Développement évolutionnaire de systèmes de systèmes avec une approche par patron de reconfiguration dynamique

Franck Petitdemange

THESE DE DOCTORAT DE

L'UNIVERSITE BRETAGNE SUD
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Franck PETITDEMANGE

Développement évolutionnaire de systèmes de systèmes avec une approche par patron de reconfiguration dynamique

Thèse présentée et soutenue à Vannes, le 3 décembre 2018
Unité de recherche : Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)
Thèse N° : 510

Rapporteurs avant soutenance :

Sophie CHABRIDON Directrice d'Étude HDR, Institut Mines-Telecom SudParis
Yann POLLET Professeur, Conservatoire National des Arts et Métiers

Composition du Jury :

Président : *(à préciser après la soutenance)*
Examinateurs : Marianne HUCHARD Professeur, Université de Montpellier
Olivier BARAIS Professeur, Université Rennes 1

Dir. de thèse : Isabelle BORNE Professeur, Université Bretagne Sud
Co-dir. de thèse : Jérémy BUISSON Maître de conférences, Écoles de Saint Cyr Coëtquidan

Remerciements

Cette thèse a été un travail difficile et long cependant je pense en avoir retiré de précieux atouts pour la suite.

Pour cela je tiens à remercier ma directrice de thèse Isabelle BORNE qui s'est montrée efficace dans son rôle de directrice. Elle m'a encouragé en particulier à publier régulièrement ce qui m'a permis de me construire une expérience du monde de la recherche en dehors du laboratoire, ce que j'ai grandement apprécié. Ensuite je veux remercier mon co-directeur de thèse Jérémy BUISSON qui m'a soutenu en particulier sur les aspects scientifiques de la thèse tant sur le plan des connaissances que de la rigueur scientifique.

Je remercie les rapporteurs Sophie CHABRIDON et Yann POLLET pour l'intérêt porté à la lecture de cette thèse que j'ai pu mesurer par les appréciations agréables à lire et les remarques pertinentes.

Je remercie également Olivier BARAIS d'avoir participé au jury. Je remercie doublement Marianne HUCHARD d'avoir accepté de participer à ce jury mais également d'avoir proposé ma candidature à cette thèse.

Je remercie aussi les personnes qui ont contribué indirectement à cette thèse de part leur accueil lors de mon arrivée, je pense en particulier à Abdel, Davy, Salma, Tu et plus généralement l'équipe ArchWare et les membres de IRISA de Vannes.

J'ai une pensée également pour la personne avec qui je partage ma vie sans qui je n'aurai pas eu l'équilibre suffisant pour persévérer dans cette thèse. Pour mes parents qui ont toujours été un pilier dans mon parcours. Et toute ma famille qui m'a encouragé depuis le début.

Table des matières

Table des figures

Liste des tableaux

Glossaire

ANTARES Adaptation nationale des transmissions aux risques et aux secours. 50

BDD Diagramme de Définition de Bloc. 25, 32, 33, 60

BDI Diagramme Interne de Bloc. 25, 33, 34, 40, 60

CML COMPASS Modelling Language. 43

CODIS Centre Opérationnel Départemental d'Incendie et de Secours. 52

COMPASS Comprehensive Modelling for Advanced Systems of Systems. 21, 31–33, 42, 88, 136

CS système constituant. 25, 31–34, 39, 41–43, 46, 60, 63–65, 69, 76, 83, 88, 110, 111, 114, 119

DANSE Designing for Adaptability and evolutioN in System of systems Engineering. 21, 27, 31, 38–40, 42–44, 88, 136

ERS Embarcation de Reconnaissance et de Sauvetage. 52

GCSL Goal Contracts Specification Language. 40

OCL Object Constraint Language. 59, 60, 63, 88, 145, 149

PC Poste de Commandement. 96

RUP Rational Unified Process. 23, 43

SDIS Service Départemental d'Incendie et de Secours. 51–53, 120

SdS système de systèmes. 9, 10, 19–22, 25, 27, 31–34, 38, 39, 41–43, 46, 47, 49–53, 57–60, 63–65, 69, 83, 88–92, 94, 95, 98–100, 102, 103, 107, 108, 110, 111, 118–122, 126, 128, 136–138, 141, 142, 145

SosADL System of Systems Architectural Description Language. 63, 88

SysML System Modeling Language. 21, 23–25, 27, 31–34, 42, 59, 60, 63, 136, 137, 145

UML Unified Modeling Language. 23, 25, 27, 31

UPDM Unified Profile for DoDAF and MODAF. 21, 27, 31, 32, 38, 39, 42, 44, 59, 60, 63, 83, 88, 136

VLC Véhicule Léger de Commandement. 52

VTU Véhicule Tout Usage. 47, 51

Chapitre 1

Introduction

La complexité croissante de notre environnement socio-économique se traduit en génie logiciel par une augmentation de la taille des systèmes et par conséquent de leur complexité. Les systèmes actuels sont le plus souvent concurrents, distribués à grande échelle et composés d'autre systèmes. Ils sont alors appelés Systèmes de Systèmes (SdS). Ils ont des caractéristiques spécifiques qui font en sorte que les méthodes et les outils existants pour le développement de systèmes ne sont pas complètement adaptés. De plus, les systèmes de systèmes présentent des architectures logicielles qui changent de manière dynamique d'une façon qui n'est pas nécessairement prévue au moment de la conception.

1.1 Contexte

Parmi les systèmes qu'un ingénieur peut être amené à concevoir, les systèmes de systèmes constituent une classe particulière, dont les constituants sont eux-mêmes des systèmes à part entière. Maier (1998) caractérise un système de systèmes par cinq propriétés. En plus de la distribution géographique des constituants, que l'on retrouve également dans les autres systèmes distribués, la présence d'un comportement émergent signifie que le système de systèmes parvient, par son organisation et par les collaborations, à accomplir des tâches au-delà de ce que ses constituants sont en mesure de faire. Contrairement aux autres systèmes distribués, les constituants d'un système de systèmes conservent leur indépendance opérationnelle, signifiant que chaque constituant peut aussi avoir ses propres objectifs, en plus de sa contribution au système de systèmes. Les constituants disposent également de leur autonomie de gestion au travers de l'indépendance managériale, c'est-à-dire, chaque constituant suit son propre cycle de vie indépendamment du système de systèmes et des autres constituants, que ce soit en termes de déploiement, maintenance, évolution ou retrait. Enfin, le développement d'un système de systèmes est évolutionnaire, au gré des acquisitions et retraits de capacités, et au fil des évolutions de la mission, pour des systèmes dont la durée de vie est typiquement de l'ordre de plusieurs décennies.

De fait, les systèmes de systèmes existent d'ores et déjà. L'organisation des services de secours pour la réponse aux sinistres est un exemple couramment cité. Les constituants de ce système de systèmes ont chacun un domaine de compétence spécifique tel

que le domaine médical, la lutte contre l'incendie, le sauvetage, l'ordre public. Organisés pour la plupart selon un maillage territorial, chaque service est rattaché aux autorités opérationnelles et administratives compétentes. D'autres systèmes de systèmes existent, par exemple, dans le domaine militaire notamment avec les projets successifs de numérisation de l'espace de bataille ou, dans le domaine civil, les terminaux de transport comme les aéroports. D'autres mouvances comme les usines du futur, les smart cities correspondent également à la classe des systèmes de systèmes.

Parmi les exemples cités au paragraphe précédent, tous les systèmes de systèmes ne laissent pas le même niveau d'indépendance opérationnelle ou managériale aux constituants. Maier (1998) définit une taxonomie spécifique par rapport à ce critère. Dans un système de systèmes dirigé, les constituants sont conçus et gérés spécifiquement pour le système de systèmes. Même si chaque constituant peut être opéré de manière indépendante, le fonctionnement normal consiste à contribuer à l'accomplissement des objectifs du système de systèmes. Lorsque l'autorité centrale (le système de systèmes) n'a pas un tel pouvoir coercitif pour imposer la collaboration aux constituants, il s'agit d'un système de systèmes collaboratif, fondé donc sur une participation volontaire des constituants. Dans un système de systèmes virtuel, aucune entité ne fait sciemment interagir les constituants. Sans qu'il n'y ait aucun objectif spécifique à cela, les constituants interagissent d'eux-mêmes conduisant à des comportements émergents. Dahmann and Baldwin (2008); ODUSD(A&T)SSE (2009) introduisent un niveau intermédiaire : les constituants d'un système de systèmes consensuel sont certes adoptés pour les besoins spécifiques du système de systèmes (comme un système de systèmes dirigé), mais suivent notamment un processus de développement indépendant (comme un système de systèmes collaboratif).

ODUSD(A&T)SSE (2009) livre son analyse des différences entre l'ingénierie système classique et l'ingénierie système dans le cas des systèmes de systèmes. Pour les aspects managériaux, il note une complexité accrue dans le cas des systèmes de systèmes, pour lesquelles plusieurs parties prenantes (une pour le système de systèmes et une par système constituant) peuvent avoir des objectifs et des intérêts divergents. Il considère que cela impose une chefferie de projet et une ingénierie nécessairement collaboratives. Dans son rapport, ODUSD(A&T)SSE (2009) attire l'attention sur le fait que le système de systèmes peut conduire à considérer un environnement opérationnel au-delà de celui considéré par la conception des systèmes constituants individuels. Aussi anecdotique que cela puisse paraître, intégrer des constituants qui ne sont pas prévu pour risque également de conduire à une expérience utilisateur peu homogène, voire incohérente au détriment de l'intérêt même du système de systèmes ainsi assemblé. Une difficulté spécifique à l'ingénierie des systèmes de systèmes consiste donc à trouver un bon équilibre avec l'ingénierie des systèmes constituants. Au niveau de l'implémentation, ODUSD(A&T)SSE (2009) note la spécificité d'intégrer des systèmes constituants étant à des phases très diverses de leur cycle de vie. Un système de systèmes peut en effet assembler des systèmes patrimoniaux, des systèmes en cours de développement ou d'acquisition, des systèmes rénovés pour en prolonger la durée de vie. Le système de systèmes doit s'accommoder et évoluer au gré de l'agenda de chacun de ses constituants, comme mis en évidence par la caractéristique de développement évolutionnaire. Enfin, la difficulté à délimiter la frontière d'un système de

systèmes est une dernière particularité mentionnée par ODUSD(A&T)SSE (2009). Barot et al. (2013) illustrent cette difficulté au travers de l'exemple d'une chaîne d'approvisionnement industrielle. Il s'agit, dans cet exemple, de trouver un équilibre entre, d'une part, la difficile exhaustivité de la liste de fournisseurs, et, d'autre part, le risque de ne pouvoir observer la criticité d'un fournisseur, au-delà d'un certain niveau à partir duquel il est fait abstraction. Dans l'exemple de Barot et al. (2013), un grand industriel cherche à s'assurer qu'il dispose de plusieurs fournisseurs pour chaque composant, afin qu'aucun fournisseur ne soit critique, c'est-à-dire qu'aucune défaillance de l'un d'eux ne force l'arrêt de la production. Ne pas avoir modélisé l'intégralité de la chaîne d'approvisionnement a empêché l'industriel de constater que, pour un composant spécifique, tous les fournisseurs dépendent d'un même fournisseur de matières premières, et, par conséquent, que ce dernier est un fournisseur critique puisque toute défaillance de ce fournisseur entraîne une défaillance de la chaîne d'approvisionnement, et empêche la production.

Pour prendre en compte ces difficultés spécifiques, plusieurs approches d'ingénierie ont été proposées. Sur la base du langage de modélisation SysML (OMG, 2015), le profil UML UPDM (OMG, 2013) unifie les frameworks architecturaux DoDAF et MODAF et liste des points de vue et des vues pertinentes pour guider l'ingénieur. Deux grands projets européens ont fait des propositions pour la modélisation des systèmes de systèmes. Le projet DANSE (Etzien et al., 2014) a sélectionné quelques-unes des vues d'UPDM, les considérant comme les plus pertinentes dans le cadre de l'ingénierie d'un système de systèmes. Le projet COMPASS (Coleman et al., 2012) a quant à lui proposé des vues spécifiques, proches de ce que le projet DANSE suggère d'exprimer.

1.2 Problématique

Nous pensons que UPDM (OMG, 2013), DANSE (Etzien et al., 2014) et COMPASS (Coleman et al., 2012) apportent des réponses pour guider l'architecte dans ses tâches d'ingénierie, de modélisation et d'analyse d'un système de systèmes. Cependant aucun de ces travaux ne traite complètement la question du développement évolutionnaire. Ce constat nous a conduit à formuler la problématique suivante :

> *Comment l'architecte doit-il procéder pour faire évoluer un système de systèmes après son déploiement, dans le cadre du développement évolutionnaire ?*

Le traitement de cette problématique s'est appuyée sur l'expertise et les orientations de recherche de l'équipe Archware. Cette équipe regroupe des enseignants-chercheurs ayant une longue expérience dans le domaine des architectures logicielles, au travers de trois axes principaux : l'analyse du comportement dynamique d'une architecture logicielle à l'aide d'une algèbre de processus, l'application de l'ingénierie dirigée par les modèles aux architectures logicielles, et la reconfiguration dynamique de logiciels. Sur cette base, nous avons décidé de privilégier une approche de conception de reconfiguration fondée sur les langages de modélisation semi-formels.

Considérant qu'un premier défi consiste à connaître l'objet reconfiguré, nous avons posé la première question de recherche suivante :

Q1. Comment l'architecte modélise-t-il la configuration du système avec le niveau de précision et d'exactitude requis ?

Afin de répondre à cette question, nous avons étudié l'état de l'art de la modélisation de l'architecture d'un système de systèmes. En étudiant UPDM, DANSE et COMPASS, nous avons énuméré les modèles que ces précédents travaux recommandent lors de la conception d'un système de systèmes. Nous avons identifié une correspondance entre les vues recommandées par chacun de ces travaux, nous fournissant ainsi un cadre unique capitalisant l'ensemble de ces précédents travaux. Cependant, poursuivant d'autres objectifs, aucun de UPDM, DANSE ou COMPASS n'aborde la question de la configuration précise et exacte du système de systèmes. Nous avons donc débuté notre travail de recherches par l'extension de ce framework unifié en sélectionnant des vues additionnelles propres à la question de la configuration. Nous avons par ailleurs effectué une sélection de manière à écarter les vues qui ne sont pas pertinentes pour notre problématique.

Un second challenge consiste à déterminer un processus de conception par lequel l'architecte du système de systèmes produit la reconfiguration. Ce challenge nous amène à poser une seconde question de recherche :

Q2. Quel processus d'ingénierie l'architecte doit-il suivre pour concevoir la reconfiguration d'évolution du système de systèmes ?

À la suite de l'étude de l'état de l'art, nous avons constaté que la plupart des travaux sur la reconfiguration, y compris dans d'autres contextes, portent essentiellement sur trois questions que sont : apporter l'assurance qu'une reconfiguration se pose pas de problème, décrire des reconfigurations et automatiser certaines tâches de reconfiguration. Pour élaborer une réponse à cette question, nous avons pu nous appuyer sur le savoir-faire présent dans l'équipe Archware, et l'expérience de conception de plusieurs reconfigurations à l'occasion de travaux précédents. Il s'est agi de conceptualiser ce savoir-faire pour en identifier les principes d'une démarche jusqu'alors implicite.

Deux derniers challenges connexes sont d'une part de permettre à l'architecte d'expliquer, voire justifier la conception qu'il a produite, et d'autre part de permettre à l'architecte de capitaliser des éléments de conception. Considérant que ces deux challenges peuvent être résolus conjointement par la capacité à documenter la conception de la reconfiguration, nous formulons la troisième question de recherche ainsi :

Q3. Comment l'architecte peut-il documenter ses choix de conception d'une reconfiguration ?

Compte-tenu de la culture de l'équipe Archware, il nous a paru naturel de nous intéresser à la notion de patron, bien connue pour répondre à la question de la documentation dans d'autres contextes. Dans le cas précis de la reconfiguration, l'étude bibliographique a mis en évidence les travaux de Gomaa and Hussein (2004); Dorn and Taylor (2015), dont nous avons remarqué qu'ils ne prennent en compte que des contextes simples. Nous avons donc décidé de les étendre afin d'aboutir à un concept de patron

de reconfiguration plus complet et mieux adapté, tirant notamment les leçons d'autres travaux autour de la documentation des reconfigurations, notamment les travaux de Allen et al. (1998); Oliveira and Barbosa (2015).

1.3 Contributions

Suite à l'étude de la problématique et des questions de recherche décrites à la section précédente, les contributions suivantes ont été apportées :

- Un alignement des processus et des vues de modélisation des projets DANSE et COMPASS. Cette contribution est une synthèse de l'étude bibliographique et de l'analyse des deux projets et nous a permis d'élaborer un cadre commun pour mettre en correspondance les vues proposées par chaque projet.
- Un processus de modélisation adapté aux systèmes de systèmes, afin d'aboutir à une description de la configuration. Construit sur la base de la contribution précédente, ce processus sélectionne et complète les vues de modélisation spécifiquement adaptées à l'élaboration d'une configuration.
- Un concept de patron de reconfiguration. Pour cette contribution, nous avons spécialisé le concept général de patron en déclinant des rubriques spécifiquement adaptées à la description de choix de conception concernant les reconfigurations. Trois premiers patrons de reconfiguration ont également été mis en forme en concordance avec notre proposition.
- Un processus de conception de reconfiguration. Ce processus guide l'architecte en charge de l'ingénierie du système de systèmes en lui indiquant les tâches à accomplir pour élaborer une reconfiguration dans le cadre du développement évolutionnaire.
- L'ensemble des modèles du cas d'étude. Afin de permettre la validation des propositions de cette thèse, nous avons produit les modèles résultant de l'application des deux processus (élaboration de la configuration et conception de la reconfiguration) correspondant à un cas d'étude réaliste. Les artéfacts ainsi produits font parties des contributions de cette thèse.

1.4 Plan du mémoire

Après ce chapitre d'introduction, le mémoire est organisé en quatre parties.

La première partie du mémoire est dédiée à l'étude bibliographique de l'état de l'art. Elle présente les éléments de l'état de l'art permettant de répondre à notre problématique dans le contexte de notre travail. Plus précisément :

Le chapitre 2 décrit la reconfiguration dynamique au travers de trois frameworks de composants représentatifs afin d'identifier les services de reconfiguration. Puis ce même chapitre analyse les précédents travaux ayant traité de la documentation des reconfigurations.

Le chapitre 3 synthétise notre étude des processus de modélisation des systèmes de systèmes. Après avoir introduit SysML et UPDM, ce chapitre se concentre spécifiquement

sur l'analyse des travaux menés dans le contexte des projets DANSE et COMPASS. Nous en déduisons un cadre général de modélisation, mettant ces deux projets en vis-à-vis.

La deuxième partie détaille les contributions de cette thèse.

Le chapitre 4 décrit le cas d'étude qui sert de support à la présentation des contributions, puis à la validation du travail.

Le chapitre 5 sur la base de l'analyse produite au chapitre 3, présente le framework de modélisation que nous proposons à l'architecte afin d'élaborer un modèle de la configuration du système de systèmes.

Le chapitre 6 décrit notre concept de patron de conception, et illustre ce concept en donnant deux patrons que nous avons conçus.

Le chapitre 7 décrit le processus de conception de reconfiguration que nous avons conçu, ainsi qu'une application de ce processus à un des scénarios identifiés au chapitre 4.

La troisième partie décrit l'expérience que nous avons conduite afin de valider notre travail.

Le chapitre 8 présente le framework, en Java et basé sur EMF, qui a été développé pour simuler le déploiement et la reconfiguration d'un système de systèmes.

Le chapitre 9 présente la méthodologie et les résultats de l'évaluation réalisée.

La dernière partie conclut ce manuscrit et propose des prolongements des travaux et des perspectives de recherches sur les thèmes développés dans ce mémoire.

Enfin nous joignons deux annexes pour compléter et expliciter certains éléments présentés dans les chapitres précédents :

l'**Annexe A** donne la description du patron de reconfiguration *re-routing*.

l'**Annexe B** donne la définition et la description des contraintes OCL utilisées dans les modèles d'architecture.

Première partie

État de l'art

Chapitre 2

Reconfiguration dynamique

Dans ce chapitre, nous allons étudier les pratiques de reconfiguration dynamique dans l'état de l'art. Tout d'abord, dans la section 2.1, nous allons poser le vocabulaire et définir ce qu'est la reconfiguration dynamique dans le cadre de cette thèse. Puis dans la section 2.2 nous allons étudier les mécanismes de reconfiguration de trois frameworks de composants. En effet, la reconfiguration dynamique a particulièrement été étudiée dans ce contexte. De plus, les systèmes distribués à base de composants partagent des similarités avec les systèmes de systèmes, malgré des différences comme l'indépendance des constituants qui est spécifique aux systèmes de systèmes. Dans la section 2.3, ceci nous permettra de mettre en lumière les choix qui se posent lors de la conception d'une reconfiguration, aboutissant à motiver le besoin de documentation ainsi que les types de documentation.

2.1 Définitions

La *reconfiguration dynamique*, c'est modifier un système pendant qu'il est utilisé. Plusieurs vocables ont été employés, parmi lesquels nous pouvons noter les références suivantes : *mise à jour dynamique du logiciel* (*dynamic software updating*) (Chen et al., 2016; Hicks and Nettles, 2005; Neamtiu et al., 2006; Baumann et al., 2005), *reconfiguration dynamique* (Batista et al., 2005; Purtilo and Hofmeister, 1991; Rasche and Polze, 2005), de *structure* ou *architecture* dynamique (Magee and Kramer, 1996; Oquendo, 2004; Allen et al., 1998), *évolution à l'exécution* (Oreizy et al., 1998; Perez et al., 2005), *auto-adaptation* ou *adaptation dynamique* (Oreizy et al., 1999; Brun et al., 2009), *autonomic computing* (Rutten et al., 2017; Kephart and Chess, 2003). En nous basant sur les travaux existants, nous allons proposer les définitions qui fixent le cadre de cette thèse. Dans cette thèse, nous nous intéressons uniquement à la reconfiguration dynamique. Une reconfiguration dynamique est *corrective* si elle corrige un bug ou modifie une décision architecturale. Elle est *fonctionnelle* si elle ajoute ou supprime une fonction au système, ou *non-fonctionnelle* si elle modifie la qualité de service du système. Pour réaliser l'une ou l'autre de ces modifications, une reconfiguration dynamique peut agir sur les composants, leurs implémentations et les liaisons qui constituent l'architecture du système. Ainsi, à la fin de la reconfiguration dynamique, le système s'exécute avec la nouvelle architecture.

S'agissant d'une reconfiguration dynamique, elle doit prendre en compte les contraintes d'exécution du système pour se réaliser harmonieusement pendant que le système est en cours d'exécution et d'utilisation, sans perturber le système. Parmi les critères habituels, une reconfiguration dynamique est évaluée par sa capacité à s'appliquer promptement (*timeliness*), avec une interruption ou dégradation de service contrôlée, et sans mettre en cause le bon fonctionnement du système.

Dans l'état de l'art, l'automatisation de la production d'une reconfiguration dynamique a été étudiée. Par exemple, Georgiadis et al. (2002); Waewsawangwong (2004) ont étudié la génération automatique d'une architecture. Plutôt que de lister exhaustivement les objets architecturaux, l'architecture est décrite par un ensemble de contraintes architecturales à résoudre en fonction des composants disponibles dans l'environnement. Par ailleurs, d'autres auteurs parmi lesquels Arshad et al. (2005); André et al. (2012); da Silva and de Lemos (2011) ont expérimenté la génération de la séquence d'actions constituant une reconfiguration à l'aide d'algorithmes de planification. Plutôt que d'utiliser un algorithme généraliste de planification, Boyer et al. (2017) définissent un algorithme ad-hoc, réalisant la même tâche pour un modèle de composants spécifique représentatif de plusieurs frameworks usuels. A contrario et comme Buisson et al. (2015), en fournissant manuellement, par exemple, de nouvelles implémentations de composants, il est possible de produire des reconfigurations évitant ou limitant la dégradation de service. Ceci montre qu'il peut y avoir un intérêt à laisser l'architecte intervenir dans la conception d'une reconfiguration dynamique. Dans ce type d'approche, l'architecte intervient dans la conception de la reconfiguration, par exemple, pour décider de relâcher certaines contraintes architecturales ou de qualité de service, ou bien, autre exemple, pour déployer des mécanismes non prévus à la conception du système pour mieux préserver certaines contraintes architecturales ou de qualité de service.

Dans cette thèse, nous adoptons résolument une approche considérant que l'architecte est impliqué dans la conception de la reconfiguration dynamique. Nous aborderons les reconfigurations qui ont un impact sur les décisions architecturales des SdSs (systèmes de systèmes).

2.2 Reconfiguration dans les systèmes distribués

L'objectif de cette section est d'étudier les mécanismes de reconfiguration dynamique. Ceux-ci ont été particulièrement étudiés dans le cadre des systèmes distribués, et plus particulièrement les systèmes distribués construits par assemblage de composants. En effet, ces systèmes distribués évoluent dans des environnements ouverts. Ils sont donc sujets à des changements dans leur environnement, y compris après déploiement, qui peuvent être pris en compte par reconfiguration dynamique. Cette section s'intéresse particulièrement aux principaux frameworks à composants. Pour assister la reconfiguration dynamique, certains frameworks à composants sont structurés en suivant les principes des architectures réflexives.

Dans cette section, nous rappellerons donc tout d'abord ce qu'est un système réflexif

d'une manière générale, section 2.2.1. Puis, en section 2.2.2, comme les systèmes de systèmes sont des systèmes distribués, constitués de composants indépendants, nous allons présenter succinctement les frameworks de composants qui servent de support à notre étude. En section 2.2.3, nous détaillerons, pour chaque service de réflexivité, sa mise en œuvre par les frameworks de composants que nous étudions. Enfin, dans une synthèse en section 2.2.4, nous conclurons par les enseignements généraux que nous tirons de cette analyse de l'état de l'art, en vue d'identifier les services de réflexivité qui pourraient être offerts par un SdS et ses constituents, compte tenu des contraintes spécifiques des SdS décrites à la section 1.1.

2.2.1 Introduction à la réflexivité

Dans les langages de programmation et dans les environnements d'exécution, la réflexion est la capacité pour un programme à manipuler comme des données quelque chose représentant son propre état, durant son exécution. Elle peut être qualifiée de réflexion structurelle ou comportementale. La réflexion est structurelle quand elle modifie la structure du programme. Par exemple, la réflexion structurelle permet de modifier la définition d'une classe ou d'ajouter de nouvelles classes dans un programme. La réflexion est comportementale quand elle modifie la sémantique du langage. Par exemple, dans les travaux de Maes (1987), un méta-objet est associé à chaque objet. Ce méta-objet implémente les primitives du langage comme l'appel de méthode. En remplaçant ce méta-objet, il est donc possible de modifier l'implémentation de ces primitives, et donc la manière dont le programme est interprété.

Un second axe de classification introduit le terme introspection pour indiquer que le programme peut s'observer et donc raisonner sur son propre état. L'intercession signifie que le programme a la capacité de modifier son propre état d'exécution ou d'altérer son interprétation ou sa signification. L'introspection et l'intercession peuvent être structurelles ou comportementales. Par exemple, l'introspection est comportementale si elle surveille les messages envoyés entre deux objets, elle est structurelle quand elle regarde les attributs d'un objet.

Ces deux axes de classification nous servent, dans l'étude des frameworks de composants réflexifs de la section 2.2.2, à organiser les mécanismes de reconfiguration qui seront listés dans la table 2.2.

Comme il est établi notamment depuis les travaux de Maes (1987), une séparation nette entre le domaine applicatif et les services de réflexivité est souhaitable. Cela a conduit à architecturer les systèmes en séparant strictement un niveau de base, en charge du domaine applicatif, et un méta-niveau, en charge des services de la réflexivité. C'est un principe de conception que l'on retrouve dans les frameworks de composants réflexifs décrits en sous-section 2.2.2.

2.2.2 Frameworks de composants réflexifs

Un système de systèmes est un système constitué de composants, qui existent toutefois indépendamment du système. Ce système repose donc essentiellement sur la composition de composants préexistants. Même si, par cette caractéristique d'indépendance de ses composants, un système de systèmes diffère des systèmes distribués ou des systèmes à composants habituels, les deux classes de systèmes ont donc en commun ce mécanisme de composition. Dans les systèmes à composants, le framework de composants fournit des primitives pour manipuler la configuration du système, c'est-à-dire pour composer les composants les uns avec les autres. Plusieurs frameworks de composants réflexifs incluent de plus des primitives pour reconfigurer le système en modifiant la manière dont les composants sont composés. Pour cette raison, étudier les frameworks de composants donne un éclairage sur la manière dont la reconfiguration peut être réalisée dans le contexte des systèmes de systèmes. Dans cette sous-section, nous introduisons les trois frameworks qui servent à notre étude des architectures réflexives. Il s'agit de Fractal (Bruneton et al., 2006), OpenCOM (Coulson et al., 2008) et Draco (Vandewoude et al., 2003), qui sont en effet trois frameworks supportant la reconfiguration dynamique. Pour aider la lecture, la table 2.1 synthétise la terminologie employée par chacun de ces frameworks. À la fin de l'étude, la table 2.2 listera les mécanismes de reconfiguration ainsi identifiés.

Fractal

Pour le framework Fractal, un composant est une entité dotée d'interfaces, qui modélisent les points d'interaction entre le composant et son environnement, c'est-à-dire les autres composants qui constituent le système. Le framework Fractal caractérise les interfaces selon trois axes orthogonaux. Tout d'abord, le framework distingue les interfaces serveur, qui déclarent les méthodes que le composant implémente, et les interfaces client, qui listent les méthodes que le composant peut utiliser. Par ailleurs, une interface peut être obligatoire ou optionnelle, indiquant que cette interface peut ne pas être connectée. Enfin, un interface peut être singleton ou collection, s'il s'agit d'un nombre arbitraire d'interfaces identiques, créées à la demande.

Optionnellement, un composant Fractal peut exposer sa structure interne, elle-même étant composée de composants. Il s'agit alors d'un composant composite, par opposition aux composants primitifs dont le contenu est strictement encapsulé. Avec le framework Fractal, un même composant peut être contenu de manière partagée par plusieurs composants.

À l'intérieur d'un composite, les composants sont connectés les uns aux autres via leurs interfaces par des liaisons. À l'intérieur d'un espace d'adressage, les liaisons primitives sont de simples références. Lorsque la communication est plus complexe, par exemple dans le cas d'un système distribué, une liaison composite consiste en un ou plusieurs composants de liaison (ou connecteurs), auxquels les composants sont connectés par des liaisons primitives. Les composants de liaisons sont chargés de mettre en œuvre le mécanisme de communication, par exemple, le protocole d'appel de méthode à distance dans le cas d'un

système distribué. Dans le framework Fractal, les liaisons sont orientées, et ne connectent qu'une interface client à une interface serveur.

Dans sa mise en œuvre, un composant Fractal repose sur deux groupes d'objets : les objets qui implémentent le comportement déclaré par les interfaces du composant, et d'autres objets, nommés la membrane, qui implémentent les services d'administration et de contrôle du composant. Correspondant au méta-niveau du système, la membrane du composant permet de gérer le contenu du composant, d'attacher des liaisons primitives aux interfaces client, d'intercepter les messages d'appel de méthode et d'introspecter l'architecture, de modifier l'état du composant. L'instanciation des composants est prise en charge par des composants fabriques.

OpenCom

OpenCOM est un framework qui définit une notion de composant similaire à celle de Fractal. Avec un vocabulaire légèrement différent, un composant déclare des interfaces (l'équivalent des interfaces serveur de Fractal) et des réceptables (l'équivalent des interfaces client de Fractal). Le framework OpenCOM, contrairement à Fractal, n'a pas de notion de composant composite ou de liaison composite.

Alors que chaque composant Fractal est doté de sa propre membrane qui fournit un méta-niveau spécifique à chaque composant, les choses sont organisées différemment avec OpenCOM. La capsule est l'environnement d'exécution dans laquelle le framework fournit ses fonctionnalités. Cette capsule se décompose en caplets, qui représentent autant d'environnements de déploiement pour les composants. Les caplets permettent, par exemple, de capturer les espaces d'adressage dans le cas d'un système distribué. Initialement, la capsule contient une unique caplet, la caplet racine (ou caplet primaire), qui contient les composants constituant le méta-niveau du système.

D'une part, le noyau d'OpenCOM permet de charger et instancier des composants dans la caplet racine, ainsi que de connecter les réceptacles aux interfaces de ces composants.

D'autre part, un mécanisme d'extensibilité permet d'utiliser des composants spécifiques pour représenter une caplet. Des composants chargeurs (*loaders*) fournissent des implémentations alternatives pour le chargement et l'instanciation d'un composant dans une caplet spécifique. Des composants lieurs (*binders*) fournissent des mécanismes ad-hoc pour connecter un réceptacle à une interface, y compris lorsque la liaison traverse la frontière des caplets. Le mécanisme d'extensibilité que constitue le framework caplet-loader-binder permet à une unique capsule OpenCOM de contrôler plusieurs environnements d'exécution, des composants avec des technologies hétérogènes, et des liaisons avec divers protocoles.

Enfin, la caplet racine fournit trois services de réflexivité. Le *métamodèle interface* permet la découverte des interfaces des composants et l'invocation dynamique de méthode. Le *métamodèle interception*, comme son nom l'indique, permet d'intercepter les messages

qui transitent via les liaisons. Et le *métamodèle architecture* représente la topologie des composants contenus dans la capsule.

OpenCOM fournit donc des services de réflexivité comparables à ceux de Fractal. On peut noter la réification du concept de caplet, et l'explicitation de l'opération de chargement d'un composant, distincte de l'instanciation. Dans Fractal, ces deux aspects sont fusionnés à l'intérieur des composants fabriques. Les composants lieurs d'OpenCOM pourraient trouver un équivalent dans Fractal au travers de composants fabriques spécifiques, instanciant des liaisons composites. Enfin, le modèle capture la topologie complète dans un unique espace, alors qu'avec Fractal, une fonctionnalité équivalente nécessite de parcourir complètement de graphe de connexion des composants afin de le reconstruire.

Draco

Draco est également un framework de composants, qui a été développé comme une ré-implémentation de l'environnement d'exécution du langage SESSCOA [1] afin de disposer d'un environnement extensible. La notion de composant définie dans ce cadre est similaire à celle de Fractal et celle d'OpenCOM, tout en précisant la terminologie des points d'interaction. Ainsi, Draco/SEESCOA distingue les ports, qui sont les points d'interaction d'un composant, et les interfaces, qui sont les contrats attachés aux ports et reprenant les niveaux de contrats identifiés par Beugnard et al. (1999). En plus des cardinalités singleton (appelés ports single dans Draco/SEESCOA) et collection (appelés multiports dans Draco/SEESCOA) de Fractal, Draco/SEESCOA introduit la notion de port multicast, qui est un port dont les messages sortants sont diffusés à tous les ports qui lui sont liés. Une telle fonctionnalité serait mise en œuvre dans Fractal par un composant de liaison composite ou dans OpenCOM par l'établissement d'une liaison par un composant lieur spécifique.

En comparaison à Fractal et OpenCOM, Draco apporte une troisième approche intermédiaire pour structurer le système. Comme Fractal, Draco est une mise en œuvre centralisée : chaque espace d'adressage exécute sa propre instance du framework, et il n'y a pas de concept similaire à la capsule OpenCOM donnant une vision globale du système. Mais comme OpenCOM, le framework est extérieur aux composants, contrairement à la membrane Fractal qui est partie intégrante du composant qu'elle contrôle.

Derrière une façade de contrôle du framework, un gestionnaire de composants assure l'instanciation des composants et maintient un ensemble des instances dans l'espace d'adressage. Un gestionnaire de connecteurs crée et maintient la liste des connecteurs entre les ports des composants. Un ordonnanceur et un gestionnaire de messages sont chargés de l'acheminement des messages entre les composants suivant les connecteurs rapportées par le gestionnaire de connecteurs. Chaque connecteur consiste en un pipeline d'interception des messages, et l'acheminement d'un message consiste à le transmettre successivement à chaque traitement de ce pipeline. En plus de ce cœur, Draco fournit des modules additionnels, dont un module de mise-à-jour dynamique.

