponse et `reply_to` est un port d'entrée sur lequel retourner la valeur.

La figure 4.5 présente la machine à état minimale d'un composant MOCAS. Cette machine déclenche toujours à son initialisation l'action interne `initMOCASProperties()` qui initialise les attributs du composant, puis entre dans l'état composite `Component Behavior` qui décrit le comportement du composant. Cet état rend disponible la lecture, l'écriture et la suppression des valeurs des attributs en fonction de la visibilité grâce aux actions UML (cf. section 4.1.2) associées aux transitions internes [15].

14. Par souci de concision, ces signaux seront respectivement notés `Get`, `Return`, `Set`, `Unset` sur les transitions des machines à états.

15. Par souci de concision, ces transitions internes ne seront pas répétées dans les prochaines machines à états. Si elles sont redéfinies dans des sous-états de `Component Behavior`, les conditions de la garde seront implicites.

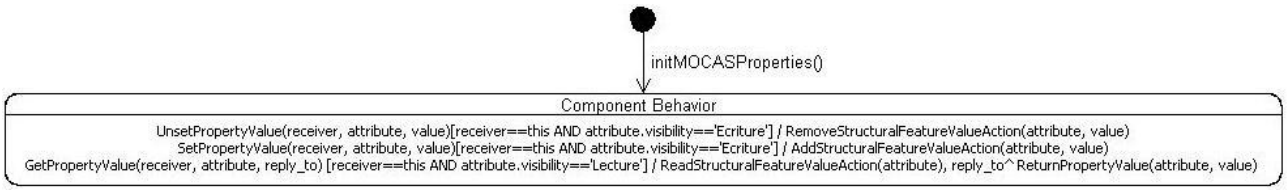

FIGURE 4.5 – Machine à états minimale d'un composant MOCAS

La machine à états a de plus un rôle de documentation dans la description du comportement et des services du composant. Elle permet la compréhension du fonctionnement du composant sans pour autant révéler l'implémentation propre à un savoir-faire. Un composant MOCAS offre donc une visibilité boîte grise afin de pouvoir consulter la machine (cf. section 1.1.3.3).

4.1.4 Le contexte fonctionnel du composant

Le contexte fonctionnel est une classe réalisant la logique fonctionnelle du composant en implémentant les actions internes de ce dernier. Les actions correspondent à des calculs et des transformations de données à partir des attributs. Ces attributs sont ceux du composant ou ceux du signal déclenchant l'action interne. Le contexte fonctionnel déclare explicitement les attributs qu'il utilise.

Les actions sont déclarées dans l'interface d'actions. Cette dernière comporte au moins l'action `initMOCASProperties()` qui initialise les attributs à l'instanciation du composant. L'interface est utilisée par la machine à états pour connaître les actions à déclencher dans le contexte fonctionnel. Elle découple ainsi le comportement et l'implémentation du composant suivant le patron de conception *stratégie* (cf. section 3.2.2).

4.2 Interaction des composants MOCAS

4.2.1 Composition horizontale

La composition horizontale des composants MOCAS se réalise à leur déploiement, c.-à-d. avant que les machines à états des composants ne soient activées. Elle est identique au mécanisme de composition des composants UML dans les collaborations : elle correspond à la mise en relation d'un port de sortie d'une instance d'un composant avec un port d'entrée d'une autre instance via un connecteur d'assemblage. Si un port de sortie est multivalué, il supporte la connection de plusieurs ports d'entrée. Les signaux émis par ce port sont alors diffusés vers tous les ports d'entrée connectés lorsque l'action `BroadcastSignalAction` est utilisée. Un port d'entrée peut être connecté à plusieurs ports de sortie : les signaux reçus sont mis dans une file d'attente unique au composant, quel que soit le port de sortie qui les a transmis.

La figure 4.6 illustre ce principe de composition : le port de sortie `gear` de l'instance du composant `Car` est connecté au port d'entrée `gear` de l'instance du composant `GearBox` ; le port

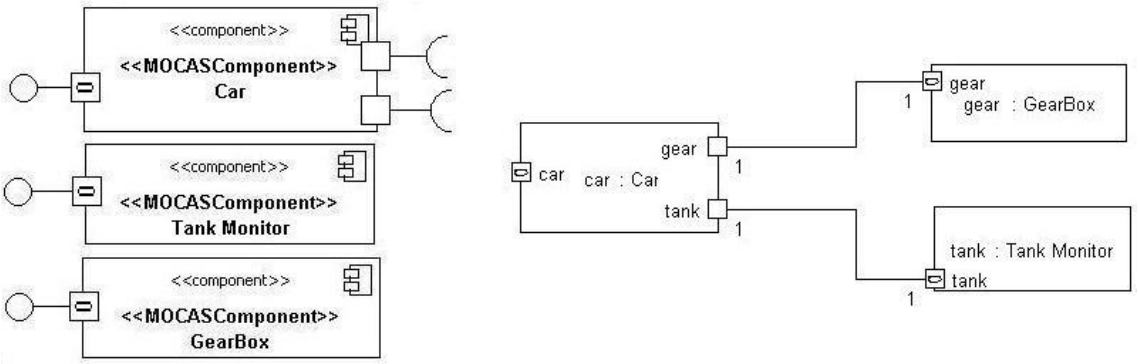

FIGURE 4.6 – Composition horizontale dans MOCAS

de sortie `tank` de l'instance du composant `Car` est connecté au port d'entrée `tank` de l'instance du composant `Tank Monitor`.

Les ports d'un composant étant aussi des attributs de ce composant, leur connexion peut être réalisée à l'exécution via le signal `SetPropertyValue`. Cependant, tant qu'un port de service requis n'est pas connecté, le composant n'est pas capable de fournir les services dépendant de ce service requis. Des mécanismes de liaisons tardives comme l'instanciation paresseuse dans Darwin [115] peuvent être une solution. Ce problème est une préoccupation de l'adaptation structurelle et est donc en dehors de notre problématique de recherche.

Toutefois, la machine à états peut être conçue de manière à apporter un début de réponse. Elle peut s'assurer que le port de service requis, dont dépend un service pour pouvoir être fourni, est bien connecté avant de requérir un service par ce port. Ainsi, sur la figure 4.7, l'état `Configuration` attend la liaison des ports. L'expression OCL `ownedPort->forAll(p|...)` teste si tous les ports requis du composants sont liés. Le déclencheur `when` permet de franchir la transition vers l'état `Stopped` dès que l'expression est vraie. De même, la garde de la transition entre l'état `Stopped` et l'état `Forward` s'assure que le port `gear` est lié avant d'autoriser l'envoi d'un signal par ce port. Ce type d'expression permet alors de gérer des liaisons incertaines entre des composants distribués.

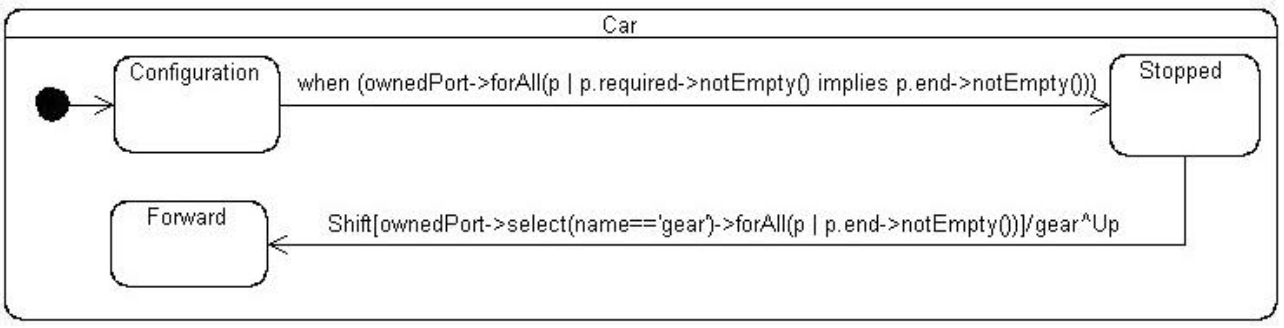

FIGURE 4.7 – Vérification de la résolution des dépendances

La composition horizontale à la conception n'a de sens que lorsqu'elle s'effectue pour former un composite. Les sous-composants constituent alors une collaboration dans le cadre de leur composite. De ce fait, un composite offre un cadre d'instanciation pour composer horizontalement ses sous-composants.

4.2.2 Composition verticale

MOCAS supporte la composition comportementale verticale de manière étendue par rapport à PauWare. Ce dernier ne supporte que la composition via l'ajout d'états orthogonaux au comportement du composite (cf. section 1.3.2). MOCAS autorise la composition depuis n'importe quel état simple, exlusif ou orthogonal, de la machine à états du composite. L'état simple désigné est alors raffiné par la machine du sous-composant et devient un état de sous-machine. Le sous-composant est intégré de façon transparente dans le composite, c.-à-d. que la composition n'altère ni la structure ni le comportement du sous-composant. Chaque composant conserve notamment son propre contexte fonctionnel comme classe d'implémentation de ses actions internes. Une fois composé, le composite est capable de réagir à tous les signaux auxquels ses sous-composants réagissent lorsque les états de sous-machine sont actifs. La figure 4.8 montre la composition verticale du composant `Car` avec les composants `GearBox` et `Tank Monitor` : `Tank Monitor` est composé de la même manière que dans le modèle PauWare alors que `GearBox` utilise le principe de MOCAS.

Cette technique de composition nécessite de garantir que deux états exlusifs, dont le premier appartient au composite et donc le second appartient au sous-composant, ne soient pas actifs simultanément. A la conception et au déploiement, cette condition est toujours vraie car les machines à états des composants composés sont inactives. En revanche, à l'exécution, les machines possédant des états actifs, cette condition doit être vérifiée pour autoriser la composition. La machine à états minimale d'un composant (cf. fig. 4.5) comprend donc des transitions internes supplémentaires (cf. fig. 4.9) afin de gérer la cohérence de la composition à l'éxécution.

Le signal `ComposeMOCASComponent(state, mocasComponent)` permet de composer le composant `mocasComponent` avec le composant destinataire du signal au niveau de l'état `state`. La composition n'est alors possible que sous certaines configurations d'états actifs :

- si le composant `mocasComponent` est inactif (sa machine à états n'a pas été démarrée) tout comme l'état `state` du composant destinaire ou bien, si les deux sont actifs, alors la composition est possible : l'action `compose(...)` transforme alors l'état `state` en état de sous-machine avec la machine à états du composant `mocasComponent` ;
- si le composant `mocasComponent` est actif (un état quelconque de sa machine est actif) et que l'état `state` du composant destinaire est inactif, alors la composition n'est pas possible. En effet, ce cas conduirait à l'activation simultanée de deux états exclusifs ;
- si le composant `mocasComponent` est inactif et que l'état `state` du composant destinaire est actif, alors la composition est possible mais le composant `mocasComponent` doit être démarré (cf. fig. 4.9, action `start()`).

Par ailleurs, cette technique soulève la question de la partageabilité du sous-composant. Si le sous-composant est partagé :

- un autre composant peut lui envoyer directement un signal. Le sous-composant rentre

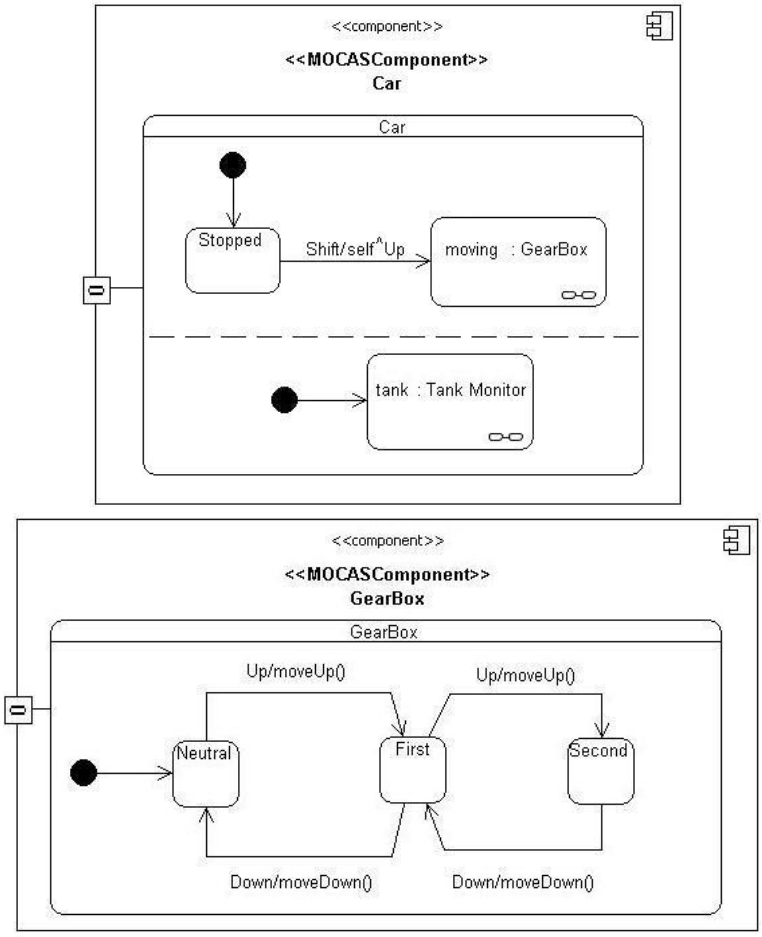

FIGURE 4.8 – Composition verticale dans MOCAS

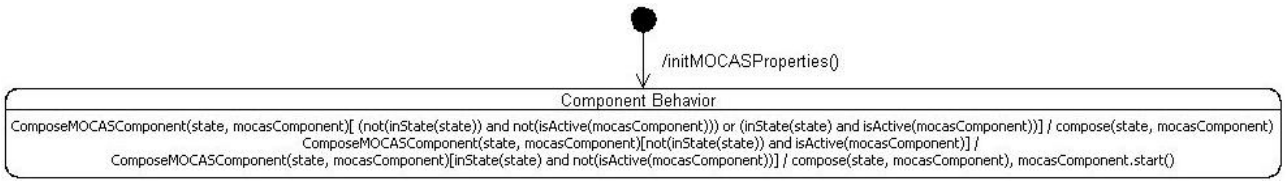

FIGURE 4.9 – Composition verticale dynamique dans MOCAS

alors dans un cycle *run-to-completion* pour traiter ce signal. Le composé qui le détient peut aussi recevoir un signal le faisant lui-même initier son propre cycle. La machine à états du composé se retrouve alors à exécuter deux cycles *run-to-completion*, ce qui est incohérent avec le mode de fonctionnement des machines à états ;
- les machines à états du composé et du sous-composant ne peuvent avoir des états exclusifs simultanément actifs. A partir du moment où l'état de sous-machine du composé devient inactif, la machine à états du sous-composant est désactivée et n'est donc plus capable de traiter les signaux directement reçus. Du point de vue du client du sous-composant, le fait que la machine soit désactivée signifie que le composant est « hors-ligne » ;

Pour ces raisons, nous n'autorisons pas le partage des sous-composants. Le composite est alors le seul à pouvoir accéder à ses sous-composants. Les ports d'entrée des sous-composants ne sont donc pas accessibles pour les composants extérieurs au composé. Les clients connectés à ces ports d'entrée sont alors reconnectés au port d'entrée du composite. Le composé gère une unique file d'événements et un unique cycle *run-to-completion* dans lequel les machines à états de ses sous-composants font partie intégrante.

Cette composition comportementale reste compatible avec la composition structurelle d'UML. En effet, un composite peut avoir un connecteur de délégation (cf. section 1.5) reliant son port d'entrée au port d'entrée d'un de ses sous-composants, conservant ainsi des machines à états parfaitement indépendantes. Cette compatibilité nous est utile lors de la définition des composants MOCAS auto-adaptables (cf. chapitre 6).

4.2.3 Communication entre composants

Chaque composant MOCAS dispose de son propre flot d'exécution et communique de façon asynchrone avec les composants qui lui sont liés. La communication se fait par envoi de signaux et est régie par la machine à états du composant. Les envois sont spécifiés sur les effets des transitions ou les actions des états avec les éléments UML `SendSignalAction` et `BroadcastSignalAction` (cf. fig. 4.4). UML ne précise pas le type des éléments destinataires des signaux. Dans MOCAS, une action `SendSignalAction` envoie une instance du signal associé (association `signal`) par un port de sortie désigné comme cible (association `target`). Une action `BroadcastSignalAction` diffuse une instance de signal vers plusieurs destinataires. UML précise que la manière de déterminer les destinataires est un « point de variation sémantique ». Dans MOCAS, il s'agit de tous les composants dont le port d'entrée est connecté au port de sortie (multivalué) désigné comme cible.

Lors de l'envoi d'une instance de signal, les valeurs des attributs de l'instance sont données par celles des attributs du composant envoyant le signal et par celles des attributs du signal reçu ayant déclenché l'envoi. Si le signal reçu possède un attribut identique à celui du composant et dont la valeur est différente, la valeur portée par le composant sera celle envoyée. Une opération de synchronisation de la valeur de l'attribut du composant avec celle de l'attribut du signal peut être spécifiée avant l'envoi.

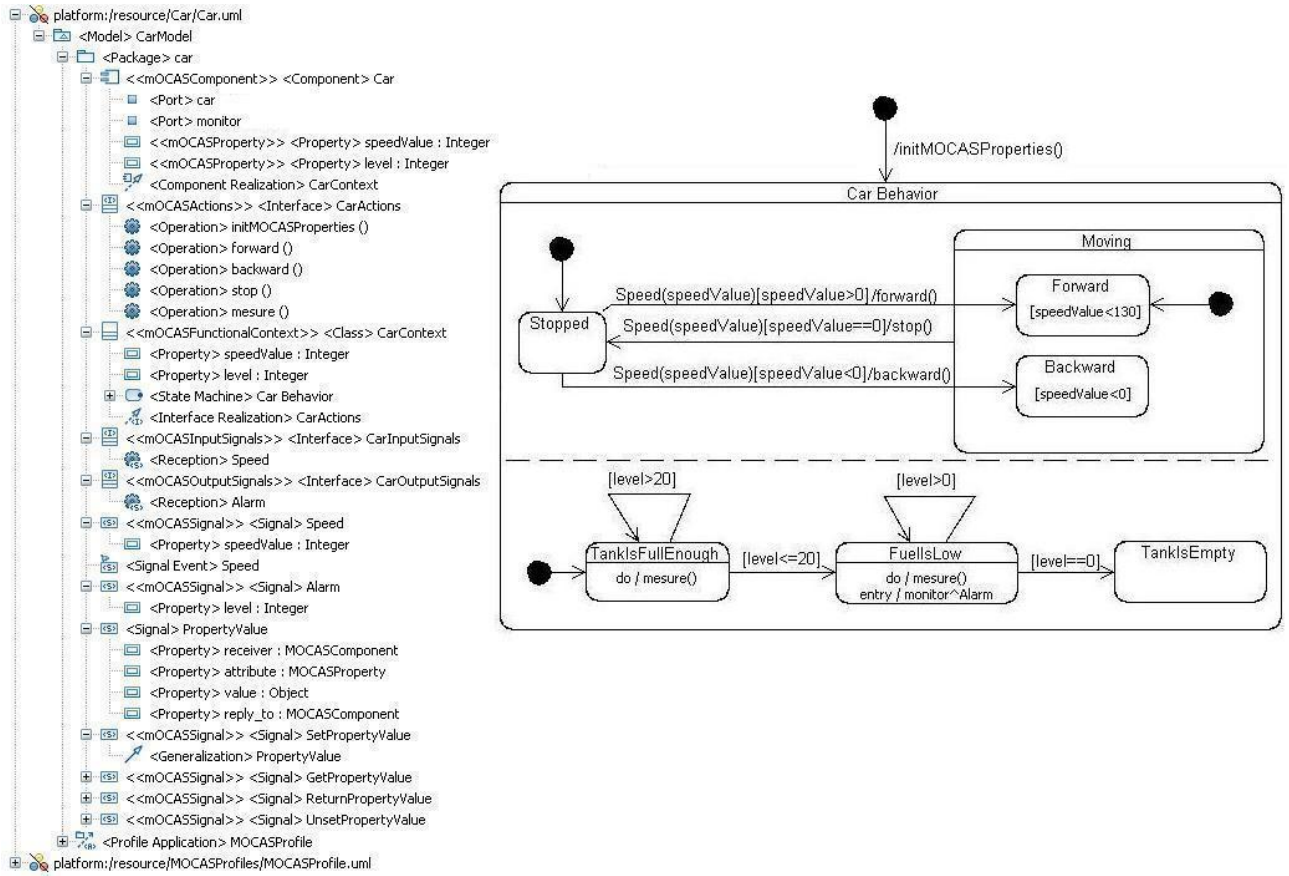

FIGURE 4.10 – Structure et comportement du composant `Car`

4.3 Exemple du composant `Car`

Nous présentons ici un premier composant MOCAS nommé `Car` qui sera repris au cours des chapitres suivants pour illustrer les spécificités de MOCAS. Dans ce chapitre, la spécificité de MOCAS tient dans la manière de structurer les composants en vue de faciliter leur adaptation. La figure 4.10 montre la spécification structurelle de ce composant et utilise les stéréotypes du profil de MOCAS. Le comportement du composant est celui décrit par la machine à états de la figure 1.8 (p. 26) augmentée des transitions par défaut pour la gestion des attributs (cf. fig. 4.5, p. 77) et la gestion de la composition verticale dynamique (cf. fig. 4.9).

Le composant `Car` possède un port d'entrée nommé `car` sur lequel il peut recevoir le signal `Speed` et les signaux affectés à la lecture/écriture des attributs. Il possède un port de sortie nommé `monitor` sur lequel il émet le signal `Alarm`. Il dispose de l'attribut `speedValue` qui correspond à la vitesse courante de la voiture et de l'attribut `level`, accessible seulement en lecture, qui correspond au niveau du réservoir. Son interface d'actions `CarActions` spécifie cinq actions internes (`initMOCASProperties()`, `forward()`, `backward()...`) implémentées par la classe `CarContext`. Son comportement est exprimé avec la machine à états `Car Behavior`.

Composition \ Moment	Conception	Déploiement	Exécution
horizontale	✓ (dans un composite)	✓	✓ (ad hoc)
verticale	✓	✓	✓

TABLE 4.2 – Support de la composition dans MOCAS

A son instanciation, le composant initialise ses attributs (action interne `initMOCASProperties()`) puis entre dans les états `Stopped` et `TankIsFullEnough`. Lors de la réception d'une instance du signal `Speed`, si la valeur de l'attribut `speedValue` est supérieure à 0, la machine déclenche l'action `forward()` puis active les états `Moving` et `Forward`. L'état `Forward` a un invariant qui précise que tant que la voiture se déplace, sa vitesse ne doit pas dépasser $130km/h$. En parallèle, lors de l'activation de l'état `TankIsFullEnough`, l'activité `mesure()` est déclenchée. Cette activité met à jour la valeur de l'attribut `level` après avoir mesuré le niveau du réservoir. Le composant connecté au port `car` peut relever à tout instant la valeur de l'attribut `level` en envoyant le signal `GetPropertyValue`. Lors de l'entrée dans l'état `FuelIsLow`, le signal `Alarm` est envoyé au composant dont le port d'entrée est connecté au port `monitor`.

4.4 Synthèse

Le modèle de composants MOCAS pose les bases d'un modèle pour l'adaptation à travers un ensemble de caractéristiques :

- la communication asynchrone par signaux permet à un composant de ne pas attendre le résultat d'un service qu'il a requis pour être prêt à satisfaire une requête d'un service qu'il fournit. La machine à états d'un composant permet de modéliser des points de synchronisation uniquement lorsqu'ils sont nécessaires. Un composant gagne ainsi en réactivité et en parallélisme de traitements ;
- la réification de la structure d'un composant avec le métamodèle UML permet d'avoir une structure manipulable avec des éléments « standardisés » ;
- la réification du comportement d'un composant avec une machine à états dont le métamodèle est connu permet d'avoir un comportement manipulable au niveau des états actifs et au niveau de la structure de la machine. Il devient dès lors possible de changer l'interprétation des invocations de services ;
- la séparation entre les attributs, le comportement et l'implémentation permet d'agir distinctement sur chacun de ces éléments.

Le tableau 4.2 résume les moments de composition supportés par MOCAS. Le moment privilégié reste le déploiement. La composition horizontale à l'exécution nécessite que la machine soit conçue de manière à gérer l'indisponibilité des services requis. La composition verticale à l'exécution est possible grâce à la définition d'un « service de composition ». Ce dernier s'assure

alors de la cohérence des machines composés.

Nous allons voir au chapitre suivant comment toutes ses caractéristiques, notamment le service de composition, sont exploitées pour concevoir des composants adaptables dynamiquement.

Chapitre 5

Adaptation des composants MOCAS

Sommaire

L'adaptation nécessite de minimiser le temps d'indisponibilité du composant adapté, de préserver son état et de permettre la continuité de son fonctionnement après l'adaptation. De plus, les mécanismes d'adaptation doivent être transparents pour le concepteur d'un composant. En effet, le concepteur ne doit fournir « aucun effort » pour rendre un composant adaptable, il peut alors se concentrer pleinement sur les services fonctionnels du composant.