1. `https://distrinet.cs.kuleuven.be/projects/SEESCOA/` – consulté le 11/09/2018

		Fractal	OpenCOM	Draco/SEESCOA
			composant	blueprint
composant		composant	instance	composant
interface	client	interface client	réceptacle	port
	serveur	interface serveur	interface	
liaison		liaison primitive	liaison	connecteur
		liaison composite		

TABLE 2.1: Correspondance entre les terminologies employées par les frameworks de composants étudiés.

	Introspection	**Intercession**
Structurelle	inspection des compositions	connexion, déconnexion, chargement, déchargement, création, destruction
Comportementale	interception des messages	démarrage, suspension, redémarrage

TABLE 2.2: Mécanismes de reconfiguration

Draco fournit donc des services de réflexivité comparables à ceux de Fractal et OpenCOM. Nous pouvons noter la réification du pipeline de traitement des messages, permettant de réorganiser ce pipeline selon les besoins, par exemple pour la reconfiguration de l'architecture du système.

Synthèse

Les trois frameworks définissent la notion de composant de façon similaire bien que les termes employés ne soient pas les mêmes. Le tableau 2.1 montre la correspondance entre les termes que nous employons dans cette thèse, et ceux utilisés par chacun de ces frameworks. Les trois frameworks séparent strictement les éléments en charge des communications entre les composants. Chaque composant, qui correspond à un processus distribué, explicite ses points d'interation avec les autres composants au travers des interfaces.

Concernant la reconfiguration, les trois frameworks séparent clairement les services applicatifs des services de réflexivité et reconfiguration. Cela présente l'avantage de l'extensibilité, permettant par exemple de rajouter dynamiquement de nouveaux mécanismes au framework. Le tableau 2.2 résume les services de reconfiguration proposés par ces frameworks. Dans le cadre de la reconfiguration, les services d'introspection sont utiles pour inspecter les compositions de composants pendant l'execution du système, et ainsi retrouver l'architecture du système tel qu'il s'exécute, par exemple dans le cas où cette architecture s'est éloignée de l'architecture initalement déployée par l'architecte. Les mécanimes d'intercession fournissent les opérations primitives pour modifier dynamiquement les compositions de composants, et donc l'architecture.

2.2.3 Mécanismes de reconfiguration

À partir de l'étude des trois frameworks de composants présentés en sous-section 2.2.2, nous avons identifié les services de réflexivité. Ces services, listés dans la table 2.2, sont des services d'instanciation, de liaison et du cycle de vie des composants. Le service d'instanciation gère les opérations de chargement, création, destruction, déchargement d'un composant. Le service de liaison est en charge de la création et de la destruction des liaisons qui connectent les interfaces des composants et correspondent aux voies d'acheminement des messages entre ces composants. Le service de liaison gère également l'interception des messages pendant leur acheminement entre les composants. Le service de gestion du cycle de vie fournit des actions d'initialisation et d'arrêt d'un composant. Dans la suite, nous allons détailler plus précisément le service de gestion du cycle de vie, qui fait l'objet de travaux spécifiques dans le domaine de la reconfiguration dynamique.

Les frameworks de composants fournissent un mécanisme de contrôle de cycle de vie. Plus que la simple question du déploiement ou du retrait du système ou des composants qui le constituent, ce mécanisme joue un rôle spécifique dans le contexte de la reconfiguration dynamique. En effet, supposons une reconfiguration qui supprime une liaison entre deux interfaces, même si cette suppression n'est que transitoire, il convient de s'assurer que les composants concernés ne réalisent aucune communication via ces interfaces pendant le laps de temps au cours duquel elles sont déconnectées. Pour résoudre ce problème, Kramer et Magee (Kramer and Magee, 1990) ont défini l'état de *quiescence*, un état dans lequel un composant n'initie aucun traitement, ne traite aucune communication provenant d'aucun autre composant, ni qu'aucun composant dépendant du composant *quiescent* n'initie aucun traitement. Lorsqu'un composant est dans cet état de *quiescence*, alors il est garanti que ce composant ne réalise aucune communication via aucune de ses interfaces.

La définition de cet état de *quiescence* repose elle-même sur un canevas considérant que le cycle de vie d'un composant, lorsqu'il est instancié, repose sur deux états : un composant est *passif* lorsqu'il ne fait que réagir aux messages provenant des autres composants ; il est *actif* lorsqu'il peut en plus envoyer des messages à destination des autres composants, de sa propre initiative. Pour être plus concret, un composant fournissant uniquement des méthodes est un composant perpétuellement *passif*. Un composant encapsulant un processus exécutant, par exemple, la boucle principale d'un serveur est *actif* lorsque le processus est en cours d'exécution, et *passif* lorsque le processus est suspendu ou arrêté. Kramer et Magee ont montré qu'il suffit, pour qu'un composant soit *quiescent*, que ce composant soit *passif*, et que tous les composants susceptibles de lui envoyer des requêtes soient *passifs* eux aussi. Dans le cas de communications imbriquées, par exemple lorsque la sémantique des communications est l'appel de méthode, l'ensemble des composants qui doivent être *passifs* est élargi pour inclure tous les composants susceptible de communiquer directement ou indirectement avec le composant *quiescent*. En rajoutant au contrôle du cycle de vie des opérations pour rendre un composant actif ou passif, il est donc possible de forcer un ou plusieurs composants dans l'état de *quiescence*, état approprié pour réaliser les opérations de reconfiguration.

Intrinsèquement, la définition de l'état de *quiescence* conduit à rendre passifs ou à

arrêter un grand nombre de composants. Pour réduire cet effet, l'état de *tranquillité*, proposé par Vandewoude et al. (2007), a une définition plus souple, autorisant les composants dépendant du composant *tranquille* à rester actifs si les traitements en cours soit ont déjà fini d'utiliser le composant *tranquille*, soit n'ont pas encore commencé à l'utiliser. L'état de *tranquillité* est connu pour plusieurs problèmes résumés par Ghafari et al. (2015), notamment le fait que la définition de Vandewoude et al. ne prenne pas en compte les communications imbriquées. La *cohérence de version* de Ma et al. (2011) corrige ce problème spécifiquement, en assurant que, dans une transaction, y compris avec imbrication, toutes les communications sont traitées par une seule version (avant ou après reconfiguration) de chaque composant.

Dans le framework Fractal, la spécification reste volontairement vague concernant le cycle de vie des composants. L'implémentation de référence Julia, par contre, choisit de n'avoir que deux états *démarré* et *arrêté* pour chaque composant. En comparaison avec les autres travaux, les états *démarré* et *actif* sont équivalents. En revanche, un composant *arrêté* ne réagit à aucun appel de méthode entrant ni ne réalise aucune action. Les messages entrant sont placés en attente dans la membrane jusqu'au démarrage du composant. Par ailleurs, l'arrêt d'un composant n'aboutit qu'une fois terminés tous les appels de méthode en cours par le composant. La conception d'un algorithme correct d'arrêt d'un composant, notamment en présence de cycles dans le graphe des liaisons entre composants, est une question particulièrement difficile comme cela est remarqué par exemple par Huynh (2017); Boyer et al. (2017).

2.2.4 Réfléxivité dans les systèmes de systèmes

Bien que partageant certaines caractéristiques, les systèmes de systèmes fonctionnent différemment des systèmes pour lesquels les frameworks de composants présentés sont utilisés. Notamment, la répartition des responsabilités est différente comme cela est indiqué en section 1.1.

Dans le cas des frameworks de composants cités, les services de réflexivité listés dans la table 2.2 peuvent être librement utilisés par le système. Par exemple, lors du déploiement du système, les composants sont instanciés pour le système. Au contraire, dans le cas d'un système de systèmes, les composants existent indépendamment du système de systèmes.

Alors que ce sont les frameworks de composants qui instancient les objets qui constituent chaque composant lors du déploiement du système, dans le cas d'un système de systèmes, cette tâche est réalisée indépendamment du système de systèmes qui ne peut que les recruter, les remercier ou faire face à leur disparition. Selon la catégorie de système de systèmes (dirigé, consensuel, collaboratif ou virtuel), l'architecte du système de systèmes peut malgré tout doter celui-ci de composants additionnels, contrôlés par le système de systèmes.

Il en va de même pour les autres services de réflexivité identifiés. Compte tenu de l'indépendance opérationnelle des systèmes qui composent un système de systèmes, ce dernier peut difficilement commander le passage en état passif d'un composant autrement que par le bon vouloir du composant lui-même. Les services d'interception ne peuvent être envisagés que lorsque le système de systèmes prend en charge les communications entre les

composants, ce qui n'est le cas que pour les systèmes de systèmes dirigés ou consensuels. Dans cette thèse nous nous concentrons sur les systèmes dirigés et consensuels.

2.3 Documentation des reconfigurations

Dans la section 2.2, nous avons vu quels mécanismes peuvent être mis en œuvre pour la reconfiguration d'un système. Nous allons tout d'abord, en sous-section 2.3.1 exposer que l'architecte a plusieurs choix lorsqu'il conçoit une reconfiguration mettant en œuvre ces mécanismes. Puis, la sous-section 2.3.2 présentera plus précisément l'état de l'art concernant la façon dont ces décisions, prises dans la conception des reconfigurations, peuvent être documentées. Nous expliquerons dans quels buts une telle documentation a été proposée, et avec quelles propriétés cela a été réalisé dans l'état de l'art.

2.3.1 Décision de conception des reconfigurations

Impliquer l'architecte dans la conception des reconfigurations permet de lui laisser l'opportunité de décider parmi plusieurs choix. Par exemple, certains travaux antérieurs comme ceux de Pissias and Coulson (2008); Kramer and Magee (1990); Vandewoude et al. (2007) ont proposé des mécanismes qui suspendent un sous-ensemble des composants de l'application à reconfigurer. Dans ces approches, certains composants sont basculés dans leur état passif afin de rendre quiescents ou tranquilles les composants concernés par la reconfiguration. D'autres travaux de Wernli et al. (2013); Ma et al. (2011) évitent au contraire de suspendre certains composants en faisant cohabiter plusieurs versions ou plusieurs variantes des mêmes composants. Les messages sont orientés vers l'une ou l'autre version afin d'assurer la cohérence globale du traitement d'une transaction, c'est-à-dire d'un ensemble de messages à considérer comme un tout. L'une ou l'autre de ces approches présente des avantages et des inconvénients. Ainsi, suspendre des composants simplifie la reconfiguration, mais les composants suspendus sont évidemment indisponibles pendant la reconfiguration. À contrario, faire coexister plusieurs versions laisse l'ensemble des composants disponibles, mais requiert un mécanisme spécifique pour synchroniser les états des versions qui coexistent.

Un autre axe de décision pour l'architecte est de savoir s'il laisse l'application atteindre par elle-même l'état idoine (par exemple quiescence ou tranquilité des composants concernés) ou s'il force l'application à l'atteindre. Comme noté par Hayden et al. (2014), dans un serveur conçu autour d'une boucle principale, les composants sont intrinsèquement quiescents entre chaque itération. On pourrait étendre le constat à un serveur qui instancierait un nouveau composant pour chaque requête, assurant ainsi intrinsèquement la cohérence de version comme dans les travaux de Ma et al. (2011). L'architecte peut au contraire décider d'une approche interventionniste : par des actions spécifiques, il conduit les composants concernés à atteindre l'état requis (quiescence ou tranquilité). C'est par exemple le cas dans les travaux de Kramer and Magee (1990); Gomaa and Hussein (2004). Dans une phase de préparation de l'application à la reconfiguration, l'architecte prévoit l'envoi de messages spécifiques afin que certains composants modifient leur comportement (par exemple passent dans l'état passif). Dans les travaux

de Vandewoude et al. (2007); Malabarba et al. (2000), c'est le comportement des connecteurs (plutôt que des composants) qui est modifié pour suspendre les communications entre composants. Pour cet axe de décision également, l'architecte arbitre entre avantages et inconvénients. Attendre que l'application atteigne d'elle-même l'état idoine peut retarder, dans le pire des cas indéfiniment, la reconfiguration. Mais pour forcer l'application à atteindre l'état idoine, encore faut-il que celle-ci fournisse les mécanismes requis, par exemple les implémentations correspondant aux états passif pour tous les composants qui la constituent, ainsi qu'une interface permettant de contrôler les transitions vers ces états passifs, par exemple par une interface de contrôle du cycle de vie.

Parfois, plutôt que de reposer sur un mécanisme générique applicable à l'intégralité de l'application, voire à toutes les applications, l'architecte a la liberté d'utiliser des mécanismes dépendant de chaque composant constituant l'application. Par exemple, dans les travaux de Pissias and Coulson (2008), l'architecte est libre d'adapter, pour chaque composant, l'algorithme permettant d'atteindre la quiescence pour ce composant. Dans les travaux de Gomaa and Hussein (2004), c'est le type du composant qui indique dans quel état le composant doit se trouver pour permettre la reconfiguration, qu'il s'agisse de la quiescence ou d'un autre état.

Une autre contrainte qui s'impose à l'architecte lors de ses décisions concerne le niveau de connaissance dont il dispose au sujet de l'architecture du système, qui peut avoir évolué depuis le déploiement initial. Par exemple, l'approche de Kramer and Magee (1990) suppose que l'architecte connaisse les composants susceptibles d'envoyer des messages aux composants affectés par la reconfiguration. Cela est suffisant pour déterminer l'ensemble des composants qui doivent être dans l'état passif pour assurer la quiescence des composants affectés par la reconfiguration. Par contre, Ma et al. (2011) supposent que l'architecte connaisse l'architecture complète, incluant toutes les liaisons entre tous les composants qui constituent le système.

2.3.2 Documentation

Dans cet état de l'art, nous identifions des approches de documentation des reconfigurations utilisant des langages formels, et d'autres utilisant des notations semi-formelles. Nous débutons notre description par les travaux établis dans un cadre formel.

Allen et al. (1998) utilisent Wright (Robert, 1997) comme notation pour modéliser l'architecture. La documentation est fournie sous la forme d'une spécification qui précise, en utilisant le formalisme CSP de Hoare (2000), le protocole pour chaque port et le comportement pour chaque composant et connecteur. Un invariant indique, sous la forme d'une contrainte, la topologie des liaisons entre les ports des composants et connecteurs de l'architecture. Conjointement à cette description de l'architecture, le comportement de la (re)configuration est décrit par un processus CSP, utilisant les opérations de suppression et création de composant, connecteur et liaisons. Dans le travail de (Allen et al., 1998), la reconfiguration dispose de messages de contrôle, lui permettant d'interagir avec les

composants de l'application reconfigurée, afin de demander, par exemple, à ce que certains composants suspendent leur activité.

Oliveira and Barbosa (2015) utilisent Reo. Il s'agit d'une notation plus souple que CSP (Bonsangue et al., 2012), mais l'approche ne fournit que des opérations architecturales qui manipulent les liaisons. Les reconfigurations documentées par cette approche ne peuvent donc pas ajouter ni supprimer de composants, mais seulement modifier la topologie des liaisons. Pour Oliveira and Barbosa (2015), il n'est pas possible de décrire comment la reconfiguration se coordonne avec l'exécution du système reconfiguré.

Ces approches formelles ont pour avantage d'être suffisamment précises pour permettre la vérification de la documentation. Allen et al. (1998) expliquent par exemple comment vérifier, à l'aide d'un model checker, que la reconfiguration documentée à l'aide de leur approche ne conduit pas à un interblocage lorsqu'elle est combinée avec l'application. Mais les approches formelles souffrent d'être généralement de trop bas niveau pour permettre une réutilisation effective.

Plutôt que des notations formelles, Gomaa and Hussein (2004) propose une approche de conception de reconfiguration réutilisable et de façon systèmatique reposant sur le langage semi-formel UML. Gomaa and Hussein (2004) proposent un notion de patron de reconfiguration, qui modélisent comment des composants collaborent pour évoluer vers une nouvelle configuration architecturale. Pour être réutilable, le patron est documenté par un contexte d'utilisation, qui prend la forme d'un diagramme de collaboration décrivant, comme son nom l'indique, la nature de la collaboration entre les composants. Des diagrammes d'états présentent d'une part le comportement nominal des composants, et d'autre part les transitions pour faire passer le composant concerné vers un état passif puis quiescent. Cette description de l'intention, notamment au travers du contexte, facilite la réutilisation de la solution représentée par le patron. Cependant, l'espace de conception des reconfigurations est très limité car Gomaa and Hussein (2004) ne se concentrent que sur des contextes de reconfiguration simplifiés, et ils font l'hypothèse que les composants disposent déjà des mécanismes de reconfiguration nécessaires.

Dorn and Taylor (2015) définissent, pour aider l'architecte pendant la phase de réflexion, un canevas d'analyse concernant la capacité à reconfigurer une architecture selon le style architectural adopté. Ce canevas traite de l'impact de la reconfiguration sur le contexte d'exécution, des moyens d'observation requis, du comportement de la reconfiguration et de la façon de changer l'état du système. Le canevas donne des indications concernant des aspects des reconfigurations qui méritent d'être documentés.

2.4 Synthèse

L'objectif de cette thèse est de fournir une approche pour la conception des reconfigurations dans le contexte des SdS. Nous avons vu que les frameworks composant nous fournissent une base de réflexion, même si des limites sont à poser. En effet, par rapport aux solutions disponibles pour la reconfiguration des systèmes distribués, les approches existantes considèrent que tous les composants disposent des mêmes mécanismes de recon-

figuration. Or, du fait de l'indépendance managériale des SdSs, les composants déployés risquent de ne pas avoir les mêmes mécanismes de reconfiguration disponibles. Comme solution, nous envisageons une approche de conception des reconfigurations mettant l'architecte à contribution. Les outils principaux de cette solution sont des patrons de reconfiguration et un processus de conception qui permettent d'aboutir à une reconfiguration de l'architecture.

L'état de l'art met en avant la variabilité des choix de conception des reconfigurations. Par exemple, pour atteindre la quiescence des composants affectés par la reconfiguration, l'architecte peut décider d'une approche interventionniste, ou au contraire utiliser sa connaissance de l'application pour attendre que celle-ci atteigne d'elle-même le bon état pour réaliser la reconfiguration. Pour être réutilisable, il s'agit de documenter les choix que réalise l'architecte, l'intention, le contexte qui est susceptible d'invalider ces choix, la problématique traitée et les forces de ces décisions. Nous avons vu les limites des langages formels pour documenter des reconfigurations. Ce type de documentation est trop bas niveau pour être réutilisable. Les langages semi-formels semblent plus adaptés à notre vision des patrons de reconfiguration. Ils sont complémentaires aux langages formels. Ils permettent une meilleure documentation de l'intention et du contexte d'une reconfiguration.

Comme nous le verrons en section 6.1, le contexte inclut les choix de conception architecturale, ainsi que le type de gouvernance du système de systèmes. Selon ce contexte, certaines opérations de reconfiguration ou certains algorithmes peuvent ne pas être possibles. C'est par exemple le cas de l'algorithme qui détermine quels composants rendre passifs pour rendre certains composants quiescents : il s'agit d'un algorithme centralisé, seulement adapté au contexte d'un système de systèmes ayant une gouvernance de type dirigé. Les solutions existantes ne prennent pas en compte les divers types de gouvernances ni les choix de conception qui ont été réalisés dans l'architecture.

Chapitre 3

Processus de modélisation des systèmes de systèmes

L'architecture est l'artefact central des processus de développement. Une architecture, comme le définissent Medvidovic and Taylor (2010), décrit l'organisation fondamentale des composants d'un système, ainsi que les relations entre ces composants et avec l'environnement. L'architecture documente les principes suivis pour concevoir le système. Elle assiste les phases de conception, implémentation, test, analyse, maintenance et évolution. Ce chapitre est consacré aux processus de développement des systèmes de systèmes, aux langages utilisés et aux outils mis en œuvre. L'objectif est de maîtriser les supports de modélisation et de faciliter le positionnement de nos travaux par rapport à la reconfiguration dynamique. Dans un premier temps, dans la section 3.1, nous nous intéressons aux pratiques de modélisation des architectures de systèmes de systèmes, en rappelant au préalable quelques définitions des concepts de base des architectures : modèle architectural, style architectural, vue et point de vue. Parmi les pratiques de modélisation nous ciblons les deux principaux langages disponibles pour assister le processus de modélisation des Systèmes de systèmes (SdSs). Le langage de modélisation SysML (System Modeling Language) (Friedenthal et al., 2014) est dédié aux systèmes complexes et constitue un préliminaire indispensable à la modélisation des systèmes de systèmes. En effet, SysML définit les concepts essentiels à la description d'un système faisant intervenir des constituants provenant de plusieurs corps d'ingénierie. Le second langage étudié est UPDM (Unified Profile for DoDAF and MODAF) (OMG, 2013). Il s'agit d'un framework architectural pour la conception de systèmes de systèmes, de systèmes d'entreprises et de systèmes complexes. En unifiant les précédents frameworks que sont DODAF, MODAF et NAF, UPDM intègre les vues qui sont pertinentes dans un processus de conception pour de tels systèmes. Les sections 3.2 et 3.3, quant à elles, étudient deux projets européens majeurs dans le processus de développement des SdSs (systèmes de systèmes) : COMPASS (Comprehensive Modelling for Advanced Systems of Systems) (Coleman et al., 2012) et DANSE (Designing for Adaptability and evolutioN in System of systems Engineering) (Shani et al., 2015). Nous y verrons les frameworks de modélisation mis en œuvre en étudiant les points de vue développés au regard des processus de développement et outils utilisés. Ces enseignements alimenteront la section 3.4 qui définira les orientations de stratégie de modélisation en termes de langage, processus et outil choisis dans la suite de

la thèse.

3.1 Pratiques de modélisation des architectures de systèmes de systèmes

D'après Medvidovic and Taylor (2010), l'architecture logicielle est l'organisation fondamentale des composants d'un système, de leurs relations entre eux et avec l'environnement, ainsi que les principes, la conception et les évolutions. L'architecture logicielle constitue la clé de voûte de la complexité des systèmes à logiciels prépondérants, en particulier dans le cas de systèmes de systèmes évolutifs, où les descriptions d'architecture fournissent un cadre pour la conception, l'implémentation et l'évolution de tels systèmes. Les architectures de systèmes de systèmes doivent être définies rigoureusement afin d'être capable d'en maîtriser les coûts et la qualité. Cette section présente, dans un premier temps, des concepts de base liés à la compréhension du chapitre. Nous présentons également deux langages de modélisation utilisés dans les domaines industriels pour la modélisation des architectures de systèmes complexes et de SdS.

3.1.1 Modélisation architecturale de grands systèmes

Style architectural Shaw and Garlan (1996) ont classifié les architectures logicielles en introduisant le concept de style d'architecture. Un style architectural est un ensemble de contraintes commun à différentes architectures et qui forme une famille d'architectures. Un modèle d'architecture peut donc inclure un style archtitectural. Un exemple typique de style architectural à flot de données est le *pipe&filter*. Les contraintes peuvent être sur le type des composants et des connecteurs. Par exemple les composants de type *pipe* et *filter* implémentent des interfaces pour lire et écrire des données. Les types ont une sémantique particulière, les *pipes* sont chargés de l'acheminement des données et les *filters* de leur traitement. De plus, les *pipes* doivent notifier les *filters* qu'une donnée peut être transmise au pipe suivant. Les contraintes peuvent porter sur les interactions et peuvent aussi être temporelles avec des protocoles d'interaction. Les contraintes d'interaction peuvent être d'ordre topologique si elles limitent les invocations entre les composants. Dans ce style, les *filters* ne peuvent pas se connecter à d'autres *filters*. Les contraintes peuvent également porter sur les comportements concurrents. Un *pipe* ne peut réaliser une opération de lecture que si le *filter* précédent a réalisé une opération d'écriture sur ce *pipe*.

Modèle architectural Le modèle architectural est l'artefact qui capture tout ou une partie des décisions architecturales. C'est la réification et la documentation des décisions architecturales. Il peut être réifié par des langages architecturaux comme définis par Oquendo (2004); Garlan and T. Monroe (2000); Robert (1997). Grâce à ces langages architecturaux, les architectes décrivent les architectures par des briques de base. Ces briques de base sont les composants qui sont les abstractions encapsulant les fonctions ou les données d'un système. Ils restreignent leurs accès par l'intermédiaire d'interfaces. Les connecteurs sont les abstractions qui affectent et régulent les interactions entre les

composants. Une configuration d'architecture est un ensemble spécifique d'associations entre composants et connecteurs.

Aspect Différents aspects sont décrits dans l'architecture d'un système. Les aspects font référence à des caractéristiques qui peuvent être modélisées de façon plus ou moins indépendantes. Dans les standards comme UML (Unified Modeling Language) et SysML (System Modeling Language), on retrouve une classification des aspects comme suit :

- statique : la description du système ne dépend pas du comportement.
- dynamique : la description du système dépend du comportement. L'aspect est dynamique si la représentation correspond au système qui s'exécute.
- aspect fonctionnel : la description de ce que fait un système. Par exemple, le système doit gérer des données.
- aspect non-fonctionnel : la description de comment une fonction est réalisée. Souvent les aspects fonctionnels sont une description de comment l'architecture réalise les objectifs décrits par les aspects non fonctionnels. Par exemple, les systèmes de gestion de données ne doivent pas être en panne plus de 5 minutes.

Kruchten (1995) propose d'autres aspects (logique, processus, développement et physique) qui capturent la description des architectures à logiciels prépondérants.

Vue et point de vue Du fait de leur taille, les modèles sont organisés en vues et points de vue. D'après Medvidovic and Taylor (2010), les vues représentent les décisions de conception d'un aspect particulier. Les points de vue définissent la perspective d'une vue et spécifient la technique de création et d'analyse utilisée par les modèles. Les points de vue sont établis selon un but (analyse, génération d'artefact, ...) et une audience (utilisateur final, architecte système, ...). Les points de vue sont donc établis dans des buts spécifiques et sont les points d'entrée des outils qui assistent les étapes de développement d'un logiciel.

Les langages et les outils intègrent des processus de développement qui définissent les tâches de modélisation, leur orchestration et les responsables. Les processus définissent quelles sont par exemple les vues qui doivent être développées et à quel moment. Un exemple de processus est RUP (Rational Unified Process) (Kruchten, 2004). C'est un processus de modélisation descendant qui commence par des vues de haut niveau manipulées par les utilisateurs vers des vues de bas niveau qui sont l'implémentation du logiciel. Dans tous les modèles développés, les processus de développement associés doivent permettre que les vues restent exactes par rapport à la réalité décrite. Un modèle est exact si il est factuel ou ne dévie pas trop du fait qu'il modélise.

Selon l'utilisation des modèles produits, il est intéressant de les comparer en fonction de leur ambiguïté et précision. Ces facteurs permettent d'organiser les vues suivant les buts de modélisation. Un modèle est ambiguë si il est ouvert à plusieurs interprétations. Les niveaux d'ambiguïté ne sont pas les mêmes et dépendent des analyses faites. Si l'architecte veut capturer les exigences de l'utilisateur final, il va fournir des langages naturels qui ne décrivent pas les aspects techniques et sont, du fait de leur nature, ouverts à plusieurs interprétations. Un modèle est précis si il est détaillé. Par exemple, quand l'architecte fait

l'analyse statique d'un modèle, il a besoin de la vue statique qui décrit la structure des objets. Il peut ainsi vérifier que son modèle, du point de vue du typage est correct. Si il souhaite faire l'analyse dynamique alors il doit fournir la description du comportement des objets de son modèle.

3.1.2 SysML

SysML (System Modeling Language) est le résultat de la collaboration de l'INCOSE [1] et de l'OMG [2]. L'idée était de fournir un langage de modélisation unifié pour les ingénieurs systèmes sur les mêmes principes que UML et qui permette d'assister les processus de développement des systèmes complexes. De fait SysML étend le langage UML. Il s'agit d'un langage de modélisation architectural à destination du domaine de l'ingénierie des systèmes, qui supporte les phases de spécification, conception, analyse, implémentation, test et déploiement et assiste la vérification et la validation des systèmes complexes. Les systèmes peuvent inclure du logiciel et du matériel. Son principal intérêt réside dans le fait de pouvoir développer des systèmes à logiciels prépondérants.

Les éléments d'un modèle

Un modèle défini avec SysML contient un certain nombre d'éléments :

- Les *blocs* représentent des unités modulaires d'un système. Ils peuvent être utilisés pour représenter des parties logicielles, matérielles ou humaines d'un système. Le bloc étend le concept de classe UML avec le concept de structure composite. Différents types d'interaction peuvent être décrits entre les blocs et peuvent être de l'ordre du logiciel ou du matériel.
- Les *ports* définissent les points d'interaction entre les blocs. SysML propose deux types de ports : le *proxy* qui expose des parties du bloc qu'il possède ou ses ports internes et le *full port* qui supporte ses propres fonctionnalités qui sont directement exposées à l'environnement. Ce type de port peut être utilisé pour modéliser directement des parties matérielles au niveau du logiciel. Les ports peuvent être imbriqués, ce qui permet de décrire des ports connectés à différents clients ou serveurs.
- Les *flux* modélisent ce qui doit circuler entre les éléments du modèle, principalement entre les ports. Cela rajoute de l'expressivité à la description des interconnexions entre les blocs pour différents contextes. Il est possible également de décrire des interconnexions entre les parties matérielles et humaines et/ou logicielles.
- Pour finir *Allocation* est un composant de diagramme qui permet de faire correspondre des éléments entre les diagrammes. La correspondance est orientée. L'ingénieur peut définir très tôt des éléments et fournir une description précise plus tard. Il peut raffiner un cas d'utilisation avec un diagramme d'activité ou encore détailler l'implémentation d'un connecteur.

1. International Council On Systems Engineering
2. Object Management Group

Les diagrammes

SysML hérite d'une partie des diagrammes de UML. Les diagrammes de séquence, machine à état, et cas d'utilisation sont repris tels quels.

- Le diagramme de classes est remplacé par le *BDD (Diagramme de Définition de Bloc)*. Le BDD montre les relations entre les blocs de type généralisation et dépendance.
- Le diagramme de structure composite est remplacé par *BDI (Diagramme Interne de Bloc)*. Le BDI décrit les connecteurs entre les propriétés de blocs. La différence majeure est que les classes sont étendues par le concept de bloc.
- Le BDI remplace également le diagramme de déploiement.
- Les diagrammes d'objets, de communication, d'interaction, temporel, et profils ne sont plus utilisés.
- Le *diagramme paramétrique* permet de décrire les contraintes entre les propriétés des blocs. Cela ajoute de l'expressivité par rapport à UML. En effet, il permet à l'architecte de construire des modèles d'analyse pour, par exemple, mesurer les performances ou la fiabilité d'un système.
- Le *diagramme d'exigence* permet d'organiser les exigences textuelles d'un système notamment en les décomposant en exigences plus simples. Il est ensuite possible d'associer les exigences à d'autres éléments des modèles avec des relations telles que *verify* ou *satify*. La figure 3.1 montre un diagramme d'exigences d'un véhicule hybride. Nous voyons que les exigences comme *Performance* peuvent être décomposées en exigences plus simple : *Braking, FuelEconomy, OffRoadCapability, Acceleration*. La figure 3.2 montre comment le diagramme permet d'associer les exigences aux autres éléments de modèle définis. Différentes relations apparaissent : *HSUVUseCases* raffine *Acceleration*, et *Max Acceleration* vérifie *Acceleration*.

Contribution de SySML à l'ingénierie des systèmes de systèmes

SySML est un framework de modélisation générique qui ignorent les outils et les processus qui l'utilisent. Dans ce sens, les bénéfices attendus de ce langage reposent essentiellement sur les processus et outils mis en œuvre dans la conception de systèmes complexes. Cependant, SysML est un langage de modélisation intéressant pour les SdSs pour au moins deux raisons. La première est qu'il assiste la gestion de modèles complexes caractérisant les SdSs par des mécanismes de traçabilité. Ces mécanismes facilitent la gestion des exigences et l'identification des relations entre les différents niveaux d'abstraction modélisés. Cela facilite l'intégration au processus de modélisation qui raffine successivement les artefacts produits. La maintenance et l'évolution des modèles sont facilitées grâce à la traçabilité permise. La seconde raison est qu'il assiste l'expressivité de la description des interconnexions dans le SdS. L'architecture des systèmes complexes comme les SdSs se concentre sur la description des connexions entre les CSs (systèmes constituants). Par rapport à UML l'expressivité est améliorée principalement grâce à la description de différents aspects des systèmes (physique, logiciel, humain). Cela permet à différents corps d'ingénierie de partager des modèles communs. Cela favorise les approches de modélisation holistique essentielles dans la conception des systèmes complexes.

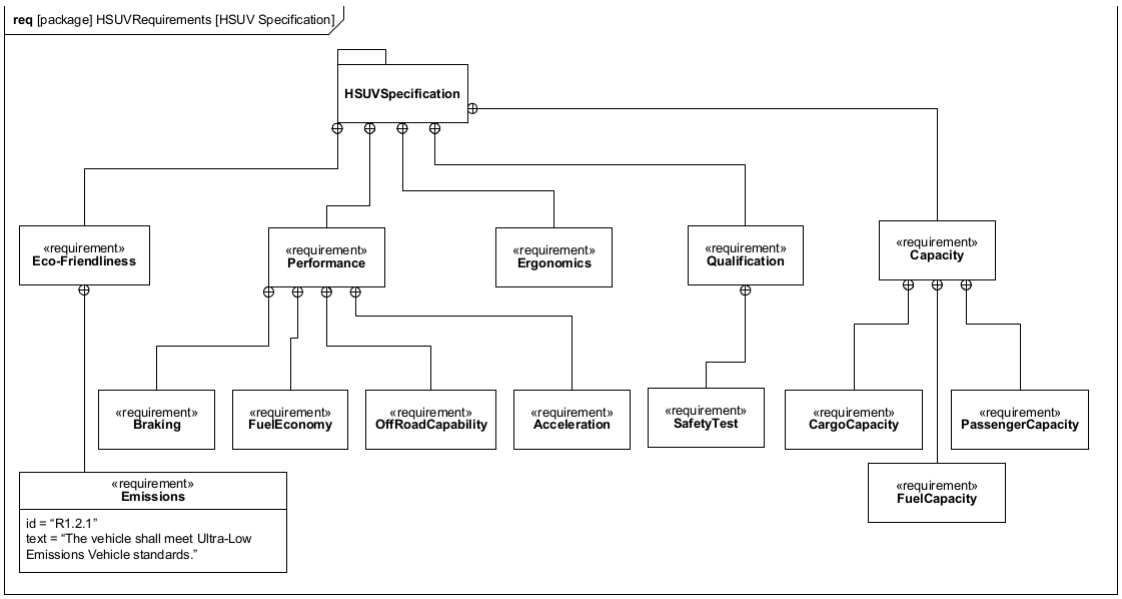

FIGURE 3.1: Décomposition d'exigences (OMG, 2015)

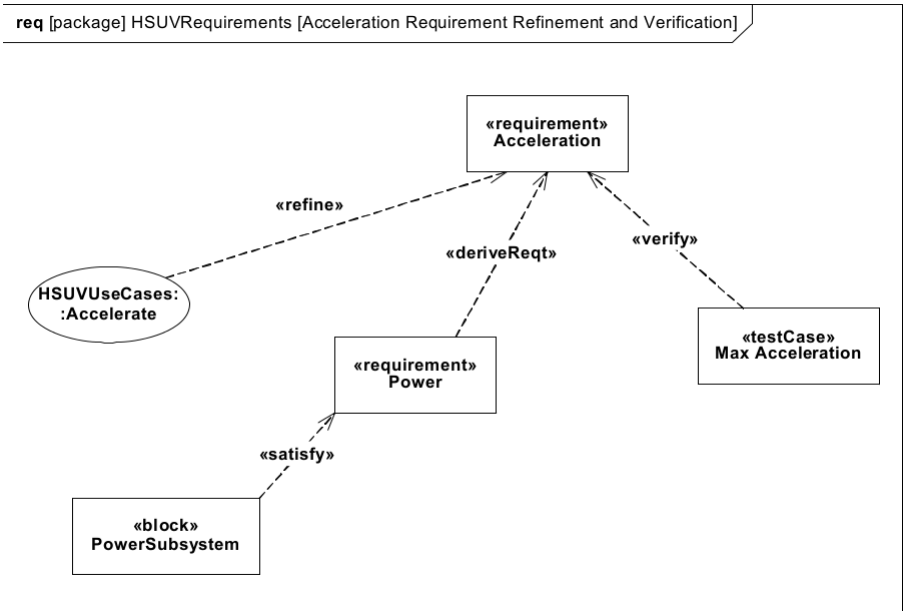

FIGURE 3.2: Exemple de relation entre exigences (OMG, 2015)

3.1.3 UDPM

UPDM (Unified Profile for DoDAF and MODAF) (OMG, 2013) est un framework de modélisation architecturale pour la conception de systèmes de systèmes. Il prescrit un ensemble de plus de 40 vues pour la modélisation de SdS. Comme son nom l'indique, il unifie les frameworks d'architecture DoDAF et MODAF préexistants, respectivement du département de la Défense des États-Unis et du ministère de la Défense du Royaume-Uni. Les principaux groupes de vues de UPDM sont :

- Les vues Acquisition / Project (AcV / PV) décrivent les aspects organisationnels des projets, y compris leur calendrier et les jalons.
- Les All-Views (AV) rassemblent des informations globales et des métadonnées sur les éléments de l'architecture.
- Les vues opérationnelles (OV) sont une collection de vues qui décrivent les activités impliquées dans le SdS, ainsi que les ressources (humaines ou mécaniques) qui exécutent ces activités.
- Les vues orientées services (SOV) décrivent les services, en termes d'interfaces, ainsi que les fonctions que les services sont censés exécuter dans le but de mettre en œuvre les activités décrites dans les vues opérationnelles.
- Les vues stratégiques / de capacités (StV / CV) sont un ensemble de vues à long terme croisées entre projets qui décrivent et organisent les capacités d'une organisation, en prévision de projets.
- Les vues système / services (SV / SvcV) sont une collection de vues qui décrivent comment les capacités opérationnelles, décrites dans les vues opérationnelles, et les exigences des utilisateurs peuvent être réalisées en termes de capacités d'équipement (ou humain), c'est-à-dire la spécification des constituants impliqués dans le système.
- Les vues techniques / standards (TV / StdV) décrivent les technologies, les règles et les normes qui sous-tendent la mise en œuvre du système, fournissant ainsi un outil pour anticiper les progrès technologiques ou les perturbations qui peuvent affecter le système.

UPDM base sa syntaxe graphique sur UML et SysML. Les tableaux 3.1 et 3.2 montrent un échantillon des vues de UPDM. L'échantillon des vues a été choisi en fonction des vues utilisées dans le projet DANSE. Les tableaux font la synthèse des composants, connecteurs et buts d'utilisation des vues. Le contenu a été identifié dans l'étude de cas du standard (OMG, 2013) de UPDM. Les composants *node* des vues opérationnelles modélisent des abstractions qui réalisent des *missions* dans le SdS. Ces missions peuvent être identifiées dans la vue OV-1d. Les missions peuvent être décrites par des *Operational Activities*. Dans les vues systèmes les *nodes* peuvent être raffinés par des types concrets. Ces types peuvent être des *configuration roles* qui modélisent des configurations, les configurations peuvent contenir des *ressources*.

Le tableau 3.3 organise les vues suivant les points de vue que nous avons identifiés (stratégique, opérationnel, ...). Les catégories proposées reflètent le but d'utilisation des vues dans le framework. Elles donnent une idée générale de l'organisation du modèle

Point de Vue	**Operational**					
Vue	ov-1a - concept op. haut niveau	ov-1d - objectif op. haut niveau	ov2 - descript. flux ressources	ov3 - matrice flux ressources	ov 4 - relation organisationnelle	ov5 - activité op.
But	décrire le contexte de la réalisation d'une capacité. Faciliter la communication avec les experts domaine non-familiarisés avec le framework architectural	montrer les objectifs des parties prenantes dans le SdS	identifier le maximum d'informations échangées entre les acteurs des opérations du SdS.	montrer l'avancement de description de l'architecture. (l'architecture est complète si, pour la description d'une capacité, tous les échanges correspondant à des activités sont identifiés.)	montrer les structures de commandement, organisationnelle, responsabilité	montrer les activités réalisées par les noeuds opérationnels, la décomposition des activités opérationnelles, les responsables de leur exécution, et leur orchestration
Composants	pictogramme, arrière plan, photo	mission, actor	Node, NodeRole, Item Flow, Port	producer(node, operational activity), consumer(node, operational activity), information(transported, conveyed)	organization, post, person, ActualOrganization, ActualPost, ActualPerson	node, NodeRole, operational activity, initial state, final state, flow, swimlane
Connecteurs	relation conceptuelle abstraite	extend, include, inherit, interact	exchange : data, equipment, energy	needline	inheritance, member, suborganization, fillsPost	perform, composition, relationship directed

Table 3.1: Echantillon des vues opérationelles

Point de Vue	**Système**					
Vue	sv-1 - descr. interface sys.	sv-2 - descr. flux ressources	sv-3 - matrice interactions et fonctions sys.	sv-4 - descr. fonctionnelle syst.	sv-5 - descr. activité opérationnelle	sv-6 - matrice flux et mesures
But	définir l'architecture du SdS en termes de type de composants et connecteurs	définir le réseau de communication physique du SdS	résumer les interactions décrites dans sv-1	définir les fonctions supportées ; montrer les ressources qui réalisent les fonctions ; planifier l'utilisation des ressources (qui peut faire quoi ?)	montrer la réalisation des activités opérationnelles et/ou service ; vérifier que les exigences des vues opérationnelles sont adressées dans les vues systèmes.	synthétiser les interactions sv1&2 ; effectuer les mesures associées.
Composants	configuration-Role, organizationRole, post, role, organization	configuration-Role, ResourceArtefactRole, flux	post, resource-Artefacts	capability-Configuration, function, post	functionalActiity,	Resource interaction, producer, consumer, interface
Connecteurs	resource-Interaction, composition	composition, connection	interaction	perfom, decomposition de fonction	implement-Operation-Activity	

TABLE 3.2: Echantillon des vues systèmes

VIEWPOINT		CATÉGORIE				
		Comportement	**Structure**	**Synthèse**	**Correspondance**	**Ontologie**
	All Views			AV-1		AV-2
Strategic			St-V-4	St-V-1	St-V-3 St-V-5 St-V-6	St-V-2
Operational		OV-5 OV-6a OV-6b OV-6c	OV-1a OV-2 OV-4 OV-7	OV-1b OV-1c OV-3		
System		SV-4 SV-10a SV-10b SV-10c SV-8	SV-1 SV-2a SV-2b SV-2c SV-11	SV-6 SV-7 SV-9	SV-3 SV-5 SV-12	
Technical				TV-1 TV-2		
Acquisition		AcV-2	AcV-1			
Service Oriented		SOV-4a SOV-4b SOV-4c SOV-5		SOV-2	SOV-3	SOV-1

Table 3.3: Catégorisation des vues UPDM

développé pour le SdS.

- La catégorie *structure* regroupe les vues qui décrivent des aspects organisationnels de l'architecture. Par exemple, dans le point de vue système, la vue SV-1 peut être utilisée pour décrire la décomposition d'une abstraction et les relations entre elles.
- La catégorie *comportement* décrit les aspects de l'architecture qui décrivent le fonctionnement de l'architecture du système. Par exemple, la vue SV-4 peut être utilisée pour décrire les fonctions d'une configuration.
- La catégorie *correspondance* regroupe les vues qui lient les éléments de modèle entre deux points de vue. Par exemple, les comportements décrits par OV-5 peuvent être exprimés avec des relations de correspondance dans le point de vue système avec SV-5. Les correspondances peuvent modéliser le même type de relation mais entre des éléments du même point de vue. Par exemple, les cas d'utilisation du SdS décrits dans ov-1a peuvent être détaillés dans OV-5. Dans ce cas le raffinement est entre éléments d'un même point de vue.
- La catégorie *synthèse* regroupe les vues qui montrent l'état d'avancement de description d'un aspect du modèle. Par exemple, la vue OV-3 met en évidence les échanges décrits par OV-2 et qui ne sont pas associés à une activité décrite par OV-5.
- La catégorie *ontologie* regroupe les vues qui définissent les termes du domaine de l'architecture et les relations entre eux.

Contribution de UPDM à l'ingénierie des systèmes de systèmes

À la différence de UML et SysML, UPDM fournit une sémantique plus précise des éléments modélisés. En effet, UPDM raffine la notion de bloc par des rôles (configuration role, etc.). De plus, comme on le verra dans le projet DANSE, UPDM fournit un point de vue SdS avec le point de vue opérationnel et un point de vue CS avec le point de vue système. Cela fournit à l'architecte une séparation des principes architecturaux qui dépendent du SdS de ceux qui dépendent des CSs. La séparation des points de vue SdS et CS permet d'expliciter les décisions architecturales qui découlent du SdS et celles qui dépendent des CSs. D'autre part, les SdSs sont formés par une composition de systèmes managérialement indépendants. Cela amène des problèmes d'intégration des CSs. En effet, chaque CS possède ses propres modèles pour représenter son architecture. UPDM fournit une base de modélisation commune. L'avantage est que cette base commune de représentation aide l'architecte à mieux comprendre l'architecture des CSs et ainsi de faciliter leur intégration dans le SdS.

3.2 Le projet COMPASS

Les objectifs du projet COMPASS (Comprehensive Modelling for Advanced Systems of Systems) sont d'améliorer la capacité d'implémentation des changements à travers un SdS . Les changements qui sont envisagés sont l'évolution des constituants, le changement d'environnement et l'ajout de nouveaux systèmes constituants. La contribution de COMPASS est le développement d'un framework de modélisation (Forcolin et al., 2013)

permettant de mettre en œuvre des techniques d'analyse des propriétés dans les systèmes de systèmes (tolérance aux fautes, comportement émergent).

Le cas d'étude de COMPASS est également un SdS dans le domaine de l'organisation de services de secours, décrit dans le rapport (Forcolin et al., 2013). Il se focalise sur la mission de déclenchement d'une intervention. Les systèmes considérés sont : un réseau téléphonique fourni par un opérateur privé (*Phone system*), un centre de traitement des appels d'urgence (*Call Center CUS*) géré par les services de secours, un système radio (*Radio System*) permettant la communication entre les ressources des services de secours et des unités de réponse urgente (*ERU*) comme des hélicoptères et des pompiers (*Fire Brigade*).

3.2.1 Framework COMPASS

Le projet COMPASS a développé son propre canevas de modélisation indépendamment des frameworks standardisés comme UPDM (Unified Profile for DoDAF and MODAF). Dans le but de comparer ce modèle avec UPDM, nous proposons un regroupement des diagrammes utilisés en *vues* puis *points de vue*, avec une nomenclature destinée à faciliter la description du projet COMPASS. Ce regroupement se base sur la précision des abstractions modélisées par les diagrammes et l'objectif des diagrammes. Le tableau 3.6 montre les vues développées dans le cas d'étude et les diagrammes utilisés pour chaque vue. Les tableaux 3.4 et 3.5 montrent une description des vues en terme de but, composant et connecteur. Le tableau 3.7 montre le regroupement par catégorie structurelle, comportementale, ontologie et correspondance. L'ordre de développement des modèles suit un processus de description descendant. Les diagrammes sont donc développés du moins précis au plus précis. Les vues décrites dans les sous-sections suivantes sont décrites dans le rapport (Forcolin et al., 2013).

3.2.2 Phase de modélisation de l'architecture niveau SdS

L'objectif des vues 1-1 à 1-6 est d'identifier les frontière du SdS en exposant les objectifs, les systèmes impliqués, les ressources déployées, en utilisant des modèles semi-formels.

La vue-1-1 *emergency response SoS boundary* capture l'environnement du SdS. La notion d'environnement permet de décrire quels sont les éléments que ne peuvent pas influencer le SdS. Dans ce cas, par exemple, la police est modélisée comme faisant partie de l'environnement car dans ce SdS la police communique des informations mais n'est pas subordonnée à des directives d'un composant du SdS. Nous remarquons que le diagramme ne correspond pas à la syntaxe concrète de SysML étant donnée la représentation graphique de la frontière utilisée.

Les vues-1-2 montrent une décomposition du SdS en systèmes constituants (CSs). Plusieurs compositions différentes sont décrites pour modéliser des aspects non fonctionnels comme la tolérance aux fautes. Les vues-1-2 utilisent un diagramme de définition de bloc (BDD).

Les vues-1-3 définissent une ontologie du domaine du SdS. Le diagramme de la vue décrit, par exemple, qu'un service d'urgence est composé de patients et d'au moins une mission.

Les vues vue-1-4 et vue-1-5 explicitent les exigences attendues des CSs dans le SdS. La vue-1-4 concerne les aspects fonctionnels des SdSs. Un exemple d'exigence fonctionnelle est le centre d'appel d'urgence(CUS) qui, pour chaque appel entrant reçu du réseau téléphonique, va évaluer la situation et activer les ressources requises. Les vues-1-5 concernent les aspects non-fonctionnels. Pour les vues 1-4 et 1-5, ce sont des diagrammes d'exigence de SysML qui sont utilisés pour la modélisation. Ce type de diagrammes permet en effet à l'architecte de traçer la réalisation des exigences à travers les autres phases de développement.

Les vues-1-6 montrent les comportements exposés par le SdS et par les ressources mettant en œuvre le SdS. Elles sont décrites par un diagramme de cas d'utilisation. Par exemple, le cas d'utilisation de surveillance des ressources exprime que l'opérateur CUS collecte les informations de surveillance et l'unité de réponse urgente (ERU) envoie ces informations. Le diagramme fait apparaître une frontière modélisant les cas d'utilisation qui sont du ressort du SdS et ceux qui ne le sont pas.

3.2.3 Phase de modélisation de l'architecture niveau CSs

Les contributions du projet COMPASS concernent la vérification du comportement émergent des SdSs. Une première solution proposée par Bryans et al. (2013) est une méthode systématique pour capturer les interfaces des systèmes constituants (CSs). Ces interfaces décrivent les contrats que devront respecter les CSs afin d'assurer le comportement émergent du SdS.

Les vues -2-1 détaillent quelles sont les ressources qui peuvent réaliser les comportements exposés par le SdS. Par exemple, le diagramme *Insiel behavioural block structure diagram* montre que le CUS peut être chargé de planifier les missions, traiter les appels et surveiller les interventions.

Les vues-2-4 montrent des aspects structurels. Elles montrent les relations entre les composants et les fonctionnalités. Par exemple, le diagramme *Call Centre CUS internal structure* montre que *Human operator* peut jouer le rôle d'opérateur du centre d'appel et participer à la fonction Traitement d'appel du CUS.

Les vues-2-2 montrent les configurations correspondant aux fonctions identifiées précédemment. Les diagrammes utilisés sont conjointement le BDD et le BDI. Par exemple, le diagramme *Insiel SoS deployment structure*' montre la décomposition de CSs en composants humains et logiciels avec un BDD. La vue *constituent system external interfaces* montre les interfaces et dépendances exposées par les CSs pour l'ensemble des fonctionnalités du SdS. Les interfaces montrent les dépendances entre les systèmes et le type des

données échangées. Les vues : CUS Phone Answering, CUS Scheduling, CUS Monitoring sont des simplifications de la vue *Call Centre CUS internal Structure*. Elles montrent, dans un diagramme BDI, les connexions pour chaque fonction. Les diagrammes présentés sont différents mais, ensemble, ces vues modélisent la configuration de l'architecture.

La vue-2-3 fait la synthèse des dépendances décrites pour chaque fonction.

La vue-2-5 capture les aspects comportementaux qui résultent de l'interaction des CSs. Chaque comportement correspond à un processus, qui est décrit par un diagramme d'activé et un diagramme de séquence. Par exemple, la vue *Process Call* montre les actions qui constituent le processus, ainsi que les CSs responsables de la réalisation de ces actions. Dans le traitement d'appel, le système téléphonique met en attente l'appel entrant puis le CUS affiche l'appel en attente. Ensuite, un diagramme de séquence est utilisé pour modéliser le flux d'information entre les CSs. La vue *Process Call* montre ainsi les envois de messages entre les CSs.

3.2.4 Phase d'analyse du modèle

Les modèles raffinés par le point de vue 2 permettent de modéliser les contrats des systèmes constituants (CSs) dans le SdS. Cette partie du modèle permet à l'architecte d'obtenir une représentation de l'architecture afin de vérifier des propriétés. Dans le projet COMPASS, la vérification est réalisée par simulation.

Coleman et al. (2012) fournissent les règles de transformation pour traduire le modèle SySML en un modèle CML (COMPASS Modelling Language (Woodcock et al., 2012)). CML est conçu pour modéliser des données mais aussi l'ordre des événements dans un système. En plus de la possibilité de vérifier le comportement émergent, Coleman et al. suggèrent de vérifier les contrats pour s'assurer que la spécification des systèmes concrets est compatible avec les contrats. La difficulté prise en compte est que la spécification des CSs diverge des contrats fournis par le SdS, car les CSs possèdent des interactions supplémentaires dans le cadre de l'indépendance opérationnelle des CSs, une des caractéristiques des SdSs. Pour ce problème précis, Coleman et al. proposent de formaliser le modèle SysML des contrats et des CSs, et d'utiliser cette version formalisée pour vérifier que la spécification des CSs est bien un raffinement de la spécification du contrat.

La notation formelle utilisée est CSP (Hoare, 2000). La transformation vers CSP n'est pas directe mais passe un réseau de Pétri (Petri, 1962). Les propriétés qui sont vérifiées sont l'absence de faute et la tolérance aux fautes. L'absence de faute est une propriété de sûreté, et sa vérification consiste donc à garantir que certains états indésirables ne sont pas atteignables. La tolérance aux fautes consiste à vérifier que le SdS pourra retourner dans un état normal s'il vient à se trouver dans un état d'erreur.

Point de Vue	**Point vue 1**					
Vue	vue-1-1	vue-1-2	vue-1-3	vue-1-4	vue-1-5	vue-1-6
But	délimiter l'environnement du SdS	décomposition du SdS en ensemble de CSs	définir une ontologie du domaine de l'architecture	définir les exigences fonctionnelles	définir les exigences non fonctionnelles	exposer la portée fonctionnelle du SdS
Composant	frontière, environnement, système	système	terme	exigence	exigence	cas d'utilisation, acteur, environnement
Connecteur	connexion, composition, aggrégation	connexion	association, composition, aggrégation	trace, contenu	trace, contenu	interaction

TABLE 3.4: Point de vue 1 du framework COMPASS

Point de Vue	**Point vue 2**				
Vue	vue-2-1	vue-2-2	vue-2-3	vue-2-4	vue-2-5
But	montrer les fonctions des CSs dans le SdS.	montrer les configurations de composant correspondantes aux fonctions du SdS	synthétiser les dépendances des CSs définies par les configurations	montrer les composants participants à la réalisation des fonctions	montrer les comportements impliqués dans le SdS
Composant	système, fonction	système, composant logiciel, composant humain, port	système, composant logiciel, composant humain, port	système, composant logiciel, composant humain, fonction	processus, système, opération
Connecteur	composition, aggrégation, association	interface fournie, interface requise, connexion	interface fournie, interface requise, connexion	a rôle, assiste, déclenché par, inclus, se réfère, représente, opéré par, utilisé par	dépend, successeur

Table 3.5: Echantillon du point de vue 2

	Vues	Nom	Diagrammes
Logique	vue-1-1	emergency response SoS boundary	schéma
	vue-1-2	SoS overview Structure Fault Tolerance [block]	bdd
		baseline SoS Overview [package]	bdd
		improved SoS Overview [package]	bdd
	vue-1-3	Insiel SoS key entity structure	bdd
	vue-1-4	Functional Requirements	diag. exigence
	vue-1-5	Non Functional Requirements	diag. exigence
	vue-1-6	case study SoS boundary	cas d'utilisation
Physique	vue-2-1	Insiel behavioural block structure diagram [block]	bdd
	vue-2-2	Insiel SoS deployment structure [block]	bdd
		CUS Monitoring [block]	bdi
		CUS Phone Answering [block]	bdi
		CUS Scheduling [block]	bdi
	vue-2-3	CUS External interfaces[block]	bdi
	vue-2-4	Call Centre "CUS" internal Structure	bdd
	vue-2-5	case study process [block]	bdd
		Process Call [ac]	diag. activité, diag. séquence
		Answer Call	diag. activité, diag. séquence
		Obtain casuality detail	diag. activité, diag. séquence

Table 3.6: Synthèse des diagrammes de l'étude de cas COMPASS

		CATEGORIE				
		Comportement	Structure	Synthèse	Correspondance	Ontologie
POINT DE VUE	Point de vue 1	vue-1-4, vue-1-5, vue-1-6	vue-1-1, vue-1-2			vue-1-3
	Point de vue 2	vue-2-5	vue-2-1, vue-2-2	vue-2-3	vue-2-4	

Table 3.7: Catégorisation des vues du framework COMPASS

3.3 Le projet DANSE

DANSE (Designing for Adaptability and evolutioN in System of systems Engineering) (Shani et al., 2015) est un projet européen qui a étudié les méthodes d'ingénierie des systèmes de systèmes, et plus particulièrement les besoins en termes d'adaptation et d'évolution. Ces besoins découlent de la caractéristique de développement évolutionnaire décrite dans le chapitre 1.1, mais aussi de la taille des SdSs. Le comportement évolutionnaire implique de vérifier que l'architecture reste cohérente chaque fois que l'architecture est modifiée par adaptation ou évolution. Pour aider l'architecte face à des SdSs, qui sont des systèmes de grande taille, le projet DANSE met en œuvre des outils de génération de code pour générer automatiquement des architectures, et des outils de simulation pour vérifier les propriétés de l'architecture développée.

L'approche de modélisation d'un système de systèmes dans le projet DANSE repose sur le framework UPDM présenté à la section 3.1.3, en sélectionnant certaines vues. La modélisation du cas d'étude développé par le projet DANSE met en évidence les vues de UPDM sélectionnées. Il s'agit d'un SdS d'organisation des services de sécurité civile, qui décrit la réponse à une urgence. Le SdS est formé temporairement par les différents services de secours pour la résolution d'une crise. Les services impliqués peuvent être la police, des pompiers, etc. La modélisation du cas d'étude se concentre sur le centre de commandement central (CCC) qui coordonne les ressources allouées pour la résolution de crise. DANSE a choisi d'utiliser le framework UPDM comme représentation standard du SdS pour ces outils d'ingénierie. Le tableau 3.8 montre la synthèse des vues employées dans le cas d'étude. Dans cette section, nous allons étudier les choix de modélisation du projet DANSE, en particulier le choix des vues de UPDM. Nous allons également étudier les processus de conception développés. Notre étude repose sur l'analyse des livrables du projet DANSE (Shani et al., 2015; Etzien et al., 2014; Lochow et al., 2013), qui présentent le cas d'étude du projet et les outils mis en œuvre dans le processus de conception des SdSs.

Le processus DANSE suit une approche de modélisation descendante. Dans un premier temps, l'architecte modélise l'architecture logique du SdS avec le point de vue opérationnel de UPDM. Puis, l'architecte raffine ce point de vue opérationnel jusqu'à obtenir une architecture physique avec le point de vue système de UPDM. Il peut utiliser des patrons pour réaliser les tâches de conception. Il s'agit de patrons opérationnels pour l'architecture logique, de patrons fonctionnels et système pour l'architecture physique (Kalawsky et al., 2014). Dans le projet DANSE, l'outillage est basé sur *IBM Rational Software Architect*.

Les modèles ainsi produits permettent de générer une architecture à partir d'une description partielle, puis une phase d'analyse sert à vérifier que le comportement du SdS est bien conforme à l'architecture.

3.3.1 Phase de modélisation de l'architecture niveau SdS

L'objectif de la phase de modélisation de l'architecture logique est de décrire les principales abstractions du SdS et leurs relations. Le modèle est développé à partir des points de vue opérationnels de UPDM. DANSE a restreint le point de vue aux vues : OV-1a (concept op. haut niveau), OV-2 (descript. flux ressources), OV-3 (matrice flux ressources) et OV-5 (activité op.).

La vue OV-1 est utilisée pour décrire statiquement le contexte du SdS en modélisant les principales ressources déployées et les relations qui les structurent. Dans le cas d'étude, le contexte contient notamment : un CCC (un centre de commande et de contrôle), un centre de contrôle de pompiers, de police et leurs unités mobiles. Toutes ces unités interagissent par des relations de commande et de contrôle. Le diagramme met en avant les ressources qui contribuent au SdS avec un diagramme de bloc. Le diagramme est informel et est destiné à l'ensemble des parties prenantes du SdS.

La vue OV-2 détaille statiquement les connexions entre les ressources contribuant au SdS. Elle modélise les interactions possibles entre les ressources ainsi que les données échangées.

La vue OV-5 présente les activités majeures du SdS. Cette vue comprend les comportements attendus, et indique quels systèmes constituants (CSs) les accomplissent. Les CSs qui collaborent sont annotés par *NodeRoles*. Dans l'étude de cas complète de DANSE, différentes perspectives sont développées (management des ressources, des accidents, etc.). La vue OV-5 montre des aspects comportementaux qui sont des scénarios d'exécution. Par exemple, dans la vue *House Fire Scenario* d'un scénario d'une maison en feu, le diagramme d'activité montre que le CCC réceptionne l'appel, collecte des informations et décide s'il faut traiter l'urgence. La vue indique quels sont les autres noeuds opérationnels qui participent au SdS dans ce scénario précis, à savoir, CC (Centre de contrôle) police et CC pompier. La vue OV-2 décrit également les échanges de données entre ces noeuds opérationnels.

La vue OV-3 fait la synthèse des informations de OV-2. Elle montre en particulier les activités produites par les interactions et les dépendances requises pour la réalisation d'une activité.

3.3.2 Phase de modélisation de l'architecture niveau CSs

L'objectif de la modélisation de l'architecture physique est de capturer les décisions de conception au niveau des CSs. Ils s'agit principalement d'identifier les interfaces des composants dans le cas d'étude. Le point de vue système de UPDM est utilisé, mais restreint aux vues SV-1 et SV-4.

La vue SV-1 indique la composition des ressources nécessaires. Les aspects sont modélisés avec un BDI, qui décrit les interfaces utilisées pour la réalisation des interactions. Dans la modélisation des relations d'autorité, la relation <<system part>> montre qu'un système a un contrôle total ou partiel sur les systèmes qu'il encapsule. Elle n'apparaît pas dans le modèle mais une relation *command* peut être utilisée pour montrer qu'un système peut commander à un autre d'exécuter une activité. La relation *control* est utilisée pour modéliser qu'un système peut changer le rôle d'un autre système.

Les vues SV-4 sont utilisées pour modéliser des scénarios d'exécutions liés aux activités décrites dans les vues OV-5. Par exemple, la vue *Receiving Call* montre, par un diagramme d'activité, les actions impliquées par la réception d'un appel.

Le modèle de l'architecture développé précédemment permet de décrire le type des ressources déployées. Il comprend une description des interfaces de chaque ressource et de leur connexion avec l'environnement. Avec un ensemble de stéréotypes (Shani et al., 2015) développé par DANSE l'architecte identifie les blocs de SV-1 avec <<catalog>> et les connecteurs avec <<TypedConnector>>. Le stéréotype <<catalog>> fait correspondre les blocs avec un ensemble des composants physiques qui peuvent se substituer aux blocs. C'est le même principe pour les <<TypedConnector>>. Les stéréotypes <<optimized>> et <<derived>> décrivent des parties du modèle qui sont calculables à partir d'autres éléments, ainsi que des parties du modèle qui modélisent des parties variables de l'architecture. Pour finir l'architecte définit des contrats qui prennent la forme de contraintes ayant pour syntaxe le langage GCSL (Goal Contracts Specification Language)(Mangeruca et al., 2013).

Le tableau 3.8 fait la synthèse des vues utilisées dans les étapes de modélisations de l'architecture (logique et physique). Il fait apparaître les diagrammes utilisées et nous donne une indication sur les aspects modélisés de l'architecture.

Points de Vues	**Vues**	**Nom**	**Diagrammes**
Opérationnel	ov-1a	ER SoS High level operational view	schéma
	ov-2	Centralized Controle	bdi
	ov-5	Manage Incident	diag. activité
	ov-5	Manage Operational Data	diag. activité
	ov-5	Receive Emergency Call	diag. activité
	ov-5	Resources Management	diag. activité
	ov-5	House Fire Scenario	diag. activité
Système	sv-1	system and service interface description	bdd
	sv-4	Receiving Call	diag. activité
	sv-4	External Resource Dispatching	diag. activité
	sv-4	Monitoring Emergency Case	diag. activité

Table 3.8: Synthèse des diagrammes du cas d'étude de DANSE

3.3.3 Phase de génération d'une architecture

Le but de cette phase est de générer une architecture qui satisfait toutes les contraintes décrites par les contrats identifiés pour chacun des CSs. L'outil utilisé est un solveur de contraintes multi-objectif [3].

Une fois l'architecture décrite s'en suit une étape d'exploration de l'espace de conception. Une instance architecturale conforme aux descriptions de l'architecture décrite précédemment est générée. Pour cela l'outil fait varier l'ensemble des paramètres décrits dans l'architecture pour générer des instances compatibles. Le générateur choisit des composants et des connecteurs physiques parmi le catalogue des éléments disponibles et les substitue aux éléments annotés par les stéeréotypes <<TypedConnector>> et <<catalog>>. Les paramètres sont les contraintes définies par les contrats. Les contraintes peuvent être minimisées/maximisées, actives/inactives, et avoir un niveau de priorité. L'architecture proposée à l'architecte est celle qui optimise le mieux l'ensemble des objectifs définis. Cette architecture correspond donc à celle qui inclut la partie physique (hardware, cablage) de l'architecture. Par exemple, deux composants doivent être connectés par un cable. Le générateur peut prendre en compte cette information pour déduire la longueur du cable nécessaire. Le générateur calcule des valeurs qui ne sont pas connues au moment de la modélisation et qui dépendent de la valeur d'autres paramètres (choisis au moment de la génération). Ces valeurs sont annotées par les stéréotypes <<optimized>> et <<derived>>. Elles sont calculées à partir de formules mathématiques.

3.3.4 Phase d'analyse de l'architecture

Les objectifs de la phase d'analyse sont principalement de prendre en compte la caractéristique de développement évolutionnaire du SdS. L'approche se fait par simulation de modèles. L'architecture est, en premier, transformée en un modèle exécutable par l'outil de *model-checking* statistique PLASMA (Jegourel et al., 2012). L'architecte décrit une boucle de contrôle qui gère l'exécution du modèle en utilisant le formalisme des grammaires de graphe (König, 2014), c'est-à-dire que l'architecte donne un ensemble de règles qui indiquent, pour chaque événement, une transformation du modèle de l'architecture, vu comme un graphe. Ces règles sont décrites par des stéréotypes qui annotent les événements dans la vue SV-4, le détail est donné dans le rapport (Etzien et al., 2014). Le *model-checking* statistique est une technique pour résoudre le problème du passage à l'échelle. Plutôt qu'une exploration exhaustive de l'espace des états, comme cela est réalisé par les approches traditionnelles de *model-checking*, les règles indiquent la distribution (au sens de la statistique) des variables : une distribution uniforme, normale ou quelconque. La loi de distribution ainsi indiquée donne la probabilité pour chaque valeur de la variable concernée, et est utilisée pour générer et sélectionner aléatoirement les chemins d'exécution les plus probablement représentatifs.

3. `https://www.ibm.com/analytics/cplex-optimizer` - consulté le 05/09/18

TABLE 3.9: Mise en correspondance des vues de COMPASS et DANSE

COMPASS \ DANSE	ov-1	ov-2	ov-5
vue-1-1	x		
vue-1-2		x	
vue-1-7	x		
vue-1-8			x

COMPASS \ DANSE	sv-1	sv-4
vue-2–2	x	
vue-2–3	x	
vue-2–5		x

3.4 Synthèse

Ce chapitre confirme l'importance des architectures dans le processus de développement des SdSs. Les architectures sont les artefacts centraux des phases d'analyse des SdSs. Nous voyons que les principales difficultés sont la définition des abstractions et des comportements ainsi que les principes architecturaux requis pour la réalisation du SdS.

Nous avons montré les principales caractéristiques des langages de modélisation architecturale et leur utilité. SysML et UPDM apportent des éléments d'expression qui facilitent la description à des niveaux d'abstraction de nature différente d'un système. La description des composants avec la notion de bloc permet de faire apparaître les parties indépendantes d'un système. Les possibilités de description des interconnexions entre ces parties indépendantes et de natures différentes enrichissent également l'expressivité du langage et les mécanismes de traçabilité. UPDM apporte, en particulier, un ensemble de concepts qui permet une séparation dans le modèle des parties qui concernent les niveaux SdS et CS. Le framework de modélisation de COMPASS est couvert par le framework UPDM comme le montre le tableau 3.9. Le tableau montre la correspondance des vues des frameworks de modélisation COMPASS et DANSE. Il nous donne une indication sur la pertinence de UPDM pour la modélisation des SdSs.

Le framework UPDM est utilisé comme description semi-formelle de l'architecture. La vue opérationnelle permet de modéliser les principales abstractions et comportements requis à la réalisation du SdS. Le projet DANSE a développé des patrons architecturaux pour assister la phase d'identification. Le projet COMPASS réutilise les mêmes concepts mais ajoute la notion d'environnement aux éléments de modélisation. Cela permet de faire apparaître clairement les frontières du SdS. Ceci est utile dans les phases de réutilisation des modèles architecturaux.

Cependant, les travaux de DANSE et COMPASS montrent que ces modèles architecturaux ne sont pas suffisants. En effet, nous avons vu qu'ils s'appuient sur des langages

formels pour les phases de génération automatique et d'analyse de l'architecture. Les langages formels permettent de capturer, en particulier, la notion de contrat. Le choix d'une approche par contrat repose sur la caractéristique d'indépendance des systèmes. Il convient donc de vérifier d'une part que les spécifications d'un système sont bien compatibles avec le SdS, et d'autre part que la composition les contrats est robuste. Les contrats permettent également de prendre en compte la contrainte que les CSs ne sont pas connus à l'avance. Les architectures abstraites ne peuvent donc être que partiellement vérifiées avant de connaître concrètement les CSs.

Concernant le processus de développement, les deux projets montrent qu'il est intégré dans un atelier de développement qui met en relation les différents outils développés dans les projets. Celui de COMPASS est basé sur Artisan Studio. Il connecte un environnement de modélisation basé sur le framework développé avec des outils de transformation de modèles et de simulation. Les modèles en entrée sont donc des modèles du framework et la sortie sont des modèles CML (COMPASS Modelling Language). Celui de DANSE fonctionne sur le même principe. L'environnement de modélisation est basé sur Rhapsody Architecture. Les outils mis en œuvre sont des outils de simulation avec PLASMA.

Le processus de développement montre que les ateliers se basent sur des descriptions partielles de l'architecture. Les processus de raffinement des architectures sont précédés des phases d'analyse qui garantissent un certain niveau de certitude sur l'implémentation du SdS. Par rapport à d'autres types de processus comme RUP (Rational Unified Process), les modèles en cascade ou le cycle en V, les phases de vérification sont réalisées dès la conception et non plus après la phase d'implémentation.

Du point de vue de la reconfiguration dynamique, les phases de génération d'architecture peuvent être vues comme des reconfigurations puisqu'elles peuvent redéfinir des configurations. Cependant les reconfigurations sont statiques et non dynamiques. En effet, pour prendre en compte la reconfiguration dynamique il faudrait que le générateur décrivent différents états de l'architecture ainsi que les transitions qui permettent de passer d'un état à l'autre.