Nous avons présenté au chapitre précédent le modèle de composants MOCAS, conçu spécialement pour faciliter l'adaptation des composants respectant ce modèle. Nous exposons à la section 5.1 de ce chapitre comment ce modèle est exploité via un conteneur de composants. Nous détaillons à la section 5.2 les opérations d'adaptation rendues possibles par la réification du comportement et de la structure des composants MOCAS. Nous reprenons à la section 5.3 l'exemple du composant `Car` du chapitre précédent afin d'illustrer les mécanismes d'adaptation.

5.1 Un conteneur pour l'adaptation

La plupart des approches à l'adaptation proposent des mécanismes centralisés dans une infrastructure d'accueil pour les composants (cf. section 3.1). Nous proposons une approche

décentralisée où chaque composant dispose de ses mécanismes d'adaptation et ne nécessite pas d'infrastructure. De plus nous voulons assurer la transparence des mécanismes d'adaptation pour le concepteur en ne mélangeant pas les aspects fonctionnels des aspects non fonctionnels comme l'adaptation. L'approche par conteneur réalise tous ces objectifs.

5.1.1 Structure du conteneur

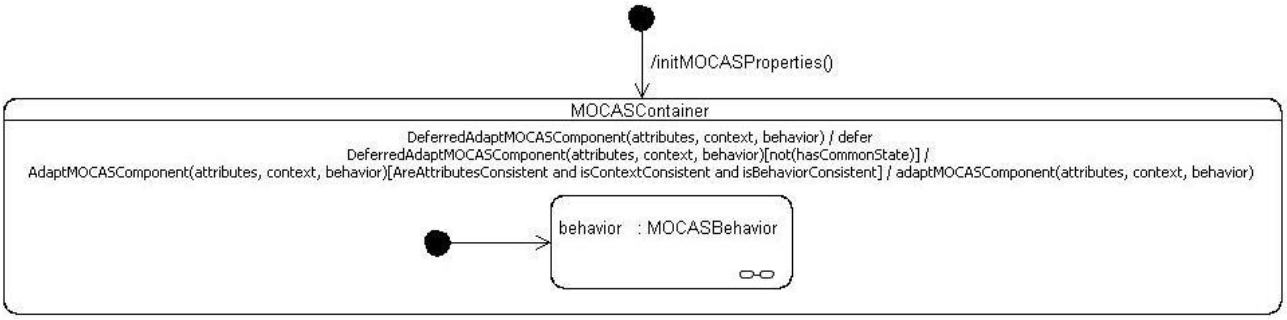

FIGURE 5.1 – Comportement du conteneur MOCAS

Le conteneur de composants MOCAS [135] est un composant MOCAS composite. La figure 5.1 montre sa machine à états. Cette dernière contient l'état de sous-machine `MOCASBehavior` destiné à accueillir la machine à états du composant à rendre adaptable, suivant le principe de composition comportementale verticale de MOCAS. Elle contient aussi l'état composite `MOCASContainer` qui détient la transition interne spécifiant le service d'adaptation. Cette transition est étiquetée par :

- le signal `AdaptMOCASComponent` dont la réception indique une demande d'adaptation du sous-composant détenu ;
- la garde, composée de trois sous-parties, vérifiant la cohérence de l'adaptation demandée ;
- l'action interne `adaptMOCASComponent(...)` qui réalise l'adaptation en parcourant le métamodèle UML du composant et en remplaçant les instances des éléments UML avec celles fournies en paramètre. Elle repose notamment sur l'action `compose` décrite à la section 4.2.2.

Le signal `AdaptMOCASComponent` dispose des attributs suivants, correspondant aux différents éléments adaptables dans le composant :

- l'attribut `attributes` de type `MOCASProperty[]`, destiné à accueillir un nouvel ensemble d'attributs pour le composant ;
- l'attribut `context` de type `MOCASFunctionalContext`, destiné à accueillir une nouvelle classe d'implémentation pour les actions internes ;
- l'attribut `behavior` de type `MOCASBehavior`, destiné à accueillir une nouvelle machine à états dans l'état de sous-machine.

Les valeurs de ces attributs sont passées en paramètre de l'action `adaptMOCASComponent(...)` une fois l'adaptation validée par la garde.

Le conteneur est aussi capable de réagir au signal `DeferredAdaptMOCASComponent`, spécialisation du signal `AdaptMOCASComponent`. Ce signal sert à reporter une adaptation tant que les conditions spécifiées dans la garde ne sont pas favorables à la réalisation de cette adaptation, c.-à-d. tant que la garde de la transition interne est fausse. Ceci est rendu possible grâce au traitement différé des événements, mécanisme natif des machines à états UML. Le signal doit être marqué comme étant différé (`/defer` sur la figure 5.1) dans un certain état. Ce signal est alors retenu dans la file des événements de la machine tant qu'il ne déclenche pas de transitions depuis l'état dans lequel il est marqué comme différé.

L'utilisation d'une transition interne permet de réaliser l'objectif de transparence du processus d'adaptation pour le concepteur du composant et pour le composant lui-même. En effet, l'exécution d'une transition interne ne provoque aucun changement d'états. De ce fait, aucune activité en cours d'exécution dans un état n'est interrompue et aucune action `exit` ou `entry` d'un état n'est exécutée à cause d'une adaptation. Le concepteur du composant a donc la garantie qu'il n'y a pas d'effets de bord au processus d'adaptation. Le composant a lui la garantie que ses services ne seront pas arrêtés. Par exemple, un composant `Système d'alarme` dont l'activation de l'un des états entraîne l'exécution d'une activité `alarme()` produisant un son strident est adaptable sans interrompre cette activité.

5.1.2 Le moment de l'adaptation

La quiescence est un état stable d'un composant dans lequel une adaptation ne compromet pas l'intégrité de ce composant (cf. section 2.3.1). Cet état de quiescence est atteint dans un composant MOCAS à chaque fin d'un cycle *run-to-completion*. En effet, losque le cycle est terminé, la machine à états est dans une configuration d'états actifs stables. Les effets des transitions et les actions *entry* et *exit* des états ont été réalisés. Seules les activités des états sont en cours d'exécution entre deux cycles. Néanmoins, l'exécution de ces activités n'empêchent pas l'adaptation étant donné que le déclenchement d'une transition interne ne provoque pas leur interruption.

Par ailleurs, le mécanisme de composition de MOCAS fait que le conteneur et le composant adapté ont une file d'événements commune. Le signal `AdaptMOCASComponent` est donc inséré dans la file de la même manière que les signaux destinés aux services fonctionnels du composant. L'adaptation s'insère donc naturellement dans le flot d'exécution des services fonctionnels. Les autres composants continuent à envoyer des signaux au composant adapté, l'adaptation est transparente pour eux, moyennant un temps de réponse accru du temps de réalisation de l'adaptation.

Même si une adaptation peut être demandée à n'importe quel moment de l'exécution du composant, sa réalisation, dans le cadre de l'adaptation de la machine à états (propriété `behavior` du signal `AdaptMOCASComponent`), nécessite que la condition `IsBehaviorConsistent` (cf. fig. 5.1, sous-partie de la garde) soit vraie. Cette condition, spécifiée dans la garde de la transition interne, permet le transfert d'état du composant et donc la continuité de son exécution (cf. section 5.1.4). Elle est vraie lorsque la machine à états courante du composant est dans une configuration d'états actifs dont la structure existe dans la machine à états candidate au remplacement (i.e. celle contenue dans l'attribut `behavior`).

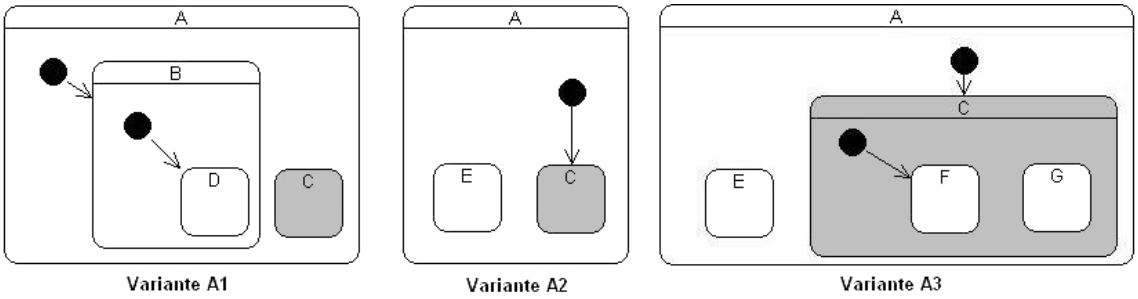

FIGURE 5.2 – Variantes cohérentes d'une machine à états

La figure 5.2 illustre cette condition. Elle présente trois variantes, `A1`, `A2` et `A3`, d'une même machine `A`. Ces trois machines possèdent une hiérarchie commune d'états : l'état `A` et son sous-état `C`. Lorsque la variante `A1` a l'état `C` actif, la variante `A2` et `A3` peuvent la remplacer car elles comporte ce même état, qui pourra donc être activé. Si la variante `A3` est choisie, son état `C` étant un état composite, son sous-état par défaut `F` sera alors activé lors de l'adaptation.

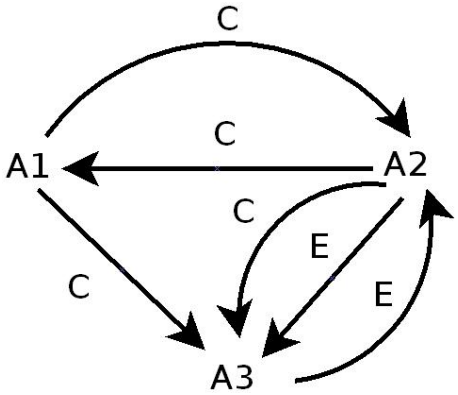

FIGURE 5.3 – Adaptation autorisée entre les variantes

La figure 5.3 présente un graphe connectant les variantes en fonction de l'état actif autorisant l'adaptation. Ainsi, la variante `A1` est directement adaptable par la variante `A3` lorsque son état `C` est actif. En revanche, la variante `A3` ne peut pas être directement adaptée par la variante `A1` lorsque son état `E` est actif car la variante `A1` ne possède pas cet état `E` (cf. fig. 5.2). La variante `A3` doit donc d'abord être adaptée avec la variante `A2` dans l'état `E`, qui sera elle-même adaptée par la variante `A1` lorsque l'état `C` sera actif. L'utilisation systématique du signal `DeferredAdaptMOCASComponent` prend tout son sens dans cet exemple : il reporte l'adaptation tant que l'état commun n'est pas atteint (i.e. activé). La transition interne `DeferredAdaptMOCASComponent(...)[not(hasCommonState)]/` (cf. fig. 5.1) sert à consommer l'événement si cet état commun n'existe pas.

La condition de quiescence de MOCAS est donc plus fine que celle de Kramer et Magee [95] ou que la condition de tranquillité de Vandewoude et al. [96] (cf. section 2.3.1 et fig. 5.4). En

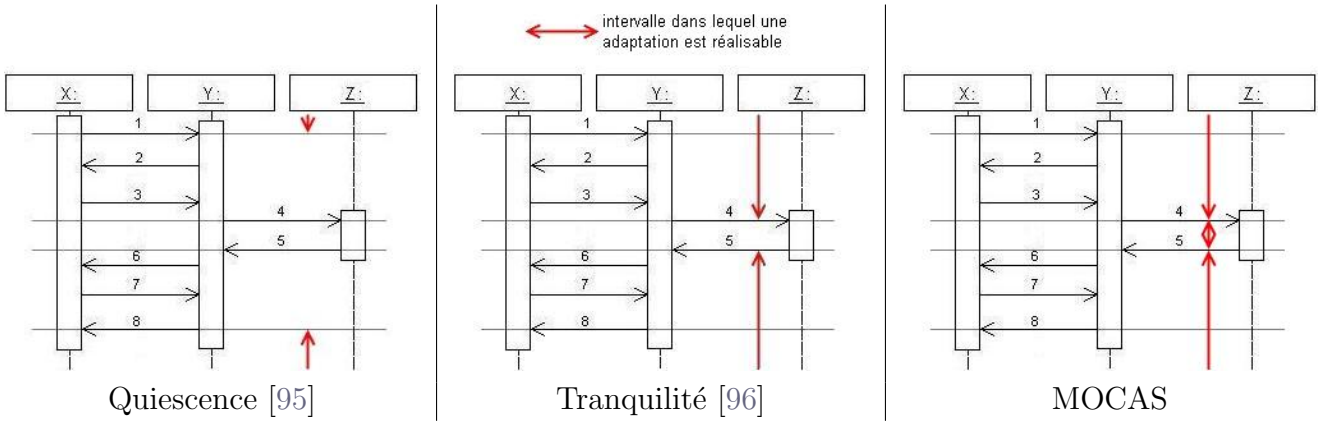

FIGURE 5.4 – Comparaison des moments d'adaptation possibles

effet, MOCAS autorise l'adaptation entre deux messages d'un protocole de communication entre composants.

5.1.3 La cohérence de l'adaptation

Nous avons vu en section 2.3.3 les trois points requérant une vérification dans un processus d'adaptation structurelle. MOCAS s'intéresse à l'adaptation comportementale et, de ce fait, la vérification à une portée locale au composant.

La garde de la transition interne pose des préconditions à l'adaptation d'un composant (cf. fig. 5.1) relatives aux éléments adaptés :

1. `AreAttributesConsistent` : cette partie de la garde concerne l'adaptation des attributs. Elle vérifie que tous les attributs qu'utilise le contexte fonctionnel sont bien contenus dans cet ensemble. En cas de réduction de l'ensemble (c.-à-d. de suppression d'attributs), elle vérifie que la configuration active ne comporte aucun état ayant des activités en cours d'exécution. En effet, une activité peut requérir des attributs du composant. Etant donnée que l'activité n'est pas interrompue pendant l'adaptation, les attributs utilisés par l'activité doivent rester disponibles ;
2. `IsContextConsistent` : cette partie concerne l'adaptation du contexte fonctionnel. Elle vérifie que le nouveau contexte fonctionnel implémente bien toutes les actions internes déclenchées par la machine à états et utilise des attributs définis dans le composant ;
3. `IsBehaviorConsistent` : cette partie concerne l'adaptation du comportement et garantit la continuité de l'exécution du composant après son adaptation.

La garde peut être étendue pour les besoins particuliers d'un composant. Par exemple, il est possible de restreindre l'ensemble des états d'un composant autorisant l'adaptation. A cet effet, l'opérateur `boolean inState(s : State)` du langage OCL [136] est utilisé pour tester si l'état `s` donné en paramètre est actif dans la machine à états.

A ces trois préconditions s'ajoutent un invariant à l'adaptation : tous les états actifs possédant des invariants ont leurs invariants vrais. La violation de cet invariant (i.e. un état actif

possède un invariant faux) est alors gérée dans le cadre de l'auto-réparation (cf. section 6.2.1), par exemple en annulant l'adaptation effectuée.

La garde `IsBehaviorConsistent` ne vérifie que la structure de la machine à états et ne s'intéresse pas au protocole lié aux services du composant. Rappelons que dans MOCAS, une opération de service correspond à un signal étiquetant une transition. L'absence de cette transition dans la nouvelle machine à états signifie la disparition d'une opération d'un service fourni par le composant. Or, un autre composant requérant cette opération se retrouve alors dans l'impossibilité d'assurer ses propres services. La vérification du protocole comme le fait Sibertin-Blanc et al. [105] (cf. section 2.3.3) coûte « cher » en temps de calcul. De plus, elle empêche les adaptations correctives : si le protocole courant du composant doit être modifié dans le but de corriger une mauvaise spécification, valider ce nouveau protocole par rapport à l'ancien n'a pas de sens.

5.1.4 Le transfert d'état

Nous avons vu à la section 2.3.2 que la continuité de l'exécution nécessite le transfert du point d'exécution et de la valeur des attributs du composant. Afin de réaliser le transfert du point d'exécution, la garde `IsBehaviorConsistent` vérifie dans la machine à états candidate au remplacement l'existence de la hiérarchie d'états correspondant à la configuration active dans la machine courante du composant.

Sans la garde `IsBehaviorConsistent`, une machine à états pourrait être remplacée par n'importe quelle autre, cette dernière serait alors activée dans son état par défaut. Le comportement du composant n'aurait pas de continuité et perdrait son « identité » : son comportement serait totalement différent de celui avant l'adaptation. La garde préserve ainsi l'identité du composant entre deux adaptations et assure la continuité de son comportement.

Le transfert des valeurs des attributs du composant n'est pas requis lors d'une adaptation avec MOCAS. En effet, l'adaptation comportementale ne nécessite pas de transfert d'état étant donné que la structure d'accueil (le conteneur) du composant persiste, à l'inverse d'une adaptation structurelle où un composant est instancié et un autre détruit. Néanmoins, ce transfert est utile lorsque les valeurs des attributs doivent être exportées d'une structure de données pour être importées dans une autre. Ce transfert entraîne pendant son exécution une indisponibilité pour l'activité qui utilise les attributs et donc une rupture de la continuité des services. De plus, le transfert nécessite l'écriture d'opérations de conversion entre les deux types. Or, nous avons choisi de privilégier la transparence de l'adaptation en évitant ce travail d'écriture et la continuité des services par rapport à la flexibilité des structures de données. Du fait de cette disponibilité constante des services, il n'est pas possible de supprimer des attributs en cours d'utilisation par une activité ni de modifier le type des attributs.

5.2 Adaptations supportées

Nous listons ci-après les adaptations supportées par un composant MOCAS. Elles peuvent être faites de manière non anticipée et être introduites dans le système après son déploiement.

5.2.1 Adaptation du comportement

MOCAS autorise différents types d'adaptation grâce à la réification du comportement des composants avec les machines à états :

- *Raffinement d'états* : un état simple de la machine à états devient composite : il est alors raffiné par un ensemble d'états exclusifs ou orthogonaux (cf. fig. 5.5). S'il appartient à la configuration active du composant, son sous-état par défaut est activé. S'il est déjà un état composite, de nouveaux états orthogonaux peuvent être introduits afin d'ajouter un comportement parallèle. Le protocole du composant est alors préservé dans la nouvelle machine et de nouveaux services, portés par les transitions entre les sous-états, peuvent être introduits ;
- *Modification du protocole* : la séquence de signaux qu'admet le composant est modifiée en réorganisant les transitions de la machine. De nouvelles transitions peuvent être introduites et étiquetées avec des signaux correspondant à de nouveaux services fournis par le composant. De nouveaux états peuvent marquer les étapes du nouveau protocole.

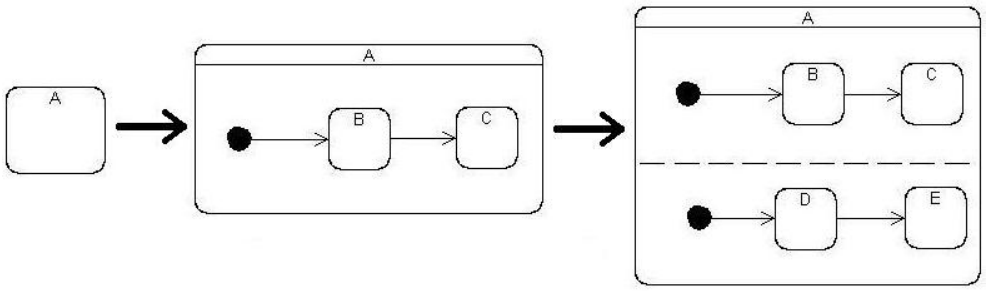

FIGURE 5.5 – Raffinement successif d'un état

La modification du protocole suppose que les composants présents dans le système ne souffrent pas de cette modification. De plus, l'introduction de nouveaux services et de nouveaux protocoles suppose que des composants capables de les exploiter existent dans le système.

5.2.2 Adaptation de l'implémentation

Le contexte fonctionnel implémente les algorithmes du composant. Il peut être changé à n'importe quel moment, sans être couplé au remplacement de la machine ou des attributs du composant. Son remplacement peut être fait dans le but de corriger ou de perfectionner le composant. Si une activité est en cours d'exécution lors de l'adaptation, elle se termine en utilisant l'ancien contexte. Le nouveau contexte sera utilisé à sa nouvelle exécution. De ce fait, si l'activité comporte un bogue rendant son exécution interminable, le composant ne pourrait pas être corrigé.

5.2.3 Remplacement du composant

L'adaptation comportementale dans MOCAS permet de modifier individuellement les attributs, la machine à états et le contexte fonctionnel du composant. Lorsque ces trois éléments sont modifiés en même temps, c.-à-d. au cours du traitement d'une seul instance du signal `AdaptMOCASComponent`, l'adaptation comportementale correspond au remplacement complet du composant détenu par le conteneur.

La garde maintient les éléments adaptés synchronisés : tant que le nouveau contexte fonctionnel implémente toutes les actions internes déclenchées par la nouvelle machine à états et qu'il n'utilise que les attributs du nouvel ensemble, les trois éléments peuvent être changés en même temps. De ce fait, le nouveau comportement du composant a en commun avec l'ancien comportement au minimum la configuration active de l'ancienne machine à états.

Néanmoins, ce remplacement ne permet pas d'introduire de nouveaux ports ayant des interfaces requises (port de sortie ou port d'entrée). En effet, ceci nécessiterait de satisfaire les nouveaux services requis et nous mettrait alors dans la situation d'une adaptation structurelle. Le signal d'adaptation devrait donc être suivi d'un signal connectant ce port à un nouveau composant. L'adaptation ne serait alors plus atomique et ne pourrait plus être demandée à n'importe quel moment de l'exécution du composant car les deux signaux pourraient s'entrelacer avec un troisième signal. Ce problème sera résolu lorsque nous nous intéresserons aux composants MOCAS autonomiques (cf. section 6.1.3) qui garantissent l'atomicité d'adaptations complexes, c.-à-d. nécessitant plusieurs signaux.

5.3 Adaptation du composant `Car`

La figure 5.6 montre la machine à états du composant `Car` de la section 4.3 après son adaptation. Le stéréotype `AdaptiveMOCASComponent` appliqué au composant spécifie à la conception que le composant est adaptable et sera de ce fait déployé dans son conteneur. Cette adaptation consiste au raffinement de l'état `Forward` avec l'introduction du service de gestion des rapports de vitesse. Ce service comporte les signaux `Up` et `Down` permettant de monter et descendre les vitesses de la voiture. Cette adaptation est possible lorsque le composant est dans l'état `Stopped` ou dans l'état `Forward`, même avec l'activité `mesure()` en train de s'exécuter. Elle ne provoque pas de changement d'état et donc pas l'exécution de l'action `entry`. Elle n'est en revanche pas possible lorsque le composant est dans l'état `Backward` étant donné que l'état est inexistant dans la nouvelle machine à états. Si l'adaptation survient lorsque le composant est dans l'état `Forward`, l'état `First`, son nouveau sous-état par défaut, est alors activé rendant ainsi possible la réception des signaux `Up` et `Down`. L'action interne `backward()` n'est plus appelée par la nouvelle machine, il est de ce fait possible de remplacer l'ancien contexte fonctionnel par un nouveau afin d'alléger la mémoire, ou bien de le conserver en cas de retour à l'ancien mode de fonctionnement. Après une adaptation lorsque l'état `Moving` est actif, l'invariant de vitesse `[speedValue<=130]` est vérifié pour s'assurer de la cohérence du nouveau comportement. L'introduction du nouveau service (et des signaux reçus associés) suppose qu'il existe un composant dans le système qui sera lié au composant `Car` afin de l'exploiter.