Par conséquent, nous envisageons de développer notre propre cas d'étude pour étudier notre proposition pour la reconfiguration dynamique des architectures de SdS. Une première contribution sera donc la modélisation de l'architecture du SdS du cas d'étude comportant des cas d'évolution. Une difficulté dans la modélisation d'un nouveau cas d'étude est la vérification de son exactitude. Dans les cas d'étude développés que nous avons étudiés, l'exactitude n'est pas prise en compte dans le développement. Nous envisageons aussi d'intégrer une approche de vérification des reconfigurations développées qu'il faudra prendre en compte dans les modèles architecturaux et processus utilisés. Le processus de développement du modèle architectural devrait reprendre les caractéristiques communes aux processus de développement des deux projets étudiés, à savoir, une phase de modélisation de l'architecture logique et physique, et une phase de vérification de l'architecture logique devrait être ajoutée.

Les modèles développés devraient reprendre des éléments de UPDM. Dans la phase de modélisation de l'architecture logique, nous envisageons d'utiliser des vues du point de vue opérationnel. Par rapport à DANSE, les choix des vues et des diagrammes doivent être différents pour faire mieux apparaître les objectifs d'interaction. Ensuite, le point de vue système de UPDM permet de capturer les décisions de conception, notamment en termes d'interfaces. Comme les deux projets COMPASS et DANSE, nous envisageons des approches plus précises. Notre objectif est de modéliser les aspects statiques pour minimiser l'effort de modélisation en intégrant les aspects dynamiques pour ajouter de la précision aux architectures. Pour vérifier les reconfigurations développées, nous utiliserons des approches par simulation, les modèles intègreront des scénarios d'exécution.

Par ailleurs, les approches de reconfiguration automatique comme la génération d'architectures est limitée comme le montre d'autres travaux (Guessi et al., 2016). Les temps de résolution peuvent être trop longs pour espérer générer automatiquement de nouvelles architectures pendant l'exécution. Une autre limite concernent les outils d'analyse qui reposent sur des modèles dynamiques, ce qui limite fortement les bénéfices des outils lorsqu'ils sont générés par des approches manuelles.

Deuxième partie

Contribution

Chapitre 4

Etude de cas : le service des secours français

Dans ce chapitre nous présentons un cas d'étude qui sera utilisé dans le manuscript pour expliquer et illustrer notre approche de modélisation et reconfiguration des systèmes de systèmes. Ce cas d'étude porte sur l'organisation des services de secours, qui est un cas typique d'exemple de SdS (système de systèmes). Plus exactement nous avons choisi de modéliser les services de secours français dont la mission principale est d'assurer le secours à la personne et la protection des biens en cas de sinistre. La section 4.1 présente le contexte du SdS. Cette section décrit la mission du SdS, les différentes organisations participant au SdS, les ressources déployées par ces organisations et les principes structurant leurs collaborations. La section 4.2 présente un scénario d'exécution du SdS mettant en avant son comportement évolutionnaire. Ce scénario met en évidence trois configurations qui seront utilisées pour étudier, en particulier, la reconfiguration dynamique dans les SdSs. Pour finir, la section 4.3 montre que notre étude de cas satisfait les définitions de SdS de l'état de l'art. Puis cette même section positionne notre étude de cas par rapport aux autres définies dans le domaine des services de secours.

4.1 Contexte

La mission principale sur laquelle se concentre notre cas d'étude est la protection des personnes, des biens et de l'environnement [1]. Il peut s'agir du transport d'une personne à l'hôpital, de limiter la progression d'une inondation, d'un incendie ou de circonscrire une pollution fluviale. De manière générale, un des objectifs de la mission est d'empêcher l'évolution d'un sinistre et de revenir à une situation normale.

Cette mission est propre aux services des pompiers. Cependant sa réalisation peut dépendre de la participation d'autres services de secours. La figure 4.1 montre un exemple de collaboration entre différents services de secours suite à une inondation. Les principaux CSs (systèmes constituants) sont :

1. `https://www.collectivites-locales.gouv.fr/files/m61tome1titre12010.pdf` consulté de 12/09/2018

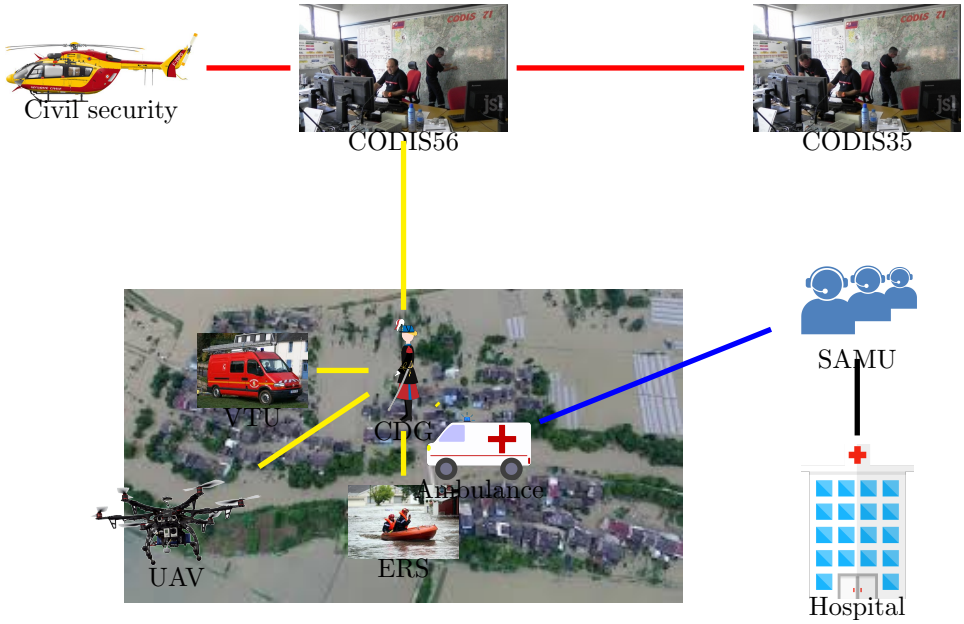

FIGURE 4.1: Exemple de collaboration des services de secours Français dans le cas d'une inondation

— Les hôpitaux prennent médicalement en charge les victimes.
— Les pompiers, à l'échelle départementale, sont structurés par un Service Départemental d'Incendie et de Secours (SDIS) qui possède ses propres ressources matérielles et sa propre structure de commandement. Ce service est responsable du transport des blessés, de la lutte contre les incendies et de la protection des biens.
— Le Service d'Aide Médicale Urgente (SAMU) est le centre de régulation médicale d'urgence qui régule les ressources de soins d'urgence (ambulances par exemple).
— Enfin, la Sécurité civile est impliquée dans des situations de crises majeures et peut fournir des moyens aériens.

L'organe de contrôle principal pour les pompiers est le Centre Opérationnel Départemental d'Incendie et de Secours (CODIS) qui traite les appels des victimes, supervise et coordonne l'ensemble des activités opérationnelles d'un service départemental d'incendie. Dans la figure 4.1. deux centres d'opérations sont impliqués CODIS56 et CODIS35. Chacun des services de secours engage ses propres ressources. Le tableau 4.1 présente les composants pouvant concourir à la réalisation de la mission du SdS. Les pompiers peuvent déployer, entre autres, un VTU (Véhicule Tout Usage) pouvant fournir un large éventail de services comme assécher des caves, tronçonner des arbres ou bâcher des toitures. La Sécurité civile peut mettre à disposition des hélicoptères pour fournir des services d'évacuation. Le SAMU fournit un centre de traitement d'appel pour fournir un service de régulation des blessés. Les ressources peuvent être humaines, par exemple des chefs d'agrès qui sont responsables de la coordination des membres de l'équipage d'un véhicule. Il peut également s'agir de chefs de groupe qui sont responsables de la coordination d'un ensemble de composants (e.g plusieurs VTU).

Ressources	**Objectif**	**Services**	**Contraintes**	**Organisme**
Antenne ANTARES	Communication	transmettre communication, permettre communication de groupe, permettre communication privée		
VTU (Véhicule Tout Usage)	Opérations diverses	assécher des locaux, débloquer des ascenseurs avec des occupants, protéger des biens, bâcher, tronçonner, détruire nid d'hyménoptères, capturer animaux ou transmettre communication, etc.	1 chef d'agrès, 1 conducteur	pompier
ERS (Embarcation de Reconnaissance et de Sauvetage)	Secours aquatique	transporter des sauveteurs de surface, transporter des plongeurs, évacuer des personnes lors d'inondation, transmettre communication, lutter contre les pollutions	1 chef 1 ou 2 plongeurs, tirant d'eau $>$ 30cm	pompier
Véhicule Léger (VL)	Reconnaissance	transmettre vidéo direct zone dangereuse ou inaccessible, capturer carte zone sinistrée	1 pilote	pompier
VL	Maintien de l'ordre	faire la circulation, transmettre communication	1 chef d'agrès, 2 ou 3 policiers	police
VLC (Véhicule Léger de Commandement)	Commandement	commander opération de maintien de l'ordre, transmettre communication, renseigner	1 chef de groupe	police

Ressources	Objectif	Services	Contraintes	Organisme
VLC (Véhicule Léger de Commandement)	Commandement	transmettre communication, commander opération de secours d'un groupe, renseigner	1 chef de groupe	pompier
CODIS (Centre Opérationnel d'Incendie et de Secours)	supervise et coordonne l'activité opérationnelle	gérer des opérations importantes et exceptionnelles (montée en puissance, crise) ou de longue durée, coordonner et acheminer des moyens, contribuer à la mise en œuvre des plans de secours, remonter des informations au niveau national, relations extérieures, anticiper des risques, soutenir logistiquement les ressources déployées	3 ou 4 opérateurs, 1 chef de salle	pompier
SAMU	régulation médicale	réguler les patients vers hôpitaux	3 ou 4 opérateurs, 1 chef de salle	SAMU
VSAV (Véhicule de Secours et d'Assistance aux Victimes)	Secours	apporter les secours nécessaires aux différentes victimes, procéder à leur évacuation sous surveillance médicalisée ou non vers un centre hospitalier.	1 chef d'agrès, 2 ou 3 sapeurs	pompier
Hélicoptère	Secours	transférer des blessés, transporter une équipe médicale, évacuations en mer, recherche de naufragé	1 pilote, 1 mécanicien, 1 médecin, piste atterrissage	sécurité civile

TABLE 4.1: Synthèse des ressources engageables par le SdS de services d'urgence

Au niveau du territoire français, les services de secours sont organisés par département. Chaque type de service est déployé sur chaque département français avec son propre financement et sa propre structure hiérarchique de commandement. La collaboration entre les services est favorisée par le réseau ANTARES (Adaptation nationale des transmissions aux risques et aux secours) [2]. Il s'agit du réseau numérique de communications intra-services (quand les communications sont limitées aux membres d'un même service) et inter-services (quand les communications permettent à des membres de services différents de communiquer). Les collaborations que peuvent former le SdS ont des échelles différentes. Elles peuvent être :

- Locales, quand il s'agit de communications entre deux ressources proches géographiquement.
- Départementales, quand les ressources qui communiquent sont distribuées à l'échelle du département.
- Inter-départementales, quand ce sont des ressources de services appartenant à deux départements différents.
- Nationales, quand elles impliquent des membres du gouvernement.

Une gestion efficace consiste à prendre les bonnes décisions au bon moment. Cette efficacité dépend de la qualité des informations et du moment où elles sont fournies. Pour ces raisons les services de secours se structurent autours d'une chaîne de commandement [3]. Les composants de la chaîne de commandement (qui sont les ressources des services de secours) sont partitionnés suivant leur niveau de responsabilité décisionnaire. Ces niveaux de décision sont d'ordre :

- Stratégique. Cela consiste à définir les objectifs et anticiper les besoins opérationnels en réservant des ressources, afin de les allouer au moment opportun.
- Tactique. Cela consiste à assigner des objectifs aux ressources allouées pour réaliser les objectifs stratégiques.
- Opérationnel. Cela consiste à décider d'opérations primitives dans le but de réaliser les objectifs tactiques.

La discipline hiérarchique impose aux composants de n'interagir qu'avec le niveau n+1 ou n-1. Lorsqu'il s'agit d'interactions avec les composants du niveau n+1, ces interactions sont des remontées d'information. Par exemple, les composants du niveau opérationnel font une synthèse de la situation observée aux composants du niveau tactique. Les communications vers le niveau n-1 concernent des directives. Par exemple, les composants du niveau tactique décident de la zone de positionnement d'un composant opérationnel et des actions qu'il doit réaliser.

Cette chaîne de commandement se reflète dans la structure du réseau de communication des pompiers. La communication entre le CODIS et ses ressources déployées se fait principalement via un canal de communication radio appelé canal opérationnel par le réseau ANTARES. Le canal opérationnel est un groupe de communication déployé pour chaque SDIS. Chaque membre du groupe connecté sur le même canal de communication peut écouter les messages envoyés par les autres membres du groupe. On parle d'écoute

2. `http://enasis.univ-lyon1.fr/clarolinepdfplayerbundle/pdf/35625` - consulté le 12/09/18
3. `https://fr.wikipedia.org/wiki/Cha%C3%AEne_de_commandement`

passive. Via le canal opérationnel, le CODIS gère toutes les interventions courantes en cours dans le département :

— Les communications se font seulement de façon verticale. Par exemple, un VTU ne peut pas communiquer avec un autre VTU.

— Les abonnés ne prennent la parole que s'ils ont l'autorisation du CODIS et s'il n'y a pas une autre communication en cours. Par exemple, le VTU demande l'autorisation de parler avant d'envoyer son rapport. Un seul abonné peut avoir la parole à un moment donné.

D'autres types de canaux sont disponibles comme le canal tactique et stratégique, pour les niveaux correspondants de la chaîne de commandement. Ces canaux fonctionnent de la même manière que les canaux opérationnels, avec les mêmes contraintes.

Ces points sont expliqués dans le document [4] de formation interne du SDIS (Service Départemental d'Incendie et de Secours) 42.

4.2 Configurations

Cette section présente un scénario d'une gestion de crise résolue par la mise en œuvre d'un SdS formé par des services de secours. Il montre, en particulier, le développement évolutionnaire du SdS à travers le déploiement de nouveaux comportements au fur et à mesure de son exécution. Le scénario présenté est celui d'une inondation de cave qui va évoluer en inondation de tout un quartier.

4.2.1 Scénario

Le premier intervenant de la chaîne de secours qui va être mise en place est l'appelant. L'appelant va constater par exemple que sa cave est inondée, et appelle donc le CODIS de son département. L'opérateur du CODIS qui traite l'appel va récupérer les informations lui permettant de dimensionner les moyens à allouer en fonction du sinistre. Dans ce cas, l'opérateur CODIS estime qu'il va falloir assécher la cave. Pour mettre en œuvre cette mission, le CODIS décide d'engager un VTU équipé d'une pompe qui est disponible. Ce type d'intervention est considéré comme courant. C'est donc le CODIS qui coordonne directement les agents déployés. Plusieurs interventions courantes peuvent être déployées simultanément. Les communications se font sur le canal opérationnel du SDIS 56.

Dans le cadre de sa mission, dès que le VTU a signalé sa présence dans le groupe opérationnel, un des opérateurs du CODIS lui envoie un ordre de mission. Ensuite, le VTU transmet des rapports régulièrement sur le canal opérationnel pour indiquer son avancement dans sa mission, donner au CODIS des informations sur l'intervention ou demander des ressources supplémentaires. Dès qu'il est arrivé sur les lieux, le chef d'équipe du VTU constate qu'entre temps la situation a évolué et que c'est tout le quartier qui est inondé. Il rend compte de la situation au CODIS et demande un soutien logistique. Le chef du VTU rend compte de l'impossibilité d'évaluer précisément l'étendue de la

4. sujet "ANTARES + INPT (Réseau de télécommunications de la sécurité civile)" du forum `http://sos112.fr`, lien raccourci `https://frama.link/ANTARES` - consulté le 12/09/2018

zone sinistrée, ni de l'explorer. À ce stade, le CODIS a des responsabilités tactiques et stratégiques. Le chef d'équipe a des responsabilités opérationnelles. La collaboration est inter service.

L'évolution des opérations conduit le CODIS à décider de lancer une mission d'exploration aérienne. Pour les besoins de l'exemple, le SDIS 56 ne possède pas de moyen d'exploration aérienne disponible. Il demande donc l'assistance du SDIS 35 voisin. À travers son CODIS (Centre Opérationnel Départemental d'Incendie et de Secours), le SDIS 56 engage pour sa part un VLC (Véhicule Léger de Commandement) équipé d'un drone de reconnaissance. Le chef d'agrès engagé dans le VLC reçoit sa mission et des informations complémentaires du CODIS 56. Dès qu'il arrive sur les lieux, il va constater les dégâts et fournir les informations au CODIS 56. Dans ce scénario, la collaboration est désormais interdépartementale, car mettant en œuvre des ressources de deux SDIS voisins. Les communications entre le VLC et le CODIS se font sur le canal opérationnel.

À la suite de ces nouvelles informations, une montée en charge est décidée par le CODIS : plusieurs ERSs (Embarcation de Reconnaissance et de Sauvetages), embarquant chacune un chef d'agrès, sont engagées. Pour accompagner cette montée en charge, il convient de modifier l'organisation du SdS : pour prendre en charge la coordination du dispositif, le CODIS décide de déployer un échelon tactique dans la chaîne de commandement. Pour sa part, le CODIS supervise désormais cette intervention uniquement du point de vue stratégique. Pour le besoin de l'exemple, le CODIS décide de limiter les ressources du SDIS 56 qu'il engage dans ce SdS afin de préserver ses capacités opérationnelles, par exemple, pour être en mesure de répondre à d'éventuelles autres opérations dans ce département sensible aux risques naturels. Le CODIS sollicite donc une nouvelle fois le SDIS 35, cette fois-ci un VLC, pour armer cette fonction de chef de groupe. Conformément à la discipline imposée par la chaîne de commandement, les ressources allouées à cette opération communiquent désormais uniquement avec le VLC du chef de groupe, lequel a désormais l'exclusivité des échanges avec le CODIS pour les comptes-rendus et les prises d'instructions. À cet effet, un nouveau canal de communication tactique, dédié à la communication entre les ressources et le chef de groupe, est déployé.

Pendant l'exécution de la mission de reconnaissance, les ERSs ont détecté plusieurs victimes, dont une dans un état grave. Le VLC communique l'information au CODIS. Le CODIS décide qu'une mission d'évacuation médicalisée aérienne et une mission d'évacuation médicalisée terrestre sont nécessaires. Le CODIS engage un hélicoptère de la Sécurité civile pour évacuer la victime et plusieurs ambulances équipées de leur équipage. L'hélicoptère communique via un canal dédié et est supervisé par le CODIS. Comme il y a des victimes blessées, le SdS doit intégrer le SAMU qui fournit un service de régulation médicale, c'est-à-dire qu'il va orienter les ambulances vers l'hôpital le plus approprié. Dans ce contexte, l'architecture évolue et prévoit que les bilans médicaux soient transmis au SAMU via un canal de communication médicale, conforme aux contraintes de confidentialité des données médicales. Un nouveau canal de communication est donc déployé. Les ambulances doivent communiquer les rapports opérationnels au VLC sur le canal tactique et

les rapports médicaux sur le canal médical pour recevoir des instructions de régulation adéquates.

4.2.2 Définition des configurations

Le scénario montre trois configurations résultant du comportement évolutionnaire du SdS.

La première configuration considère que deux SDIS ont une collaboration opérationnelle interdépartementale, c'est-à-dire qu'ils communiquent directement, seulement pour ce qui concerne les actions de terrain réalisées pour remplir la mission. Une telle collaboration est mise en place, par exemple, lorsqu'un SDIS fait face à un manque de personnel, et demande donc des ressources manquantes auprès d'un SDIS voisin pour une mission de petite taille. Dans notre scénario, cette configuration est utilisée lorsque les habitants appellent des services d'urgence en réponse à ce que l'on pense être une inondation localisée. La figure 4.2 structure les exigences liées à cette configuration. Le diagramme des exigences utilisé capture les fonctions que doit réaliser le SdS. Ce diagramme permet d'organiser la traçabilité des exigences et des artefacts produits à partir de ces dernières. Les exigences peuvent être structurées avec différents connecteurs : nous avons utilisé la relation de *containement* pour décomposer une exigence, la relation *refine* est utilisée pour associer un modèle ou ses éléments avec une exigence

Dans la deuxième configuration, deux SDIS ont une collaboration tactique et stratégique interdépartementale, c'est-à-dire qu'une chaîne de commandement hiérarchique est mise en place. En plus de la collaboration opérationnelle, la collaboration tactique permet aux commandants de communiquer avec leurs subordonnés, afin de leur donner des instructions et de recueillir des rapports sur les actions sur le terrain. La collaboration stratégique garantit que le SDIS collaborateur adopte des orientations cohérentes afin d'atteindre les objectifs de niveau supérieur définis par la mission. Cette deuxième configuration est utilisée lorsque la crise est plus importante, ce qui nécessite plus de ressources que dans la première configuration, et lorsque ces ressources nécessitent une coordination étroite. Dans notre scénario, le SdS évolue vers cette seconde configuration lorsque les services d'urgence découvrent que l'inondation est plus critique qu'ils ne le pensaient initialement. La figure 4.3 structure les exigences liées à cette configuration.

Enfin, dans la troisième configuration, un SDIS collabore avec le SAMU et la Sécurité civile. Cette configuration se produit lorsque les services d'incendie et de secours ont besoin de l'aide d'autres services pour faire face à la crise. Dans notre scénario, le SAMU est impliqué afin de réguler l'évacuation des blessés vers les hôpitaux les plus proches, et la Sécurité civile fournit des ressources de type hélicoptère lorsque l'inondation rend les routes inutilisables. La figure 4.4 structure les exigences liées à cette configuration.

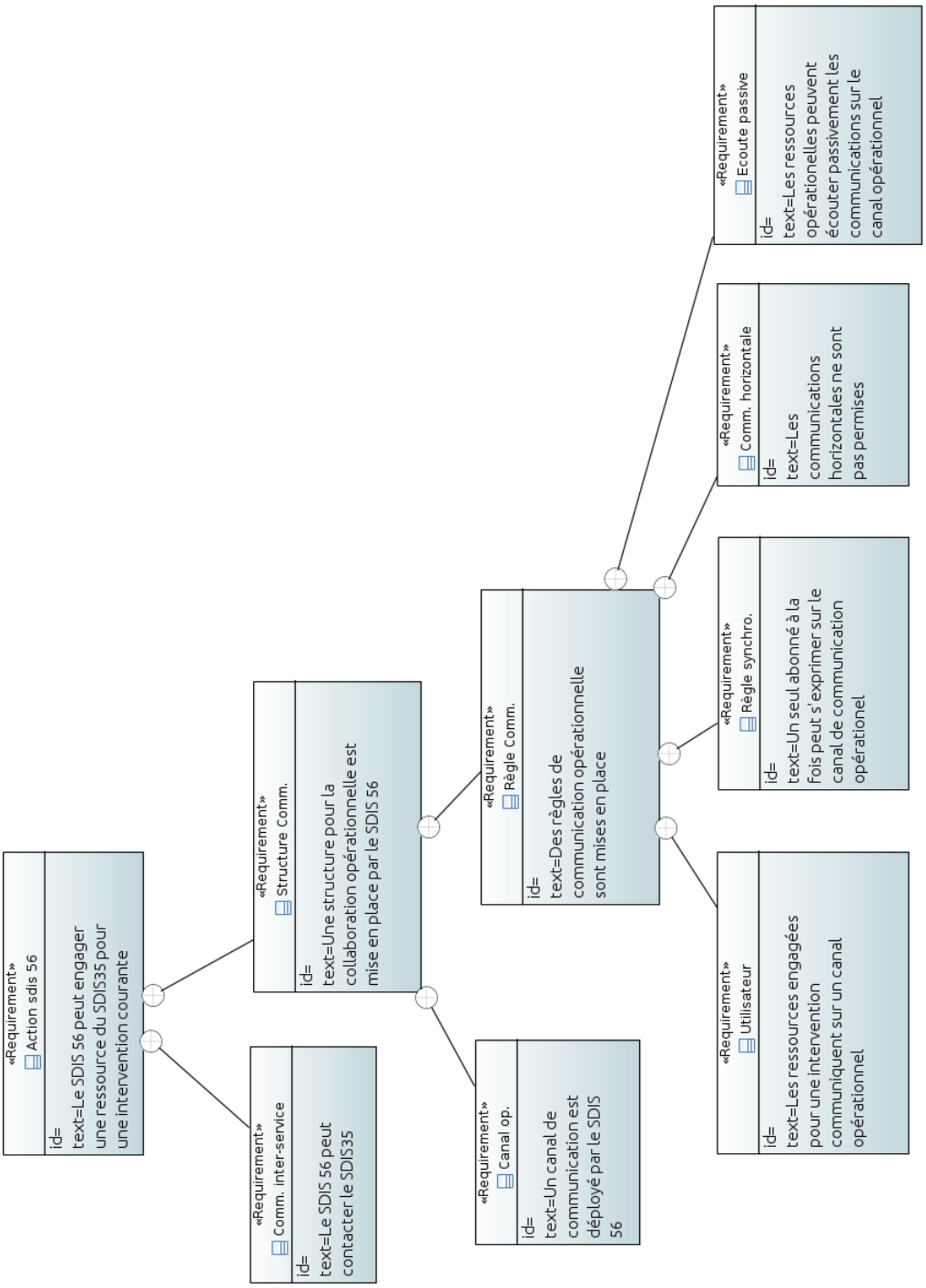

FIGURE 4.2: Exigences collaboration opérationnelle interdepartementale

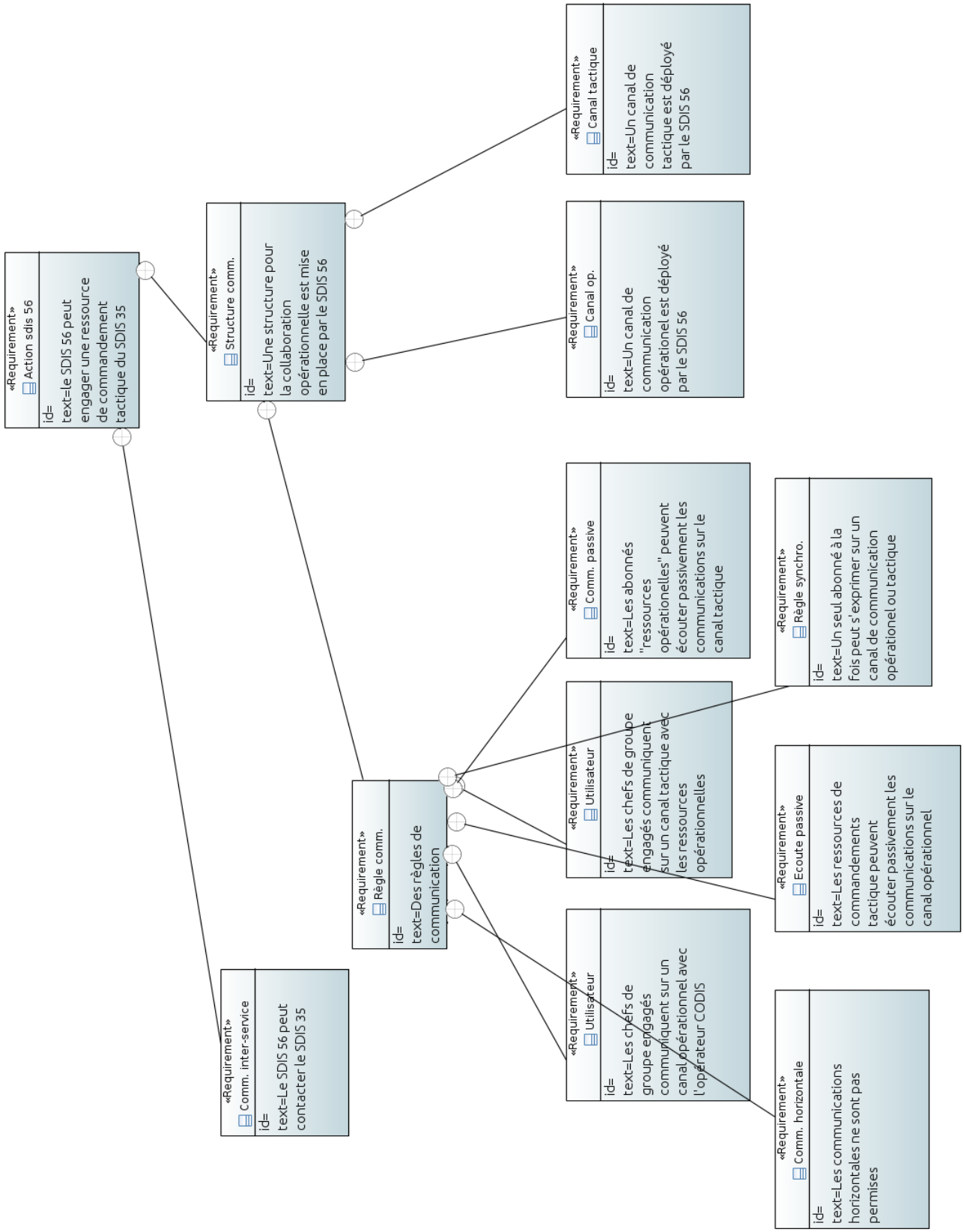

FIGURE 4.3: Exigences collaboration tactique interdepartementale

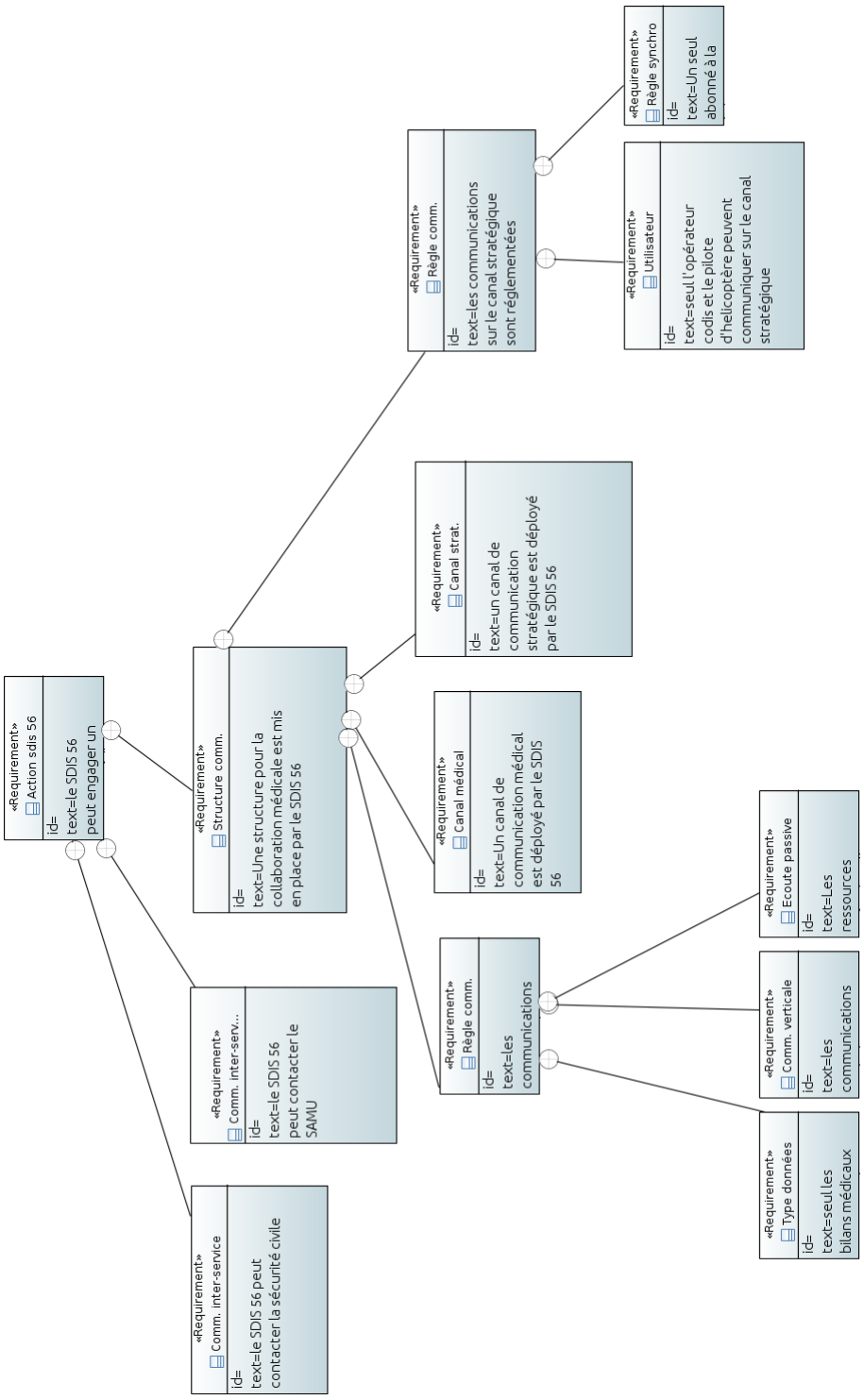

FIGURE 4.4: Exigences collaboration médicale

4.3 Synthèse

Dans ce cas d'étude, les entités que nous allons considérer comme des systèmes constituants sont les différentes organisations de secours : les SDISs, les services de police, le SAMU, la Sécurité civile et les hôpitaux. Chaque organisation met à disposition ses ressources pour réaliser la mission. Par exemple, le SDIS 35 fournit un VLC, la Sécurité civile un hélicoptère. Le système de systèmes se forme dès que les systèmes se mettent à interagir, par exemple, deux SDISs (35 et 56), un SDIS et le SAMU, un SDIS, le SAMU et la Sécurité civile.

Le SdS décrit dans l'étude de cas correspond bien aux principales définitions de système de systèmes trouvée dans la littérature. Si nous considérons d'abord la définition de Maier (1998), chaque système possède bien son indépendance opérationnelle et managériale et est capable de réaliser un objectif indépendamment du SdS. Par exemple, le SDIS possède sa propre structure de commandement et dispose de moyens suffisants pour lutter contre un incendie indépendamment des autres systèmes. Par ailleurs, l'interaction des différents systèmes produit un comportement qui n'est pas possible par un seul des systèmes. Par exemple, le SDIS 56 ne pourrait pas réaliser son comportement d'exploration sans les ressources du SDIS 35, ni transporter une victime avec une certaine contrainte de temps sans les ressources de la Sécurité civile. Le développement est évolutionnaire car les comportements déployés dans le SdS évoluent au cours de son exécution. Le scénario a montré que de nouveaux systèmes sont intégrés et l'architecture est restructurée pour réaliser les nouveaux comportements.

La définition de Firesmith (2010) met en avant les caractéristiques suivantes : complexité, évolution constante, comportement émergent, taille et variabilité. Notre cas d'étude correspond bien à ces caractéristiques. En effet, ce type de système évolue constamment. Dans le cadre de l'intervention engagée par le SDIS 56, il est nécessaire de déployer de nouvelles ressources pour réaliser de nouveaux comportements au fur et à mesure de l'exécution de l'intervention. Par ailleurs, au fil de l'exécution, la taille du SdS tend à augmenter et progresser vers une catégorie de SdS à ultra large échelle. Cette caractéristique pourrait être atteinte, par exemple, en cas de sinistre avec un pays frontalier : dans un tel cas, des ressources provenant des deux pays devraient être déployées au sein du même SdS. Enfin, la variabilité est possible dans notre cas d'étude en considérant les arbitrages du CODIS qui conduisent à solliciter ou non un autre SDIS, à estimer que la gravité appelle ou non à déployer l'échelon tactique, et cela indépendamment de la mission à accomplir.

Dans le contexte de notre thèse, nous nous intéressons particulièrement au développement évolutionnaire. Nous adoptons la perspective que le SdS découvre au fur et à mesure les comportements qu'il doit réaliser. Dans le cas d'étude, le SdS doit réaliser un comportement d'exploration aérienne, puis maritime. Plus tard, il doit réaliser un comportement d'évacuation des victimes découvertes pendant l'accomplissement de sa mission. Dans les projets COMPASS et DANSE, les études de cas s'intéressent plus à la dimension de taille des SdSs et les contributions sont orientées vers des techniques de vérification et génération d'architecture au moment de la conception de l'architecture du

SdS. L'étude de cas se concentre sur les moyens de communication qui sont peu sujets à des évolutions à court terme. Notre cas d'étude met en avant les évolutions constantes et à court terme. Il s'agit des ressources engagées ponctuellement pour permettre de nouveaux comportements et les décisions architecturales adoptées. L'architecte doit donc réviser l'architecture de son SdS pendant son exécution et sans que le SdS tombe dans un état qui ne lui permet plus de fonctionner correctement.

Chapitre 5

Framework de modélisation pour les systèmes de systèmes

Nous avons vu dans le chapitre 3 que les architectures sont les artefacts centraux dans les développements logiciels. Elles sont utilisées dans toutes les phases des processus de développement. Du fait de la complexité des SdSs (systèmes de systèmes), la modélisation des architectures est une tâche non triviale. Ce chapitre se concentre sur la problématique d'identifier correctement les frontières du SdS et de vérifier des propriétés de reconfiguration. La section 5.1 expose les stratégies mises en oeuvre pour résoudre ces problèmes et analyse les critères d'un framework de modélisation adéquat pour ces stratégies. L'objectif est de raffiner des frameworks existants suivant des critères qui sont : la portée des analyses mises en oeuvre, les aspects nécessaires, le type d'analyse et le niveau de formalisme. Nous proposons un framework de modélisation basé sur UPDM (Unified Profile for DoDAF and MODAF), SysML (System Modeling Language) et OCL (Object Constraint Language). Suite à l'étude des processus de modélisation des SdSs dans le chapitre 3 nous sélectionnons les vues, composants, connecteurs et types de diagramme les plus adaptés à notre stratégie de modélisation. Puis nous définissons un processus de modélisation, les objectifs de modélisation et les outils mis en oeuvre.

Le processus de modélisation est décomposé en cinq phases qui se structurent autour d'un atelier utilisant comme modeleur Papyrus et le framework développé pour la simulation. La section 5.2 définit trois modèles architecturaux de l'étude de cas présentée dans le chapitre 4. Ces trois modèles correspondent aux architectures définies à partir des comportements évolutionnaires identifiés dans l'étude de cas. Les modèles sont développés suivant le framework et le processus expliqués dans la section précédente. Pour finir la section 5.3 synthétise les contributions de ce chapitre.

5.1 Choix de modélisation

Dans un premier temps nous allons expliquer les langages de modélisation que nous avons retenus. Pour cela nous développons leur utilité du point de vue de la modélisation, les éléments qu'ils permettent de modéliser en termes de composant et connecteur, puis le choix du type de diagramme. Dans un second temps nous justifions ces choix au regard des problématiques que nous adressons, qui sont l'analyse de l'exactitude du modèle et la

vérification des propriétés de reconfiguration. Ces choix sont guidés par l'analyse du but des modèles développés, la portée, les aspects qui seront analysés, le niveau de formalité, le type d'analyse et la technique utilisée.

5.1.1 Langage de modélisation

Au regard des caractéristiques du framework de modélisation attendu, nous avons choisi comme support à la modélisation du cas d'étude les frameworks SysML, OCL et UPDM. Suite à l'analyse de ces frameworks présentée au chapitre 3 et plus particulièrement la synthèse que nous en avons fait en section 3.4, nous avons sélectionné les vues pertinentes pour notre travail. Identifier les concepts associés à chacune de ces vues nous a conduit à élaborer des méta-modèles, que les figures 5.1 et 5.2 présentent de manière simplifiée.

La figure 5.1 décrit un méta-modèle simplifié du point de vue opérationnel :

- Les vues OV-1a permettent avec une description picturale de montrer les principaux composants du SdS. Ces composants sont les principales ressources déployées dans le contexte de la mission principale du SdS.
- Les vues de type OV-1b permettent de modéliser avec les diagrammes de cas d'utilsatition quels sont les objectifs des interactions entre les CSs (systèmes constituants). Ces diagrammes sont associés à des acteurs qui modélisent des ressources impliquées dans les interactions. Nous utiliserons le diagramme de cas d'utilisation de SysML car cette partie du modèle architectural est comportementale et statique.
- Les vues OV-5 permettent d'exprimer l'intention des interactions par la décomposition des actions et leur responsable. Nous constatons de plus qu'elles permettent de faire apparaître des ressources qui n'étaient pas identifiées dès le début. Le diagramme le plus adapté est le diagramme d'activité qui modélise des aspects comportementaux et des analyses de la dynamique des interactions.
- Pour finir la vue OV-4 fait la synthèse des CSs impliqués ainsi que des ressources qu'ils déploient. Elle permet de faire apparaître clairement les CSs en montrant à quelles organisations les acteurs appartiennent. Nous utiliserons un diagramme de package pour modéliser des aspects structurels et permettre des analyses statiques. Les diagrammes de package mettent en avant l'indépendance des ressources d'un CS vis-à-vis des autres CS.

Le point de vue système de UPDM fournit un ensemble de vues qui donnent un niveau de détail supplémentaire à la vue opérationnelle. La figure 5.2 montre un méta-modèle simplifié à partir duquel nous développons une partie du modèle. Nous avons choisi la vue SV-1 qui raffine les ressources en termes de composant humain, matériel et logiciel. Elle précise le type des ressources, le nombre d'instance et les connexions. Les connexions sont décrites via la notion de port et d'interface fournie et requise. Le BDD (Diagramme de Définition de Bloc) et le BDI (Diagramme Interne de Bloc) sont des diagrammes structurels qui permettent d'exprimer les instances minimales et maximales pour le BDD et les interconnexions pour le BDI. Ces diagrammes permettent d'exprimer en termes de contrainte le nombre de composants de l'architecture et la dépendance entre ces composants. Cela permet de capturer le style architectural mais aussi le comportement des CSs.

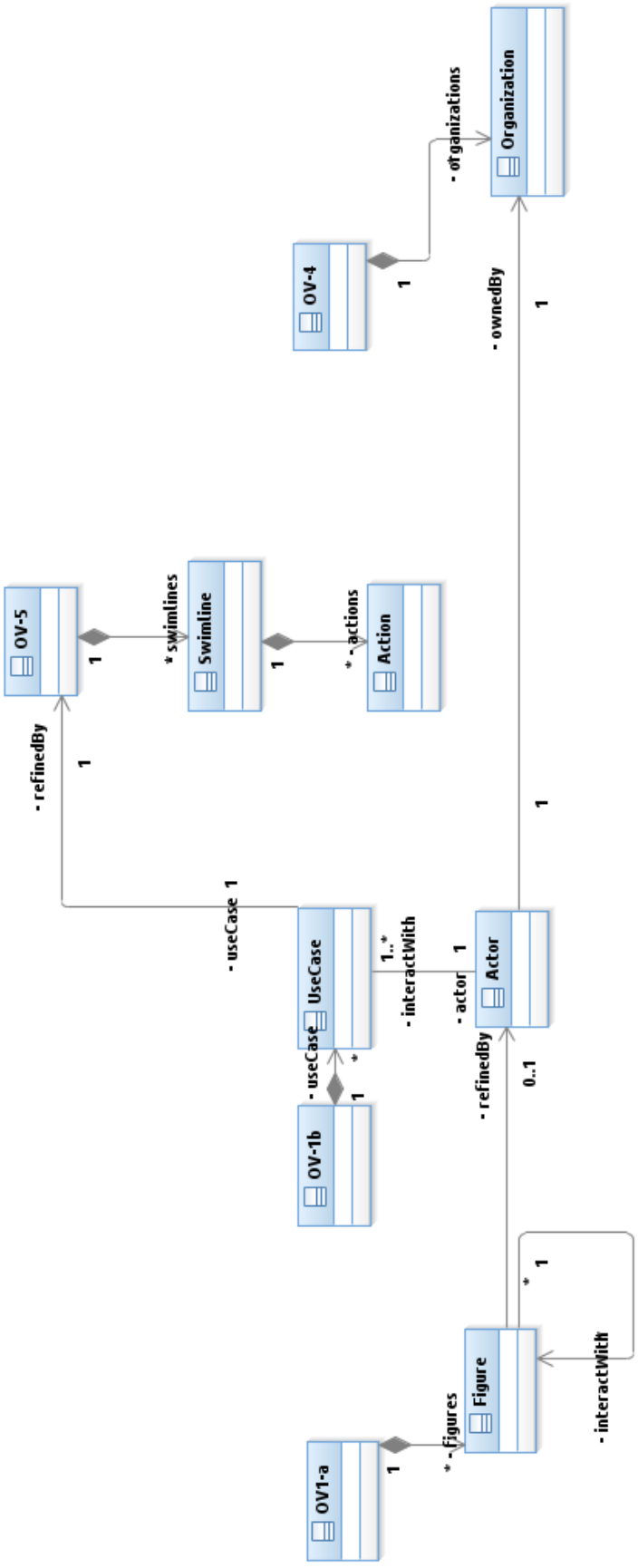

FIGURE 5.1: Métamodèle simplifié du point de vue opérationnel

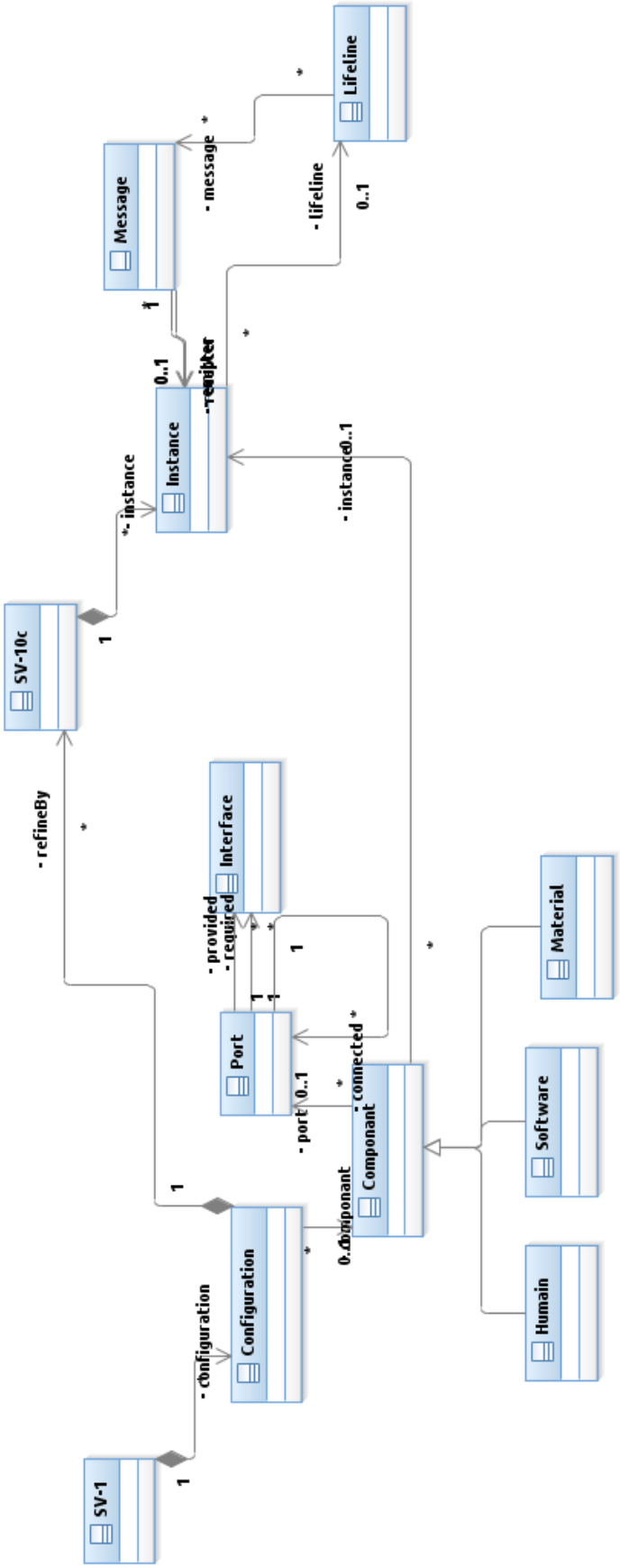

FIGURE 5.2: Métamodèle simplifié du point de vue système

Ces diagrammes limitent les propriétés que l'on souhaiterait modéliser. Un exemple de propriété que nous serons amené à modéliser est la définition de la stratification entre des groupes de composants. Le but est de vérifier que des composants ne puissent pas communiquer directement avec des composants d'autres stratifications.

Pour étendre la précision de SysML et UPDM nous intégrons OCL qui ajoute plus de précision aux propriétés exprimées et permet leur vérification automatique. Nous utiliserons OCL pour définir des primitives architecturales comme expliquées par Zdun and Avgeriou (2008). Ensuite, nous verrons que l'approche met en œuvre des outils de simulation se focalisant sur des scénarios d'exécution. Les vues OV-5 nous permettent d'identifier un ensemble de scénarios d'exécution possibles, mais la description n'est pas assez formelle. La vue SV-10c nous permet de décrire les scénarios d'exécution en termes de messages échangés. UPDM n'étant pas assez précis, nous avons intégré des primitives comportementales qui capturent les concepts d'envoi et de réception de messages comme celle de SosADL (System of Systems Architectural Description Language) Oquendo (2017). Un diagramme de séquence est utilisé et montre les interactions en termes d'envoi de messages et de réception de messages. Il permet de modéliser des aspects d'interaction, alors que le diagramme d'activité permet de modéliser des envois de messages.

Notre framework de modélisation se limite aux vues opérationnelles et systèmes. Notre intérêt se porte sur les reconfigurations à court terme dans les SdSs. Les points de vue *toutes vues* et *stratégique* s'intéressent à des éléments qui portent sur des évolutions à long terme plutôt que des évolutions à court terme qui impliquent des solutions de reconfiguration non anticipées. Le point de vue *service* est couvert par la description des interfaces de SV-1. Le point de vue *technique* est également couvert par la vue SV-1 en capturant le style architectural suivi.

5.1.2 Outils et processus

Caractéristiques du framework pour l'analyse

Un des buts est d'analyser l'exactitude du modèle architectural de l'étude de cas. Nous avons vu qu'une difficulté lors de la modélisation des SdSs est d'en définir la frontière et les objectifs d'interaction des CSs. La portée de l'analyse sera du niveau SdS mais aussi CS. Il s'agira de modéliser quels sont les CSs impliqués mais également les ressources déployées. Les aspects modélisés seront structurels et comportementaux. Nous modéliserons les ressources engagées par les CSs. Puis nous devrons modéliser les interactions entre les ressources engagées. Pour vérifier l'exactitude du modèle architectural nous ferons une simple revue manuelle de modèle. Nous ne ferons pas vérifier les modèles par des experts métiers mais par des partis prenants externes au travail de modélisation. Pour cela le modèle devra être facilement vérifiable. Nous favoriserons une analyse statique des modèles.

Dans l'état de l'art sur la reconfiguration nous avons vu que les propriétés de reconfiguration concernent la maîtrise de qualité de service. Cela consiste en pratique à vérifier qu'une connexion reste disponible pendant la reconfiguration ou qu'un comportement est

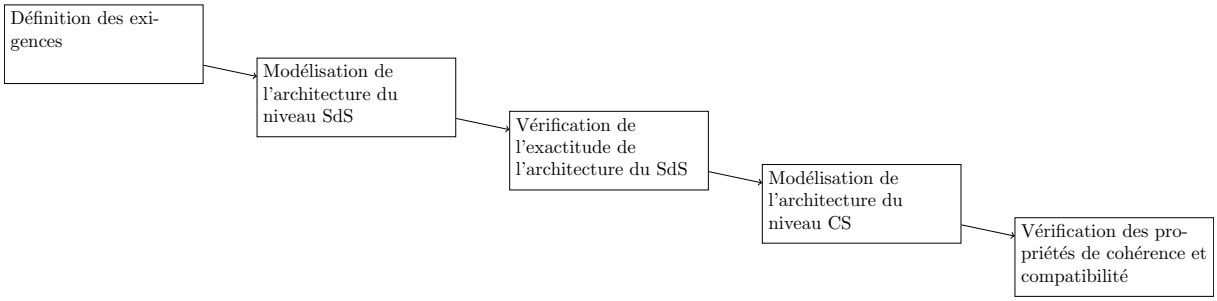

FIGURE 5.3: Processus de modélisation

réalisé lorsqu'un événement se produit. Les propriétés à vérifier sont donc structurelles et comportementales. Nous souhaitons donc vérifier la cohérence comportementale du SdS pendant sa reconfiguration ainsi que la compatibilité à un style architectural (pour les propriétés statiques). Ce qui nous intéresse est le comportement du système pendant la reconfiguration. La portée de l'analyse se concentre sur les flux de données entre les CSs et la composition des CSs. Les aspects qui doivent être analysés sont les interactions entre les CSs. L'analyse structurelle précise des points d'interaction entre les ressources et consiste à vérifier la compatibilité avec le style architectural. Pour analyser la compatibilité du style architectural et la cohérence du comportement pendant la reconfiguration, nous utiliserons une technique d'analyse basée sur la simulation d'exécution du SdS. La technique de vérification manuelle n'est pas envisageable car peu fiable. Pour éviter une explosion des états à analyser nous envisageons une analyse centrée sur des scénarios d'exécution. Comme les techniques de simulation requièrent des modèles exécutables, les interactions doivent être décrites avec des langages dont la sémantique repose sur le π-calcul comme c'est la cas pour le langage pi-ADL Oquendo (2004).

Processus

Le développement du modèle de l'architecture du SdS est décomposé en plusieurs phases. La figure 5.3 montre l'enchaînement de ces phases. Le processus choisi est descendant puisque les premières phases du modèle sont une description haut niveau de SdS, les phases suivantes raffinent le modèle architectural vers des abstractions de plus bas niveau.

La première phase est la définition et description des exigences du SdS. Nous utilisons le diagramme des exigences de SysML, qui permet de structurer les exigences du SdS en les décomposant en exigences plus simples telles que nous l'avons mentionné dans le chapitre 3.

La seconde phase est la modélisation de l'architecture du niveau SdS . La conception de l'architecture du SdS utilise les vues choisies de la vue opérationnelle. Nous commençons par la définition de la vue OV-1a qui montre un éventail représentatif des ressources engagées et leurs relations. La vue OV-1b est développée à partir de OV-1a. Elle se compose des objectifs d'interaction entre les ressources modélisées par les cas d'utilisation. La vue OV-5 est construite à partir de OV-1b, et fournit pour un cas d'utilisation une

description comportementale des interactions. Pour finir la vue OV-4 fait la synthèse des organisations impliquées dans le SdSs.

La troisième phase est la vérification de l'exactitude de l'architecture du SdS. Elle consiste à vérifier la partie du modèle développé dans la phase précédente. Nous avons choisi une technique manuelle. L'architecte du modèle de la phase deux identifie les parties du modèle qui correspondent aux exigences avec des relations de *satisfy* ou *refine* . Le parti prenant tiers valide les relations en analysant les diagrammes produits.

La quatrième phase est la conception de l'architecture du SdS du point de vue des CSs. Elle se concentre sur la composition des constituants. Nous commençons par la vue SV-1 qui décrit les composants impliqués dans chaque configuration et leur connectivité. Puis la vue SV-10c, décrit les scénarios d'exécution en termes d'envois de messages.

Pour finir la phase cinquième phase vérifie la cohérence et la compatibilité des modèles développés. Le modèle architectural est transformé vers la plateforme de simulation. Le modèle exécuté par la plateforme est construit à partir des vues SV-1 et SV-10c. Après l'exécution des scénarios du modèle, l'architecte vérifie les traces d'exécution pour voir si des propriétés ont été violées pendant la reconfiguration.

5.2 Modélisation de l'étude de cas

Dans le chapitre 4 consacré à l'étude de cas nous avons identifié trois configurations pour l'architecture de notre système de systèmes d'organisation des services de secours. Nous allons modéliser chacune de ces configurations en suivant le processus de développement et en donnant les vues associées. La première configuration est une collaboration opérationnelle interdépartementale entre deux SDIS. La deuxième configuration est une collaboration tactique et stratégique interdépartementale entre deux SDIS. Enfin la troisième configuration concerne la collaboration entre les SDIS et les équipes médicales du SAMU et de la sécurité civile. La première phase du processus concernant la description des exigences a été traitée dans le chapitre précédent donc n'indiquerons ici que les figures correspondantes dans le chapitre 4.

5.2.1 Cas 1 : collaboration opérationnelle entre SDIS 56 et 35

Exigences

La première phase du processus concerne les exigences. La figure 4.2 du chapitre 4 structure les exigences de la collaboration opérationnelle interdépartementale. L'exigence principale est *le SDIS 56 peut engager une ressource au SDIS 35 pour une intervention courante.*

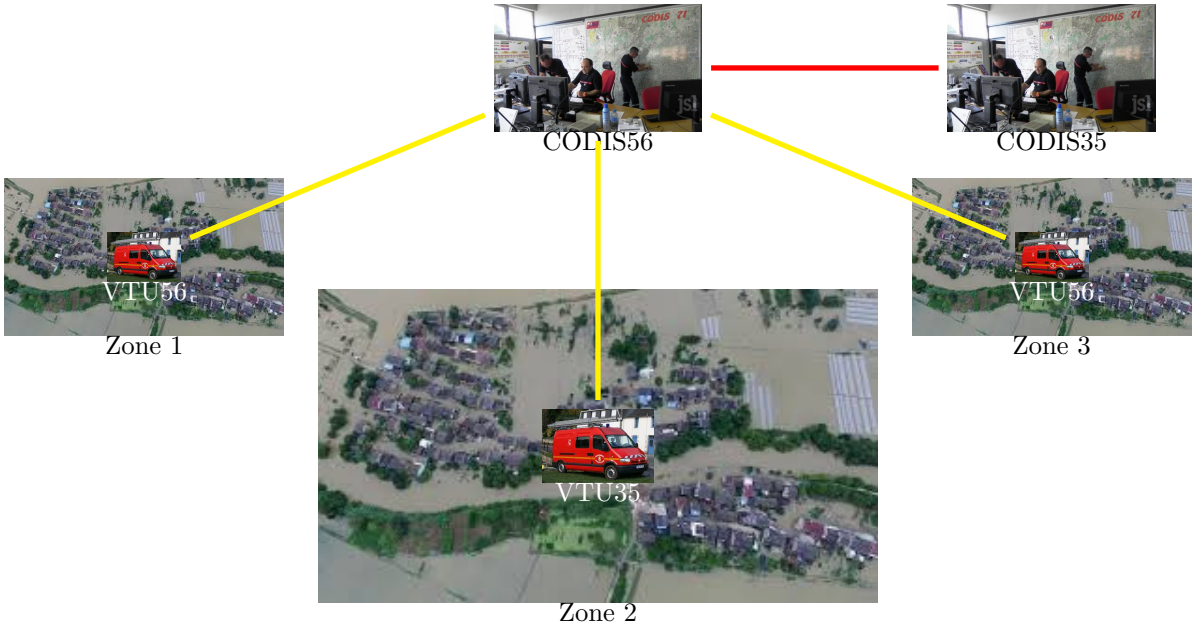

FIGURE 5.4: OV-1a - collaboration opérationnelle entre SDIS 56 et 35

OV-1a

La deuxième phase du processus est la modélisation de l'architecture du niveau SdS et va se décliner avec les vues qui suivent. La figure 5.4 présente la vue OV-1a correspondant à la première configuration de l'étude de cas. Il s'agit d'un diagramme informel qui décrit statiquement le contexte du SdS avec les principales ressources et relations.

OV1-b : objectifs d'interaction de la collaboration opérationnelle entre SDIS 56 et 35

La vue OV1-b est le diagramme de cas d'utilisation présenté dans la figure 5.5 . Elle montre que l'opérateur CODIS utilise le SDS pour engager différents types de ressource. On note que le déploiement d'une ressource implique des capacités de coordination opérationnelle utiles à l'opérateur du CODIS pour gérer toutes les interventions qui se déroulent sur son secteur. Cette capacité est utilisée par le chef d'agrès pour avoir toutes les informations nécessaires pour diriger ses effectifs, et requiert un canal de communication opérationnelle.

OV-5 - les activités opérationnelles

Les diagrammes d'activités montrent les activités associées à un cas d'utilisation. Elles décrivent comment sont réalisés les cas d'utilisation. Les figures 5.6 et 5.7 montrent les activités associées au cas d'utilisation : collaboration opérationnelle. Les partitions des diagrammes montrent les acteurs responsables de la réalisation des activités qu'ils englobent. Les connecteurs entre les activités montrent leurs orchestrations dans le temps. Dans la figure 5.6, nous voyons que le chef d'agrès 35 attend la fin des communications

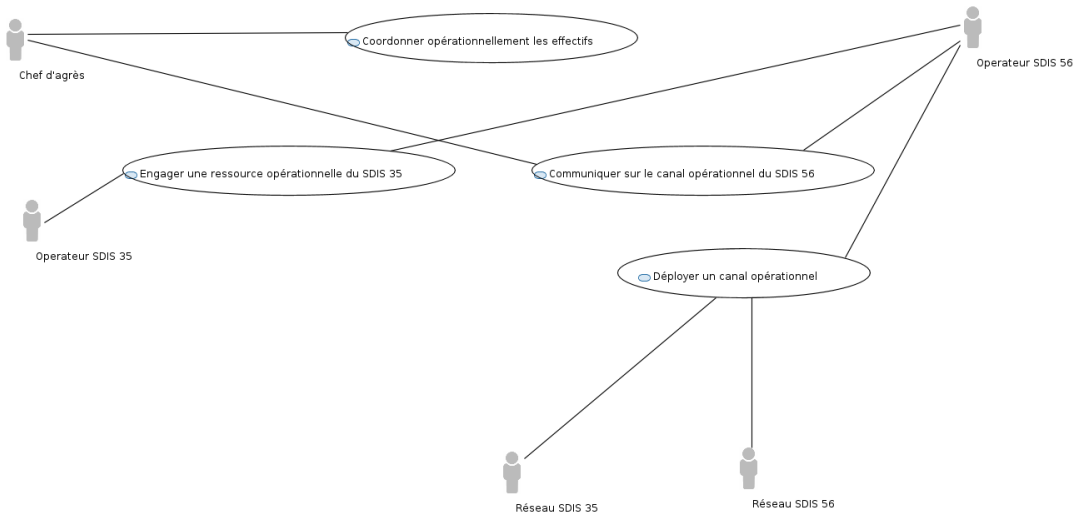

FIGURE 5.5: OV1-b : objectifs d'interaction de la collaboration opérationnelle entre SDIS 56 et 35

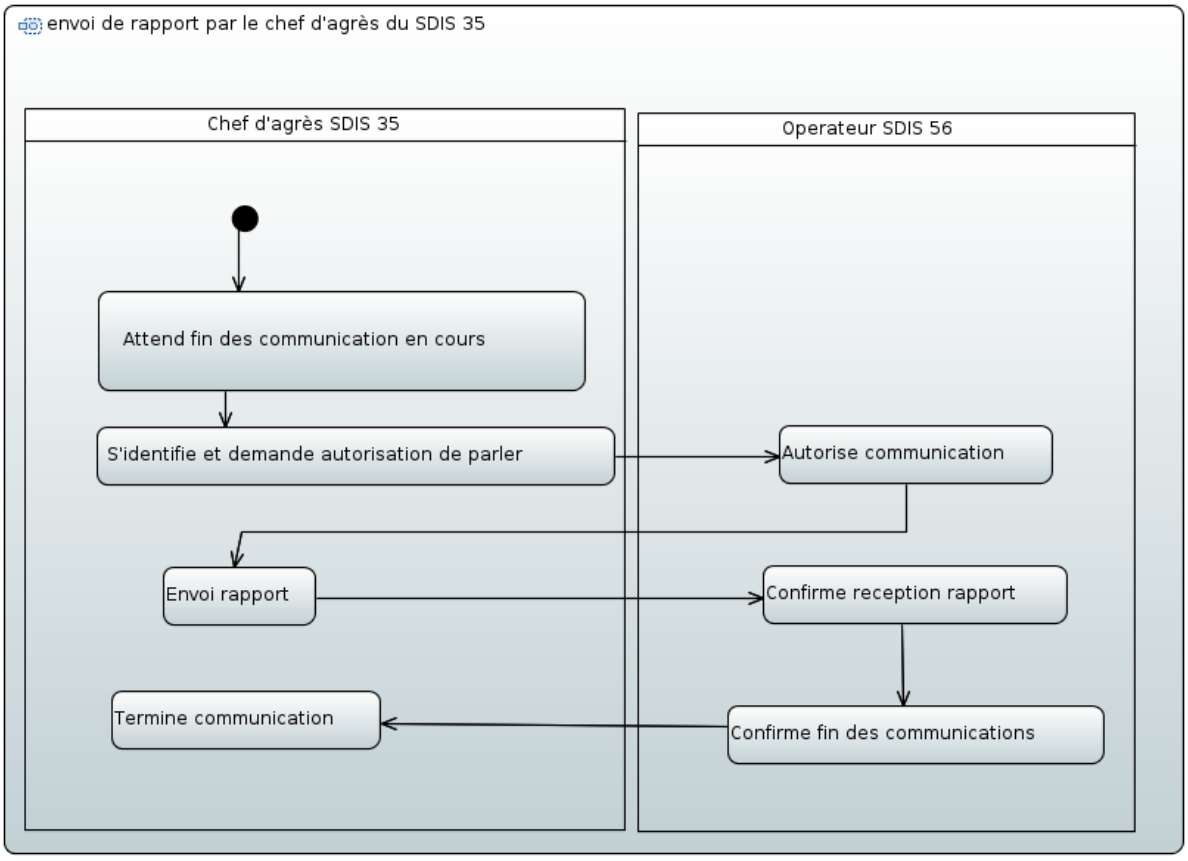

FIGURE 5.6: envoi de rapport

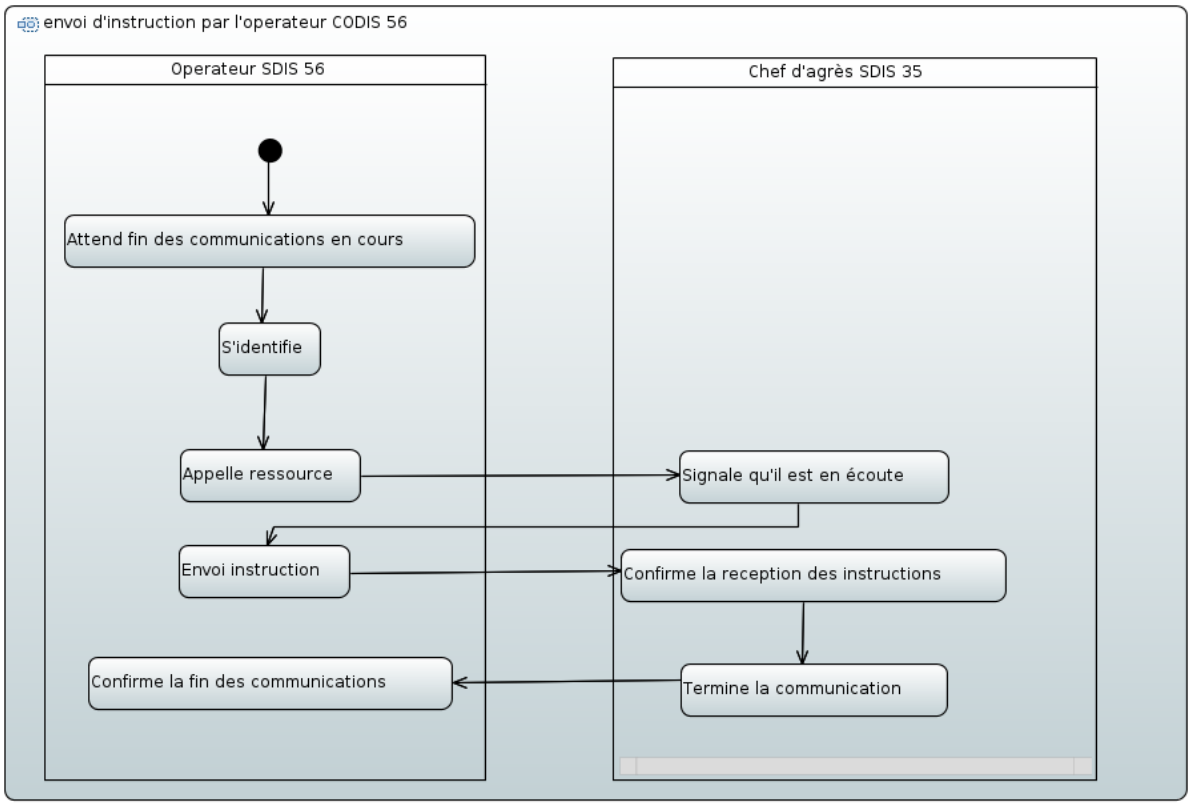

FIGURE 5.7: envoi d'instruction

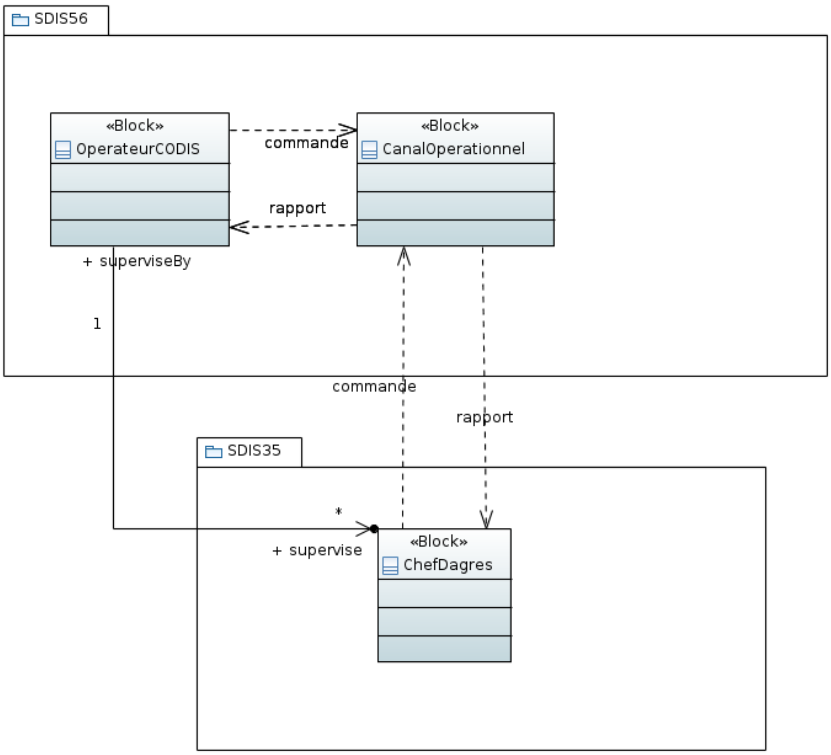

FIGURE 5.8: OV-4 - synthèse des systèmes constituants

sur le canal opérationnel pour envoyer son rapport. Nous voyons également que le protocole de communication est le suivante : il doit s'identifier, attendre la confirmation de l'opérateur CODIS et finalement envoyer son rapport. La figure 5.7 montre les activités qui concernent l'envoi d'instructions par l'opérateur CODIS 56 au chef d'agrès 35. Le diagramme montre aussi que l'opérateur attend la fin des communications et le protocole pour envoyer ses instructions.

OV-4 - synthèse des systèmes constituants

La figure 5.8 montre la synthèse des systèmes constituants représentés dans le diagramme par les packages. Nous retrouvons les CSs impliqués dans le SdS. Ils sont identifiés dans les vues OV-1b et OV-5. Dans ce cas il s'agit des organisations qui déploient les ressources.

Validation

La troisième phase du processus est la vérification de l'exactitude de l'architecture du SdS. La 5.9 montre l'étape de validation du point de vue du système de systèmes. L'architecte vérifie la partie du modèle développé et la correspondance avec les exigences. Pour cela l'architecte place une relation *satisfy* entre l'exigence et la partie du modèle correspondante.