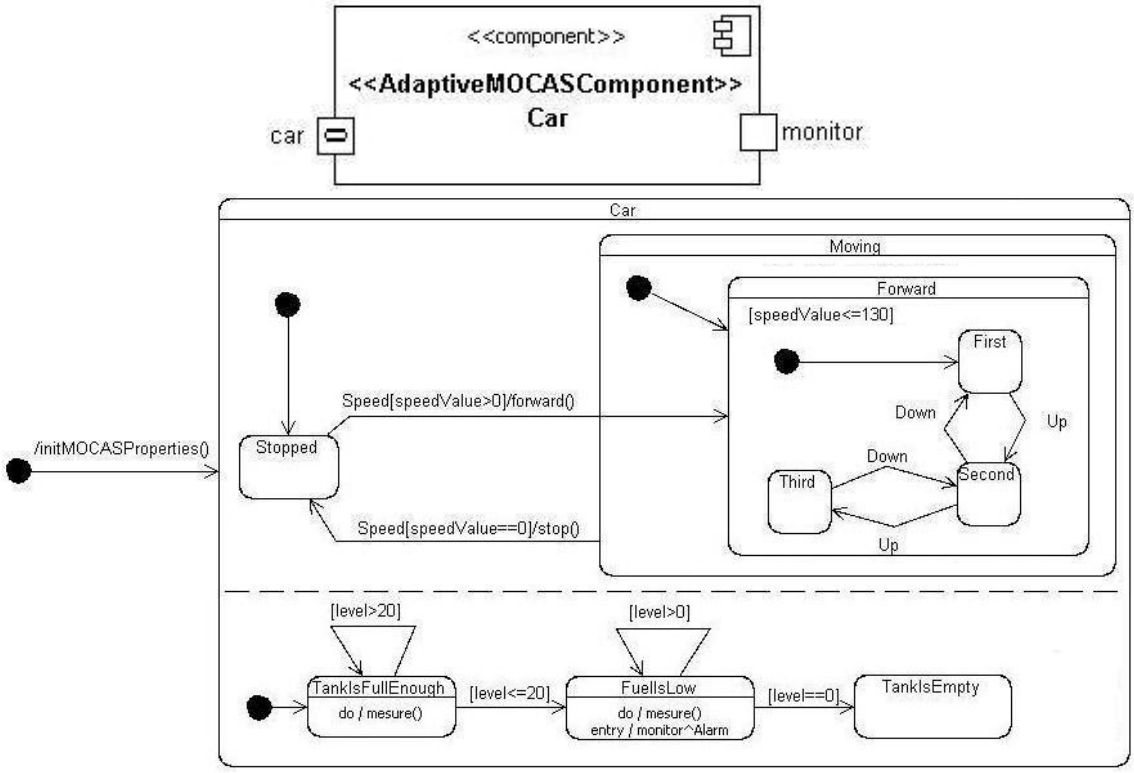

FIGURE 5.6 – Raffinement du comportement du composant `Car`

5.4 Synthèse

Dans ce chapitre, nous avons vu comment un conteneur pour les composants MOCAS a été conçu dans le but de les rendre adaptables. Tout en satisfaisant les contraintes de l'adaptation dynamique (quiescence, transfert d'état, cohérence), ce conteneur a plusieurs avantages :

- il réalise une adaptation transparente :
  - à la conception, pour le concepteur du composant adaptable, qui n'a pas à se soucier des mécanismes d'adaptation,
  - à l'exécution, pour les clients du composant adapté, qui continuent à lui envoyer des signaux, et pour le composant adapté lui-même qui n'interrompt pas ses activités en cours ;
- il ne nécessite pas le déploiement d'une infrastructure pour les composants, qui centraliserait l'adaptation ;
- il s'appuie sur les éléments natifs du langage UML et des machines à états sans en altérer la sémantique ;
- il permet de mettre à jour le composant dans le cas de correction d'erreurs ou d'évolution de ses spécifications.

Le chapitre suivant introduit les composants réalisant une boucle de contrôle attachée à un composant adaptable, de manière à doter ce composant de capacités d'auto-configuration et d'auto-réparation.

Chapitre 6

Auto-adaptation des composants MOCAS

Sommaire

La manière communément admise en génie logiciel pour doter un système de propriétés autonomiques consiste à lui adjoindre une boucle de contrôle (cf. section 2.1.2). Dans les systèmes à base de composants, cette boucle est le plus souvent unique et centralisée dans l'infrastructure gérant les composants. De plus, la flexibilité de la boucle se limite à pouvoir changer la politique qui gouverne le comportement de cette boucle. La structure est, quant à elle, fixée à la conception du système.

Afin d'augmenter la robustesse et la flexibilité d'un système, le contrôle tend à être décentralisé. De multiples boucles de contrôle coexistent alors dans le système. Malheureusement, ceci implique de coordonner les actions des boucles afin de préserver la cohérence du système (cf. section 2.1.3).

MOCAS utilise une approche décentralisée pour rendre autonomique un système à base de composants MOCAS. Chacun des composants adaptables du système est alors doté de sa propre boucle de contrôle. Cette boucle est elle-même à base de composants MOCAS. La boucle présente

donc les avantages que lui confère le modèle MOCAS : flexibilité et dynamicité de sa structure et de son comportement. De plus, MOCAS propose de coordonner ces boucles en s'appuyant sur les protocoles d'interaction.

Nous introduisons à la section 6.1 les différents composants constituant une boucle de contrôle dans MOCAS. Nous présentons à la section 6.2 différentes politiques autonomiques rendues possibles par notre modèle de composants. Enfin, nous détaillons à la section 6.3 la manière dont sont coordonnées les différentes boucles dans un système MOCAS.

6.1 Une boucle de contrôle à base de composants MOCAS

Un composant MOCAS autonomique (CMA) [137] est un composant MOCAS adaptable dont le conteneur a été étendu pour supporter une boucle de contrôle. La figure 6.1 présente le profil du modèle de composants autonomiques de MOCAS. Un CMA est stéréotypé `AutonomicMOCASComponent`. Son conteneur possède un évaluateur (`MOCASEvaluator`) et un répartiteur (`MOCASDispatcher`) instanciés en même temps que lui et un ensemble de capteurs (`MOCASSensor`) attachés suivant les besoins du CMA. Des effecteurs (`MOCASEffector`) sont chargés de réaliser les opérations d'adaptation.

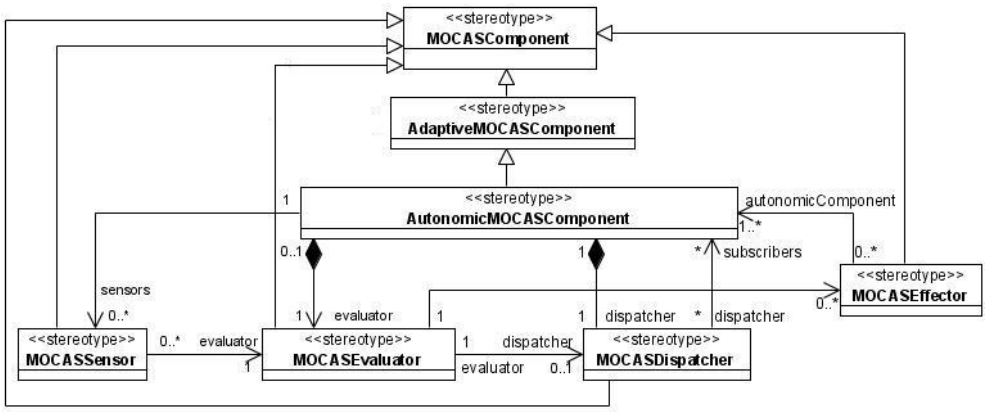

FIGURE 6.1 – Profil du modèle de composants autonomiques de MOCAS

La figure 6.2 montre les connexions internes au conteneur. Ce dernier est composé avec son évaluateur de manière à lui déléguer les signaux qu'il reçoit. Sa machine à états est spécialisée pour maintenir les connexions entre les composants de la boucle lors de leur remplacement (cf. redéfinition des transitions internes `SetPropertyValue` sur la figure 6.2).

6.1.1 Les capteurs

Un CMA possède des capteurs chargés de surveiller son fonctionnement et de sonder son environnement. Un capteur est un composant MOCAS stéréotypé `MOCASSensor`. Il est lié à

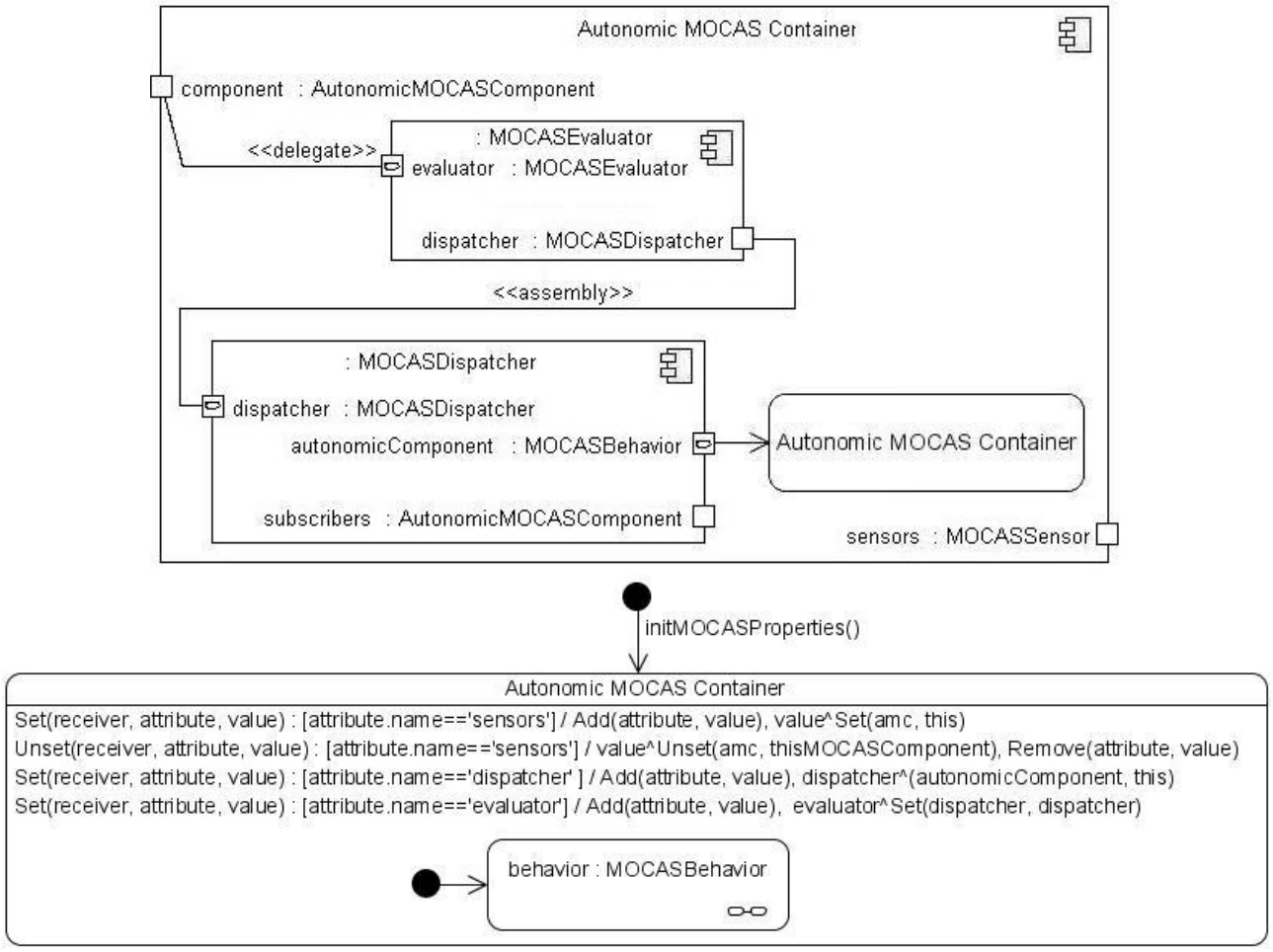

FIGURE 6.2 – Le conteneur autonomique MOCAS

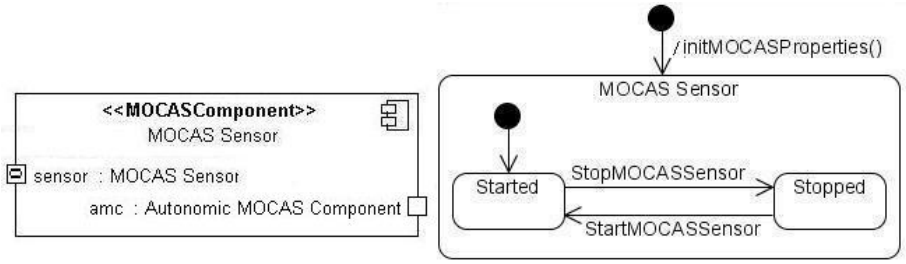

FIGURE 6.3 – Le capteur MOCAS réactif

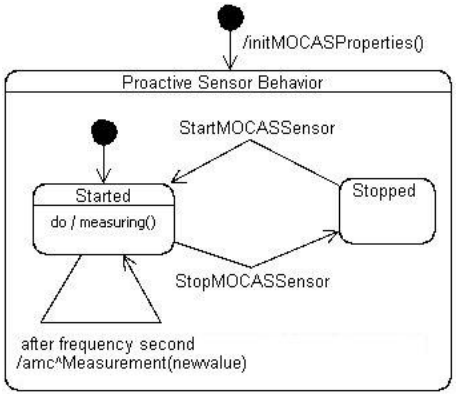

FIGURE 6.4 – Le capteur MOCAS proactif

l'exécution au CMA qui le nécessite. Il est soit réactif – il est alors activé à la réception d'un signal particulier – soit proactif pour anticiper les besoins du composant – il sonde alors le système à intervalle régulier (et par conséquent accroît la charge du système).

La figure 6.3 montre la machine à états générique d'un capteur. Le concepteur de ce dernier raffine l'état `Started` avec la fonction qu'il souhaite : surveillance de l'environnement (charge du processeur, taux d'occupation de la mémoire...) ou du composant (exceptions dans le CMA, des signaux qui transitent dans le CMA, du changement de la valeur d'un attribut...). Il spécifie aussi le signal qui est émis par le capteur vers le CMA.

La figure 6.4 montre la machine à états d'un capteur proactif. Le développeur de ce dernier fournit un contexte fonctionnel implémentant l'action interne `measuring()`. Il précise aussi dans l'action `initMOCASProperties()` la valeur de l'attribut `frequency` indiquant le nombre de secondes entre deux mesures. Une bibliothèque logicielle spécialisée, comme WildCat dans SAFRAN (cf. section 3.3.3.1), peut être utilisée pour la surveillance des ressources de l'environnement. La valeur mesurée est ensuite envoyée au CMA et est analysée par l'évaluateur décrit ci-après.

Lorsqu'une exception interne survient (p. ex. violation d'invariant ou rupture d'un canal de communication), le CMA envoie le signal `Failure` à tous ses capteurs. Quand le capteur détecte un événement anormal, il envoie un signal à l'évaluateur du composant auquel il est lié.

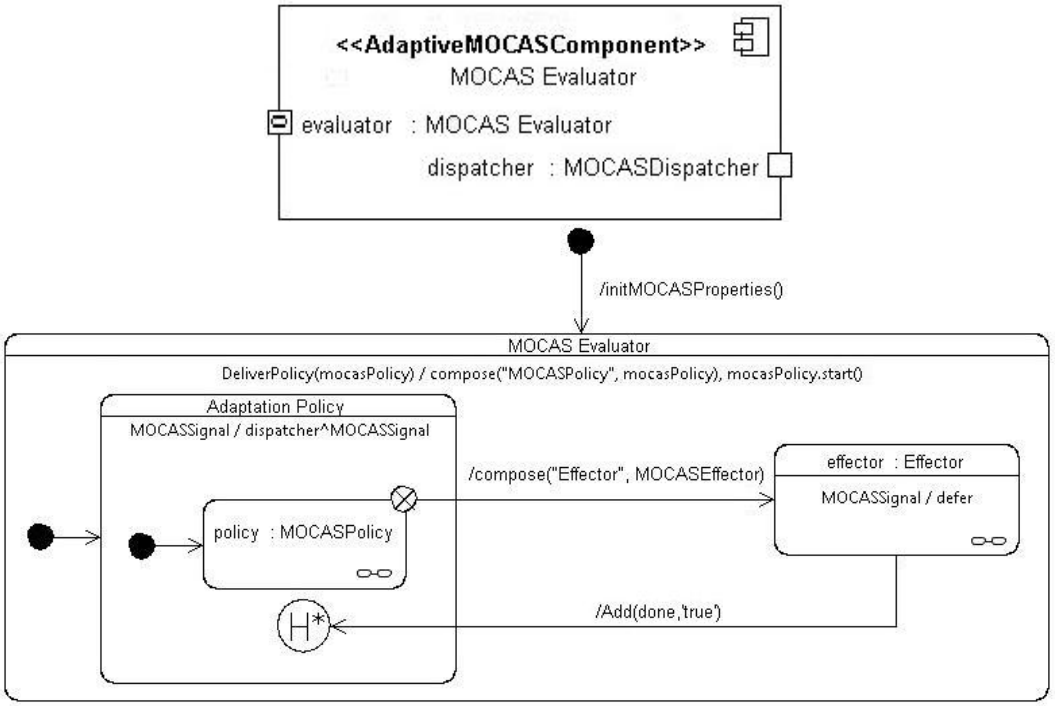

FIGURE 6.5 – L'évaluateur MOCAS

Le capteur peut être désactivé en lui envoyant le signal `StopMOCASSensor` (puis réactivé avec `StartMOCASSensor`). Ceci l'empêche d'envoyer des signaux qui ont déjà été pris en compte ou qui n'ont pas de sens dans un environnement particulier, et préserve de plus les ressources du système.

6.1.2 L'évaluateur

Un CMA possède un unique évaluateur [16] gérant sa politique. L'évaluateur est un composant MOCAS stéréotypé `MOCASEvaluator`. Il est lié au moment de l'appel à l'action `initMOCASProperties()` implémentée par le contexte fonctionnel du conteneur du CMA. La liaison se fait via un connecteur de délégation (cf. fig. 6.2), de telle manière que tous les signaux arrivant au CMA lui sont transférés. Il est de plus lié au répartiteur auquel il transmet une partie des signaux qu'il reçoit. La figure 6.5 montre sa spécification générique.

Tous les signaux reçus par l'évaluateur sont analysés suivant la politique contenue dans son état de sous-machine `MOCASPolicy`. En l'absence de politique, les signaux sont directement transmis au répartiteur. La politique est délivrée et remplacée via le signal `DeliverPolicy(mocasPolicy)`. Ce signal déclenche l'action `compose` qui compose la machine à états `mocasPolicy` avec l'état `MOCASPolicy`. L'action `start` est ensuite invoquée afin de démarrer la politique.

16. L'évaluateur correspond à l'*analyseur* dans la boucle de contrôle d'IBM, montrée sur la figure 2.1, page 36.

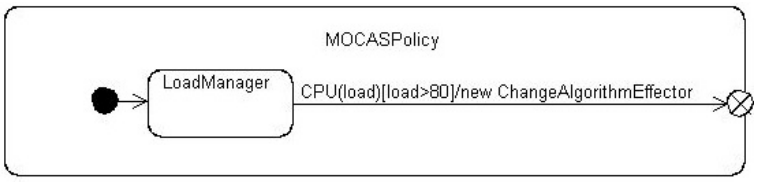

FIGURE 6.6 – Une règle MOCAS

La politique supportée est une politique d'actions (cf. section 2.1.2) dont les règles sont structurées par une machine à états. Ces règles sont de type *événement-condition-action* et sont réalisées par les transitions entre les états de la machine sous la forme *signal-garde-effets*. L'activation d'un état de la politique ne permet de déclencher qu'un sous-ensemble des règles contenues dans cette politique. L'attente de l'occurrence de plusieurs signaux pour déclencher une règle nécessite la définition de signaux différés dans les états.

La figure 6.6 montre un exemple de règle qui nécessite l'occurrence du signal `CPU(load)` pour se déclencher. Si la garde `load>80` est validée, la règle instancie alors l'effecteur `ChangeAlgorithmEffector` grâce à l'action UML `CreateObjectAction` (cf. section 4.1.3). Cet effecteur est chargé d'adapter le CMA avec un contexte fonctionnel consommant moins de ressources. La transition a pour cible un « point de sortie » (*Exit point* dans UML). Ce point de sortie est un pseudo-état qui permet de faire la liaison vers l'état `Effector`. Cet état est destiné à être composé avec un effecteur. Il permet de désactiver la politique le temps de réaliser l'adaptation, de manière à éviter des adaptations concurrentes au sein du même composant. Tous les signaux reçus de l'extérieur pendant ce temps-là sont différés jusqu'à la fin de l'adaptation.

L'administrateur du composant a la responsabilité de définir la politique. Cette dernière, en étant spécifiée avec une machine à états, facilite son travail :

- elle peut être réalisée graphiquement en dessinant la machine ;
- elle peut être comprise rapidement en visualisant ses états actifs (chaque état peut par exemple correspondre à un mode de fonctionnement particulier du composant).

L'administrateur a aussi la possibilité de déployer un évaluateur indépendant pour gérer plusieurs politiques d'adaptation d'un CMA. Etant donné que l'évaluateur est un composant adaptable, la politique qu'il détient est modifiable dynamiquement. L'évaluateur indépendant gérant les politiques peut donc envoyer à l'évaluateur du CMA un signal `AdaptMOCASComponent` contenant une nouvelle politique, plus restrictive par exemple, correspondant à de nouvelles conditions environnementales (à la manière du contrôle direct illustré avec la figure 2.4 de la section 2.1.3).

6.1.3 Les effecteurs

Un effecteur est un composant MOCAS stéréotypé `MOCASEffector` chargé de réaliser les actions d'adaptation qu'il implémente. Il est instancié par une action associée à une règle de la politique d'un CMA. Il a accès aux attributs du signal de la règle qui l'a initiée et à ceux de son CMA. Il est destiné à être composé dans l'état de sous-machine `Effector` de l'évaluateur. Il

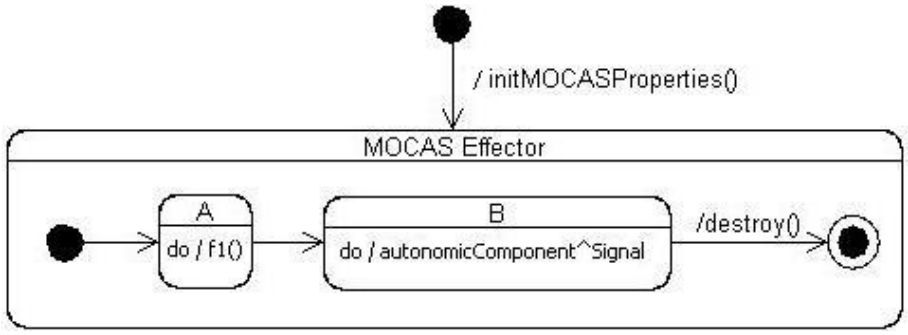

FIGURE 6.7 – Exemple de machine à états d'un effecteur

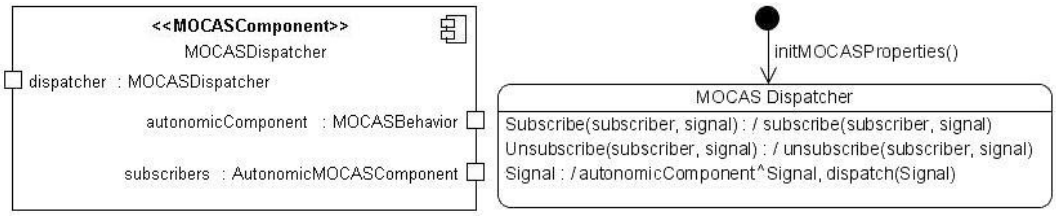

FIGURE 6.8 – Le répartiteur MOCAS

peut ainsi envoyer des signaux au répartiteur. Un seul effecteur est actif à la fois dans le CMA et sa durée de vie est limitée aux temps de réalisation des actions.

La figure 6.7 montre un exemple de machine à états d'un effecteur. Cette machine a typiquement des états qui possèdent des activités et qui sont reliés par des transitions d'achèvement [17]. La machine possède un état final marquant la fin de l'adaptation et de l'existence de l'effecteur (cf. action interne `destroy()`). Lorsque cet état final est atteint, la transition d'achèvement issue de l'état `Effector` dans le comportement de l'évaluateur est franchie afin de réactiver la politique. Les erreurs survenant dans un effecteur donnent aussi lieu à un signal `Failure` envoyé aux capteurs de son CMA de sorte qu'une solution alternative d'adaptation puisse être trouvée.

6.1.4 Le répartiteur

Un CMA possède un unique répartiteur intervenant dans la logique de coordination globale des adaptations du système (cf. section 6.3). Le répartiteur est un composant MOCAS stéréotypé `MOCASDispatcher`. Tout comme l'évaluateur, il est instancié au moment de l'appel à l'action `initMOCASProperties()`.

La figure 6.8 montre sa spécification. Le répartiteur dispose d'un port de sortie `autonomicComponent` connecté directement à la machine à états du conteneur autonomique.

17. Une transition d'achèvement (*completion transition* dans UML) n'est déclenchée que lorsque l'activité de son état source se termine.

Il dispose aussi d'un port de sortie `subscribers` connecté à d'autres CMA. Un CMA peut se mettre en écoute d'un signal particulier auprès d'un autre CMA. Ce signal peut correspondre à un signal d'entrée ou bien à un signal issu de la politique d'adaptation. Le composant s'abonne auprès du CMA en lui envoyant le signal `Subscribe` précisant le signal qu'il souhaite écouter. Il peut révoquer son abonnement grâce au signal `Unsubscribe`. Par défaut, les signaux reçus de l'évaluateur sont systématiquement transmis au conteneur autonomique et aux CMAs en écoute (action interne `dispatch(Signal)`).

6.2 Réalisation de politiques autonomiques

Une politique dirige le comportement de la boucle de contrôle d'un CMA afin de réaliser les propriétés autonomiques. Nous proposons ici des politiques spécifiques exploitant les particularités du modèle de composants MOCAS pour réaliser certains aspects de ces propriétés.

6.2.1 Auto-réparation

Les politiques d'auto-réparation dans MOCAS visent à résoudre les problèmes survenant lors de l'exécution des actions internes d'un CMA. En cas d'exception de fonctionnement ou de violation de l'invariant d'un état actif, le CMA envoie un signal *Failure* dont les attributs contiennent les détails de l'exception venant de se produire. Ce signal est addressé à tous ses capteurs. Si un capteur est capable d'interpréter ce signal, il envoie alors un nouveau signal synthétisant son interprétation de l'exception. La politique définit la stratégie à adopter lorsqu'un tel signal survient. Toutes ces stratégies sont basées sur le service d'administration `to_state(configuration)` qui force la machine à états à activer la configuration en paramètre. Ce service ne déclenche ni les actions *entry* et *exit* ni les activités d'un état pour ne pas altérer les valeurs des attributs, mais il provoque un événement d'achèvement, au cas où une transition d'achèvement existe. Si une des feuilles de la configuration à activer est un état composite, la configuration par défaut de cet état est activée en déclenchant effets, actions et activités.