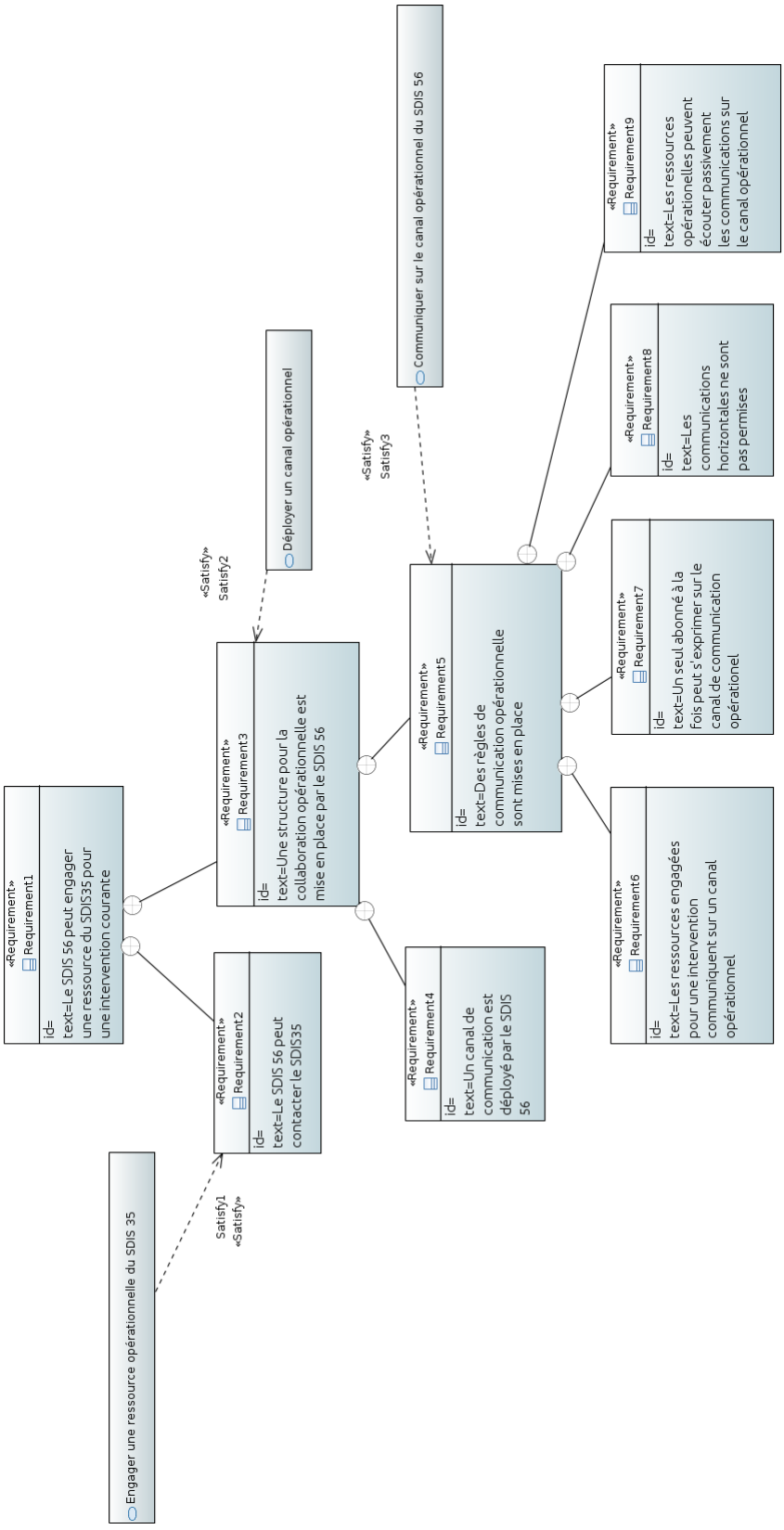

FIGURE 5.9: Étape de validation du point de vue SdS

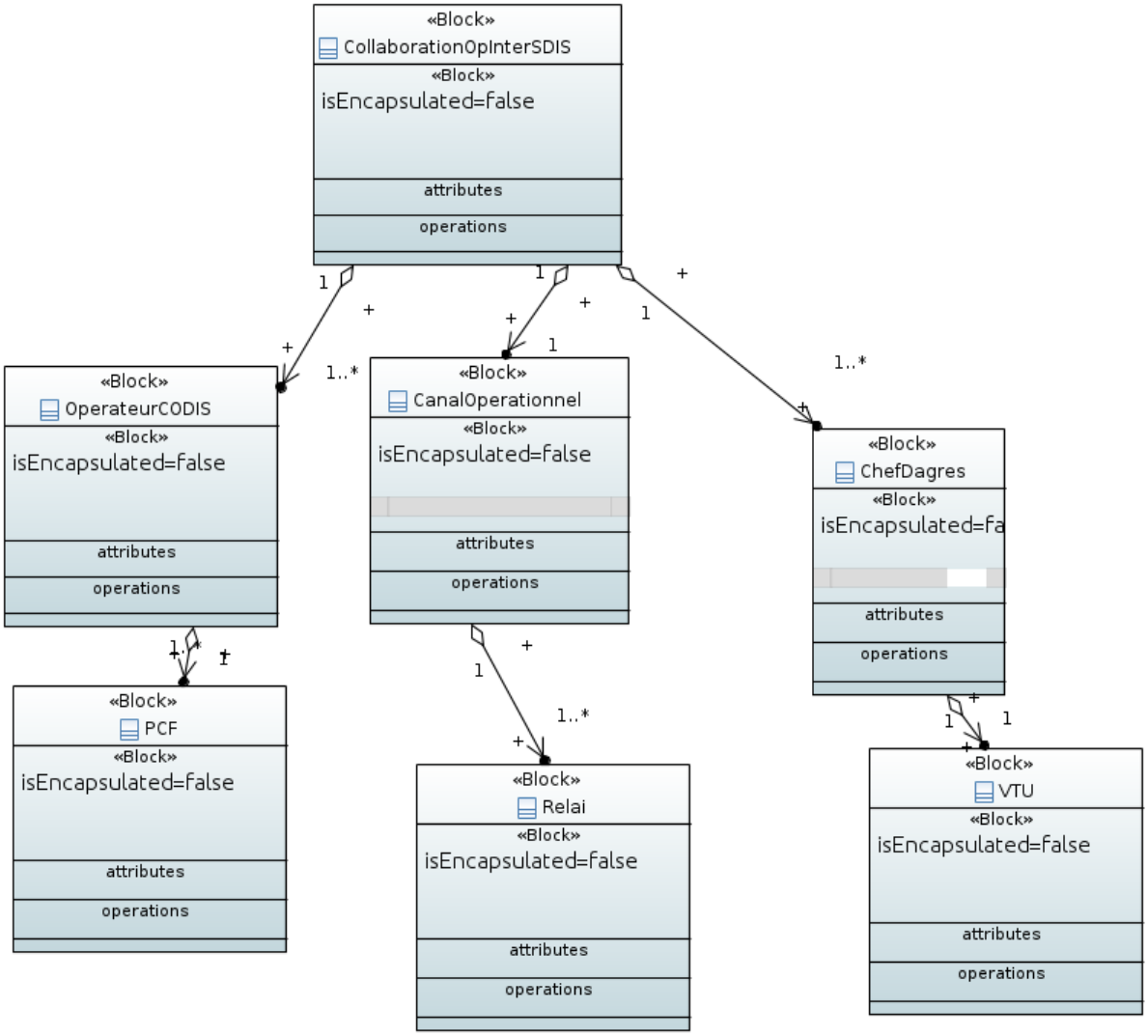

FIGURE 5.10: SV-1 BDD de la collaboration opérationnelle entre SDIS 56 et 35

SV-1 - configuration

Nous sommes à la quatrième phase qui se concentre sur la composition des constituants. La figure 5.10 montre la décomposition du SdS dans le contexte d'une collaboration opérationnelle entre SDIS. La figure 5.11 montre la résolution des dépendances entre les composants. Nous voyons qu'un canal opérationnel peut par exemple connecter un nombre quelconque de chef d'agrès. La configuration intègre la primitive architecturale *Shield* proposée par Zdun and Avgeriou (2008) qui permet à un ensemble de composants de ne pas être accédé directement par un client externe, mais uniquement à travers un *shield*. Ici nous remarquons que les chefs d'agrès ne peuvent pas se connecter directement à l'opérateur CODIS mais par l'intermédiaire du canal opérationnel. Finalement la figure 5.12 est le diagramme de package de cette collaboration.

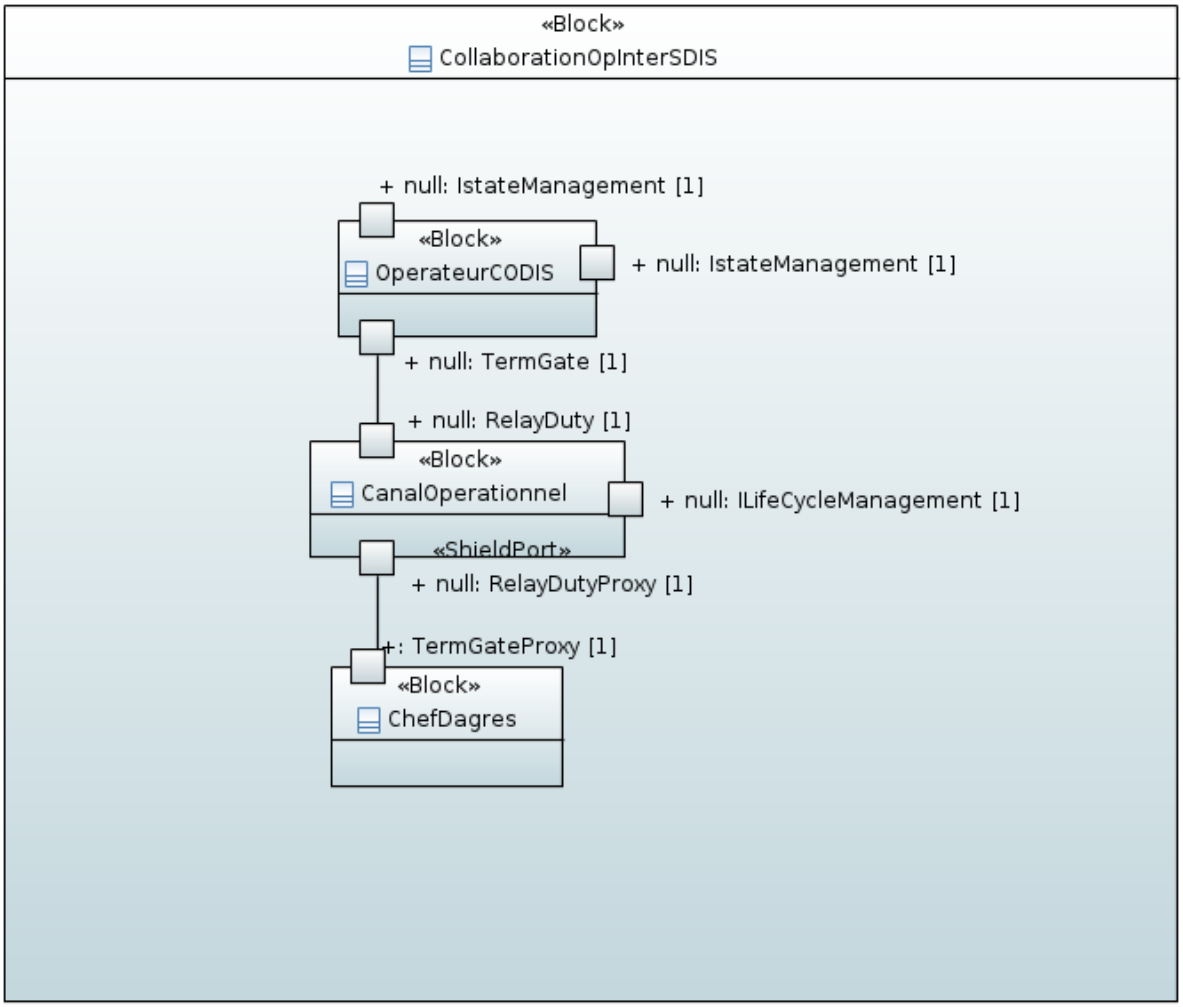

FIGURE 5.11: SV-1 BDI de la collaboration opérationnelle entre SDIS 56 et 35

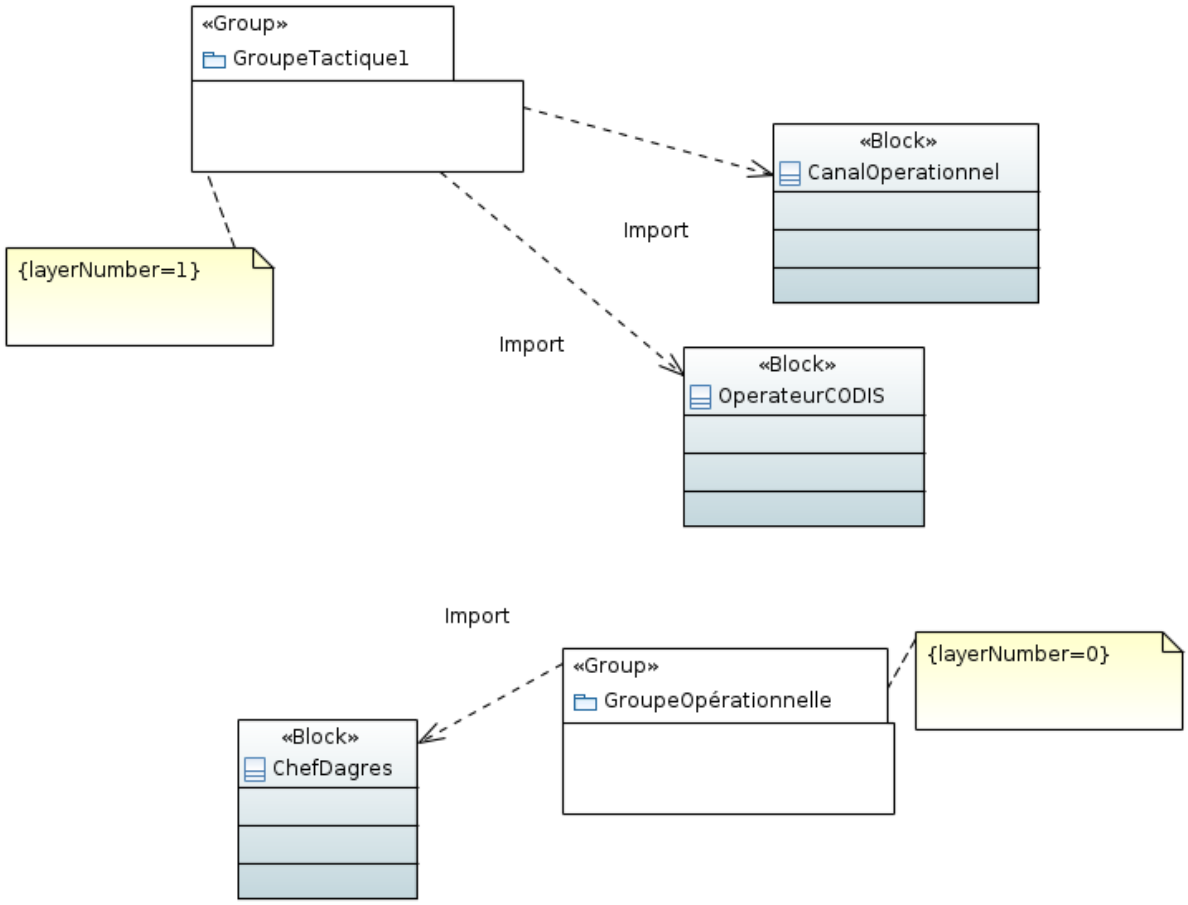

FIGURE 5.12: SV-1 Diagramme de package de la collaboration opérationnelle entre SDIS 56 et 35

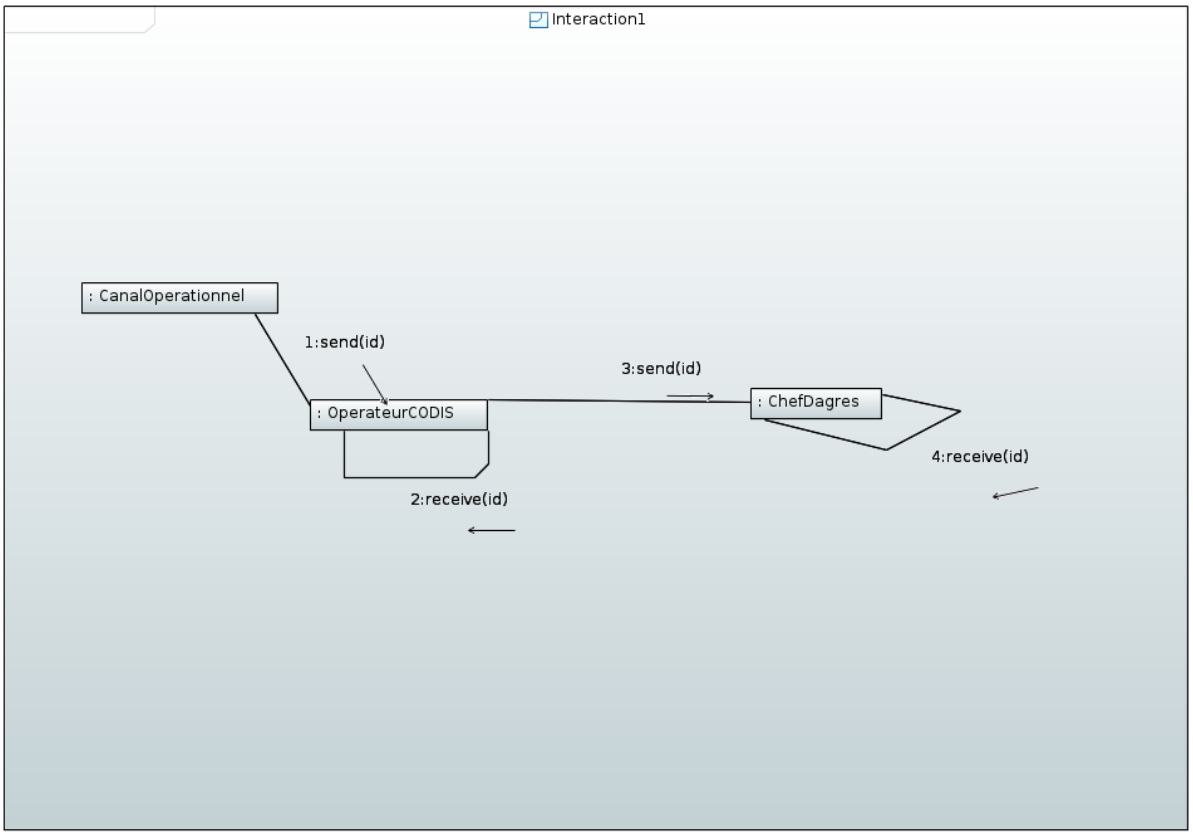

FIGURE 5.13: Activité "s'identifier" : envoi de message

SV-10c - Scénarios d'exécution

Pour les scénarios d'exécution nous utilisons des diagrammes d'interaction (séquence ou communication). Ces diagrammes montrent les données échangées et les opérations invoquées entre les acteurs pendant la réalisation des activités. Ils modélisent une exécution possible avec les acteurs participants et les ressources physiques utilisées. Les connecteurs, entre l'acteur et une ressource, montrent que l'acteur utilise deux ressources physiques un envoi de message. La figure 5.13 est un diagramme de communication qui montre les envois de messages correspondant à l'activité d'identification réalisée par le chef d'agrès du SDIS 35.

Nous modéliserons aussi des comportements d'erreur dans le SDS. Par exemple, alors qu'un VTU a engagé une communication avec l'opérateur CODIS, un second VTU tente d'engager une communication.

5.2.2 Cas 2 : collaboration tactique entre SDIS 56 et 35

Nous modélisons maintenant la deuxième configuration qui représente la collaboration tactique entre les SDIS 56et 35.

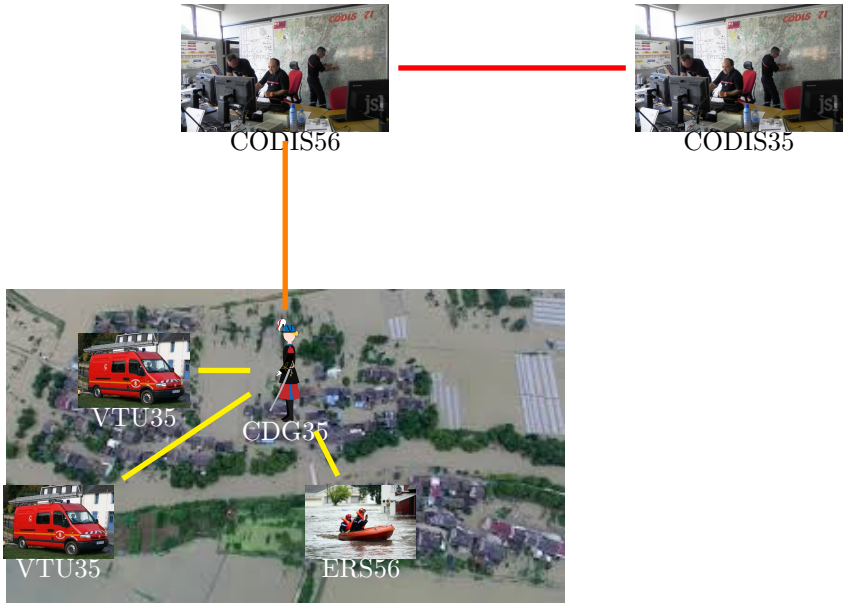

FIGURE 5.14: OV-1a - collaboration tactique entre SDIS 56 et 35

Exigences

Comme pour la première configurations le diagramme des exigences est celui du chapitre 4 où la figure 4.3 montre la structuration des exigences de la collaboration tactique interdépartementale.

OV-1a

D'après l'étude de cas, la collaboration tactique implique un chef de groupe en plus, par rapport au cas précédent, comme nous le voyons sur la figure 5.14.

OV-1b - Cas d'utilisation

Le diagramme de cas d'utilisation 5.15 montre les comportements impliqués dans le cas d'une montée en charge d'une intervention. C'est l'opérateur du CODIS qui peut la déclencher dans le cas où il engage un véhicule de commandement léger du SDIS 35. L'engagement d'un véhicule de commandement nous oblige à ajouter, en plus de la coordination tactique, une coordination stratégique et un canal de communication tactique.

OV-5

Les activités de communication de la collaboration tactique suivent le même principe que dans la première configuration.

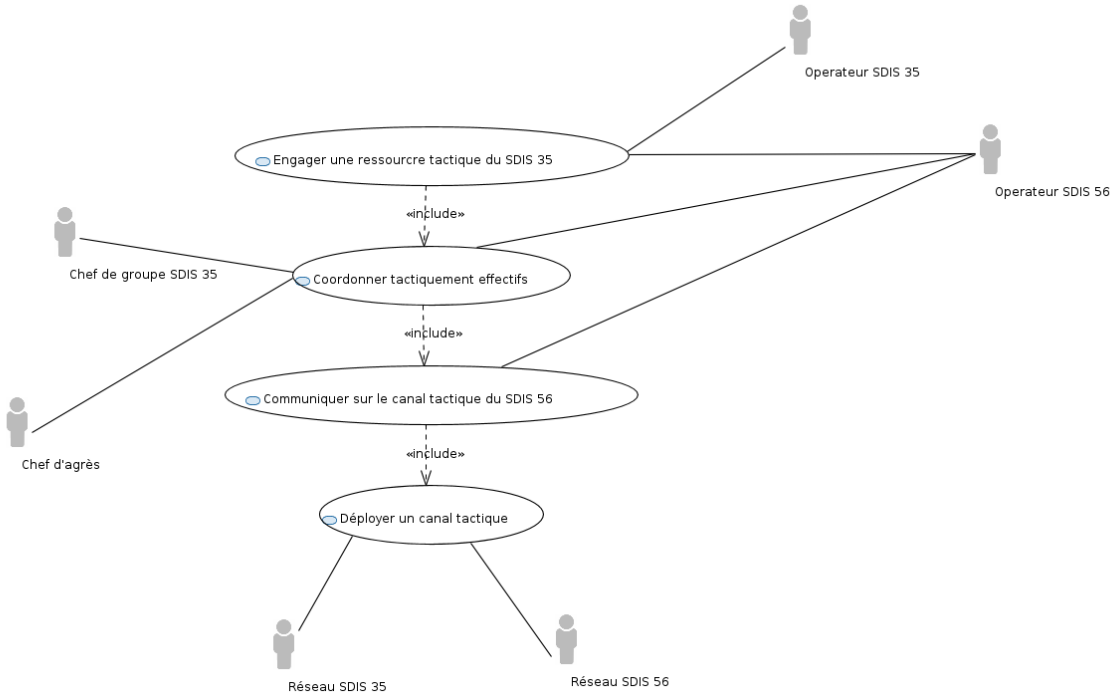

FIGURE 5.15: OV-1b - collaboration tactique entre SDIS 56 et 35

OV-4 - Synthèse des systèmes constituants de la collaboration tactique entre SDIS 56 et 35

Par rapport au cas précédent, l'ensemble des CSs n'évolue pas, cependant de nouvelles ressources sont déployées. La figure 5.16 fait la synthèse des CSs.

SV-1 - Configuration

Nous sommes à la quatrième phase qui se concentre sur la composition des constituants. Le diagramme de définition de blocs (5.17) et celui de package (5.19) nous montrent le cas où l'architecture organise la collaboration tactique et stratégique entre les deux SDIS. L'architecture inclut, en plus de la primitive *Shield*, la primitive *Layer* qui restreint les communications par niveau hiérarchique. Nous voyons sur la figure 5.18 que les composants du groupe opérationnel ne peuvent pas communiquer avec ceux du groupe tactique.

SV-10c - Scénarios d'exécution

Les interactions, dans ce deuxième cas, respectent le même patron de communication que dans le cas précédent. Ici, les communications se déroulent entre le chef d'agrès et le chef de groupe, puis en entre le chef de groupe et l'opérateur CODIS. Le protocole de communication reste le même : si le chef d'agrès veut communiquer un rapport, il doit s'identifier et demander l'autorisation de parler, etc.

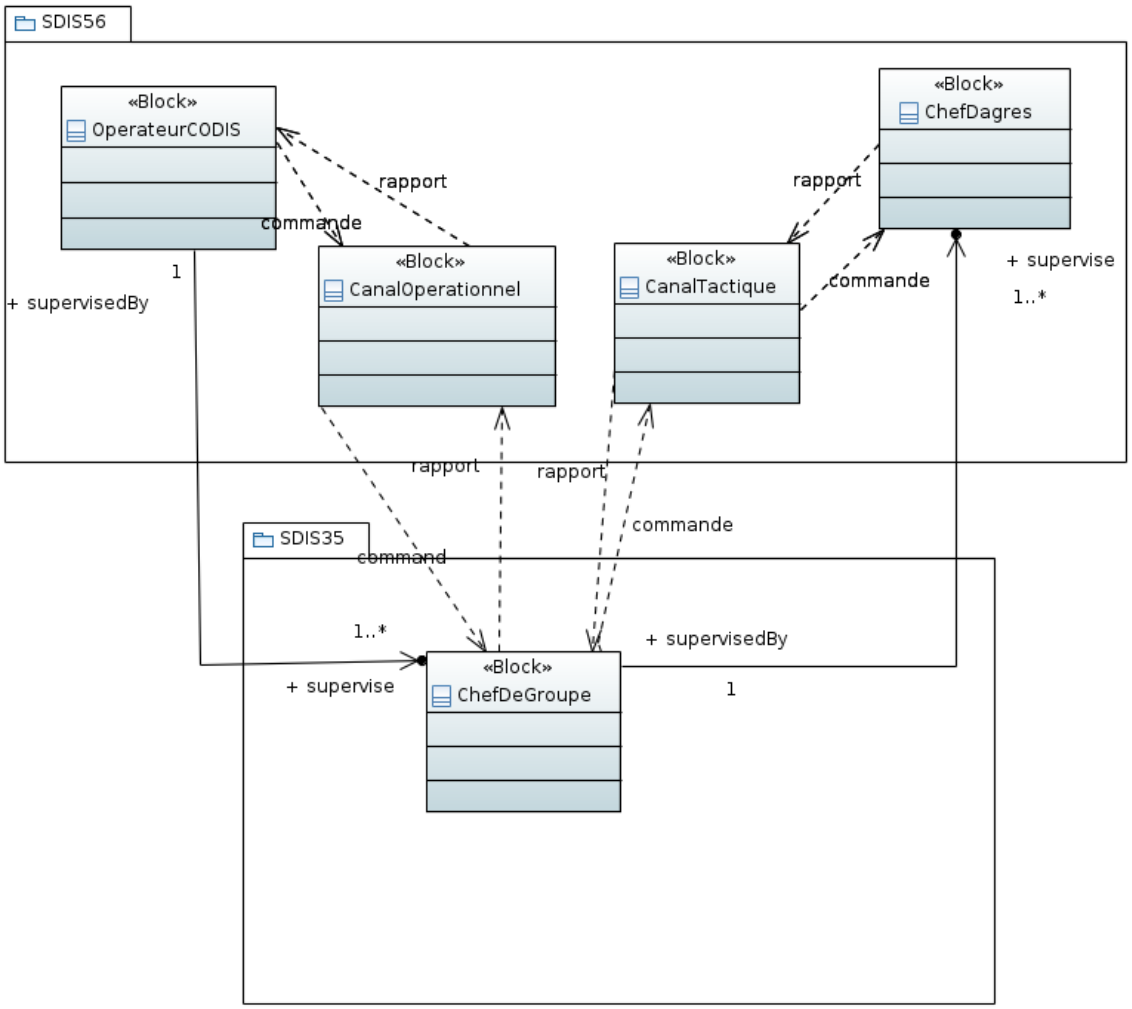

FIGURE 5.16: OV-4 - synthèse des systèmes constituants de la collaboration tactique entre SDIS 56 et 35

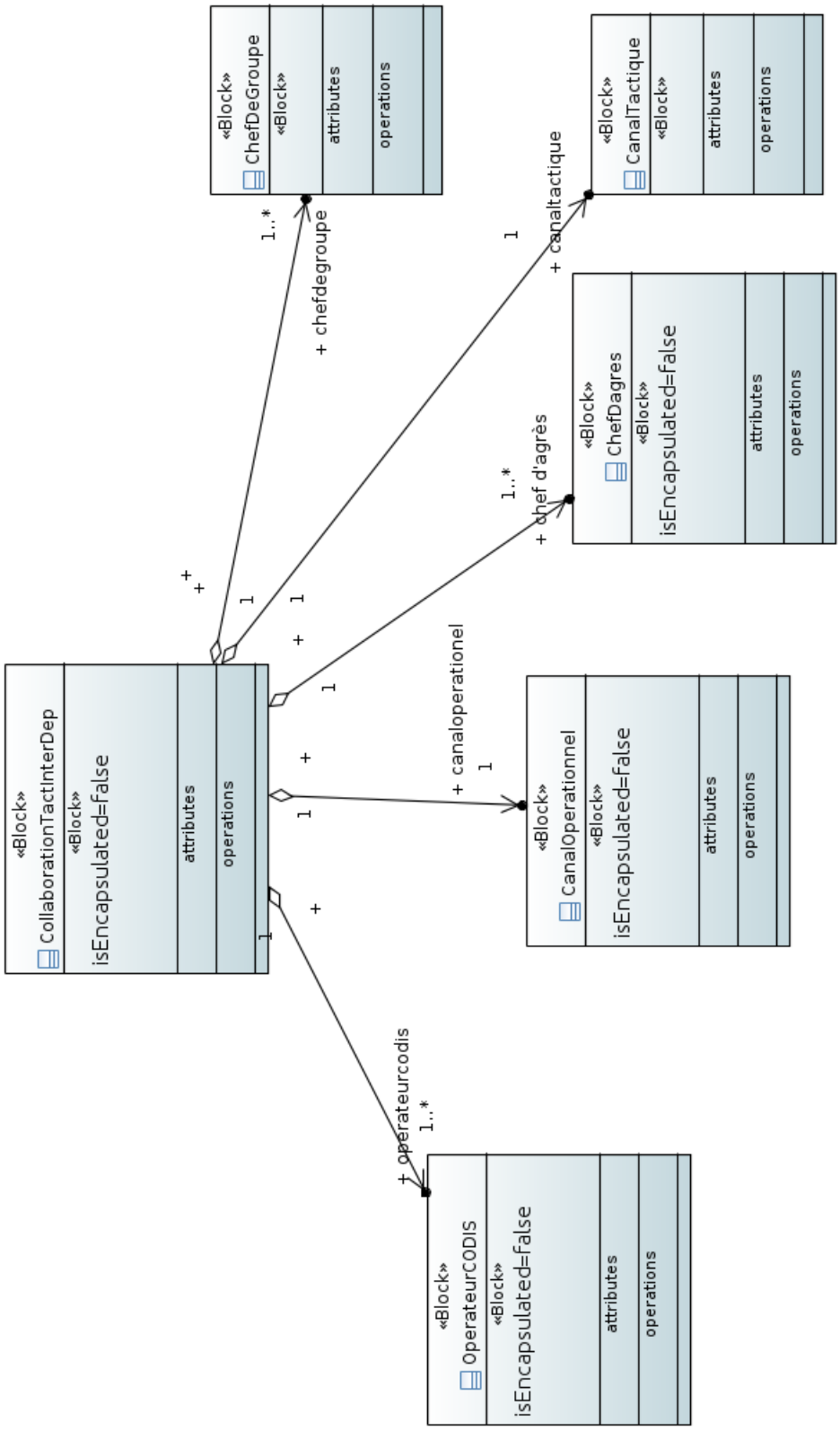

FIGURE 5.17: SV-1 BDD de la collaboration tactique entre SDIS 56 et 35

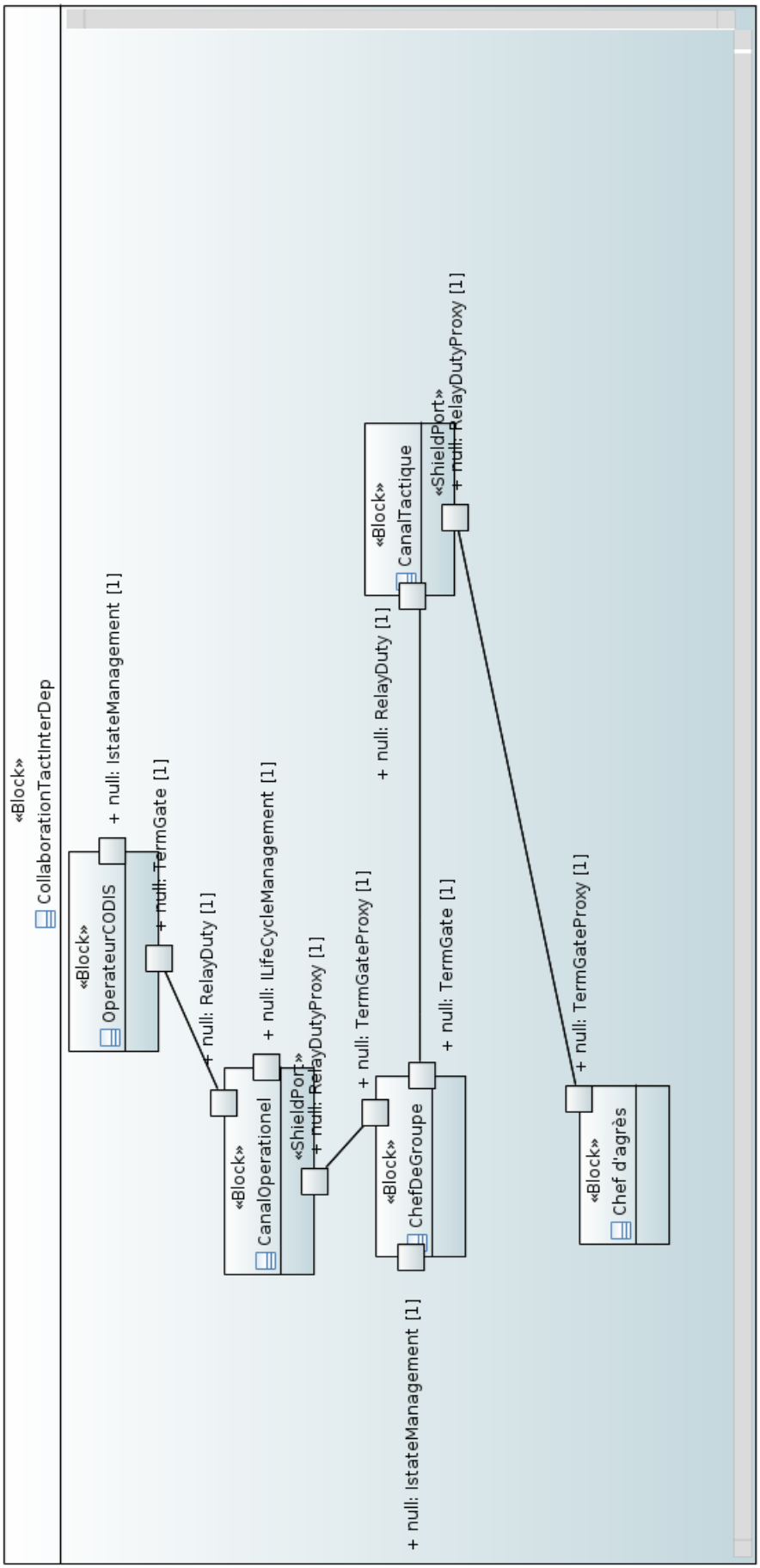

FIGURE 5.18: SV-1 BDI de la collaboration tactique entre SDIS 56 et 35

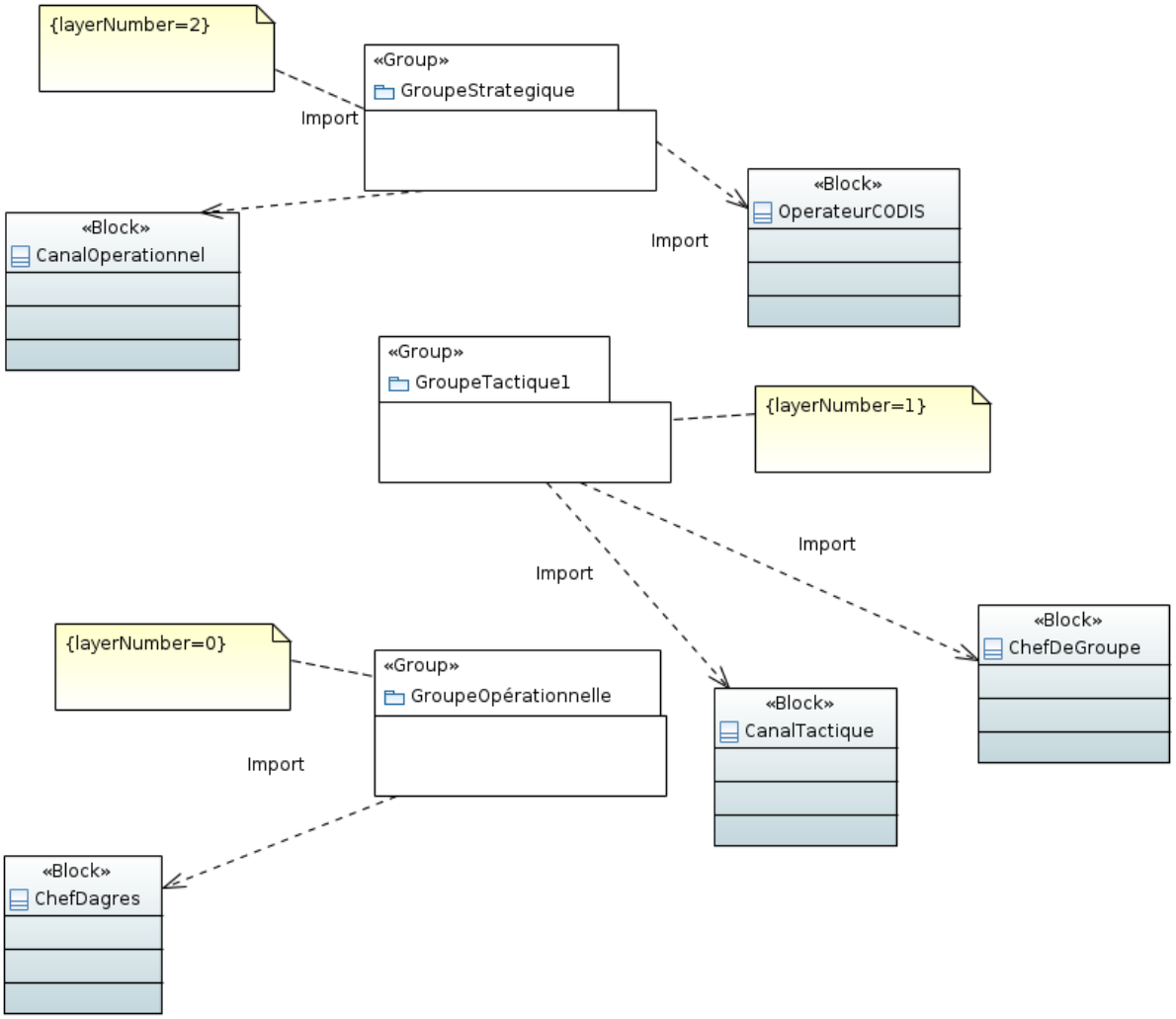

FIGURE 5.19: SV-1 Diagramme de package de la collaboration tactique entre SDIS 56 et 35

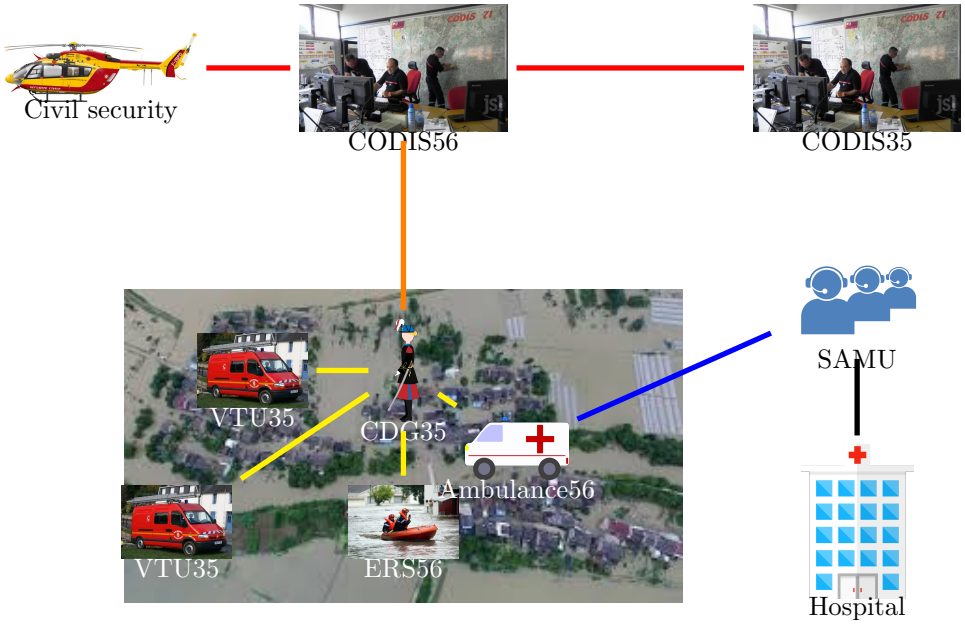

FIGURE 5.20: OV-1a - collaboration médicale

5.2.3 Cas 3 : collaboration médicale entre SDIS 56 et 35, le SAMU et la sécurité civile

Exigences

De même pour ce dernier cas le diagramme des exigences est celui du chapitre 4 où la figure 4.4 montre la structuration des exigences de la collaboration médicale.

OV-1a

La première étape de la deuxième phase du processus présente avec la figure 5.20 une vue générale de la collaboration médicale expliquée dans le cas d'étude.

OV-1b - Cas d'utilisation

La figure 5.21 montre les cas d'utilisation associés à la collaboration médicale. La régulation médicale implique un chef d'agrès et l'opérateur du SAMU. En effet, c'est le chef d'agrès qui transmet directement les rapports médicaux à l'opérateur du SAMU. Ce comportement dépend du canal de communication médicale. Ce canal est fourni par la conjonction du réseau de télécommunication du SAMU et du SDIS 56. La collaboration médicale implique également l'évacuation urgente des victimes. C'est l'opérateur CODIS qui déclenche et coordonne le pilote d'hélicoptère pour ce cas d'utilisation. Ce cas d'utilisation dépend également d'un canal de communication, qui est de niveau stratégique, et qui est déployé sur le réseau du SDIS 56.

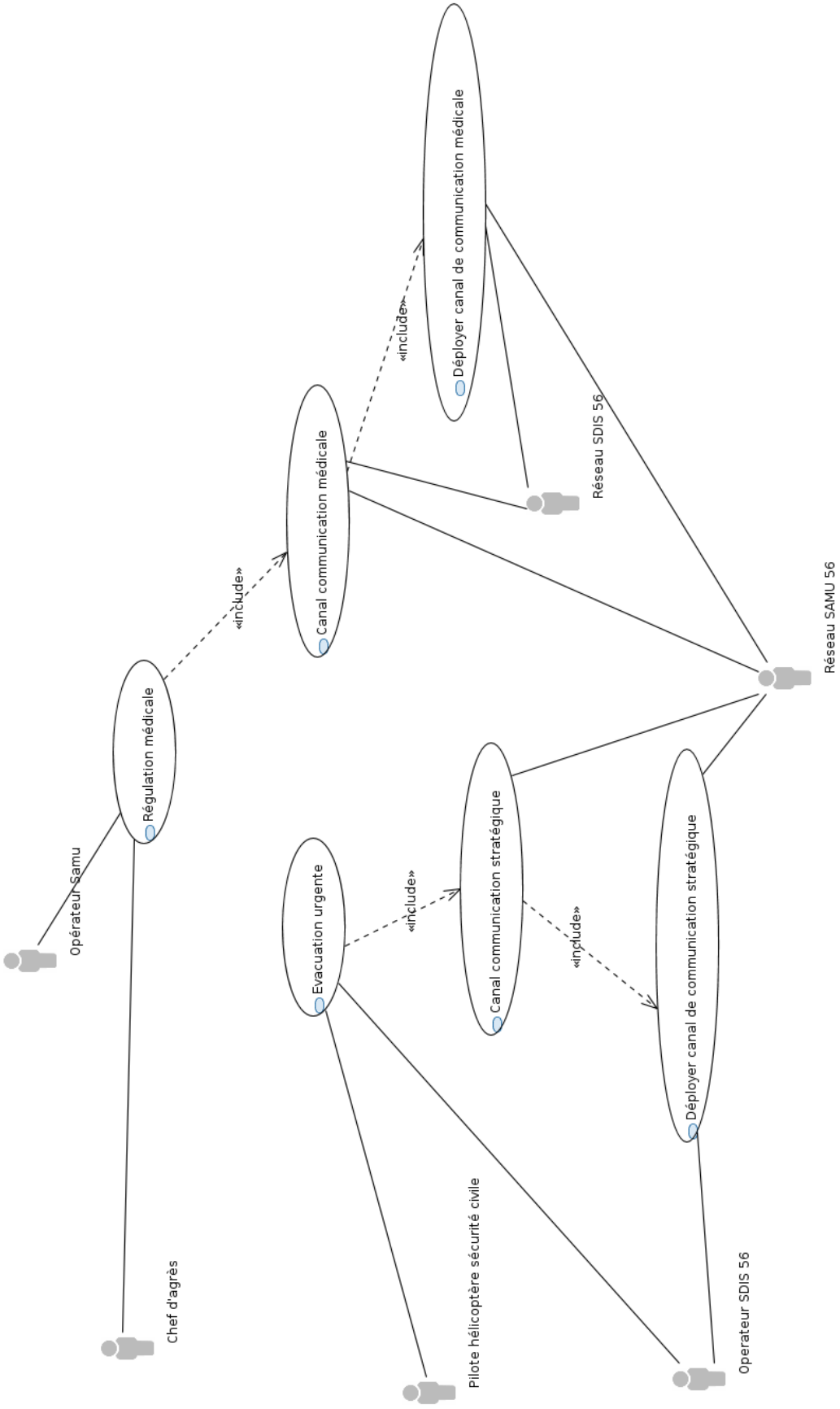

FIGURE 5.21: OV-1b collaboration médicale

OV-5

Les activités de communication de la collaboration médicale suivent le même principe que dans les autres configurations.

OV-4 - Modélisation des systèmes participants

La figure 5.22 montre les CSs impliqués dans le SdS qui sont le SAMU, le SDIS 56, le SDIS 35 et la sécurité civile.

SV-1 - Configuration

Nous sommes à la quatrième phase du processus qui se concentre sur la composition des constituants. Le diagramme de définition de blocs (5.23) et celui de package (5.25) nous montrent le cas où l'architecture organise la collaboration médicale ente tous les systèmes constituants de la collaboration médicale. La figure 5.24 est le diagramme de blocs internes qui décrit les connexions entre les différents CSs. Les différents canaux de communication (opérationnel, stratégique, et médical) montrent comment la communication peut s'établir entre les constituants du SdS.

5.3 Synthèse

L'objectif de ce chapitre était de préparer la phase de validation de notre approche de reconfiguration par patron avec la définition d'un cadre de modélisation d'architecture de SdS. Ce cadre nous permettra de modéliser des SdSs pour vérifier les propriétés de reconfiguration. Nous abordons deux problématiques qui sont, d'une part, la définition d'architecture exacte pour les SdSs et, d'autre part, le développement des modèles pour supporter la vérification des propriétés de reconfiguration par simulation, en limitant l'impact de l'effort de modélisation des propriétés dynamiques et l'explosion des états à vérifier. Nous avons proposé pour cela un framework de modélisation basé sur un langage de modélisation graphique, comprenant des vues et des diagrammes, accompagné d'un processus de modélisation descendant.

La résolution de la problématique de l'exactitude de l'architecture réside dans l'approche du processus de modélisation. Le processus de modélisation nous a permis d'identifier les comportements et les principales abstractions du SdS, modélisés dans le point de vue SdS, puis de les raffiner et d'en capturer les contraintes majeures à vérifier, de les modéliser dans le point de vue CS. Le choix de UPDM a facilité la phase de vérification de l'exactitude en permettant de valider manuellement que le choix des abstractions et des comportements étaient pertinents par rapport à la documentation étudiée.

La vérification des propriétés de reconfiguration est fondée sur une approche par simulation. Pour limiter les efforts de modélisation des propriétés dynamiques du SdS, notre stratégie de modélisation consiste à capturer les régularités des architectures, plutôt que de modéliser des comportements qui dépendent de l'exécution. Cette caractéristique est

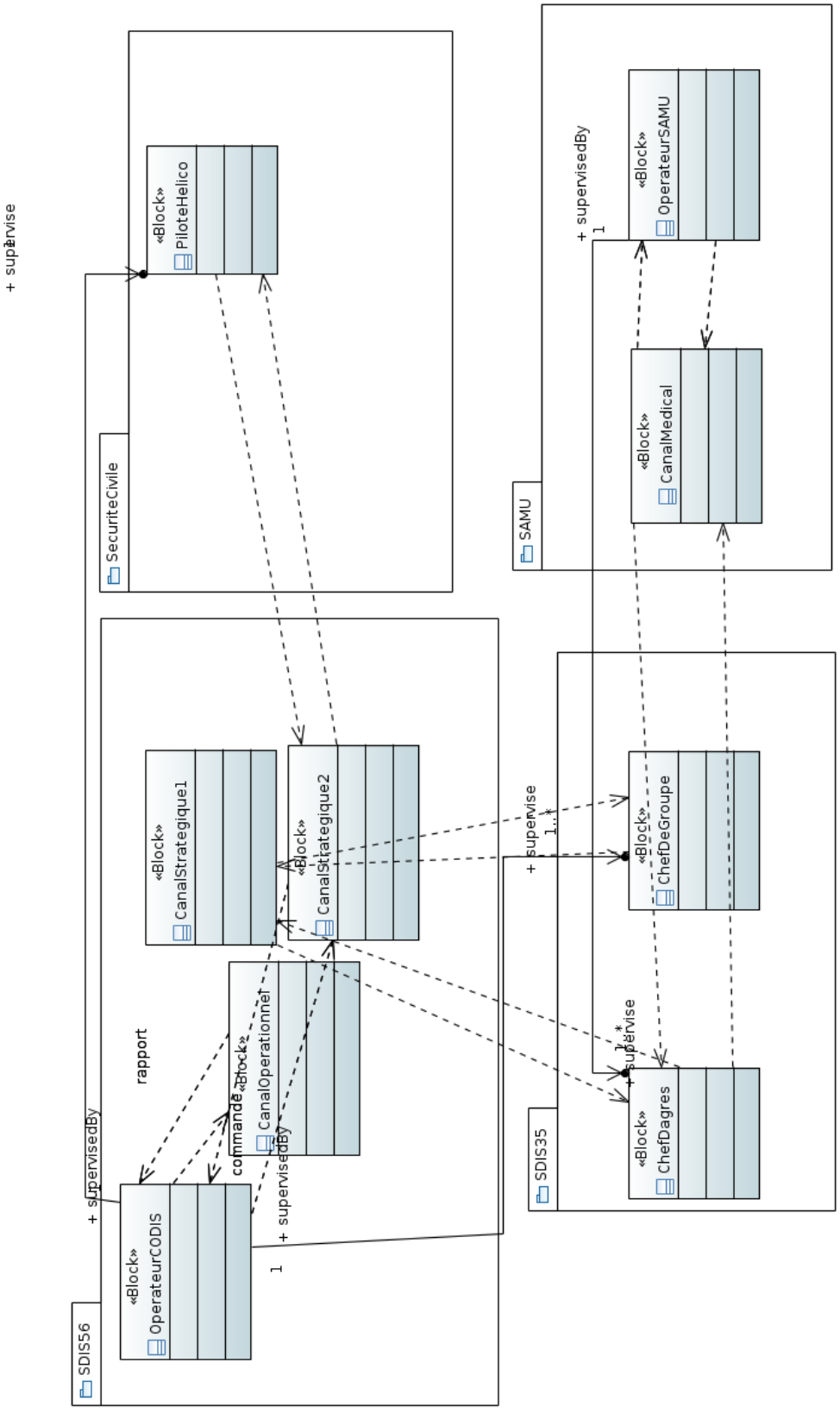

FIGURE 5.22: OV-4 - synthèse des systèmes constituants

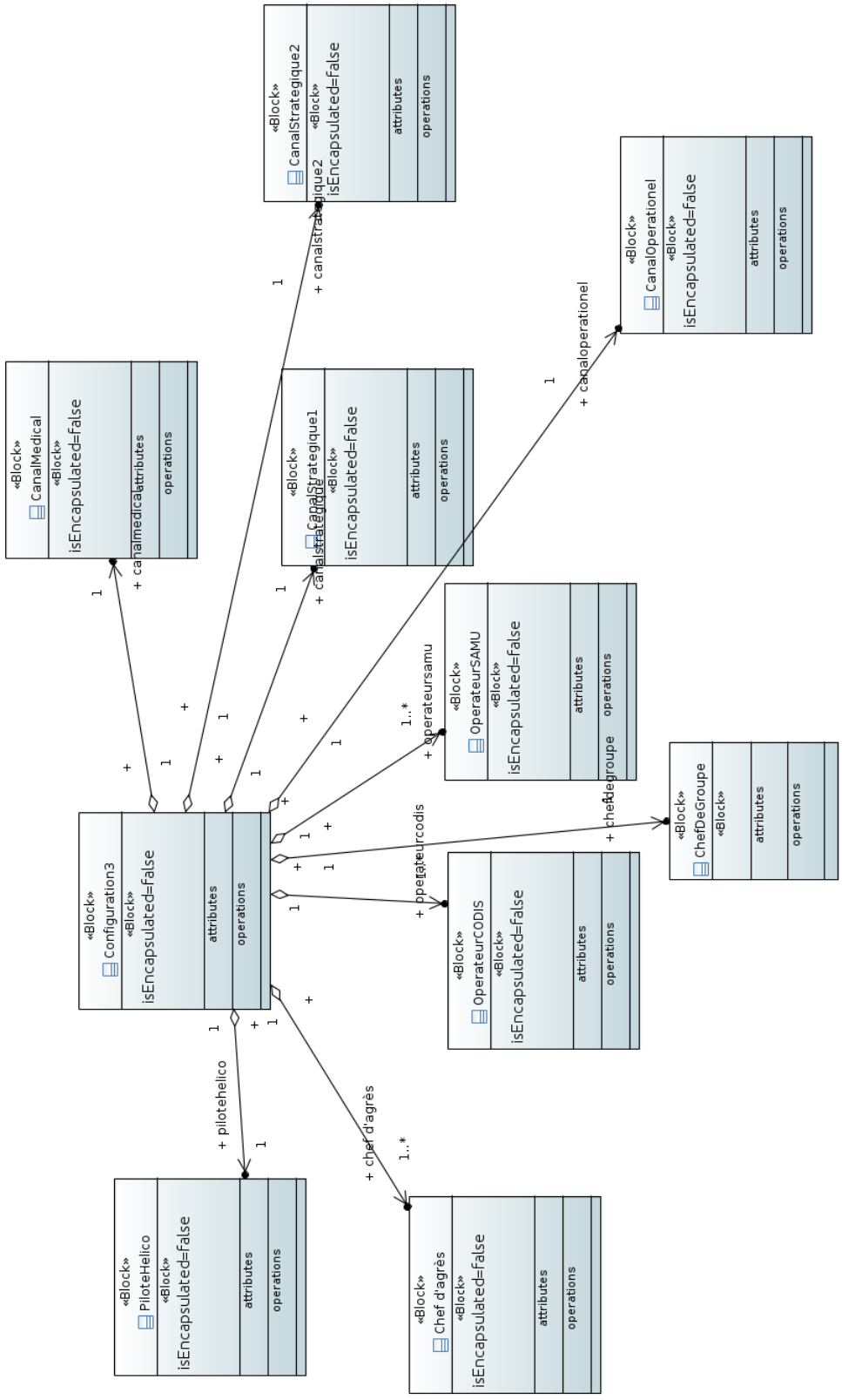

FIGURE 5.23: BDD collaboration médicale

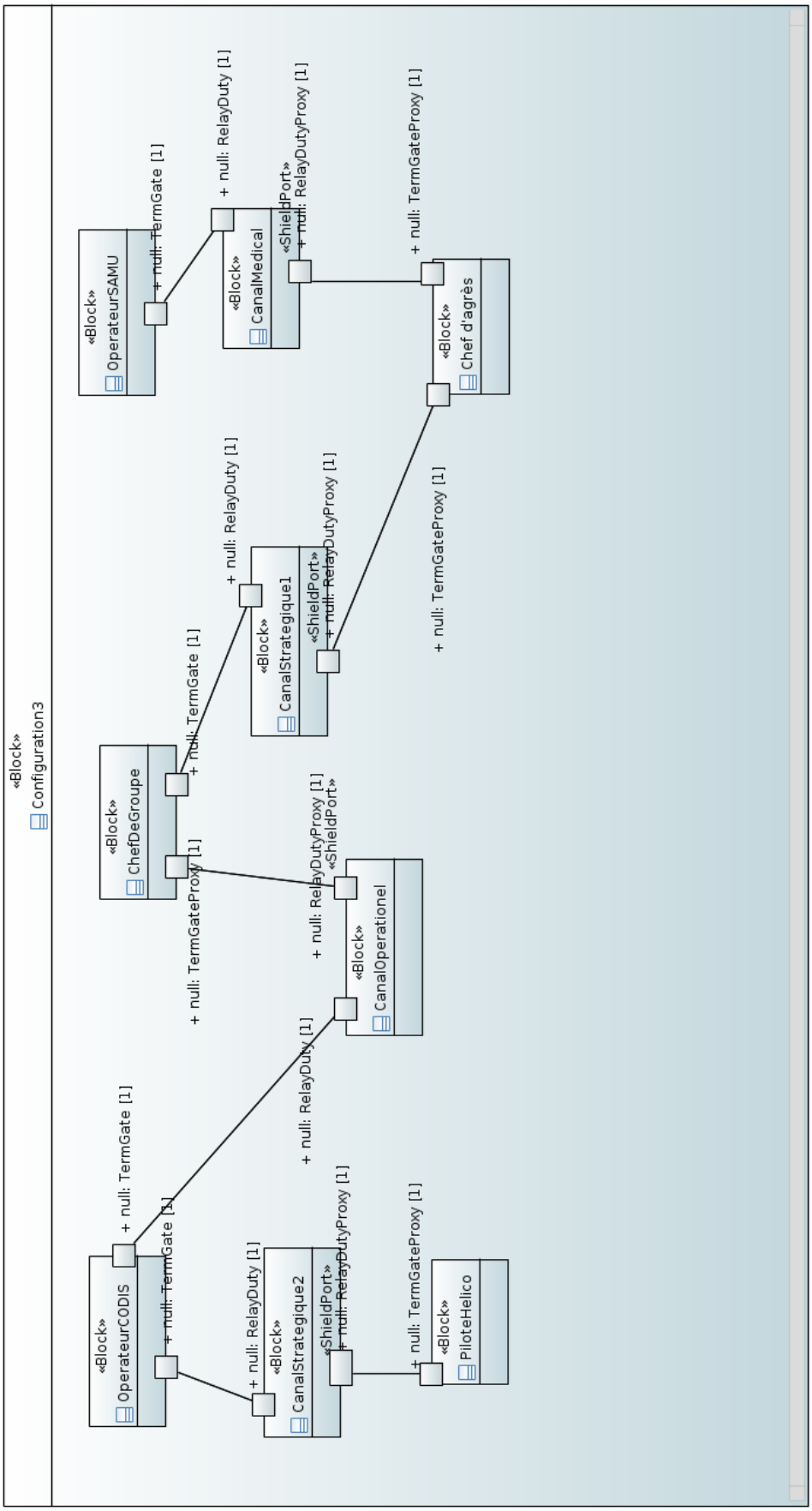

FIGURE 5.24: BDI collaboration médicale

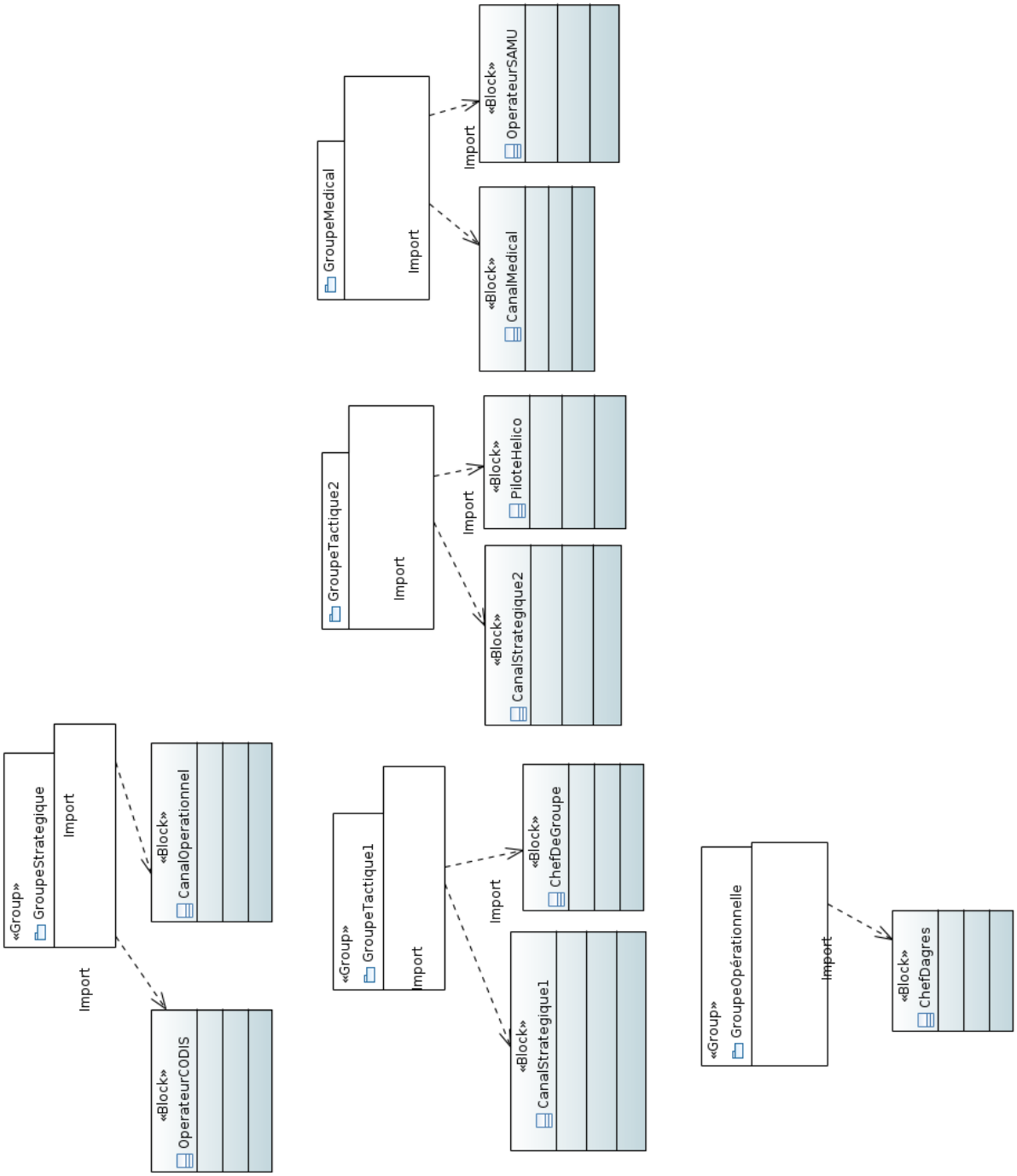

FIGURE 5.25: SV-1 Diagramme de package de la collaboration médicale

supportée par le choix de diagrammes statiques pour les vues SV-1 mais aussi par le fait d'intégrer l'utilisation de contraintes OCL dans les diagrammes. Dans les projets DANSE (Designing for Adaptability and evolutioN in System of systems Engineering) et COMPASS (Comprehensive Modelling for Advanced Systems of Systems) les contraintes exprimables avec OCL sont décrites après transformation des diagrammes vers des langages formels. Les contraintes qui sont ajoutées, après transformation, deviennent moins vérifiables manuellement par la suite. Une seconde difficulté des approches par simulation est l'explosion des états à vérifier. Nous avons donc choisi de restreindre les exécutions possibles du SdS à quelques scénarios d'exécution. Ces scénarios sont décrits par les vues SV-10c et l'utilisation de diagrammes de séquence ou de communication. UPDM n'a pas une description suffisamment précise pour la description du comportement dynamique et cela nous a poussé à préciser une sémantique d'exécution inspiré du π-calcul. Pour le choix des opérations à implémenter nous nous somme inspirés du langage SosADL, développé dans l'équipe Archware, qui supporte cette sémantique opérationnelle.

Si nous comparons nos travaux à ceux de COMPASS et DANSE, nous pouvons voir que le processus de modélisation est très similaire. Notre processus de modélisation est descendant avec un point de vue SdS et CS. Il inclut également une phase de vérification dynamique. Dans notre approche du processus de modélisation, nous avons explicité une phase de vérification de l'exactitude. Nous choisissons également un langage de modélisation graphique. À la différence de COMPASS nous avons choisi UPDM comme dans le projet DANSE. Bien que les frameworks de modélisation soient équivalents nous avons préféré choisir un standard. Du point de vue du choix des diagrammes pour les vues OV-1b nous avons préféré utiliser un diagramme de cas d'utilisation plutôt qu'un diagramme de bloc. Le diagramme de cas d'utilisation nous a permis une meilleure modélisation des relations complexes entre les acteurs du SdS et facilite la correspondance avec les vues OV-5.

Le framework a permis de développer trois modèles architecturaux pour trois configurations. Les modèles développés sont facilement exécutables. Ils sont développés avec des scénarios d'exécution qui permettent de tester des scripts de reconfiguration et de vérifier quelles sont les propriétés préservées pendant la reconfiguration. Cependant, les modèles développés pour chacune des architectures de SdS ne sont pas complets. En effet, il manque une description des activités associées pour chaque cas d'utilisation. Pour l'instant ces cas d'utilisation sont décrits textuellement et de façon succincte.

Chapitre 6

Patron de reconfiguration

Nous avons vu dans l'état de l'art que nous envisageons une approche de reconfiguration centrée sur l'architecte. La raison principale est que, dans les SdSs (systèmes de systèmes), les reconfigurations ne peuvent pas être anticipées à la conception du SdS. Par conséquent, un certain nombre de choix sont reportés au moment de l'exécution.

L'approche proposée repose sur l'utilisation de documentation dont l'architecte dispose pour la conception des reconfigurations dynamiques. Dans ce chapitre nous définissons le concept de *patron de reconfiguration* comme étant un élément de cette documentation à des fins de réutilisation. Dans l'état de l'art, et indépendamment du contexte de la reconfiguration, les patrons sont des solutions identifiées à des problèmes de conception récurrents. Dans le contexte de la reconfiguration, les patrons de reconfiguration décrivent en particulier comment passer d'une architecture à une autre en maintenant des propriétés bien définies. Pour cette raison les patrons de reconfiguration explicitent, pour un problème et sa solution, l'intention, le contexte et les conséquences.

La section 6.1 définit le contenu d'un patron de reconfiguration. Dans la section 6.2 nous expliquons comment nous construisons nos patrons de reconfiguration. Comme nous le verrons, une manière de procéder consiste à synthétiser la bibliographie abordant une solution particulière à un problème particulier. Enfin nous verrons comment mettre en forme ces connaissances dans un patron de reconfiguration.

6.1 Contenu du patron

Le contenu des patrons de reconfiguration reprend les principes des patrons d'une manière générale, à savoir, les sections *nom*, *intention*, *contexte*, *problème*, *solution* et *conséquences*. Pour certaines de ces sections, le contenu est adapté spécifiquement au domaine de la reconfiguration. Ainsi, la description du problème est centré sur la modification de l'architecture espérée, présentée par exemple sous la forme d'extraits génériques d'architectures. Pour la solution, le mécanisme présenté par le patron est accompagné d'une description des opérations utilisées et de la sémantique associée. Ce complément d'information doit aider l'architecte à identifier les opérations que les composants doivent

fournir, ou, le cas échéant, qu'il doit ajouter (en complétant la reconfiguration) dans l'architecture.

Ce dernier cas, où la reconfiguration doit être complétée pour ajouter des opérations de reconfiguration dans l'architecture, nous permet de remarquer que tous les patrons de reconfiguration n'adressent pas des problèmes au même niveau. Ainsi, le patron *Quiescence* servant d'exemple dans cette section est un patron qui résoud un problème de plus bas niveau que le patron *Co-evolution* décrit en section 6.2.2.

Nous détaillons ci-après un patron faisant référence à la quiescence présentée par Kramer and Magee (1990).

Nom

Le nom du patron identifie le patron dans le catalogue sous la forme d'une métaphore. Cela doit aider l'architecte à comprendre ce que fait le patron et permettre d'identifier un patron plus rapidement dans la phase de conception. Par exemple, dans le domaine de la biologie, la quiescence fait référence à une phase de repos durant laquelle la cellule cesse ces activités (e.g elle arrête de se diviser) [1]. Dans le contexte de la reconfiguration, l'architecte peut s'attendre à ce que le patron nommé *Quiescence* suspende les processus du système qui s'exécute.

Intention

L'intention résume ce que fait le patron. Elle décrit l'approche générale que le patron suit pour mettre en œuvre la reconfiguration et indique quand le patron peut être appliqué avec pertinence. Dans le cas de la quiescence, l'intention devrait décrire que l'architecte veut éviter des problèmes de synchronisation entre la reconfiguration et le système, notamment lorsque certaines liaisons doivent être déconnectées (de manière transitoire). Puis le patron devrait présenter une approche générale (comme : arrêter complètement le système) et les inconvénients de cette approche générale. Ensuite, le patron devrait décrire l'approche souhaitable qui est de rendre passifs les composants qui dépendent du composant à manipuler afin de rendre quiescent ce dernier. Enfin, le patron devrait indiquer les avantages par rapport à l'approche générale. Dans cet exemple du patron *Quiescence*, l'approche préconisée réduit l'impact de la reconfiguration sur le système qui s'exécute, tout en garantissant l'absence de problème de synchronisation avec l'exécution du système.