6.2.1.1 Repli

La réification du comportement d'un composant avec une machine à états autorise la mise en place d'une stratégie de repli (*rollback*). La configuration active ainsi que les valeurs des attributs du CMA sont alors sauvegardées avant le déclenchement d'un cycle *run-to-completion*. Ce point de sauvegarde constitue un point de reprise en cas d'erreur. Le service `to_state` est alors appelé par l'effecteur de repli (`RollbackEffector`) avec en paramètre la dernière configuration active connue. La restauration des valeurs des attributs annule les effets des actions internes mais pas ceux des actions agissant sur l'environnement, comme par exemple l'envoi de signaux.

La figure 6.9 illustre un scénario de repli d'un CMA. Lorsque le composant `A` est dans l'état `D`, il reçoit un signal `S2` avec une valeur -10 pour son attribut `p2`. Ce signal déclenche le passage de l'état `D` à `E` avec l'exécution d'une action `f(p2)` utilisant la valeur véhiculée par le signal. `f` modifie alors la valeur de l'attribut `p1` du composant `A` qui devient négative. Or l'état `E` a un invariant spécifiant que `p1` doit avoir une valeur positive lorsqu'il est actif. Une exception est

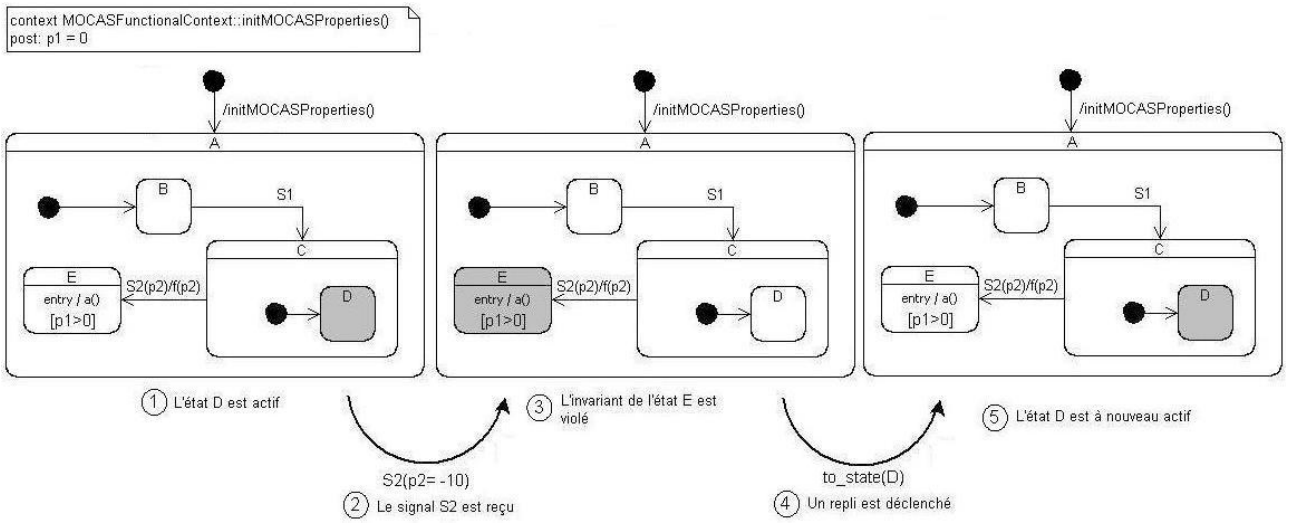

FIGURE 6.9 – Exemple de scénario de repli d'un composant autonomique

alors levée signifiant la violation de l'invariant. La politique spécifie dans ce cas de faire appel au `RollbackEffector`. Celui-ci met à jour les attributs avec leur valeur avant le changement d'états et invoque le service `to_state` pour réactiver l'état `D`.

Ce scénario suppose soit que l'implémentation des actions internes du composant est incorrecte, soit que l'attribut `p1` contenait déjà une valeur erronée, soit que la valeur véhiculée par le signal `S` était fausse. Ce dernier cas nécessite que le composant émetteur du signal soit averti de l'erreur. Il le sera s'il s'est abonné pour écouter les signaux `Failure` se produisant chez les destinataire des signaux qu'il envoie. Dans ce cas-là, il pourra lui-même procéder à un repli. Néanmoins, le comportement asynchrone des composants fait que l'émetteur aura sûrement déjà lui-même changé d'états depuis l'envoi de son signal. Une solution de repli global (c.-à-d. impliquant plusieurs composants du système) nécessite que chaque composant tienne un historique de son fonctionnement, corrélé aux signaux émis. Cette solution engendre un surcoût quant à la mémoire nécessaire à l'historique et quant aux interactions qu'elle engendre. Le diagnostic et l'analyse des conséquences des erreurs dépassent le cadre de cette thèse.

6.2.1.2 Réinitialisation

Un composant est réinitialisable par l'effecteur de réinitalisation `ResetEffector`. Cet effecteur désigne directement la machine à états du CMA comme paramètre du service `to_state`. Ceci a pour effet d'activer la configuration par défaut de la machine. L'action interne `initMOCASProperties()` est alors déclenchée, réinitialisant les attributs du composant avec leur valeur par défaut.

Le scénario de la figure 6.10 est identique au précédent mais cette fois-ci la règle d'adaptation fait appel à l'effecteur de réinitialisation `ResetEffector`. L'état initial par défaut `B` est alors activé et l'attribut `p1` remis à zero par l'action `initMOCASProperties()`.

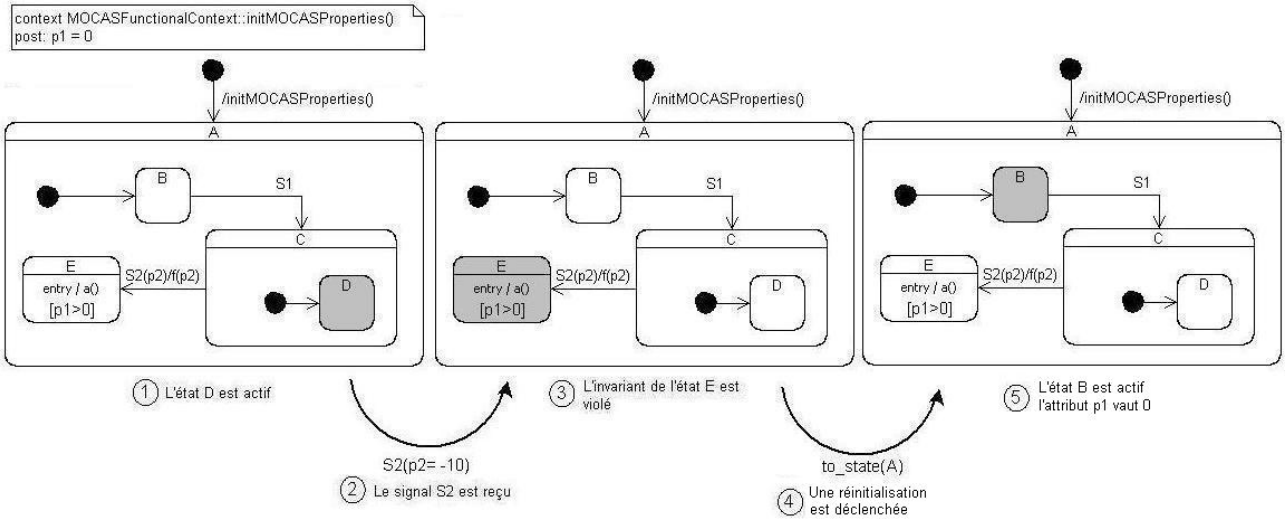

FIGURE 6.10 – Exemple de scénario de réinitialisation d'un composant autonomique

6.2.1.3 Configuration cohérente

Chaque état a un invariant associé qui doit être vrai tant que cet état est actif. De ce fait, l'effecteur `ConsistentConfigurationEffector` recherche s'il existe dans le composant une unique configuration pour laquelle les invariants sont vrais. Si une telle configuration existe, elle sera passée en paramètre du service `to_state`. Sinon, un signal `Failure` est envoyé afin de chercher une autre stratégie.

6.2.1.4 Configuration cible

Le service `to_state` active la configuration donnée en paramètre. L'effecteur `AbstractToStateEffector` permet de désigner n'importe quelle configuration comme point de reprise et d'affecter aux attributs du CMA les valeurs appropriées. Le concepteur de l'effecteur ou l'administrateur du composant désigne alors, en fontion de l'état ayant provoqué une erreur, la configuration la plus appropriée pour poursuivre le fonctionnement du composant.

6.2.2 Auto-configuration

Chaque possibilité de comportement d'un CMA correspond à un mode de fonctionnement. Un mode de fonctionnement est un triplet spécifiant un ensemble d'attributs, un contexte fonctionnel et une machine à états. Chaque mode est associé à des conditions environnementales à l'aide d'une politique d'adaptation.

6.2.2.1 Configuration au déploiement

La configuration d'un composant consiste à déterminer le comportement que ce composant doit adopter en fonction de l'environnement dans lequel il est déployé. Ceci suppose qu'il

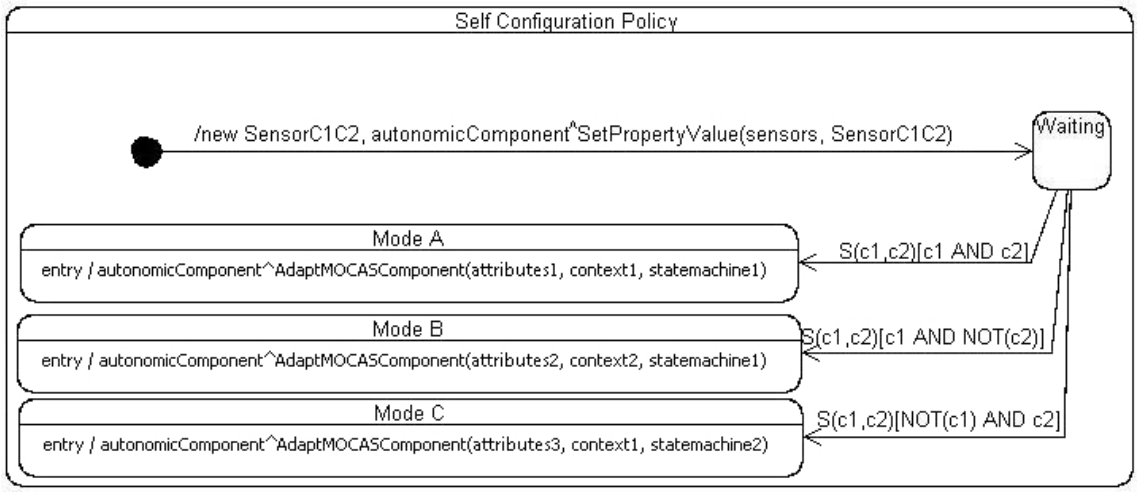

FIGURE 6.11 – Exemple de politique d'auto-configuration

existe des capteurs dans l'environnement de déploiement capable de fournir les informations qui permettent de choisir ce comportement.

La figure 6.11 montre une politique de configuration. Lorsque cette dernière est activée dans l'évaluateur, elle déploie le capteur `SensorC1C2`[18] qui est alors lié au CMA. Le capteur `SensorC1C2` mesure deux paramètres `c1` et `c2` dans l'environnement et envoie les valeurs à l'évaluateur. La politique permet à l'évaluateur de choisir l'adaptation à demander au CMA. Ainsi si les paramètres réalisent les conditions `c1` et `c2`, le mode `A` est activé. L'entrée dans ce mode provoque l'envoi au CMA du signal `AdaptMOCASComponent(attributes1, context1, statemachine1)` provoquant sa configuration avec les éléments contenus dans le signal.

6.2.2.2 Reconfiguration à l'exécution

La reconfiguration d'un composant consiste à adapter le mode de fonctionnement aux conditions courantes de l'environnement. Alors que la configuration est requise au déploiement du composant, la reconfiguration peut être requise à n'importe quel moment de l'exécution du composant. Elle fait appel au même type de politique que celles de configuration à la différence qu'elle connecte les modes de fonctionnement entre eux en fonction des conditions environnementales (cf. fig. 6.12) [94, 138, 139].

Le concepteur du composant peut définir un nouveau mode de fonctionnement pour le CMA après que ce dernier ait été déployé. L'administrateur pourra délivrer le nouveau triplet correspond au mode et une politique le prenant en compte à n'importe quel moment de l'exécution du CMA via le signal `DeliverPolicy` (cf. `Mode D` sur la figure 6.12).

De plus, une reconfiguration peut être requise lorsqu'un composant invoque une opération

18. L'action `new` correspond à l'action UML `CreateObjectAction` permettant d'instancier l'élément en paramètre.

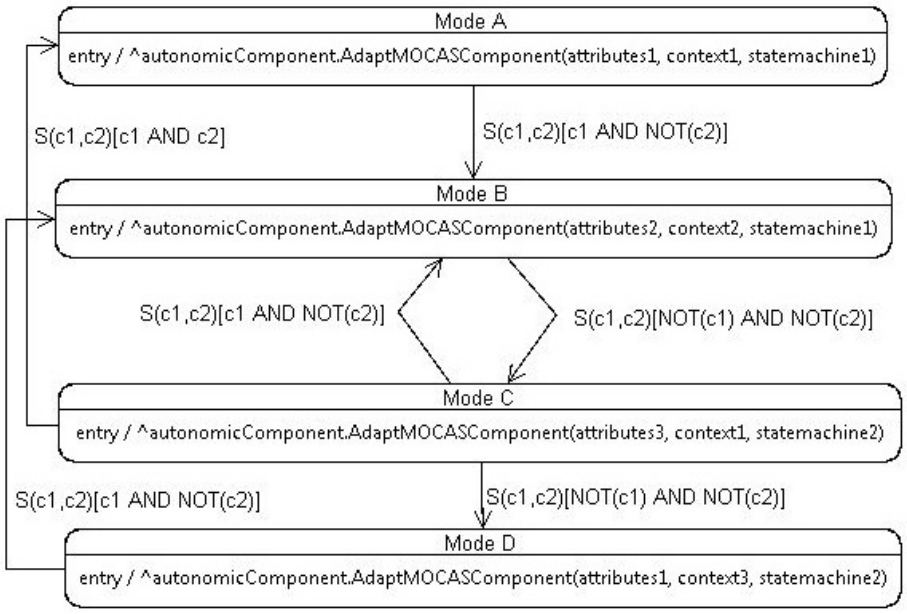

FIGURE 6.12 – Exemple de politique de reconfiguration

de service qui a été masquée suite à une précédente reconfiguration (les conditions environnementales ne permettaient plus d'assurer cette opération). Le composant envoie pour cela un signal au fournisseur du service. Le signal ne correspond alors à aucune transition dans la machine courante du fournisseur. Un capteur détecte le fait qu'aucune transition n'est déclenchée suite à la réception du signal. Si les conditions environnementales le permettent à nouveau, le composant est adapté afin de remplacer son mode de fonctionnement par un mode dans lequel le service est disponible.

6.3 Adaptation d'un système MOCAS

Toutes les politiques d'auto-adaptation évoquées dans la section 6.2 sont locales à un composant autonomique. Ces politiques n'impliquent et n'ont d'effets que sur un unique CMA. Or, nous avons vu à la section 2.1.3 qu'un composant ne peut s'adapter isolément sans menacer la cohérence du système auquel il appartient. De ce fait, MOCAS propose deux modes de coordination pour réaliser une adaptation globale dans le cadre de composants composés horizontalement.

6.3.1 Coordination par propagation

Le mode de coordination par défaut est le mode par propagation. Grâce à ce mode, un CMA a l'opportunité d'adapter son comportement en conséquence de l'adaptation d'un autre composant. Ce mode est approprié si deux composants ont été conçus indépendamment l'un de l'autre et si, dans le système dans lequel ils sont tous les deux composés horizontalement, l'adaptation

du second composant est dépendante de celle du premier. Ce mode est intéressant, par exemple, lorsqu un CMA propose de nouveaux services après s'être adapté. D'autres composants seront ainsi avertis de la disponibilité de ces nouveaux services et vont pouvoir eux-mêmes s'adapter afin de les utiliser.

Ce mode est réalisable grâce aux connexions réalisées entre les CMAs, via leur répartiteur. Un graphe est ainsi établi dans lequel les signaux transitant par les répartiteurs sont propagés. Ce graphe ne doit pas comporter de cycles afin d'assurer l'arrêt de la propagation. L'invariant OCL du listing 6.1 attaché au stéréotype `AutonomicMOCASComponent` de la figure 6.1 spécifie une telle contrainte. La vérification de cet invariant implique de parcourir les liaisons entre les composants. A l'exécution, elle nécessite que le modèle de l'architecture du système déployé soit mis à jour au gré de l'évolution des liaisons entre les composants. Ceci rejoint les problématiques de l'adaptation structurelle et dépasse le cadre de cette thèse.

Listing 6.1 – Invariant OCL spécifiant l'absence de cycles dans une composition horizontale de CMAs

```
1   context AutonomicMOCASComponent
2   def: noCycle(component: AutonomicMOCASComponent): Boolean =
3     if(dispatcher.subscribers->isEmpty())
4       result = true
5     else
6       result = dispatcher.subscribers.excludes(component) and dispatcher.subscribers->
            forAll(s | s.noCycle(component))
7     endif
8
9   context AutonomicMOCASComponent
10  inv: self.noCycle(self)
```

6.3.2 Coordination par un protocole d'interaction

Une adaptation globale à un système implique plusieurs composants. Elle peut nécessiter une séquence particulière d'actions effectuées par des composants différents. Il devient alors nécessaire de les coordonner pour maintenir la cohérence du système [140]. Pour cela, nous proposons de faire communiquer les CMAs par un protocole d'interaction.

6.3.2.1 Définition d'un protocole d'interaction

Les protocoles d'interaction sont surtout utilisés dans les systèmes multi-agents afin de structurer les communications entre les agents [141]. De façon général, un protocole d'interaction décrit les scénarios de communication possibles entre les entités d'un système. Il spécifie :

- les différents rôles que peuvent jouer les entités dans le protocole ;
- l'enchaînement des messages échangés entre les entités en fonction du scénario et du rôle joué.

Dans ce contexte, une « conversation » est la suite de messages échangés qui correspond à un scénario du protocole. Un protocole, en « bornant » l'espace des conversations possibles, permet de vérifier la terminaison de toutes ces conversations en s'assurant, aussi, de l'absence de situations d'interblocage.

6.3.2.2 Un protocole pour l'adaptation

Nous avons spécifié un protocole d'interaction pour coordonner l'adaptation de plusieurs composants MOCAS [142]. Ce protocole comporte un rôle *initiateur* et un rôle *participant* ainsi que trois phases :

1. une phase d'initiation dans laquelle *l'initiateur*, le composant à l'initiative de la communication, prend contact avec les *participants*, les différents composants impliqués dans le protocole ;
2. une phase d'exécution du plan d'adaptation décrivant les différentes actions à réaliser ;
3. une phase de terminaison dans laquelle les différents participants informent l'initiateur de la réussite ou de l'échec de la réalisation du plan d'adaptation.

Chaque message du protocole comporte les attributs ci-après, permettant de suivre la conversation en cours :

- l'attribut *sender* correspond au composant qui envoie le message ;
- l'attribut *receiver* correspond au (aux) composant(s) à qui le message est envoyé ;
- l'attribut *reply-to* correspond au composant à qui répondre ;
- l'attribut *plan* correspond à une machine à états décrivant le plan d'adaptation, il sert à paramétrer la phase d'exécution ;
- l'attribut *content* correspond à diverses informations nécessaires à la réalisation de l'adaptation.

Le protocole se déroule de la manière suivante (cf. fig. 6.13) :

1. dans la phase d'initiation :
   - un CMA qui a besoin d'une adaptation prend le rôle d'initiateur. Il contacte les participants – les CMAs dépendants de l'adaptation de l'initiateur – en leur envoyant le message *Request(sender, reply-to, receiver, plan)* qui décrit la requête d'adaptation,
   - chaque participant, en fonction de sa politique et du plan en paramètre du message *Request*, accepte la requête (en répondant avec le message *Agree(sender, reply-to, receiver, plan)*) ou bien la refuse (en répondant avec le message *Refuse(sender, reply-to, receiver, plan)*),
   - si tous les participants ont accepté, l'initiateur leur envoie le message *Confirm(sender, reply-to, receiver, plan)* afin de prévenir que le consensus est atteint ;
2. dans la phase d'exécution du plan, après la confirmation de l'initiateur, les participants et l'initiateur réalisent leur plan d'adaptation. Ce plan peut impliquer d'autres échanges de messages ;
3. dans la phase de terminaison :
   - si un participant échoue lors de la réalisation de son adaptation, il prévient l'initiateur avec le message *Failure(sender, reply-to, receiver, plan, content)*, dont l'attribut *content* contient le détail de l'erreur survenue,
   - si un participant réalise le plan d'adaptation sans erreur, il prévient l'initiateur avec le message *InformDone(sender, reply-to, receiver, plan)*,
   - le protocole se termine lorsque l'initiateur envoie à tous les participants :

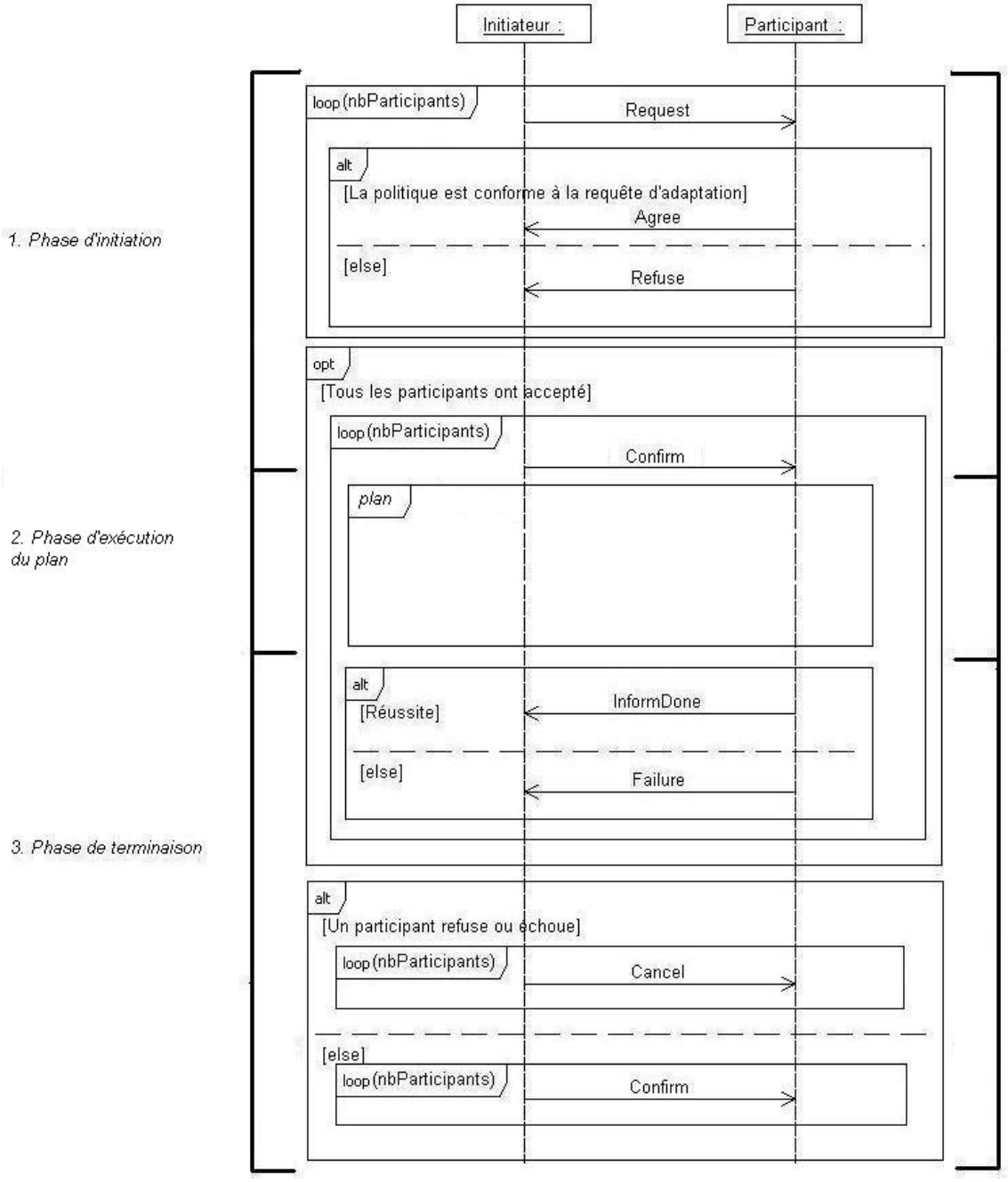

FIGURE 6.13 – Un protocole pour l'adaptation

- le message *Cancel(sender, reply-to, receiver, plan)*, si un des participants a refusé la requête d'adaptation ou bien échoué dans sa réalisation,
- le message *Confirm(sender, reply-to, receiver, plan)*, si tous les participants ont attesté de leur adaptation.

Chacun des rôles du protocole de la figure 6.13 est ensuite spécifié sous forme d'une machine à états (cf. fig. 6.14). Les messages correspondent alors à des signaux émis ou reçus par la machine à états et les états permettent d'attendre la réception de ces signaux. Chaque rôle est tenu par un effecteur d'un CMA. Cet effecteur a son comportement décrit par la machine à états correspondant au rôle. L'état `AdaptationRôle` est alors raffiné par la machine décrivant le plan d'adaptation. L'effecteur implémente un contexte fonctionnel propre à ce plan.

Notons qu'un CMA ne peut être impliqué que dans une seule conversation à la fois. Cela est garanti par le fait qu'un seul effecteur ne peut être actif à la fois dans un CMA (cf. section 6.1.3). Ainsi, les adaptations concurrentes sont évitées dans le but de garantir la cohérence globale du système.

6.3.2.3 Exemple

Nous considérons un système dans lequel un composant diffuse des messages à deux autres composants. Pour des raisons de sécurité, les messages envoyés ont besoin d'être cryptés alors qu'ils ne l'étaient pas. Les trois composants ont besoin de se coordonner afin que l'émetteur n'envoie des messages cryptés que lorsque les récepteurs sont prêts à les décrypter. Le composant émettant les messages initie alors le protocole d'adaptation et les deux autres y participent.

La figure 6.15 montre les plans d'adaptation réalisés dans le cadre de ce protocole. L'effecteur `EncryptEffector` implémente le rôle d'initiateur : il raffine l'état `AdaptationInitiateur` avec le plan de la partie supérieure de la figure et donne la valeur `DecryptPlan` à l'attribut *plan*. L'effecteur `DecryptEffector` implémente quant à lui le rôle de participant : il raffine l'état `AdaptationParticipant` avec le plan d'actions reçu en paramètre de la requête, c.-à-d. celui de la partie inférieure de la figure. La politique d'adaptation des CMAs participants permet de mettre en relation le plan sollicité lors de la requête d'adaption avec l'effecteur prenant en charge le protocole.

Au départ, la politique d'adaptation de l'émetteur déclenche l'effecteur `EncryptEffector` qui initie le protocole. L'initiateur contacte ensuite les participants via la liste des abonnés au répartiteur de l'émetteur. La réception du signal `Request` déclenche chez les participants l'instanciation de l'effecteur `DecryptEffector` prenant en charge le protocole. Si la requête est acceptée, le comportement décrit dans le plan est composé dans l'état de sous-machine `AdaptationParticipant` pour être réalisé : le CMA est adapté pour réaliser le décryptage des messages reçus. En parallèle, l'initiateur réalise le plan `EncryptPlan` composé dans l'état de sous-machine `AdaptationInitiateur` : le CMA est adapté pour effectuer le cryptage du contenu des messages qu'il envoie. Lorsque tous les participants ont attesté via le signal `InformDone` de la réussite de leur plan et que l'initiateur a terminé le sien, ce dernier termine le protocole en envoyant aux participants le signal `Confirm`. Pendant l'exécution du protocole, tous les autres messages reçus par les CMAs sont mis en file d'attente. Ils seront traités de façon différée à la fin du protocole.

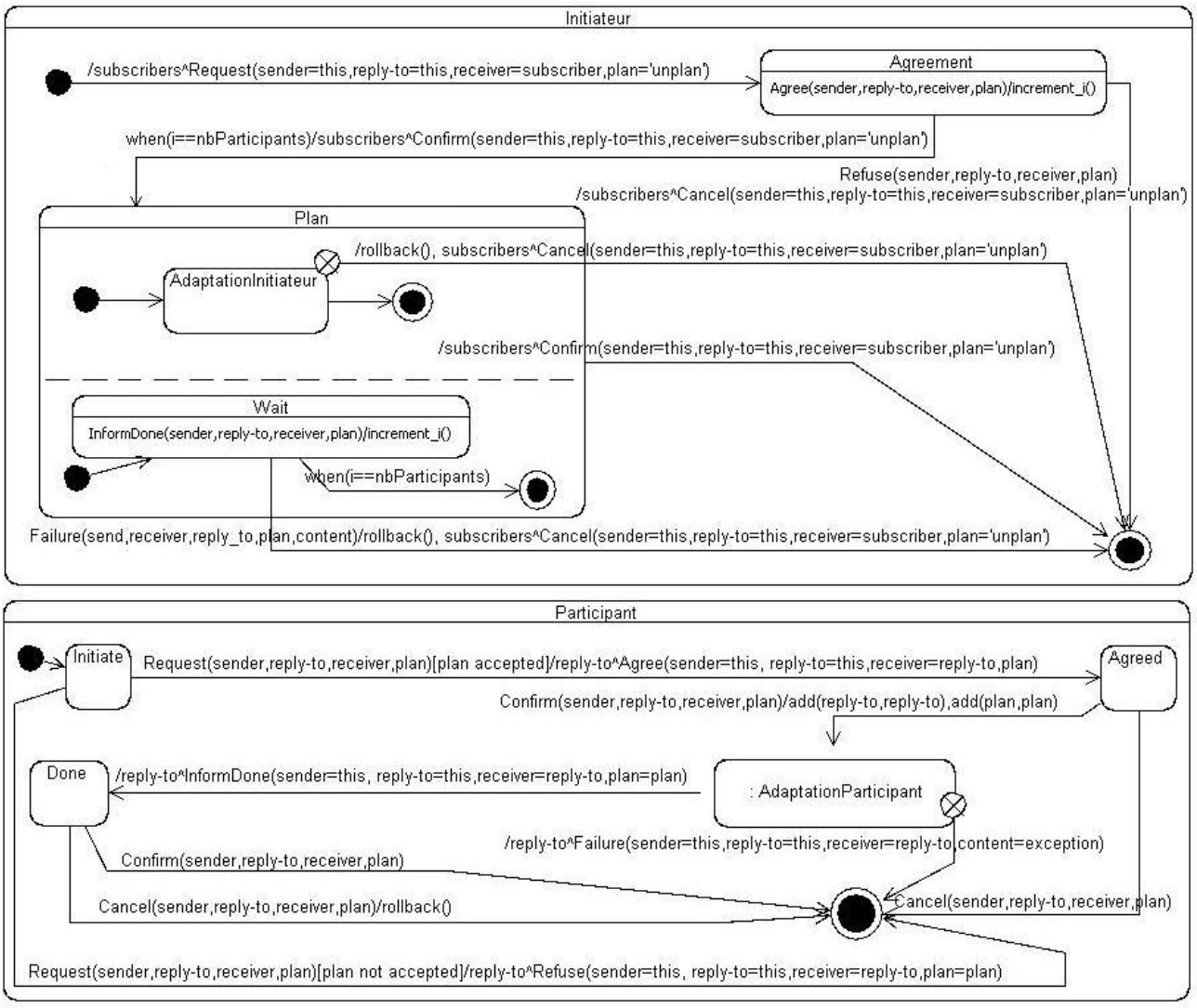

FIGURE 6.14 – Machines à états correspondant aux rôles du protocole d'adaptation

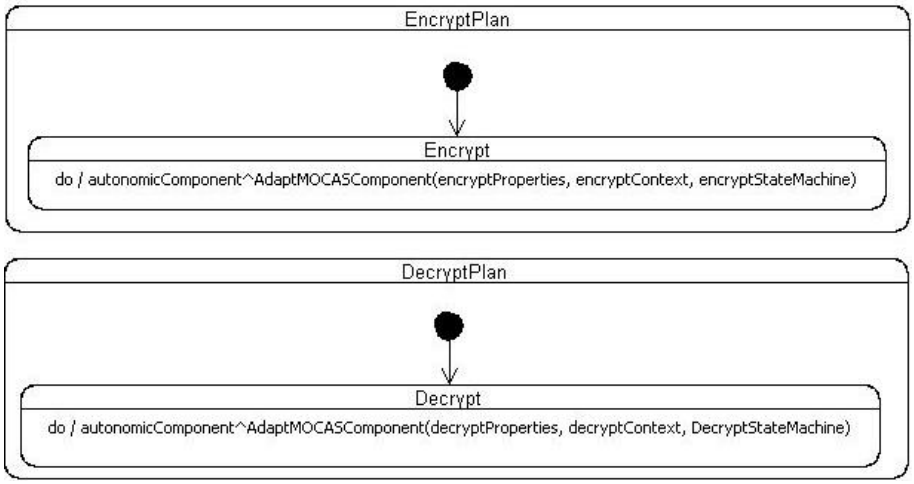

FIGURE 6.15 – Plans d'actions de l'initiateur et des participants du protocole d'adaptation

6.4 Synthèse

Nous avons vu au cours des trois chapitres précédents comment le modèle de composants MOCAS a été construit (chapitre 4) dans le but de simplifier la construction des systèmes adaptatifs (chapitre 5) pour réaliser des propriétés de l'informatique autonomique (chapitre 6).

MOCAS s'appuie pour cela sur la métamodélisation, le langage de modélisation UML et les mécanismes des machines à états. Il utilise les éléments du métamodèle UML et les associations préexistantes. Le comportement d'un composant est décrit avec une machine à états. La réification de la structure et du comportement des composants permet de donner aux composants conscience d'eux-mêmes (cf. section 2.1). Tous ces éléments deviennent alors manipulables.

Un composant MOCAS devient adaptable lorsqu'il est composé verticalement avec le conteneur MOCAS. Il devient alors un sous-composant du conteneur. Ce dernier s'assure de la cohérence de l'adaptation demandée et de sa réalisation. La technique d'adaptation consiste à remplacer dynamiquement la machine à états et/ou le contexte fonctionnel du sous-composant détenu par le conteneur. Le conteneur réalise la séparation de l'aspect non-fonctionnel de l'adaptation avec les aspects fonctionnels assurés par le sous-composant. Il réalise aussi l'objectif de transparence : le concepteur du sous-composant n'a pas à se soucier des mécanismes d'adaptation, le sous-composant est adaptable sans interrompre ses services et les clients liés au conteneur n'ont pas à interrompre leurs requêtes.

Le conteneur est spécialisé dans une version autonomique afin de construire une boucle de contrôle surveillant le sous-composant. Des capteurs sont chargés de recueillir des informations sur l'environnement et sur le fonctionnement de ce sous-composant. Un évaluateur analyse ces informations à l'aide de la politique qu'il embarque. Cette politique est spécifiée avec une machine à états et déclenche des effecteurs. Les effecteurs réalisent des actions d'adaptation sur le sous-composant. Un répartiteur assure la coordination de l'adaptation avec les autres composants autonomiques du système. La coordination est structurée par des protocoles d'interaction

exprimés avec des machines à états.

MOCAS propose ainsi une approche uniforme pour la construction d'un système autonomique en s'appuyant sur un unique modèle de composants et sur un unique formalisme. Le système est ouvert : de nouveaux comportements et de nouvelles politiques d'adaptation sont délivrables aux composants à n'importe quel moment de l'exécution.

Troisième partie

Mise en œuvre de l'approche et évaluation

Chapitre 7

Outils pour la réalisation de composants MOCAS

Sommaire

Un ensemble d'outils logiciels a été développé afin de couvrir le cycle de réalisation d'un système à base de composants MOCAS [143]. Ces outils facilitent la conception, le développement, l'exécution et l'administration du système. Ils reposent sur l'utilisation unique du langage UML et des machines à états. Ces différents outils, en s'appuyant directement sur les modèles et en exécutant directement les machines à états issues de la phase de conception, limitent les risques d'erreurs liées aux interventions humaines. Il n'y a ainsi pas de « distance » entre la spécification et l'implémentation du système.

La section 7.1 présente la méthodologie de conception et de développement des composants MOCAS. La section 7.2 détaille le moteur d'exécution MOCASEngine destiné à exécuter dans un environnement Java les machines à états embarquées dans les composants MOCAS. La section 7.3 décrit la plate-forme de déploiement et d'administration d'un système MOCAS.

7.1 MOCAS4TopCased : un plugiciel pour la conception et le développement

7.1.1 Conception

Un composant MOCAS est spécifié à l'aide du langage de modélisation UML. Les éléments de sa structure sont identifiés par l'application des stéréotypes du profil de MOCAS (cf. fig. 4.2). Son comportement est décrit avec une machine à états. De ce fait, les composants MOCAS sont faits pour être conçus dans des ateliers de génie logiciel (AGL, ou CASE tools) supportant l'ingénierie des modèles.

L'environnement TopCased [19] [144] a été utilisé pour la spécification des composants MOCAS. TopCased est un projet libre dédié principalement à la conception de systèmes embarqués pour l'aéronautique. Il est conçu pour favoriser l'utilisation des modèles dans le processus de réalisation de ces systèmes. Il repose sur l'environnement Eclipse. Ce dernier est déjà largement tourné vers l'ingénierie des modèles grâce à son support d'EMF (*Eclipse Modeling Framework*) [145] et son implémentation du métamodèle UML. TopCased ajoute à Eclipse de nombreux plugiciels permettant la spécification de contraintes OCL, la génération de code, la vérification de modèles...

Les profils UML de MOCAS ont été saisis et validés dans TopCased. De plus, un modèle générique – le *template* – spécifiant la structure et le comportement canoniques d'un composant MOCAS a été saisi comme base à la spécification d'autres composants (cf. fig. 7.1). A partir du *template*, le concepteur spécifie les attributs du composant, les actions internes ainsi que les signaux émis et reçus. Il complète ensuite la machine à états en rajoutant les états, les transitions entre ces états ainsi que les invariants et les gardes. Le modèle est ensuite sauvegardé au format XMI [146].

7.1.2 Développement

Un plugiciel a été développé pour TopCased afin de générer le code Java d'un composant MOCAS ainsi qu'une archive `jar` contenant les fichiers nécessaires au déploiement du composant. A partir du modèle, le plugiciel génère le squelette de la classe correspondant au contexte fonctionnel du composant et des classes correspondant aux contraintes, c.-à-d. aux invariants et aux gardes. Le développeur peut alors compléter les différentes méthodes correspondant aux actions internes et à l'évaluation des contraintes. Une fois le code des différentes classes complété, le plugiciel permet de créer une archive `jar` contenant le fichier XMI ainsi que les classes.

19. http ://www.topcased.org

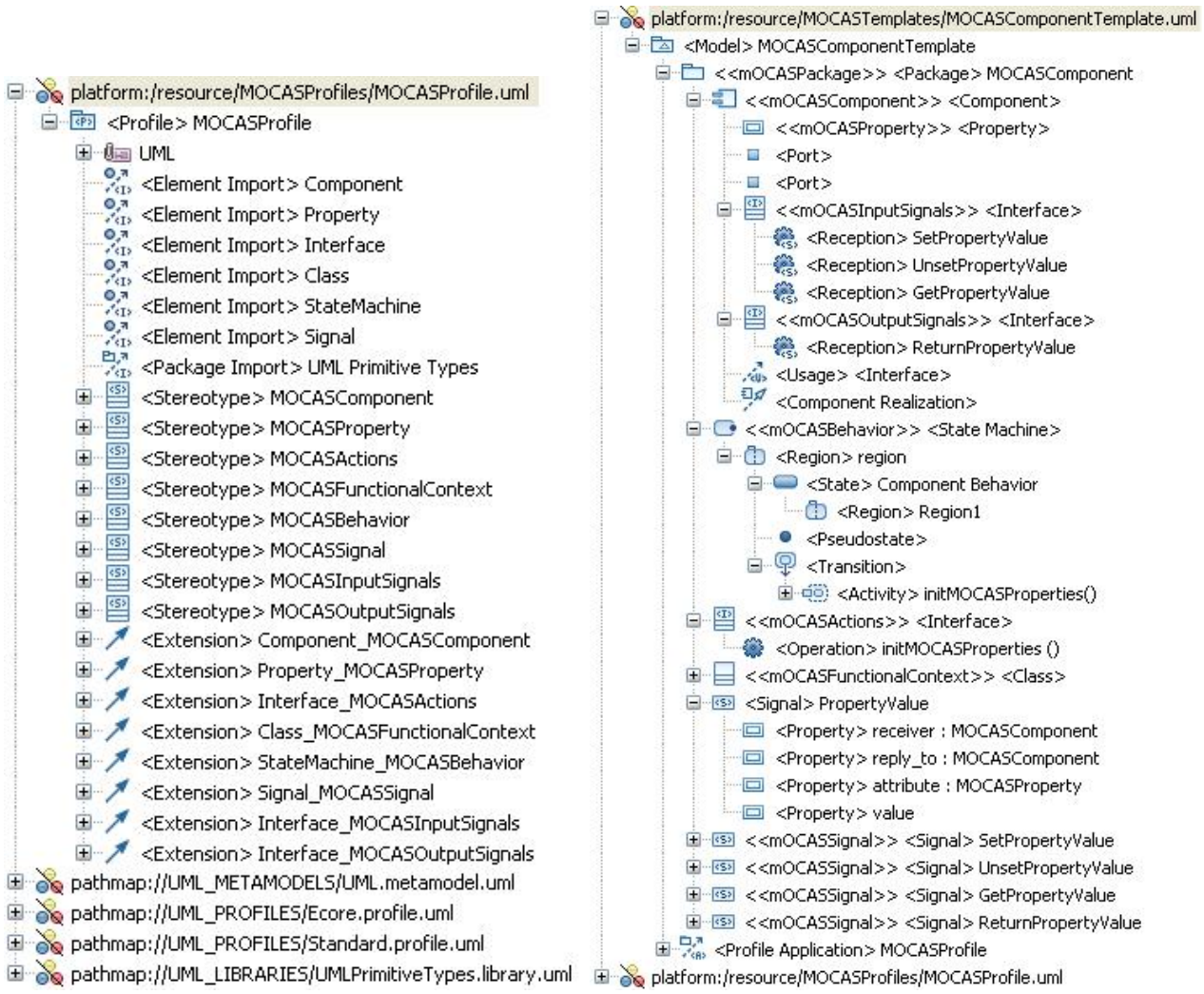

FIGURE 7.1 – Spécification du profil et du *template* MOCAS dans TopCased

7.2 MOCASEngine : un moteur pour l'exécution de machines à états UML

Le moteur MOCASEngine[20] a été développé afin d'exécuter les machines à états UML. Il s'appuie sur l'expérience acquise lors du développement du moteur PauWare[21] réalisé au sein du laboratoire d'informatique de l'université de Pau et des Pays de l'Adour (LIUPPA). Il se présente sous la forme d'une librairie Java et repose sur l'*Eclipse Modeling Framework* (EMF).

7.2.1 Sémantique

La spécification UML des machines à états [1] comporte de nombreux points de variation sémantique ainsi que de nombreuses ambiguïtés (par exemple celles concernant les états historiques et les points de connexion entre une machine et ses sous-machines [147]). L'implémentation d'un moteur d'exécution requiert de fixer ces différents points [148].

Nous avons adopté dans le moteur MOCASEngine la sémantique suivante :

- une région doit posséder un pseudo-état initial désignant un état de cette région (cf. point de variation p. 551 [1]), c.-à-d. que l'activation d'un état composite implique l'activation de ses régions et de leur sous-état par défaut. Cette sémantique est nécessaire pour permettre le raffinement dynamique d'un état simple ;
- les événements sont enregistrés dans une file de type FIFO propre à chaque machine. Les événements d'achèvement sont prioritaires et sont donc systématiquement insérés en tête de file (la politique de priorité est libre, cf. p. 565 [1]). Ainsi, les transitions dont l'état-source est un pseudo-état initial et n'ayant pas de déclencheur, sont traitées par le moteur de la même manière que les transitions d'achèvement[22]. L'activation des sous-états directs d'un état composite nécessite donc un cycle *run-to-completion* ;
- les événements n'ayant pas de déclencheurs sont ignorés par le moteur (cf. points de variation p. 456 et p. 538 [1]). Néanmoins, la réception d'un événement non interprétable peut constituer une violation du contrat d'interaction entre les composants. Les suites à donner à cette violation sont spécifiables dans la politique du composant autonomique dont la machine a ignoré l'événement ;
- si deux transitions peuvent être déclenchées à partir du même état ou de deux états de trouvant sur le même chemin de la configuration active, alors la machine à états est mal formée car indéterministe (cf. point de variation p. 566), une exception est alors levée ;
- l'ordre d'exécution des actions décrivant les effets d'une transition correspond à l'ordre dans lequel ces actions apparaissent sur la transition. Ainsi, la séquence des actions est significative. Ceci permet d'utiliser les sorties d'une action comme entrées de l'action suivante ;
- les destinataires d'un signal diffusé (`BroadcasSignalAction`) correspondent aux valeurs

20. http ://mocasengine.sourceforge.net/
21. http ://www.pauware.com/
22. Rappelons qu'une transition d'achèvement n'a pas de déclencheur explicite et est franchie automatiquement lorsque l'activité de son état-source se termine.

d'un attribut multivalué [23]. Cet attribut est détenu par la classe qui est le contexte de la machine à états réalisant l'action de diffusion (cf. point de variation p. 242 [1]).

Par ailleurs, MOCAS et UML distinguent quatre types de visibilité pour les attributs. Si les types « Lecture/Ecriture » et « Aucun accès » des attributs MOCAS ont respectivement une sémantique équivalente aux types UML `public` et `private`, il n'y a pas d'équivalence directe pour les deux autres types (« Lecture » et « Ecriture »). Un attribut est décrit par la métaclasse `Property`. Cette métaclasse dispose de l'attribut booléen `isReadOnly` qui, s'il est vrai, rend l'attribut décrit par la métaclasse seulement accessible en lecture. Néanmoins, afin que le moteur gère uniformément les différents types de visibilité, nous avons établi une correspondance syntaxique (et non sémantique) entre les visibilités UML et MOCAS. Cette correspondance est consignée dans le tableau 7.1. Elle nous est utile afin de distinguer les attributs dont la valeur peut être « configurée » par un composant associé (par exemple une valeur d'un *time-out* ou bien la connection d'un port à un autre), de ceux dont la valeur n'est modifiable que par le composant qui les possède.

MOCAS	UML
Lecture/Ecriture	public
Aucun accès	private
Lecture	protected
Ecriture	package

TABLE 7.1 – Correspondance des visibilités des propriétés MOCAS et UML

7.2.2 Principes de fonctionnement

Un de nos objectifs est de réduire la distance entre la conception, le développement, l'exécution et l'administration des systèmes. Nous souhaitons pour cela conserver le même formalisme au cours de ces différentes phases. Le modèle de composants MOCAS étant un modèle réflexif s'appuyant sur UML, nous avons décidé de réutiliser les modèles de conception en les interprétant. Néanmoins, UML ne permettant pas de spécifier la logique opérationnelle des actions internes, un langage de programmation reste nécessaire pour implémenter ces dernières dans le contexte fonctionnel des composants. La plupart des outils étant déjà développés en Java, nous avons conservé ce langage pour la réalisation des composants MOCAS. MOCASEngine est ainsi une approche hybride entre l'interprétation du modèle spécifié et l'exécution du code compilé [39].

Le moteur d'exécution s'appuie sur les modèles d'instances UML (cf. fig. 7.2). La métaclasse `InstanceSpecification` permet de décrire une instance du classifieur auquel elle est associée (cf. association `classifier`). De plus, un `slot` est créé (cf. association `slot`) pour chaque attribut du classifieur (cf. association `definingFeature`) afin d'en retenir la valeur (cf.

23. Un attribut multivalué est une collection d'élements du type de l'attribut.

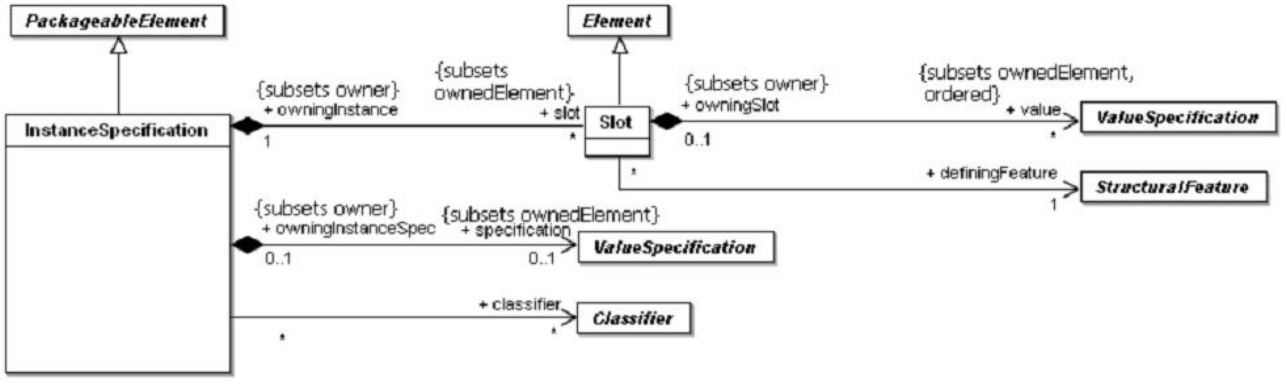

FIGURE 7.2 – Métaclasses UML liées aux modèles d'instances et supportées par le moteur MOCASEngine [1]

association `value`). L'instance est aussi associée à une spécification de valeur (cf. association `specification`) qui décrit la manière dont l'instance est construite.

Un composant, une classe, une machine à états et un signal sont des classifieurs UML pour lesquels une spécification d'instance peut être créée. Le lien `specification` permet d'associer l'instance d'un composant à l'instance de son contexte fonctionnel et cette dernière à l'instance de son comportement. Nous avons spécialisé la métaclasse `InstanceSpecification` pour chaque type de classifieur. Le moteur d'exécution utilise alors la classe spécialisée correspondant à l'élément MOCAS considéré. La spécification d'instance de la machine à états du composant est par exemple chargée d'instancier la configuration active par défaut de la machine à états.

La phase de déploiement d'un composant MOCAS consiste alors à créer le modèle d'instances correspondant aux différents éléments du composant. Ainsi, à partir du fichier `jar` contenant le composant MOCAS à instancier, le moteur charge le fichier XMI décrivant le composant. La structure d'objets EMF est ainsi créée à l'identique de celle manipulée par l'environnement TopCased au moment de la conception du composant. Suivant le stéréotype qui lui est appliqué, le composant est déployé directement ou composé verticalement avec le conteneur pour l'adaptation ou le conteneur autonomique.

De la même manière qu'un classifieur est mis en correspondance avec une spécialisation d'une spécification d'instance, une action est mise en correspondance avec un exécuteur. Ainsi, l'exécuteur d'une invocation d'opération (`CallOperationAction`) : 1) effectue une instrospection de la classe d'implémentation réalisant le contexte fonctionnel afin de trouver la méthode Java correspondant à l'opération à invoquer, 2) injecte les valeurs des paramètres de l'opération (ces valeurs proviennent des *slots* associés aux spécifications d'instance du signal déclenchant l'action et du composant possédant la machine à états). L'exécuteur d'un envoi de signal (`SendSignalAction`) invoque la méthode `onSignal(InstanceSpecification instanceSignal)` sur la spécification d'instance d'un port de sortie. Les correspondances entre exécuteurs/méta-élements et spécification d'instance/méta-éléments sont écrites dans un fichier de configuration du moteur. Il est ainsi aisé de changer le fonctionnement du moteur pour, par exemple, envoyer les signaux en utilisant un exécuteur reposant sur la bibliothèque JMS (*Java Message Service*).

7.2.3 Fonctionalités et limites

Les éléments de modélisation du métamodèle UML supportés par MOCASEngine correspondent aux métaclasses décrivant la structure des composants et des machines à états ainsi qu'aux actions associées aux effets des transitions (modification de la valeur des attributs, invocation d'opération, envoi de signal, cf. fig. 4.3 et fig. 4.4). Les principaux éléments non supportés par le moteur sont :

- certains pseudo-états correspondant aux transitions dites « composées » (*join, fork, junction, terminate*) ;
- la redéfinition de machines à états.

Le mécanisme de redéfinition des machines à états est à considérer dans des travaux futurs car il permettrait de maintenir l'historique des adaptations du comportement d'un composant : les différentes versions de la machine à états seraient ainsi « chaînées » et exploitées lors d'un repli.

Par ailleurs, le moteur ne garantit pas l'ordre de réception des signaux envoyés d'un composant à un autre. Ainsi, si un composant `A` envoie un signal `s2` à un composant `B` après lui avoir envoyé un signal `s1` (cf. fig. 7.3, machine à états `A`), `s2` peut très bien être reçu avant `s1`. Si le medium de communication entre les deux composants ne garantit pas non plus l'ordre de réception, le concepteur peut utiliser des signaux différés. Ainsi, sur la figure 7.3, la machine `B` précise que si le signal `s2` est reçu alors que l'état `B1` est actif, son traitement est différé. Lorsque l'état `B2` sera actif, `s2` sera traité et conduira au déclenchement de la transition sortante. Sans ce mécanisme, `s2` serait traité dans `B1` : il ne déclencherait aucune action et serait ignoré.

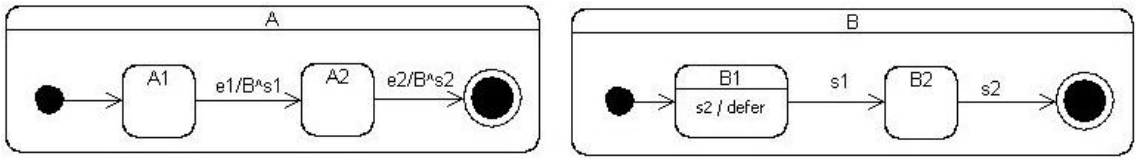

FIGURE 7.3 – Gestion d'un ordre aléatoire de réception de signaux

Enfin, MOCASEngine s'appuie sur l'implémentation du métamodèle UML réalisée avec EMF (*Eclipse Modeling Framework*). EMF est un cadriciel complexe, destiné à un environnement de développement et non d'exécution. De ce fait, il n'est pas soumis aux mêmes contraintes de performance. Nous avons donc implémenté la partie du métamodèle UML [24] qui nous était nécessaire en nous restreignant à l'environnement Java pour appareils mobiles (J2ME). Nous avons déjà eu des résultats dans cet environnement avec le moteur PauWare [149]. Le moteur MOCASEngine est quant à lui actuellement en cours de portage.

24. http ://uml2forjava.sourceforge.net/

7.3 MOCASA : une plateforme pour le déploiement et l'administration

Nous avons développé la plateforme MOCASA (cf. fig. 7.4) afin de tester les composants MOCAS et les assemblages utilisant ces composants. MOCASA permet pour cela de déployer des composants MOCAS et d'effectuer des tâches d'administration.

7.3.1 Déploiement

MOCASA permet de déployer les composants contenus dans un fichier `jar` ainsi qu'un assemblage de composants décrit avec un modèle de structure composite (cf. fig. 8.4). Lorsqu'un `jar` est sélectionné, tout son contenu est parcouru afin d'identifier les fichiers `.uml` contenant un composant MOCAS ainsi que ceux contenant un classifieur stéréotypé `MOCASArchitecture`. L'administrateur choisit alors le nombre d'instances qu'il souhaite pour un type de composant ou bien l'architecture qu'il souhaite déployer. Il peut très bien instancier le conteneur pour rendre adaptable un composant qui n'a pas été spécifié comme tel (c.-à-d. seulement stéréotypé `MOCASComponent`). L'administrateur a enfin la possibilité de réaliser les connexions entre les composants de façon dynamique afin de créer un assemblage sur mesure.

7.3.2 Administration des composants

La plateforme offre la possibilité, pour chaque composant instancié, d'observer les états actifs de la machine à états (cf. fig. 7.4, fenêtre `MOCASBehavior`), de voir les valeurs courantes des attributs (cf. fenêtre `MOCASProperties`) et de suivre les signaux reçus et les transitions franchies (cf. fenêtre `Tracing`).

En plus de l'observation, MOCASA permet de contrôler le comportement d'un composant. Il permet à l'administrateur d'envoyer des signaux à un composant (cf. fenêtre `MOCASInputSignals` afin de modifier la valeur des attributs, ou bien afin de déclencher des adaptations (pour délivrer des mises à jour ou de nouvelles politiques). La fenêtre `MOCASBehavior` permet à l'administrateur d'activer par un double clic une configuration d'états particulière (pour par exemple remettre en cohérence le comportement d'un composant avec un autre). Dans ce cas-là, les actions associées aux états activés ne sont pas effectuées et l'administrateur a la charge d'assurer la cohérence des valeurs des attributs.

7.4 Synthèse

Avec les différents outils présentés, nous couvrons un ensemble d'étapes du cycle de vie d'un système MOCAS :

1. Conception : la plateforme TopCased permet de spécifier les composants MOCAS en respectant le métamodèle UML et le profile de MOCAS ;
2. Développement : le module d'extension *MOCAS4TopCased* permet de générer le squelette du contexte fonctionnel et des contraintes de chaque composant ;

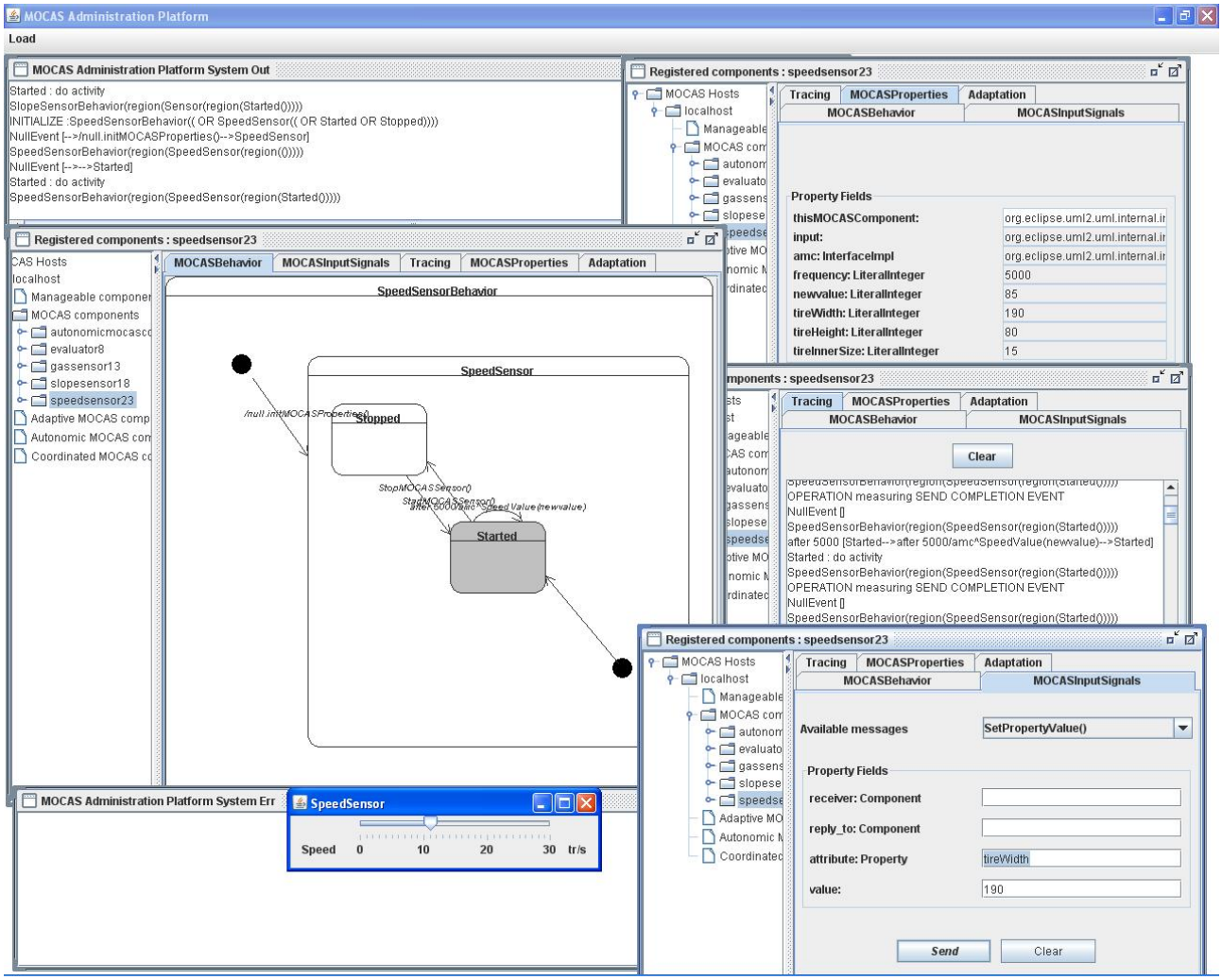

FIGURE 7.4 – Plateforme d'administration MOCASA

3. Test : le moteur MOCASEngine permet d'exécuter la spécification de chaque composant et d'en vérifier le comportement ;
4. Mise en production : une fois que le développeur a complété le code des classes générées, le module *MOCAS4TopCased* permet de packager chaque composant dans des fichiers `jar` indépendants ;
5. Déploiement : la plateforme MOCASA permet de déployer les composants MOCAS, de les paramétrer et de les assembler ;
6. Maintenance : la plateforme MOCASA permet de mettre à jour un composant en lui délivrant une nouvelle machine à états, un nouveau contexte fonctionnel, de nouvelles propriétés.

Chapitre 8

Mise en œuvre

Sommaire

Le modèle de composants MOCAS permet de construire des systèmes autonomiques à base de composants logiciels. Pour cela, il promeut l'utilisation du langage UML et, plus particulièrement, des modèles UML de machines à états. Ces derniers tiennent une place prépondérante tout au long du cycle de vie d 'un système MOCAS. Nous l'illustrons dans ce chapitre en présentant la réalisation d'un système permettant le contrôle d'une boîte de vitesse robotisée. Ce système supporte son auto-reconfiguration en modifiant dynamiquement son mode de fonctionnement ainsi que son auto-réparation en remettant en cohérence la configuration active d'un composant dont un invariant est violé. Nous terminons ce chapitre par une évaluation de l'impact sur la mémoire et sur le temps d'exécution de l'utilisation des modèles et d'une approche par conteneur.

8.1 La boîte de vitesses robotisée

8.1.1 Description

Issue du monde de la course automobile, la boîte de vitesse robotisée [150] réalise le mariage de la boîte manuelle traditionnelle et de la boîte automatique. Elle se retrouve chez la plupart des constructeurs, sous l'appellation Sensodrive chez Citroën, Quickshift chez Renault, DSG

chez Volkswagen... Le but de cette technologie est d'allier les avantages des boîtes manuelles – un meilleur rendement, une excellente robustesse, un coût de fabrication réduit et le plaisir de la conduite sportive – à ceux des boîtes automatiques – le confort d'utilisation, un changement de rapport sans à-coups et sans rupture de traction, l'économie de carburant. De plus, cette boîte de vitesse libère de l'espace dans l'habitable car elle ne nécessite pas de pédale d'embrayage et occupe un volume réduit. Elle comporte donc :

- un mode automatique, dans lequel la boîte change les rapports au moment le plus opportun en fonction des conditions de conduite (vitesse du véhicule, pente de la route, enfoncement de l'accélérateur...) ;
- un mode manuel, dans lequel le conducteur peut changer les rapports lui-même et jouer avec le sur/sous-régime du moteur. Le conducteur utilise pour cela des boutons, palettes ou le levier classique.

8.1.2 Conception

Notre boîte de vitesse robotisée utilise [151] :

- un composant `SlopeSensor` rapportant l'inclinaison du véhicule sous forme d'un pourcentage positif (si l'avant du véhicule est plus haut que l'arrière) ou négatif (si l'arrière du véhicule est plus haut que l'avant) ;
- un composant `SpeedSensor` rapportant la vitesse du véhicule, en kilomètre par heure, et qui se base sur le nombre de tours effectué par la roue ;
- un composant `GasSensor` rapportant l'enfoncement de la pédale d'accélérateur sous forme d'un pourcentage ;
- un composant `Control` permettant au conducteur d'activer soit le mode manuel, soit le mode automatique de la boîte ;
- un composant `GearBox` qui permet de déclencher le changement de rapport d'une boîte à six vitesses.

Chaque composant est spécifié dans TopCased en utilisant le *template* approprié au type de composant. Pour le composant `GearBox`, nous utilisons le *template* correspondant à un composant MOCAS (cf. fig. 7.1, partie droite). La figure 8.1 présente la machine à états de ce composant. Elle comprend les états simples `Park`, `Rear`, `Neutral` et l'état composite `Drive`, correspondant respectivement au frein de parking, à la marche arrière, au point mort et à la conduite en marche avant. Les signaux `Up` et `Down` sont respectivement émis lorsque le conducteur actionne le levier de vitesses d'une position vers le haut ou vers le bas (cf. fig. 8.2). Lorsque l'état `Drive` est activé, le composant sélectionne le rapport le plus approprié en fonction des valeurs des attributs `gas`,`slope` et `speed` correspondant respectivement à l'enfoncement de l'accélérateur, à l'inclinaison et à la vitesse du véhicule. Ces valeurs proviennent des signaux émis par les différents capteurs. Lorsqu'une nouvelle valeur est affectée à un de ces attributs, un événement de changement (matérialisé par le mot `when` sur la figure) se produit et conduit à l'évaluation de la fonction `f(speed,slope,gas)`. En fonction de la valeur retournée par cette fonction, la transition entre deux vitesses se fait ou non. L'activation d'un sous-état de l'état `Drive` conduit à la réalisation de son activité `do` effectuant alors le changement *physique* de rapport.

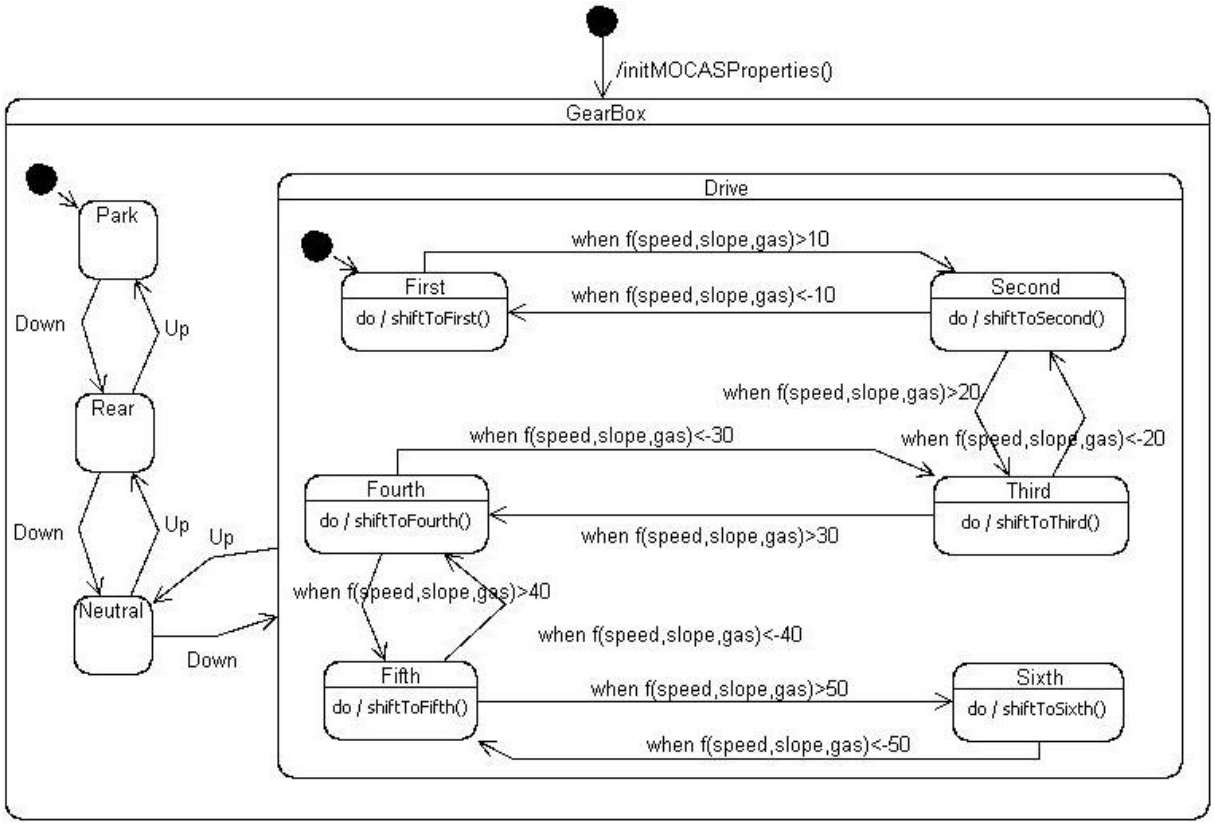

FIGURE 8.1 – Comportement du composant `GearBox`

FIGURE 8.2 – Exemple de levier de boîte automatique

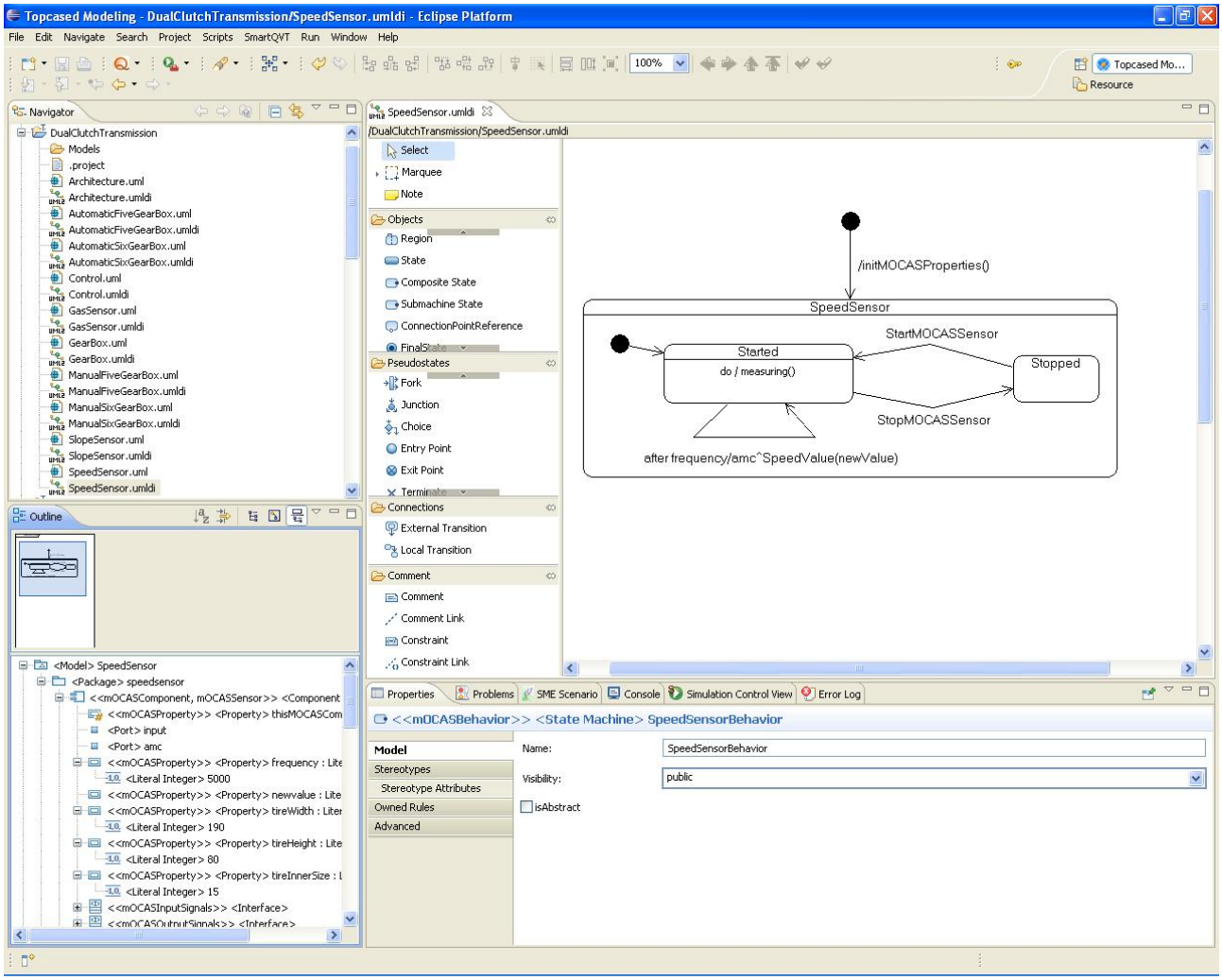

FIGURE 8.3 – Spécification du capteur de vitesse dans TopCased

La figure 8.3 présente une capture de l'environnement TopCased avec la spécification du capteur de vitesse (`SpeedSensor`). La fenêtre `Navigator` montre la liste des fichiers au format XMI comprenant les spécifications UML des différents composants du système. La fenêtre `SpeedSensor.umldi` montre la machine à états du capteur basée sur le *template* des capteurs proactifs (cf. fig. 6.4, p. 98). Enfin, la fenêtre `Outline` montre une partie des différents éléments UML utilisés. Les attributs du composant ont été mis en avant afin de montrer la spécification de leur valeur par défaut.

8.1.3 Développement

Une fois qu'un composant est spécifié, *MOCAS4TopCased* permet de générer le squelette des classes correspondant au contexte fonctionnel et aux différentes contraintes (garde, invariant,

Pour un pneu de dimension $205/65R15$, la largeur est de $205mm$, le rapport H/L est de 65%, le diamètre intérieur est de 15 pouces.
La circonférence C (en m) du pneu est alors telle que :

$$C = \pi \left(2 * \frac{largeur}{1000} * \frac{rapportH/L}{100} + \frac{diametre * 2.54}{100} \right)$$

$$C = \frac{\pi}{100} \left(2 * \frac{largeur}{10} * \frac{rapportH/L}{100} + diametre * 2.54 \right)$$

$$C = \frac{\pi}{100} \left(\frac{largeur * rapportH/L}{500} + diametre * 2.54 \right)$$

TABLE 8.1 – Calcul de la circonférence d'un pneu

condition des événements de changement). Le listing 8.1 montre le code du contexte fonctionnel du capteur de vitesse. *MOCAS4TopCased* a généré le prototype de la méthode `measuring()`, propre aux capteurs proactifs, en inspectant la liste des opérations définies dans l'interface d'actions. La méthode `initMOCASProperties()` n'ayant pas été redéfinie dans le contexte, les attributs du composant sont initialisés par le moteur avec leur valeur par défaut, c.-à-d. celle issue de la spécification. Le développeur initialise quant à lui des attributs propres à l'implémentation (cf. la `JFrame` sur la ligne 5). Ensuite la méthode `measuring()` est renseignée par le développeur : à partir des paramètres du pneu (cf. tab. 8.1, diamètre de la jante, largeur du pneu et rapport hauteur/largeur) pour lequel le capteur a été configuré et en fonction du nombre de rotations mesuré, la vitesse en kilomètre par heure est calculée et mise à jour (cf. ligne 10, `setValue(...)`). Cette valeur sera par la suite envoyée au CMA (Composant MOCAS autonomique) auquel le capteur sera associé, le composant `GearBox` dans notre cas.

Listing 8.1 – Implémentation du contexte fonctionnel du capteur de vitesse

```
1  public class SpeedSensorContext extends MOCASFunctionalContext{
2      private JFrame _frame;
3      public SpeedSensorContext() throws MOCASException{
4          super();
5          _frame = new SpeedSensorFrame();
6          _frame.setVisible(true);
7      }
8      public void measuring() throws MOCASException{
9          Double circumference = Math.PI * (((Integer)getValueAsObject("tireWidth")).
               doubleValue() * ((Integer)getValueAsObject("tireHeight")).doubleValue() /500D
                + ((Integer)getValueAsObject("tireInnerSize")).doubleValue() * 2.54D) /100D;
10         setValue("newvalue", ((Double)(_frame.getRpm() * 3600D * circumference /1000D)).
               intValue());
11     }
12 }
```

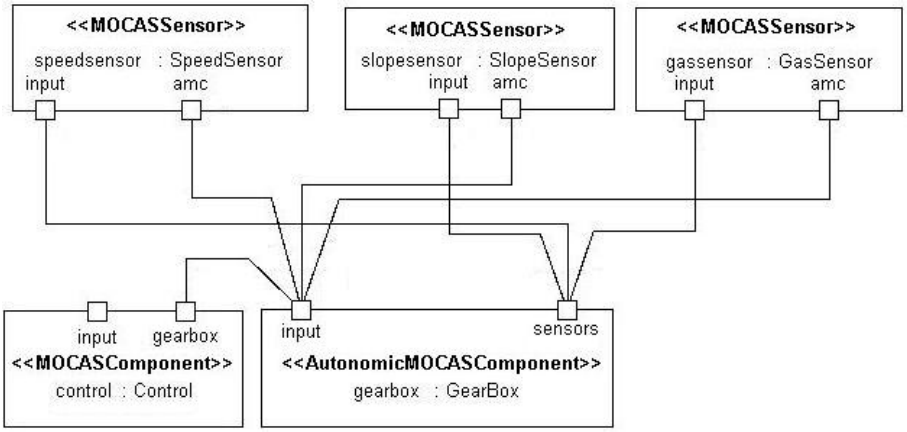

FIGURE 8.4 – Modèle de structure composite de la boîte robotisée

8.1.4 Administration

8.1.4.1 Déploiement

Le moteur est capable de déployer un assemblage à partir d'un modèle de structure composite (cf. fig. 8.4). La structure composite est un classifieur qui comprend des attributs et des connecteurs : chaque attribut correspond à une instance d'un classifieur ; chaque connecteur relie deux instances de classifieurs grâce aux ports que possèdent ces derniers. Dans notre contexte, les classifieurs sont des composants MOCAS.

Le modèle de la figure 8.4 montre l'architecture du système constituant la boîte de vitesse robotisée. Ce système comprend une instance du composant `SpeedSensor`, une instance du composant `SlopeSensor`, une instance du composant `GasSensor`, une instance du composant MOCAS autonomique `GearBox` et une instance du composant `Control`. Les trois instances de capteurs sont reliées au composant autonomique afin de rapporter les valeurs qu'ils mesurent et le composant de contrôle est relié au composant autonomique afin de requérir un changement de mode de fonctionnement. L'administrateur a toujours la possibilité d'instancier, par exemple, d'autres capteurs et de les connecter dynamiquement au CMA.

8.1.4.2 Politique autonomique

Dans sa version actuelle, la boîte de vitesse robotisée est une boite automatique six vitesses. Afin de changer dynamiquement son mode de fonctionnement, nous avons spécifié une politique d'auto-reconfiguration (cf. fig. 8.5). Cette politique spécifie trois modes : le mode manuel (état `Manual Mode`), le mode automatique cinq vitesses (état `Five Gear Mode`) et le mode automatique six vitesses (état `Six Gear Mode`) activé par défaut. Chaque état est connecté par une transition d'achèvement au point de sortie de l'état composite `GearBoxPolicy`. Ce point sera lui-même connecté au point de sortie de l'état `Policy` de la machine de l'évaluateur (cf. fig. 6.5) lorsque la politique sera délivrée au CMA par l'administrateur.

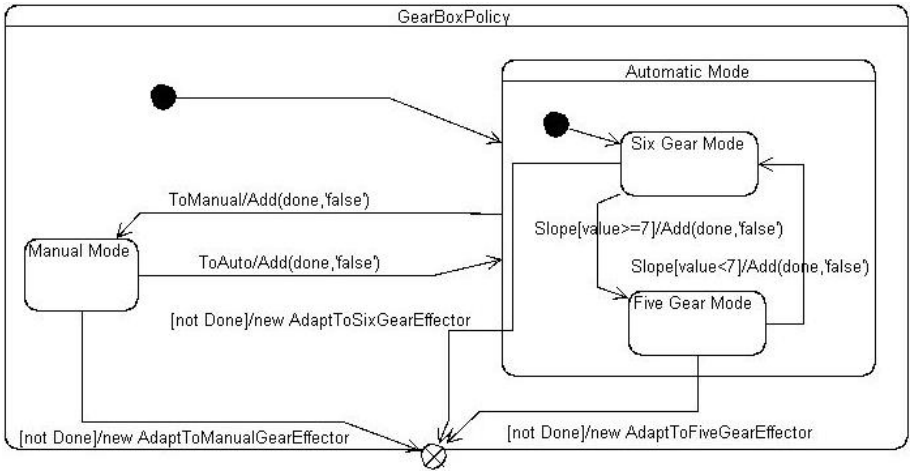

FIGURE 8.5 – Politique d'auto-reconfiguration du composant `GearBox`

Lorsqu'une des transitions d'achèvement est franchie, elle a pour effet de créer l'effecteur spécifié en paramètre de l'action `CreateObjectAction` (`new` sur les transitions). Chacun des trois effecteurs consiste à envoyer au conteneur du composant `GearBox` le signal `AdaptMOCASComponent`. La valeur de l'attribut `behavior` de ce signal dépend de l'effecteur instancié et correspond à une nouvelle machine à états destinée à remplacer celle composée verticalement avec le conteneur. La machine à états du mode manuel (cf. fig. 8.6) fournit notamment le service de changement des vitesses avec les signaux `ToFirst`, `ToSecond`... La réception d'un de ces signaux permet d'activer directement le rapport voulu, indépendamment du précédent rapport sélectionné (par opposition avec le mode automatique dans lequel les rapports sont strictement montés ou descendus).

La politique spécifie de plus le passage du mode six vitesses à un mode cinq vitesses lorsque la pente de la route devient trop forte (cf. transition `Slope[value<=7]`). Dans ce mode, l'étagement de la boîte est modifié afin de permettre au moteur de monter plus haut dans les tours. Ceci se matérialise par un changement des conditions associées aux événements de changement des transitions et de la méthode de calcul de la fonction `f(speed,slope,gas)` précédente (cf. fig. 8.1).

8.1.4.3 Maintenance

Afin d'éviter de forts sous/sur-régimes du moteur, le concepteur du composant `GearBox` décide de rajouter à certains états de la machine des invariants en rapport avec la vitesse du véhicule (cf. tab. 8.2). La nouvelle machine incorporant ces invariants peut alors être délivrée par l'administrateur via l'envoi du signal d'adaptation. Ce signal sera alors mis dans la file d'attente des événements du CMA et interprété entre deux changements de rapport. Si, au moment de la requête, le sixième rapport est selectionné et que la nouvelle machine ne contient que cinq rapports et que l'administrateur a choisi le signal d'adaptation différée, la mise à jour sera appliquée après que le cinquième rapport de la machine courante ait été sélectionné

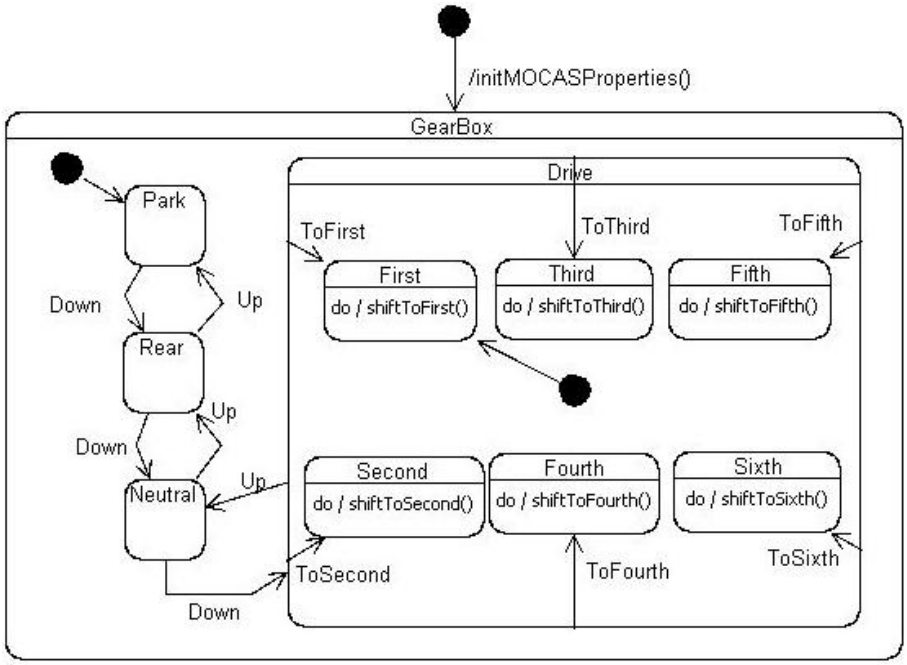

FIGURE 8.6 – Machine à états du mode manuel six vitesses

(conformément à la condition `IsBehaviorConsistent` du conteneur, cf. section 5.1.3).

Ces invariants permettent d'envisager une politique d'auto-réparation de la boîte (cf. fig. 8.7). Cette politique a pour but de sélectionner directement le rapport le plus adapté lorsque l'invariant d'état est violé, c.-à-d. lorsque la vitesse devient supérieure (ou inférieure) à celle tolérée par le rapport courant. La politique est rajoutée en parallèle de la politique d'auto-reconfiguration (c.-à-d. dans une région orthogonale à l'état `GearBoxPolicy`).

Etat	Invariant
`First`	speed>=0 and speed<15
`Second`	speed>=15 and speed<40
`Third`	speed>=40 and speed<70
`Fourth`	speed>=70 and speed<85
`Fifth`	speed>=85 and speed<120
`Sixth`	speed>=120 and speed<180

TABLE 8.2 – Invariants associés aux états du composant `GearBox`

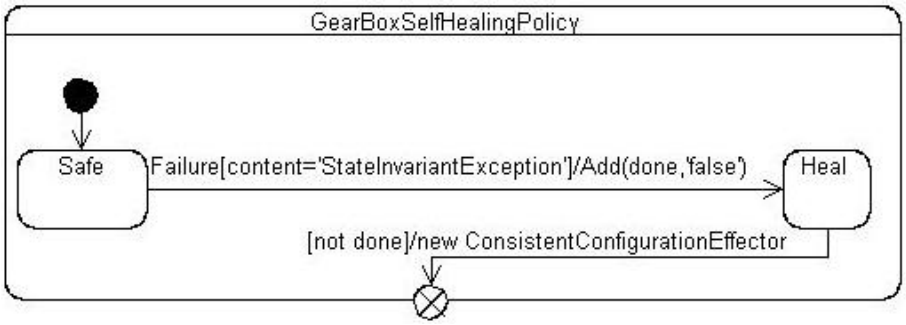

FIGURE 8.7 – Politique d'auto-réparation du composant `GearBox`

8.2 Evaluation quantitative

MOCAS, en interprétant les modèles UML et en utilisant un conteneur pour l'adaptation de chaque composant, peut poser problème dans les environnements où la performance est une préoccupation majeure. Nous avons donc réalisé un ensemble de mesures afin d'évaluer le temps d'exécution et la mémoire requise pour un composant MOCAS. Pour cela, nous avons utilisé le composant `GearBox` dans son mode « automatique à six rapports ». Nous l'avons décliné en version simple (c.-à-d. stéréotypé « MOCASComponent »), en version adaptable (c.-à-d. stéréotypé « AdaptiveMOCASComponent ») et en version autonomique (c.-à-d. stéréotypé « AutonomicMOCASComponent »). De plus, nous nous sommes appuyés sur deux implémentations du moteur :

- JavaSE + EMF : cette implémentation est celle décrite dans la section 7.2. Elle s'appuie sur l'*Eclipse Modeling Framework* (EMF) et s'exécute dans un environnement JavaSE ;
- JavaME + UML2ForJava : cette implémentation a été faite pour l'environnement JavaME, c.-à-d. la version « mobile » de Java. Cet environnement présente une bibliothèque Java limitée qui ne possède pas d'API de réflexion et qui ne dispose, par exemple, que des classes `Vector` et `Hashtable` comme collections disponibles. Nous avons donc implémenté le métamodèle UML à l'aide de cette bibliothèque sous la forme du projet *opensource* UML2ForJava [25]. Comme précisé précédemment, le moteur est actuellement en cours de portage pour cet environnement : seuls les composants MOCAS simples sont exécutables pour le moment.

Les différentes mesures ont été réalisées sur un Pentium D 820 à 2,8GHz avec 2Go de RAM, exécutant la machine virtuelle Java 1.6 sur Windows XP. Afin de ne pas biaiser les résultats des mesures, l'implémentation sous JavaME a aussi été exécutée sur cette machine directement, c.-à-d. sans passer par l'émulateur de périphériques mobiles.

25. http ://uml2forjava.sourceforge.net/

Implémentation	Composant GearBox	Instanciation	Changement d'état
JavaSE + EMF	Simple	1083 ms	2498 μs
	Adaptable	2341 ms	2954 μs
	Autonomique	3555 ms	5320 μs
JavaME + UML2ForJava	Simple	0.220 ms	26 μs

TABLE 8.3 – Temps d'exécution moyen du composant `GearBox`

8.2.1 Temps d'exécution

Nous avons comparé les temps d'instanciation du composant ainsi que le temps pour opérer un changement d'états en fonction de la version et de l'implémentation. Les résultats sont consignés dans le tableau 8.3. Ils révèlent :

- la charge du cadriciel EMF : nous observons un facteur de l'ordre de 5000 pour le temps d'instanciation et de 100 pour le temps de changement d'état d'un composant simple entre les environnements JavaSE et JavaME. En effet, l'instanciation sous EMF nécessite notamment de charger les fichiers des différents métamodèles (celui à la base d'EMF – *ECore*, celui pour les profiles *ECore Profile* –, celui d'UML...) ainsi que les profiles de MOCAS (cf. fig. 4.2 et 6.1). Ces fichiers sont au format XML et necessitent d'être analysés pour recréer la structure d'objets correspondante. L'impact sur le changement d'état est moins important étant donné que les deux implémentations sont plus proches une fois les modèles instanciés ;
- la charge du conteneur : nous observons un facteur de 2,16 entre le temps d'instanciation du composant dans sa version adaptable et dans sa version simple. Ceci s'explique par le fait que la version adaptable nécessite l'instanciation de deux composants simples (le conteneur adaptable et le composant) puis leur composition. De même, nous observons un facteur de 3,28 entre le temps d'instanciation du composant dans sa version autonomique et dans sa version simple. Ceci s'explique par le fait que la version autonomique nécessite l'instanciation de quatre composants simples (le conteneur autonomique, l'évaluateur, le répartiteur et le composant) puis leur composition. Bien que la version autonomique nécessite deux fois plus de composants que la version adaptable, le facteur n'est pas doublé car l'évaluateur et le répartiteur utilise une instance commmune de ECore et du profil de MOCAS, contrairement aux autres composants. Par ailleurs, un changement d'état dans un composant adaptable nécessite 18% de temps supplémentaire par rapport à un composant simple. Ceci s'explique par une augmentation du nombre de niveaux dans la structure de la machine à états (liée à la composition). Un changement d'état dans un composant autonomique nécessite 113% de temps supplémentaire. Ceci s'explique par la nécessité pour le signal entraînant le changement d'état de traverser l'évaluateur puis le répartiteur avant d'être interprété par la machine à états du composant autonomique.

Implémentation	Composant GearBox	Occupation mémoire	Nombres d'instances (tas de 128Mo)
JavaSE + EMF	Simple	4.4 Mo	28 (théorique 29)
	Adaptable	8.8 Mo	13 (théorique 14)
	Autonomique	13.3 Mo	9 (théorique 9)
JavaME + UML2ForJava	Simple	0.040 Mo	3137 (théorique 3200)

TABLE 8.4 – Occupation mémoire du composant `GearBox`

8.2.2 Occupation mémoire

La deuxième série de mesures que nous avons effectuées concerne l'espace mémoire requis pour un composant MOCAS en fonction de sa version. Bien que Java ne dispose pas d'un opérateur `sizeof` comme en C, la mesure de la taille d'un objet n'est pas impossible, même si elle est certes imprécise [26]. Les résultats sont consignés dans le tableau 8.4. Ils révèlent une taille impressionnante pour un unique composant MOCAS mais ils reflètent bien la réalité. En effet, ils ont été confirmés en vérifiant le nombre maximum de composants instanciables en fonction de la taille du tas (*Java heap size*), c.-à-d. jusqu'à obtenir une exception `OutOfMemoryError`.

Nous retrouvons un facteur 2 enre la taille d'un composant adaptable et d'un composant simple, correspondant à l'ajout du conteneur, ainsi qu'un facteur 3 entre le composant autonomique et le composant simple, correspondant à l'ajout de l'évaluateur et du répartiteur. Ceci révèle que l'espace mémoire est majoritairement occupé par EMF. Ces résultats témoignent encore une fois de la lourdeur de EMF.

8.3 Synthèse

Nous avons illustré à travers ce chapitre le cycle de réalisation d'un système à base de composants MOCAS. Nous avons aussi mis en avant l'utilisation de politiques d'auto-reconfiguration et d'auto-réparation afin :

- d'alterner les modes de fonctionnement d'un composant, en remplaçant la machine à états qui définit le comportement de ce composant ;
- de modifier les services disponibles, en activant ou non les services de changement manuel de rapport ;
- de réagir à une violation des contraintes matérielles de fonctionnement d'un rapport de vitesse, en utilisant des invariants sur les états.

Par ailleurs, nous avons aussi révélé les problèmes de performance de notre approche. Notre utilisation du cadriciel EMF n'est pas optimisée, et ce n'était pas là notre but. EMF est un

26. Nous renvoyons le lecteur à l'article de la page `http://www.javaworld.com/javaworld/javatips/jw-javatip130.html` qui documente la manière de procéder pour mesurer la taille d'un objet en Java.

cadriciel qui a été conçu pour une exploitation dans un environnement de conception/développement. De ce fait, notre implémentation actuelle est faite pour du prototypage rapide ainsi que du test. Une exécution performante du modèle de composants MOCAS ne saurait s'appuyer sur EMF et requerrait donc une étape de génération de code vers une implémentation plus performante (comme celle sous JavaME). Notre but étant d'éviter justement cela, nous préconisons l'utilisation d'un environnement de conception necessitant moins de ressources et reposant sur une autre implémentation du métamodèle UML. Néanmoins, l'utilisation d'EMF, dans sa stricte implémentation d'UML, nous permet de valider notre approche par les modèles.

Conclusion et perspectives

1 Bilan et contributions

Dans cette thèse, nous nous sommes intéressés à la conception de systèmes autonomiques, c.-à-d. de systèmes ayant des capacités d'auto-gestion. Pour cela, nous nous sommes concentrés sur la capacité d'auto-adaptation, que nous estimons primordiale pour la réalisation de ces systèmes. Alors que les approches actuelles se concentrent sur l'adaptation structurelle des systèmes, nous avons proposé de mettre l'accent sur l'adaptation comportementale des entités constituant le système.

Pour la modularité qu'ils permettent et pour le niveau d'abstraction qu'ils offrent, nous nous sommes appuyés respectivement sur les composants logiciels et sur les modèles. Nous avons donc proposé le modèle de composants MOCAS. MOCAS permet de construire un système adaptatif ayant des capacités autonomiques en utilisant les composants logiciels et le langage de modélisation UML. Il propose une approche uniforme de la conception à l'administration de ces systèmes en mettant au cœur de l'approche les modèles de machines à états UML.

Ainsi, un composant MOCAS est spécifié avec le langage de modélisation UML. Sa structure est contrainte par un profil UML et son comportement est décrit par une machine à états. Un composant MOCAS est composable horizontalement grâce à des ports de connexions et verticalement grâce aux états de sa machine.

Pour devenir dynamiquement adaptable, un composant MOCAS est déployé dans un conteneur dédié respectant le modèle MOCAS. Le conteneur est ainsi composé verticalement avec le composant qu'il encapsule. Il est chargé de vérifier et de valider les adaptations demandées au composant. Une adaptation consiste principalement en la modification de la machine à états du composant, mais elle permet aussi de remplacer les fonctions utilisées par le composant pour traiter les données.

Pour devenir autonomique, un composant MOCAS est déployé dans un conteneur étendu pour supporter une boucle de contrôle. Cette boucle de contrôle est elle-même constituée de composants MOCAS. Elle supporte des politiques d'auto-reconfiguration et d'auto-réparation exploitant les capacités d'adaptation des composants.

Afin de maintenir la cohérence globale du système dans lequel il est intégré, un composant MOCAS autonomique est capable de se coordonner avec les autres composants présents. Ainsi, lorsque l'adaptation implique plusieurs composants, il communique en utilisant des protocoles d'interaction dont les propriétés, telles que la terminaison, peuvent être vérifiées à la spécification.

Par ailleurs, nous avons développé un ensemble d'outils logiciels pour aider à la construction des composants et des systèmes MOCAS. Le plugiciel *MOCAS4TopCased* et la richesse du

moteur d'interprétation MOCASEngine limite le travail du développeur. En effet, seul le code correspondant à celui constituant les actions métiers et constituant les expressions booléennes des contraintes restent à implémenter. Enfin, la plateforme MOCASA permet de tester les composants MOCAS, les assemblages de ces composants, de délivrer des mises à jours aux composants ainsi que de nouvelles politiques autonomiques.

Ainsi, MOCAS permet d'atteindre les objectifs suivants :

- la *transparence* : l'utilisation d'un conteneur permet d'affranchir le concepteur des préoccupations liées à l'adaptation des composants qu'il conçoit ;
- la *cohérence* : le conteneur vérifie les caractéristiques des éléments à adapter ainsi que le moment où réaliser l'adaptation ;
- la *flexibilité* : la politique embarquée par un composant autonomique peut être remplacée dans le respect des principes d'adaptation définis ;
- la *réutilisation* : les modèles de conception sont présents tout au long du cycle de réalisation d'un composant. Le modèle de machines à états spécifiant le comportement d'un composant est utilisé en documentation et pour l'administration de ce composant.

Les deux derniers objectifs fixés sont en revanche plus mitigés :

- l'*utilisabilité* : si MOCAS est une approche uniforme via l'utilisation exclusive du langage UML pour la spécification de la structure, du comportement et du déploiement des composants, il nécessite une très forte connaissance et maîtrise du métamodèle UML ;
- l'*indépendance* : un composant MOCAS est réalisable et déployable indépendamment des autres composants. Néanmoins, lorsque les composants sont assemblés, les signaux émis par un composant nécessitent qu'ils soient « compris » par leur destinataire afin que le système progresse. La mise en correspondance des interfaces requises et fournies peut donc nécessiter la création d'un composant traduisant les signaux d'un composant vers un autre.

2 Perspectives

Nos travaux laissent entrevoir un ensemble de perspectives de recherche, que cela soit à court terme au niveau des outils ou bien à long terme au niveau des modèles, au niveau des composants ou bien encore au niveau de la coordination.

2.1 Au niveau des outils

Nous avons vu que l'implémentation actuelle du moteur MOCASEngine est relativement coûteuse en mémoire et en temps d'exécution. Nous sommes en train de terminer l'implémentation du moteur pour JavaME afin de rapprocher MOCAS de WMX [149], la plateforme développée dans notre laboratoire et qui permet l'administration de composants PauWare déployés sur des systèmes sans fil. Par ailleurs, la plateforme MOCASA est aussi en cours de modification pour supporter le déploiement distant de composants grâce à la technologie JINI[27]. Enfin, nous en-

27. http ://www.jini.org/

visageons d'exploiter un interpréteur OCL au niveau du moteur afin d'éviter de devoir générer des classes pour les contraintes.

2.2 Au niveau des modèles

Les résultats de notre équipe de recherche en matière d'exécution de modèles et d'administration dirigée par les modèles [149], ainsi que la richesse encore inexploitée du langage UML (notamment au niveau des modèles d'activités et de séquences), amène la question de la viabilité d'une « Programmation orientée modèle ». La prochaine génération de langages de programmation sera-t-elle une génération de langages graphiques ? Cet engouement pour les modèles est prouvé par des projets récents en matière d'exécution de modèles (comme le projet Eclipse MXF, *Model Execution Framework*) ainsi que par les grands noms de l'industrie aéronautique groupés autour du projet TopCased. Cette perspective de recherche requiert notamment un langage de modélisation qui soit formel, ce que les détracteurs d'UML ont toujours réclamé.

2.3 Au niveau des composants

Si MOCAS répond à des problématiques liées à l'adaptation dynamique et à la construction de politiques autonomiques pour les composants, son intégration avec les autres problématiques de la conception à base de composants reste à étudier. En effet, un modèle de composants industriel comme les EJB permet l'équilibrage de charge et la persistence des composants, un modèle de composants comme Fractal permet la partageabilité d'un sous-composant entre plusieurs composites, un modèle comme CCM est tourné vers la distribution des composants. Comment MOCAS se comporte-t-il face à ces préoccupations ? Nous avons déjà pu mesurer la compatibilité du modèle PauWare avec les EJB et étudier l'administration distante de composants déployés sur des systèmes sans fils. Il nous reste à fusionner MOCAS et ces précédents résultats.

2.4 Au niveau de la coordination

Nous avons proposé un protocole d'interaction pour coordonner des adaptations globales à un système MOCAS. Nous devons encore évaluer ce protocole, notamment au niveau des aspects décisionnels (grâce à l'introduction de fonctions d'utilité par exemple) et au niveau du passage à l'échelle. Par ailleurs, les composants MOCAS disposent de leur propre flot d'exécution, communiquent en échangeant des signaux et sont capables d'interagir via des protocoles. Nous voyons ainsi que MOCAS suit la tendance actuelle qui consiste à réduire la distance entre composants et agents [152–154]. Nous projetons donc de pousuivre l'*agentification* des composants MOCAS en étudiant les aspects liés aux connaissances et à la mobilité.

Bibliographie

[1] *OMG Unified Modeling Language (OMG UML), Superstructure Specification, V2.2*, 2009.

[2] Fabien Romeo, Cyril Ballagny, and Franck Barbier. Pauware : a state-based component model. In *Actes des Journées Composants (JC2006)*, pages 1–10, octobre 2006.

[3] IBM. An architectural blueprint for autonomic computing, Third Edition, 2005.

[4] Guillaume Grondin. *MaDcAr-Agent : un modèle d'agents auto-adaptables à base de composants*. PhD thesis, Ecole Nationale Supérieure des Mines de Saint-Etienne, Saint-Etienne, France, 24 novembre 2008.

[5] Jim Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, Dept of Computer Science, Trinity College, Dublin, Ireland, 2004.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[7] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing : emerging trends and open problems. *SIGSOFT Softw. Eng. Notes*, 30(4) :1–7, 2005.

[8] P. Horn. Autonomic computing : IBM perspective on the state of information technology. 2001.

[9] John D. Strunk and Gregory R. Ganger. A human organization analogy for self-* systems. In *In First Workshop on Algorithms and Architectures for Self-Managing Systems, in conjunction with Federated Computing Research Conference*, pages 1–6, 2003.

[10] M. D. McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1968. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[11] *Report of the NATO Software Engineering Conference*, Garmisch, Germany, october 1968.

[12] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[13] Bertrand Meyer. *Conception et programmation orientées objet*. Eyrolles, 2000.

[14] Ivica Crnkovic. Component-based software engineering - new challenges in software development. In *Journal of Computing and Information Technology*, volume 11 of *3*, pages 151–161. University Computing Centre, Zagreb, Croatia, 2003.

[15] Mourad Oussalah et al. *Ingénierie des composants*. Vuibert Informatique, 2005.

[16] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Linda DeMichiel and Michael Keith. JSR 220 : Enterprise JavaBeans TM,Version 3.0. Technical report, Sun Microsystems, 2006.

[18] Johannes Sametinger. *Software engineering with reusable components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[19] Bill Councill and George T. Heineman. Definition of a software component and its elements. pages 5–19, 2001.

[20] Bertrand Meyer. The grand challenge of trusted components. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 660–667, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Christopher H. Lovelock. *Services marketing*. Prentice Hall, 1996.

[22] Philip A. Bernstein. Middleware : a model for distributed system services. *Commun. ACM*, 39(2) :86–98, 1996.

[23] Steve Vinoski. Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14 :46–55, 1997.

[24] OASIS : Organization for the Advancement of Structured Information Standards. *Reference Model for Service Oriented Architecture 1.0*, August 2006.

[25] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to web services architecture. *IBM Systems Journal*, 41(2) :170–177, 2002.

[26] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10) :709–724, 2007.

[27] Nicolas Belloir. *Composition logicielle basée sur la relation Tout-Partie*. PhD thesis, Université de Pau et des Pays de l'Adour, Pau, France, december 2004.

[28] ISO : International Organization for Standardization. *Reference Model for Open Distributed Processing : Foundations*, 1996.

[29] Luca Pazzi. Part-whole statecharts, or how i learned to turn the explicit attitude of composition languages on my side. In *Proceedings of the Second International Workshop on Composition Languages*, Malaga, Spain, june 2002.

[30] Mikaël Beauvois. *Composition comportementale de composants*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 2005.

[31] Martin Buchi and Wolfgang Weck. A plea for grey-box components. Technical report, August 1997.

[32] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4) :369–385, 2006.

[33] Raul Silaghi and Alfred Strohmeier. Integrating CBSE, SoC, MDA, and AOP in a Software Development Method. In *EDOC '03 : Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 136, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Kathryn Anne Weiss, Elwin C. Ong, and Nancy G. Leveson. Reusable specification components for model-driven development. In *In Proceedings of the International Conference on System Engineering, INCOSE*, 2003.

[35] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *WiSME 2004 : Proceedings of the 3rd Workshop in Software Model Engineering*, 2004.

[36] An Phung-Khac, Antoine Beugnard, Jean-Marie Gilliot, and Maria-Teresa Segarra. Model-driven development of component-based adaptive distributed applications. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 2186–2191, New York, NY, USA, 2008. ACM.

[37] Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Sci. Comput. Program.*, 74(4) :168–196, 2009.

[38] *MDA Guide version 1.0.1*, 2003.

[39] Andrei Kirshin, Dolev Dotan, and Alan Hartman. A uml simulator based on a generic model execution engine. In Kühne [155], pages 324–326.

[40] B. Combemale, X. Crégut, J.P. Giacometti, P. Michel, and M. Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *4th European Congress ERTS : EMBEDDED REAL TIME SOFTWARE*, Toulouse, France, 2008.

[41] Michael Soden and Hajo Eichler. Towards a model execution framework for eclipse. In *BM-MDA '09 : Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–7, New York, NY, USA, 2009. ACM.

[42] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[43] Arthur Gill. *Introduction to the Theory of Finite-State MacHines*. McGraw Hill, 1962.

[44] David Harel and Eran Gery. Executable object modeling with statecharts. In *ICSE '96 : Proceedings of the 18th international conference on Software engineering*, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society.

[45] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the corba component model. pages 557–571, 2001.

[46] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.

[47] Franck Barbier. An enhanced composition model for conversational enterprise javabeans. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 344–351. Springer, 2006.

[48] Franck Barbier and Xabier Aretxandieta. State-based composition in uml 2. *International Journal of Software Engineering and Knowledge Engineering*, 18(8) :987–1011, 2008.

[49] Manish Parashar and Salim Hariri. Autonomic computing : an overview. *Lecture notes in computer science*, 3566 :247–259, 2005.

[50] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[51] Roy Sterritt. Autonomic computing. Springer-Verlag.

[52] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3) :181–187, 2005.

[53] Mohammad Reza Nami and Koen Bertels. A survey of autonomic computing systems. In *The 3rd IEEE International Conference on Autonomic and Autonomous Systems*, page 26, Washington, DC, USA, 2007.

[54] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1) :5–18, 2003.

[55] Peter Van Roy. Self management and the future of software design. *Formal Aspects of Component Software (FACS '06)*, september 2006.

[56] Seyed Masoud Sadjadi, Philip K. McKinley, R. E. Kurt Stirewalt, and Betty H. C. Cheng. Generation of self-optimizing wireless network applications. In *ICAC* [156], pages 310–311.

[57] Hiroo Ishikawa, Alexandre Courbot, and Tatsuo Nakajima. A framework for self-healing device drivers. In *SASO '08 : Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 277–286, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Marc Leger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants*. PhD thesis, Ecole Nationale Supérieure des Mines De Paris, Paris, France, 2009.

[59] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY*, pages 3–12. IEEE Computer Society, 2004.

[60] Tom De Wolf, Giovanni Samaey, Tom Holvoet, and Dirk Roose. Decentralised autonomic computing : Analysing self-organising emergent behaviour using advanced numerical methods. In *ICAC '05 : Proceedings of the Second International Conference on Automatic Computing*, pages 52–63, Washington, DC, USA, 2005. IEEE Computer Society.

[61] Roy Sterritt, Darren Gunning, Alan Meban, and Phillip Henning. Exploring autonomic options in an unified fault management architecture through reflex reactions via pulse monitoring. In *ECBS '04 : Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, page 449, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Rodney A. Brooks. A robust layered control system for a mobile robot. pages 2–27, 1990.

[63] Yuan-Shun Dai, Tom Marshall, and Xiaohong Guan. Autonomic and dependable computing : Moving towards a model-driven approach. *Journal of Computer Science*, 2(6) :496–504, 2006.

[64] Pierre-Charles David and Thomas Ledoux. Wildcat : a generic framework for context-aware applications. In *MPAC '05 : Proceedings of the 3rd international workshop on*

Middleware for pervasive and ad-hoc computing, pages 1–7, New York, NY, USA, 2005. ACM.

[65] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004.

[66] Richard Anthony, Achim Rettberg, Dejiu Chen, Isabell Jahnich, Gerrit de Boer, and Cecilia Ekelin. Towards a dynamically reconfigurable automotive control system architecture. In *Embedded System Design : Topics, Techniques and Trends*, volume 231/2007 of *IFIP International Federation for Information Processing*, pages 71–84. Springer-Verlag, May 2007.

[67] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. In *AAMAS '04 : Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Brian Melcher and Bradley Mitchell. Towards an autonomic framework : Self-configuring network services and developing autonomic applications. *Intel Technology Journal*, 8(4) :279–290, November 2004.

[69] Jérémy Dubus and Philippe Merle. Applying omg d&c specification and eca rules for autonomous distributed component-based systems. In Kühne [155], pages 242–251.

[70] Salim Hariri, Lizhi Xue, Huoping Chen, Ming Zhang, Sathija Pavuluri, and Soujanya Rao. Autonomia : an autonomic computing environment. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pages 61–68, 2003.

[71] David S. Wile and Alexander Egyed. An externalized infrastructure for self-healing systems. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004)*, pages 285–290, Oslo, Norway, June 2004.

[72] Laurent Broto, Daniel Hagimont, Estella Annoni, Benoit Combemale, and Jean-Paul Bahsoun. Towards a model driven autonomic management system. In *ITNG '08 : Proceedings of the Fifth International Conference on Information Technology : New Generations*, pages 63–69, Washington, DC, USA, 2008. IEEE Computer Society.

[73] Roy Sterritt. Autonomic networks : engineering the self-healing property. *Engineering Applications of Artificial Intelligence*, 17(7) :727 – 739, 2004. Autonomic Computing Systems.

[74] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *ICAC* [156], pages 70–77.

[75] J. A. McCann, P. Howlett, and J. S. Crane. Kendra : adaptive internet system. *J. Syst. Softw.*, 55(1) :3–17, 2000.

[76] Oussama Layaida and Daniel Hagimont. Plasma : a component-based framework for building self-adaptive applications. In *SPIE/IS&T Symposium On Electronic Imaging,*

Conference on Embedded Multimedia Processing and Communications, pages 185–196, San Jose, CA, USA, 2005.

[77] D. M. Chess, C. Palmer, and S. R. White. Security in an autonomic computing environment. *IBM Syst. J.*, 42(1) :107–118, 2003.

[78] O. Patrick Kreidl and Tiffany M. Frazier. Feedback control applied to survivability : a host-based autonomic defense system. *IEEE Transactions on Reliability*, 52 :148–166, 2002.

[79] Guangzhi Qu, Salim Hariri, Santosh Jangiti, Jayprakash Rudraraju, Seungchan Oh, and Samer Fayssal. Online monitoring and analysis for self-protection against network attacks. In *Proc. Intl. Conf. on Autonomic Computing*, pages 324–325, 2004.

[80] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, Tim Hollebeek, and Alexander Egyed. Self-adaptive systems for information survivability : Pmop and awdrat. In *SASO '07 : Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems*, pages 332–335, Washington, DC, USA, 2007. IEEE Computer Society.

[81] Miranda Mowbray and Alexandre Bronstein. What kind of self-aware systems does the grid need ? Technical report, HP Laboratories, 2005.

[82] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *WMCSA '94 : Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[83] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web : An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2) :86–93, 2002.

[84] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[85] Robert Laddaga. Active software. In *IWSAS' 2000 : Proceedings of the first international workshop on Self-adaptive software*, pages 11–26, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[86] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1) :3–30, 2001.

[87] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Dynamic adaptation : Experimentation on javabeans component model. In *ICSSEA'2002 : Proceedings of the 15th International Conference Software and Systems Engineering and their Applications*, Paris, France, december 2002.

[88] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 7th International Symposium, CBSE*

2004, Edinburgh, UK, May 24-25, 2004, Proceedings, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 2004.

[89] Alexander Egyed. Architecture differencing for self management. In *The 1st ACM SIGSOFT workshop on Self-managed systems*, pages 44–48, New York, USA, 2004.

[90] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99 : Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, London, UK, 1999. Springer-Verlag.

[91] Heather J. Goldsby and Betty H. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *MoDELS '08 : Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 568–583, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande Berbers. Pitfalls in unanticipated dynamic software evolution. In *Proc. of Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, pages 41–49, Glasgow, Schotland, July 2005.

[93] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. Technical report cs-tr-3210, University of Maryland, January 1994.

[94] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[95] J. Kramer and J. Magee. The evolving philosophers problem : Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11) :1293–1306, 1990.

[96] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. An alternative to quiescence : Tranquility. In *ICSM '06 : Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 73–82, Washington, DC, USA, 2006. IEEE Computer Society.

[97] N. De Palma, P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. In *6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, pages 18–24, 2001.

[98] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[99] 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*, pages 43–52, Washington, DC, USA, 1998. IEEE Computer Society.

[100] Yves Vandewoude and Yolande Berbers. Component state mapping for runtime evolution. In *In Proceedings of the 2005 International Conference on Programming Languages and Compilers*, pages 230–236, Las Vegas, Nevada, USA, June 2005.

[101] Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. Component reassembling and state transfer in MaDcAr-based self-adaptive software. In Richard F. Paige and Bertrand Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns*, pages 258–277, Zurich, Switzerland, 2008.

[102] Y. Berbers Y. Vandewoude. A meta-model driven methodology for state transfer in component oriented systems. In *USE 03 : Proceedings of the Second International Workshop on Unanticipated Software Evolution*, Warshau, Poland, April 2003.

[103] Goudarzi Kaveh Moazami. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, London, England, 1999.

[104] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley. Enabling safe dynamic component-based software adaptation. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *WADS*, volume 3549 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2004.

[105] Christophe Sibertin-Blanc, Philippe Mauran, and Gérard Padiou. Safe adaptation of component coordination. *Electronic Notes in Theorical Computer Science*, 189 :69–85, 2007.

[106] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.

[107] National Research Council Staff. *Embedded Everywhere : A Research Agenda for Networked Systems of Embedded Computers*. National Academy Press, Washington, DC, USA, 2001.

[108] Mehmet Aksit and Zièd Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *ICDCSW '03 : Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 84, Washington, DC, USA, 2003. IEEE Computer Society.

[109] Dhouha Ayed, Chantal Taconet, and Guy Bernard. Architecture à base de composants pour le déploiement adaptatif des applications multi-composants. In *Actes des Journées Composants*, mars 2004.

[110] N. Parlavantzas, G. Coulson, and G.S. Blair. A resource adaptation framework for reflective middleware. In *Proceedings of the 2nd International Workshop on Reflective and Adaptive Middleware*, June 2003.

[111] G.S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in OpenORB. In *Proceedings of the first workshop on Self-healing systems*, pages 9–14. ACM New York, NY, USA, 2002.

[112] F. Briclet, C. Contreras, and P. Merle. Openccm : une infrastructure à composants pour le déploiement d'applications à base de composants corba. In *DECOR'04, Déploiement et (Re) Configuration de Logiciels*, 2004.

[113] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report.

[114] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

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

[116] Alexander Egyed and Dave Wile. Statechart simulator for modeling architectural dynamics. In *WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 87, Washington, DC, USA, 2001. IEEE Computer Society.

[117] M. Riveill and A. Senart. Aspects dynamiques des langages de description d'architecture logicielle. *L'Objet, Revue des Sciences et Technologies de l'Information*, 8(3), 2002.

[118] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[119] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for gui software. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering*, pages 295–304, New York, NY, USA, 1995. ACM.

[120] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[121] Djalel Chefrour and Francoise André. Aceel : modèle de composants auto-adaptatifs - application aux environnements mobiles. *Systèmes à composants adaptables et extensibles*, october 2002.

[122] Djalel Chefrour. *Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique*. PhD thesis, Université de Rennes 1, Rennes, France, 2005.

[123] P. Boinot, R. Marlet, Noyé J., G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000. IEEE Computer Society Press.

[124] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[125] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02 : Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[126] Swen Aussmann and Michael Haupt. Axon - dynamic aop through runtime inspection and monitoring. In *Proceedings of the 1st Workshop on Advancing the State-of-the-Art in Run-Time Inspection*, 2003.

[127] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 85–92, New York, NY, USA, 2002. ACM.

[128] Eddy Truyen and Wouter Joosen. Towards an aspect-oriented architecture for self-adaptive frameworks. In *ACP4IS '08 : Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–8, New York, NY, USA, 2008. ACM.

[129] Javier Alonso, Jordi Torres, Rean Griffith, Gail Kaiser, and Luis Silva. Towards self-adaptable monitoring framework for self-healing. In *Proceedings of the Third CoreGrid Workshop on Middleware*, June 2008.

[130] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[131] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[132] H. Conrad Cunningham. Reflexive metaprogramming in ruby : tutorial presentation. *J. Comput. Small Coll.*, 22(5) :145–146, 2007.

[133] Richard S. Sutton. *Reinforcement Learning*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[134] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Ecole des Mines de Nantes, Nantes, France, 2005.

[135] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Dynamic adaptive software components : the MOCAS approach. In *ASBS'08 : Proceedings of The First IEEE International Workshop on Autonomous and Autonomic Software-Based Systems*, pages 517–524, Cergy-Pontoise, France, 2008. ACM.

[136] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.0 Specification*, May 2006.

[137] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. MOCAS : a State-Based Component Model for Self-Adaptation. In *SASO'09 : The Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 517–524.

[138] Nelly Bencomo, Paul Grace, Carlos A. Flores-Cortés, Danny Hughes, and Gordon S. Blair. Genie : supporting the model driven development of reflective, component-based adaptive systems. In *The 30th International Conference on Software Engineering*, pages 811–814, Leipzig, Germany, may 2008. ACM.

[139] Franck Chauvel, Olivier Barais, Isabelle Borne, and Jean-Marc Jézéquel. Un processus à base de modèles pour les systèmes auto-adaptatifs. *Revue de l'Electricité et de l'Electronique (REE)*, (2) :38–44, February 2009.

[140] Najla Hadj Kacem, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Towards modelling and analysis of a coordination protocol for dynamic software adaptation. In *CSTST '08 : Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, pages 499–507, New York, NY, USA, 2008. ACM.

[141] Stefan Poslad and Patricia Charlton. Standardizing agent interoperability : the fipa approach. pages 98–117, 2001.

[142] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Endowing software components with autonomic capabilities based on modeling language executability. In *The 1st Workshop on Model-driven Software Adaptation M-ADAPT'07 at ECOOP 2007*, pages 55–60, Berlin, Germany, July 2007.

[143] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. MOCAS : a Model-Based Approach for Building Self-Adaptive Software Components. In *ECMDA 2009, Tools and Consultancy Track*, pages 5–11, June 2009.

[144] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Cregut, and Marc Pantel. The TOPCASED project : a toolkit in open source for critical aeronautic systems design. In *The 3rd Embedded Real Time Software Conference*, pages 54–59. ACM, 2006.

[145] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects*. Addison-Wesley Professional, 2008.

[146] Object Management Group, Inc. *XML Metadata Interchange Specification*, May 2003.

[147] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[148] Thomas Santen and Dirk Seifert. Executing uml state machines. Technical report, Technische Universit¨at Berlin, Softwaretechnik.

[149] Fabien Romeo. *Administration de composants logiciels pour systèmes sans fil*. PhD thesis, Université de Pau et des Pays de l'Adour, Pau, France, 2007.

[150] Adriana LE Van. Mechanical gearbox control system including wheel skid preventing means. Technical report, European Patent Application, 1998.

[151] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. A Model-based Design Environment for Executing Self-* Software Components. In *WASELF'09 : The Second Workshop on Autonomic and SELF-adaptive systems at JISBD 2009*, pages 32–41, September 2009.

[152] Martin L. Griss and Gilda Pour. Accelerating development with agent components. *Computer*, 34(5) :37–43, 2001.

[153] Martin L. Griss. Software agents as next generation software components. *Component-based software engineering : putting the pieces together*, pages 641–657, 2001.

[154] Federico Bergenti and Michael N. Huhns. On the use of agents as components of software systems. In *Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering handbook*, pages 19–32, New York, 2004. Kluwer Academic Publishing.

[155] Thomas Kühne, editor. *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*. Springer, 2007.

[156] *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*. IEEE Computer Society, 2004.

Résumé

Les administrateurs, les développeurs, les concepteurs logiciels ont besoin de nouvelles approches pour faire face à la complexité croissante des systèmes informatiques. Afin de libérer les administrateurs de tâches répétitives et d'améliorer la réactivité des systèmes, ces systèmes, alors appelés *systèmes autonomiques*, tendent à être dotés de capacités d'auto-gestion telles que l'auto-configuration et l'auto-réparation. Ces capacités d'auto-gestion sont fortement couplées à la capacité du système à s'auto-adapter, c.-à-d. à modifier sa structure et son comportement alors qu'il est en cours d'exécution. Les approches se sont jusqu'alors concentrées sur l'auto-adaptation structurelle des systèmes à base de composants logiciels en remaniant les liaisons entre les composants.

Dans notre approche, nous nous focalisons sur l'adaptation comportementale. Nous avons défini le modèle de composants MOCAS (*Model Of Components for Adaptive Systems*) pour permettre la réalisation de systèmes autonomiques à base de composants logiciels auto-adaptatifs. MOCAS repose sur une approche uniforme exploitant l'ingénierie des modèles : les capacités d'adaptation et les propriétés autonomiques de chaque composant reposent sur l'usage qui est fait par MOCAS du langage de modélisation UML. La structure d'un composant MOCAS repose sur les éléments natifs du langage UML. Le comportement du composant est décrit avec une machine à états UML. Cette machine est ensuite exécutée par le composant pour réaliser son comportement. Pour devenir adaptable, le composant est déployé dans un conteneur respectant ce même modèle. Une boucle de contrôle à base de composants MOCAS permet de réaliser des composants auto-adaptables de façon décentralisée et de les doter de propriétés autonomiques. Les politiques autonomiques exploitent alors les modèles de machines à états afin d'auto-configurer et d'auto-réparer les composants.

Mots-clés: adaptation comportementale, composants logiciels, UML, machines à états, informatique autonomique.

Abstract

Software administrators, developers and designers need original means to deal with the growing complexity of IT systems. For administrators to be freed from tedious tasks and for systems to be more reactives, these systems, a.k.a *autonomic systems*, tend to be endowed with self-management capabilities such as self-configuration and self-healing. These capabilities are strongly tied to the *self-adaptive* one, which enables a system to modify its structure and its behavior while it is running. Hitherto, current approaches focus on adaptations related to the structure of component-based systems by altering links between components.

In our approach, we focus on behavioral adaptation. We defined the MOCAS component model (*Model Of Components for Adaptive Systems*) to allow the construction of autonomic systems by using self-adaptive software components. MOCAS is based on model-driven engineering techniques and only relies on the Unified Modeling Language (UML) to endow each software component with self-adaptive capabilities. The structure of a MOCAS component is specified with UML native elements. The behavior of the component is specified with a UML state machine. This state machine is later executed by the component in order to realize its behavior. The component is then deployed in a dedicated container, a MOCAS component too, in order to become adaptive. A set of MOCAS components enables to build a decentralized control loop for making components self-adaptive and autonomic. Finally, a MOCAS component used autonomic policies also specified with state machines to configure itself and to heal itself.

Keywords: behavioral adaptation, software components, UML, statemachine, autonomic computing.