Contexte

Cette section décrit l'historique d'utilisation du patron. Elle permet d'identifier tous les facteurs qui empêcheraient d'appliquer le patron, ou, à l'inverse, les conditions préalables à son application. Nous documentons en particulier le contexte architectural dans lequel il est utilisable. Comme expliqué par Taylor et al. (2009); Dorn and Taylor (2015) le style architectural représente une contrainte pour l'application de la reconfiguration. Nous capturons également dans le contexte le type de gouvernance du SdS. Pour le cas de la quiescence le patron décrit comment déterminer les composants à rendre passifs. Cet

1. `https://fr.wikipedia.org/wiki/Quiescence` - consulté le 20/09/2018

algorithme est centralisé. Donc, au regard de ces caractéristiques, le contexte indique que le patron *Quiescence* ne peut être utilisé que dans le contexte d'un SdS ayant une gouvernance de type dirigé.

Problème

La section décrit le problème résolu par le patron. Cela se traduit par un ensemble de besoins que le patron cherche à satisfaire. Pour ce faire, la section problème indique deux architectures (ou deux extraits de l'architecture), l'une initiale et l'autre cible, qui modélisent les changements obtenus par l'application du patron. Le choix de modélisation doit supporter la compréhension du problème. Par exemple, dans le cas de la quiescence, les architectures doivent modéliser, d'une part, au minimum les dépendances requises, les dépendances fournies par les composants et les liaisons pour capturer le problème de dépendance entre les composants. D'autre part, le choix des architectures source et cible doit être pertinent. Dans le cas de la quiescence, elles doivent montrer la modification des liaisons puisque, dans le cas contraire, la quiescence n'aurait pas d'intérêt.

Ensuite, la section problème indique les propriétés que la reconfiguration doit satisfaire. Les propriétés sont exprimées en termes d'invariants structurels et d'invariants comportementaux. Les invariants structurels peuvent être des dépendances qui doivent toujours être maintenues pendant la reconfiguration. Les invariants comportementaux sont des contraintes sur le comportement tels que des comportements déclenchés ou à éviter lorsque des événements surviennent. Dans le cas de la quiescence, le patron devrait imposer un invariant comportemental afin de ne permettre la déconnexion d'une liaison que lorsque les composants envoyant des messages via cette liaison sont dans l'état de quiescence.

Enfin, la section problème liste les forces du patron qui documentent les principes mis en œuvre par le patron dans la solution. Le patron *Quiescence* devrait citer et synthétiser l'article de (Kramer and Magee, 1990) pour définir les états actif et passif, ainsi que l'état de quiescence. Ainsi, le patron devrait indiquer que tous les composants, qu'ils soient dans l'état actif ou dans l'état passif, doivent prendre en compte les messages qu'ils reçoivent, mais que seuls les composants dans l'état actif ont le droit d'engager de nouvelles transactions. Le patron devrait également préciser qu'un composant est quiescent s'il est possible de garantir qu'il n'engage aucune transaction et si aucun autre composant ne lui envoie de message. Le patron devrait enfin décrire que le principe mis en œuvre pour rendre un composant quiescent consiste à rendre passif ce composant, ainsi que tous les autres composants susceptible d'impliquer ce composant dans une transaction, directement ou indirectement.

Solution

La section solution indique comment est résolu le problème décrit par la section précédente. Elle pointe les problèmes que peut introduire la solution, notamment les mécanismes d'évolution utilisés par sa mise en œuvre. La documentation de la solution est accompagnée d'une grammaire de reconfiguration qui donne la sémantique précise des opérations de reconfiguration. Modélisée par un diagramme d'état ou par un diagramme de collaboration et accompagnée d'un texte descriptif, la grammaire de reconfiguration d'un

patron indique les opérations de reconfiguration sur laquelle repose la solution préconisée par ce patron, ainsi que les enchaînements autorisés de ces opérations selon l'état des systèmes et les événements susceptibles de survenir. Pour le cas de la quiescence, le patron devrait décrire le protocole de reconfiguration consistant à rendre passifs les composants dépendant (directement ou indirectement) du composant ciblé, d'appliquer les opérations de déconnexion/reconnexion, puis remettre tous les composants dans leur état actif.

Conséquences

La section discute l'impact du patron sur les aspects non fonctionnels du SdS. Il peut s'agir de la disponibilité des services pendant la reconfiguration, la performance de ces services et leur fiabilité. Dans l'exemple de la quiescence, le patron devrait expliquer les limites et avantages de la quiescence. Le patron devrait, par exemple, expliquer qu'en rendant passif l'ensemble des composants dépendant directement ou indirectement du composant ciblé, de nombreux services peuvent être indisponibles. L'impact peut par exemple être plus large que le seul ensemble des composants rendus passifs, notamment dans le cas où d'autres composants en apparence indemnes se retrouvent isolés des composants actifs.

6.2 Processus de définition d'un patron de reconfiguration

Dans cette section, nous allons montrer sur des exemples spécifiques comment, à partir d'un problème de reconfiguration et d'une solution clairement établie dans la bibliographie, nous en extrayons un patron de reconfiguration.

6.2.1 Données en entrée du processus

Les patrons sont par définition des artefacts qui capturent des solutions éprouvées et reconnues à des problèmes récurrents. Par conséquent, l'élaboration d'un patron requiert l'étude de solutions préexistantes pour le constituer, par exemple par une étude bibliographique du domaine de la reconfiguration ou par la connaissance du domaine de la reconfiguration dynamique.

Problème récurrent

Une application de la reconfiguration consiste à mettre à jour un composant vers une nouvelle version. Nous pouvons généraliser le problème du remplacement d'un composant par un autre dans une architecture, qu'il s'agisse d'une nouvelle version ou d'une variante alternative, et qui est en effet un problème particulièrement fréquent.

Présenté ainsi, cependant, ce problème est certes récurrent mais trop général dans la mesure où des solutions différentes les unes des autres ont été proposées. Ce problème peut par exemple être résolu par la suspension du composant en le mettant dans l'état de quiescence de Kramer and Magee (1990), ou en coordonnant la cohabitation transitoire des deux composants (ou versions ou variantes) comme Wernli et al. (2013); Hjálmtýsson

and Gray (1998). À chacune de ces solutions correspondra un patron distinct, pour lequel la formulation du problème récurrent traité est précisée pour exprimer, par exemple, la présence ou l'absence de contrôle sur le cycle de vie des composants.

Ayant déjà utilisé le patron *Quiescence* pour illustrer la section précédente, et afin de donner un second exemple, nous considérons que le problème récurrent est le remplacement d'un composant lorsque l'architecte ne peut pas contrôler le cycle de vie des composants, c'est-à-dire lorsque les composants disposent de leur indépendance opérationnelle.

Solutions

Dans l'étude bibliographique, nous avons donc identifié les travaux de Wernli et al. (2013) et ceux de Hjálmtýsson and Gray (1998) comme traitant le problème tel que nous l'avons formulé. Définir la solution du patron en cours d'élaboration consiste à identifier, dans chacun de ces travaux, la manière dont le problème est résolu. Certes, chacun de ces travaux a ses spécificités : soit nous parvenons à faire abstraction de ces différences pour dégager une solution commune ; soit il s'agit d'un indicateur que le problème doit encore être précisé, conduisant dans ce cas à deux patrons distincts.

Comme nous allons le voir, nous allons effectivement parvenir, malgré les différences entre les deux travaux, à identifier une solution commune au problème tel que nous l'avons formulé.

Les travaux de Wernli et al. (2013) introduisent la notion de contexte pour désigner la version utilisée pour les traitements. Lorsque le développeur modifie l'implémentation d'une classe, un nouveau contexte est créé. Conceptuellement, toutes les instances de cette classe existent dans tous les contextes, y compris dans le contexte nouvellement créé, et un mécanisme de synchronisation bidirectionnelle assure que toute opération réalisée sur une instance dans un contexte particulier produit aussi ses effets dans l'ensemble des autres contextes. Les détails de la synchronisation entre les contextes peuvent être ajustés selon une stratégie paresseuse ou immédiate. Chaque fil d'exécution s'exécute dans un contexte, c'est-à-dire que les appels de méthodes sont dirigés vers l'implémentation de la classe correspondant au contexte. Wernli et al. (2013) argumentent que le passage d'un contexte à l'autre doit être explicite, donnant l'exemple d'un site web dont chaque requête est intégralement traitée dans un même et unique contexte. Cette approche est conforme au critère de cohérence de version décrit par exemple par Ma et al. (2011) selon lequel des traitements interdépendants doivent tous s'exécuter avec une même version du système.

Hjálmtýsson and Gray (1998) proposent le concept d'une classe dynamique, c'est-à-dire une classe dont l'implémentation évolue lors des mises à jour. Pour mettre en œuvre ce concept, Hjálmtýsson and Gray (1998) proposent d'insérer un proxy pour encapsuler l'accès à l'implémentation de la classe dynamique. Lorsqu'un objet est instancié, c'est toujours la dernière version de la classe qui est utilisée par le proxy. Mais si un objet existait avant la mise à jour, alors le proxy redirige les appels de méthode vers la version qui a servi à instancier l'objet. Il s'agit, ici encore, d'une interprétation du critère de cohérence de version de Ma et al. (2011), puisque chaque instance reste pour toujours

dans la même version. Au fur et à mesure que les objets sont détruits, les anciennes versions de la classe sont de moins en moins utilisées, jusqu'à ne plus être utilisées du tout. Une opération spécifique permet de détruire tous les objets d'une version précédente, afin d'assurer que les versions précédentes ne sont effectivement plus utilisées.

Ces travaux ont des différences. Par exemple, Wernli et al. (2013) définissent un protocole de synchronisation de l'état des objets entre les différentes versions, alors que Hjálmtýsson and Gray (1998) n'imposent pas la migration vers la nouvelle version. Ces derniers forcent le développeur à adapter spécifiquement son programme, alors que Wernli et al. (2013) utilisent le niveau méta de Smalltalk pour préserver une meilleure transparence, bien que demandant au développer d'indiquer explicitement dans quelle version une instruction s'exécute.

Malgré ces différences, l'approche choisie repose dans les deux cas sur un même principe : plusieurs versions du programme mises à jour s'exécutent en même temps, de manière transitoire, pour préserver la continuité de service, et ce jusqu'à ce que l'ancien composant ne soit plus nécessaire au fonctionnement du système. C'est sur cette base que la section solution du patron est formulée.

Domaine de l'exemple

Les travaux qui servent de base à l'élaboration du patron peuvent provenir de divers domaines applicatifs ou peuvent concerner des niveaux de reconfiguration différents. Dans l'exemple que nous considérons, Wernli et al. (2013); Hjálmtýsson and Gray (1998) traitent le niveau objet, et nous avons fait référence également aux travaux de Ma et al. (2011) qui se situent au niveau composant. Cela illustre que nous nous autorisons à transposer les solutions au niveau qui nous intéresse, ici le niveau SdS, dès lors que nous pensons que la solution peut y être pertinente.

L'exemple de la cohabitation de plusieurs versions ou variantes est une solution pertinente pour le SdS, présenté au chapitre 4, qui sert de cas d'étude. En effet, il s'agit d'une solution respectueuse de l'indépendance des constituants du SdS. Par ailleurs, le cas d'étude requiert le remplacement de l'échelon de coordination des agents déployés, rôle initialement assuré par le CODIS, puis par le VLC du chef de groupe lors de la création d'un échelon tactique. Enfin, il est tout à fait acceptable que, de manière transitoire, le CODIS et le VLC du chef de groupe se synchronisent le temps que les agents déployés prennent en compte la modification de la chaîne de commandement.

6.2.2 Patron de co-évolution

Nous donnons ici un exemple de patron de reconfiguration, élaboré à partir de la description et du processus de définition donnés précédemment.

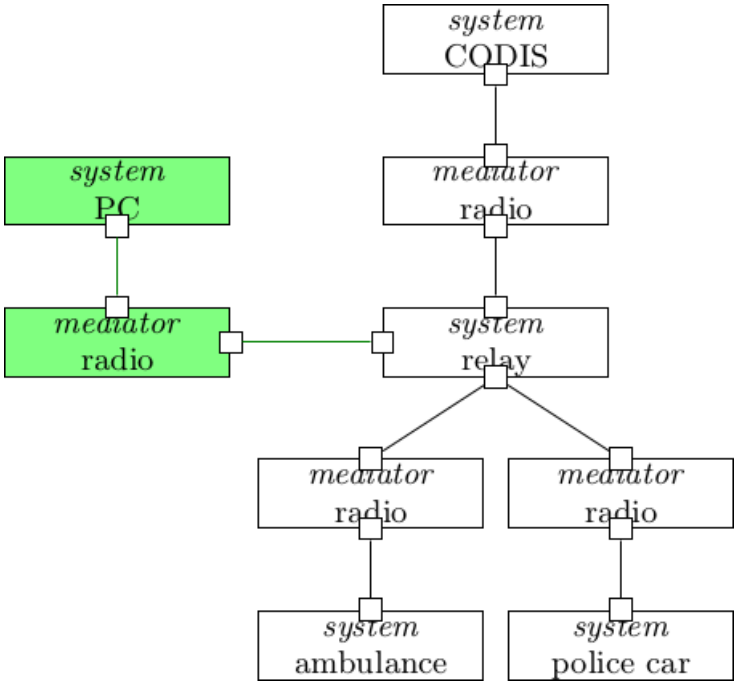

FIGURE 6.1: Exemple de SdS d'urgence reconfiguré

Nom : Co-évolution

Intention

Remplacer un composant par un autre.

Il faut donc mettre à jour les connexions vers ce composant pour reconnecter le reste du SdS à la nouvelle version. Une façon simple de procéder consiste à arrêter les composants qui interagissent avec le composant à remplacer. Néanmoins, dans le contexte des systèmes de systèmes, l'architecte est susceptible de ne pas disposer des opérations pour suspendre ou rendre passifs certains composants. De plus, si beaucoup d'autres composants dépendent directement ou indirectement du composant remplacé, le délai avant l'application effective de la reconfiguration tout comme l'indisponibilité de services risquent d'être élevés.

Une autre approche, mieux adaptée dans le contexte considéré, consiste à faire coexister les deux composants, celui à remplacer et son substitut. Les nouvelles requêtes sont dirigées vers le substitut. Dans une telle situation, les deux composants sont liés et évoluent conjointement. Ils peuvent également partager leur état. C'est pour cela que le patron se nomme co-évolution.

Une telle reconfiguration s'avère utile dans le contexte d'un SdS d'organisation des services d'urgence et de réponse à un incident. La figure 6.1 montre un exemple où le composant CODIS supervise initialement des composants opérationnels. Suite à une augmentation des ressources mises en œuvre pour la mission, l'architecte décide de déléguer

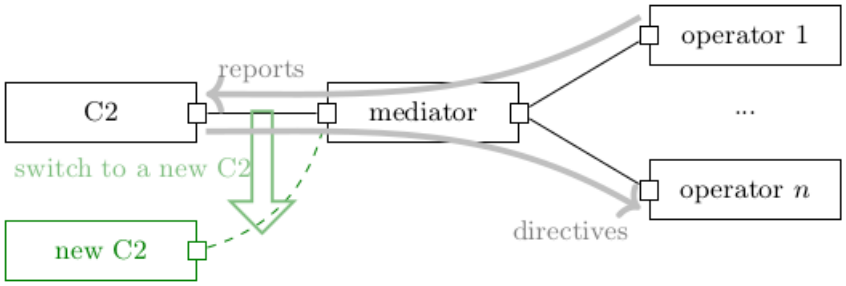

FIGURE 6.2: Contexte architectural

l'orchestration des composants opérationnels à un Poste de Commandement (PC). Les composants opérationnels doivent donc se déconnecter du CODIS pour se reconnecter à la place au composant PC. Cependant, les composants opérationnels ne peuvent pas réaliser cette opération instantanément pour diverses raisons : tous les composants ne sont peut-être pas joignables au même moment, ou bien l'indépendance managériale peut conduire certains composants à privilégier leurs autres missions quitte à retarder la reconfiguration. Les composants de l'échelon opérationnel décident de façon autonome du moment auquel ils réalisent le changement de connexion. Le CODIS ne peut donc pas totalement abandonner son rôle de coordination tant que tous les composants de l'échelon opérationnel ne se sont pas reconnectés au PC. Pendant cette phase transitoire, le CODIS et le PC doivent collaborer pour assurer correctement la coordination.

Contexte

Le patron intervient dans un contexte du subordination fort de certains composants, en présence d'une chaîne de commandement hiérarchique, comme c'est le cas dans le domaine de l'organisation des secours. De manière générale une ressource est chargée de coordonner les actions des composants engagés pour résoudre une situation de crise. Le composant chargé de cette coordination est un composant de *Comand&Control (C2)* et les composants qui exécutent les actions sont les composants opérationnels. Le C2 fournit des directives qui sont exécutées par les composants opérationnels. Les composants opérationnels fournissent des rapports sur l'avancement de leur tâche. Comme le montre la figure 6.2, la reconfiguration cible le *C2*. Un médiateur est chargé d'expliciter la politique de communication, qui consiste à demander l'autorisation avant la transmission d'un rapport par un composant opérationnel.

Pour gérer la co-évolution, deux mécanismes sont mis en œuvre. Pour pouvoir synchroniser les états internes des composants C2, le patron dote la reconfiguration d'un fort pouvoir d'ingérence vis-à-vis de ces composants par une capacité d'inspection de leurs états internes. Afin de surveiller les messages entre les C2 et les composants opérationnels, une indirection est mise en place dans le système de communications, par exemple par l'implémentation du patron observateur (Gamma et al., 2015).

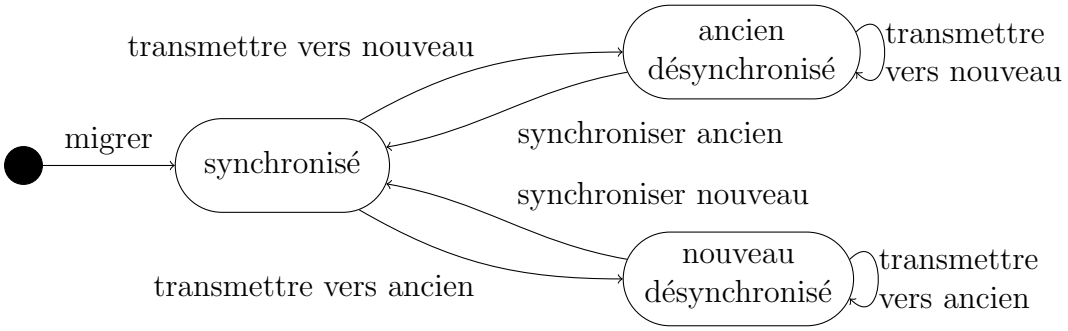

FIGURE 6.3: Grammaire de reconfiguration pour le patron co-évolution.

Problème

Le patron de reconfiguration a pour objectif d'assurer les deux invariants suivants :

- Pendant la reconfiguration, les deux C2 partagent le même état de mission. Ainsi, par exemple, si le CODIS est informé du nombre de victimes pour le sinistre, alors le PC l'est aussi.
- Pendant la reconfiguration, tous les opérateurs sont connectés à exactement un C2. Ainsi, tout composant opérationnel peut communiquer avec son supérieur hiérarchique lorsqu'il en a besoin. Par exemple, tout agent blessé peut se signaler rapidement.

L'intention et le contexte montrent qu'il n'est pas possible d'appliquer de façon atomique la reconfiguration. Pour résoudre cela, la solution doit avoir les forces suivantes :

- instancier une deuxième version du composant qui coexiste avec la version initiale ;
- et synchroniser les états partagés entre les versions du composant.

Solution

La solution repose sur la grammaire de reconfiguration de la figure 6.3, pour synchroniser l'état des deux C2 pendant la reconfiguration. Les deux états peuvent se désynchroniser lorsqu'un message est envoyé à l'un des deux C2. L'autre C2 n'ayant pas connaissance de ce dernier compte-rendu, les deux C2 ne partagent pas le même état concernant leur perception de la mission. Des opérations de reconfiguration doivent être utilisées pour transférer l'état afin de re-synchroniser les deux C2.

Pour ce patron co-évoution, la grammaire de reconfiguration définit donc trois états :

- un état *synchronisé* qui signifie que les deux états des deux C2 représentent les mêmes informations.
- un état *ancien désynchronisé* qui signifie que le C2 remplacé n'est plus synchronisé avec l'état de la nouvelle version du C2. Le nouveau C2 a l'information la plus récente.
- un état *nouveau désynchronisé* qui signifie que le nouveau C2 n'est plus synchronisé avec l'ancien. Cette fois-ci, c'est l'ancien C2 qui a l'information la plus récente.

Les opérations qui composent la grammaire de reconfiguration du patron co-évolution sont :

- une opération *migrer*. Elle est appelée lorsque le nouveau C2 est déployé et démarré. Elle met à jour l'état du nouveau C2 avec celui de l'ancien.
- une opération *transmettre vers nouveau*. Elle est appelée quand un message est envoyé au nouveau C2. Elle consiste à transmettre le message envoyé au nouveau C2.
- une opération *transmettre vers ancien*. Elle est appelée quand un message est envoyé à l'ancien C2. Elle consiste à transmettre le message envoyé vers l'ancien C2.
- une opération *synchroniser nouveau*. Elle consiste à synchroniser l'état du nouveau C2 avec l'ancien.
- une opération *synchroniser ancien*. Elle consiste à synchroniser l'état de l'ancien C2 avec le nouveau.

Des variantes de l'événement qui déclenche les opérations *synchroniser vers nouveau* et *synchroniser vers ancien* sont possibles. Plutôt que d'appeler les opérations dès que le composant entre dans un état de dé-synchronisation, l'opération peut être appelée seulement au moment où le C2 désynchronisé est accédé. Cela correspond à une stratégie paresseuse.

La reconfiguration se termine quand tous les composants opérationnels sont connectés à la nouvelle version du C2. Dans le cas où des composants opérationnels tardent à réaliser l'opération, l'architecte doit prévoir des protocoles pour prendre en compte ce cas, par exemple, en les excluant du SdS.

Conséquences

- Les communications sont maintenues pendant la reconfiguration. Les C2 ont toujours connaissance de l'historique des communications des composants opérationnels. Ils connaissent le contexte du message et ils peuvent donc répondre de façon pertinente.
- La disponibilité des services du C2 est maintenue pendant la reconfiguration.
- La fiabilité est partiellement maintenue. Le patron minimise les erreurs dues à la reconfiguration mais n'empêche pas le cas d'un C2 désynchronisé envoyant des directives.
- Les performances du SdS sont fortement affectées pendant la reconfiguration si des opérations de synchronisation sont réalisées. Cette baisse de performance peut être compensée par des mécanismes de transactions concurrentes.

6.3 Synthèse

L'objectif de ce chapitre était de proposer une manière de documenter la conception des reconfigurations.

Dans notre étude de l'état de l'art en section 2.3, nous avons noté des propositions pour le concept de patron de reconfiguration. La proposition de Oliveira and Barbosa (2015) repose sur la composition de primitives de reconfiguration garantissant des attributs qualités. Comme nous l'avons noté, une telle approche ne répond pas au besoin de réutilisation. Gomaa and Hussein (2004) propose une description moins formelle que Oliveira and Barbosa (2015), et nous avons adopté ce niveau de description semi-formel pour les solutions de reconfiguration. Au regard de cet existant, nous améliorons la réutilisabilité du patron, notamment par la description de l'intention, du contexte et des conséquences. Nous décrivons également quels patrons de reconfiguration ou de conception peuvent être utilisés conjointement au patron décrit pour expliciter les dépendances vis-à-vis des choix de conception de l'architecture reconfigurée.

Le concept patron de reconfiguration proposé favorise les approches de conception systématique. La capacité de composition des patrons de reconfiguration introduit la flexibilité nécessaire à la reconfiguration dans les SdSs. Les patrons étant basés sur des solutions éprouvées et reconnues de l'état de l'art, les reconfigurations conçues à l'aide de ces patrons sont de qualité.

Dans ce chapitre nous avons également définis plusieurs patrons de reconfiguration qui sont le patron *quiescence* et le patron *co-evolution*. Un troisième patron, appelé *re-routage*, est défini dans les annexes de ce mémoire. Dans le chapitre suivant, nous expliquons comment l'architecte peut utiliser ces patrons dans son processus de conception des reconfigurations.

Chapitre 7

Processus de reconfiguration

Dans le chapitre 5, nous avons expliqué le processus de modélisation et le langage utilisé pour définir l'architecture d'un système de systèmes. Appliqué au cas d'étude, nous avons ainsi produit plusieurs configurations correspondant à des évolutions successives du système de systèmes étudié, dans l'esprit du développement évolutionnaire. Dans ce chapitre, nous allons nous intéresser à la conception des reconfigurations dynamiques requises dans un tel scénario. Pour cela, nous nous appuyons sur le cycle de vie d'une reconfiguration dynamique tel qu'il est présenté dans la figure 7.1. Une reconfiguration dynamique peut avoir deux origines. D'une part, la configuration du système de systèmes peut évoluer spontanément, indépendamment du contrôle de l'architecte. Cela peut se produire, par exemple, lorsqu'un système constituant ne souhaite plus participer au système de systèmes. Dans une telle situation, il convient de réévaluer l'architecture afin de déterminer si elle reste réalisable dans le nouvel environnement du système de systèmes. Si ce n'est pas le cas, l'architecte doit faire évoluer l'architecture. D'autre part, l'architecte peut décider de faire évoluer l'architecture comme dans le cas d'étude. Dans les deux cas, une fois que l'architecte a défini une nouvelle architecture, il doit concevoir la reconfiguration dynamique permettant au système de systèmes d'atteindre une configuration correspondant à cette nouvelle architecture. Pour cela, nous verrons dans la section 7.1 que l'architecte commence par définir des contraintes architecturales de transition, qui correspondent au niveau de service, éventuellement dégradé, qu'il accepte pendant la reconfiguration. Puis, nous verrons dans la section 7.2 qu'il conçoit et documente une reconfiguration conforme aux contraintes précédemment spécifiées et comment l'architecte peut utiliser les patrons de reconfiguration. Dans la section 7.3, nous conclurons sur l'apport du processus de reconfiguration proposé.

7.1 Besoin de spécification d'une architecture de transition

Le processus de reconfiguration s'incrit dans un cadre général présenté par la figure 7.1. L'étape 1 concerne l'événement qui déclenche la reconfiguration. C'est soit le SdS (système de systèmes) qui diverge de l'architecture (cas 1.1), soit l'architecte qui fait évoluer l'architecture du SdS à cause de l'évolution des exigences pour le SdS (cas 1.2.1). Spécifier une

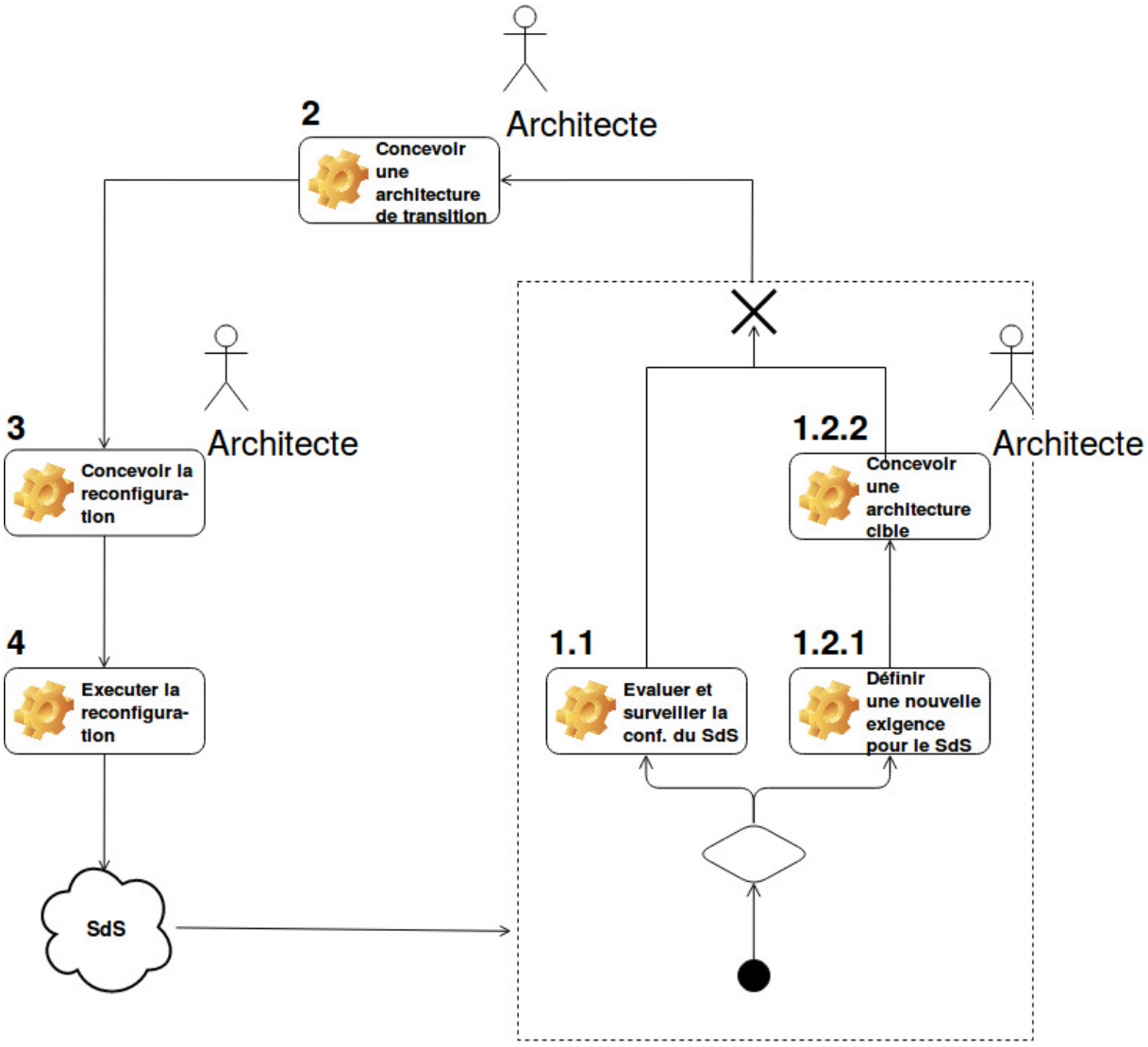

FIGURE 7.1: Contexte global d'une reconfiguration

reconfiguration inclut également plusieurs étapes comme définir l'architecture ciblée (cas 1.2.2), définir une architecture de transition pour la reconfiguration (cas 2) puis définir les opérations de reconfiguration permettant au SdS d'atteindre l'architecture ciblée (cas 3). Enfin, la reconfiguration est soumise à une infrastructure en charge de l'appliquer au système de systèmes (cas 4). Dans des contextes où les changements sont identifiés dès la conception, la description de l'architecture ciblée peut suffire.

Dans notre cas, la spécification de la reconfiguration ne se limite donc pas à décrire l'architecture ciblée et requiert des efforts de conception importants si le changement n'a pas été anticipé. En effet, quelle que soit la cause qui a créé le besoin de reconfiguration, il est peu probable que la reconfiguration puisse être réalisée si l'architecte s'impose de respecter les spécifications du système de systèmes. Si la configuration du système de systèmes a été affectée, par exemple, par le départ non anticipé d'un système constituant, alors, de fait, la spécification n'est pas respectée. Par ailleurs, quand la reconfiguration est à l'initiative de l'architecte, la question se pose de savoir dans quelles conditions se fait le passage vers celle-ci, à partir du moment où la nouvelle spécification est à respecter, et quelles sont les conséquences sur le SdS. Le respect strict de la spécification capturée par l'architecture n'a donc pas de sens. Ne rien dire concernant la phase transitoire, pendant la reconfiguration, certes facilite la conception d'une reconfiguration qui n'a pas à assurer que le système de systèmes continue à fournir des services pendant la reconfiguration. Mais cela questionne le qualificatif dynamique de la reconfiguration, qui pourrait valablement interrompre la fourniture des services du système de systèmes reconfiguré, c'est-à-dire, interrompre le système de systèmes.

En demandant à l'architecte de spécifier le comportement du système de systèmes pendant la reconfiguration, nous avons pour objectif de répondre aux points suivants :

— Dans le cas 1 de la figure 7.1, l'architecte prend acte, le cas échéant, de la configuration du système de systèmes, dans le cas où celle-ci a dévié de son intention. L'architecte réconcilie l'architecture avec la configuration observée.
— Puis, dans le cas 2 et 3 de la figure 7.1, l'architecte spécifie les services que le système de systèmes doit rendre pendant la reconfiguration et, par conséquent, il fournit un cadre aux actions de reconfiguration.

Pour définir l'architecture de transition, on peut s'attendre à ce qu'il s'agisse d'un sous-ensemble des services rendus avant reconfiguration, et de ceux rendus après reconfiguration, avec une qualité inférieure ou égale. Ainsi, l'architecte décrit et documente la dégradation de service qu'il accepte pendant les actions de reconfiguration. La sémantique de recouvrement et celle de restriction/remplacement atomique formalisées par Zhang and Cheng (2005, 2006), bien que poursuivant des objectifs différents, fournissent un cadre possible pour définir la spécification de transition par la restriction du comportement du système de systèmes. Nous adoptons un cadre plus souple, dans lequel l'architecte est totalement libre dans sa spécification de la transition. Nous anticipons que ces contraintes doivent être dynamiques, dans le sens où la dégradation de service acceptée par l'architecte est susceptible de varier au cours de la reconfiguration, par exemple pour exprimer qu'une succession de dégradations de service peut être acceptable même si leur cumul ne l'est pas.

Dans le contexte du cas d'étude, considérons le besoin de déléguer la supervision du champ opérationnel à un chef de groupe, ce qui se traduit par une évolution des exigences du SdS. L'architecture courante de la section 5.2.1 du chapitre 5 n'est pas compatible donc l'architecte décide d'une nouvelle architecture décrite dans la section 5.2.2. Ce cas se place clairement dans la situation où c'est l'architecte qui prend l'initiative de faire évoluer l'architecture, dans le but d'ajouter un niveau de commandement additionnel entre le CODIS et le terrain opérationnel. À ce niveau l'architecte doit spécifier une architecture, celle décrite dans la section 5.2.2, puis il doit définir ensuite une architecture de transition. Celle-ci exprime les contraintes que doit préserver le script de reconfiguration lorsqu'il transforme l'architecture. Par exemple, du fait que le chef d'agrès est en train d'opérer, l'architecte identifie comme contrainte que la communication doit être maintenue en permanence entre le chef d'agrès et son superviseur, initialement l'opérateur CODIS, puis, après reconfiguration, le chef de groupe. De plus, après reconfiguration, le chef de groupe doit connaître les informations au sujet de la mission dont disposait l'opérateur CODIS avant la reconfiguration, éventuellement mises à jour selon les actions que le chef d'agrès a réalisées pendant la reconfiguration.

7.2 Processus de conception d'une reconfiguration

Nous avons identifié le contexte de reconfiguration qui comprend une phase importante, la conception de l'architecture de transition. Cette architecture de transition pose les contraintes, qui peuvent elles-mêmes évoluer pendant la reconfiguration. Dans cette section on verra quelles sont les phases du processus de reconfiguration et sa nature. Nous identifions le type des décisions auxquelles l'architecte doit répondre et verrons comment l'architecte peut utiliser les patrons de reconfiguration pour l'assister dans sa prise de décision.

D'une manière générale, pour expliquer notre processus de conception de reconfiguration, on peut décomposer la reconfiguration en trois phases comme indiqué sur la figure 7.2 :

- Une première *phase de préparation* du système à reconfigurer. Il s'agit de la phase qui modifie le système de manière transitoire, le temps de la reconfiguration, pour permettre la réalisation de celle-ci. La phase de préparation assure que certains constituants du système sont dans un état de quiescence si cela est nécessaire, par exemple, la phase *down* de (Boyer et al., 2013, 2017; Durán and Salaün, 2016), ou modifie la mise en œuvre de constituants (Buisson et al., 2015). Cette phase de préparation peut aussi recruter des constituants additionnels, par exemple chargés d'assurer la continuité de service pendant l'indisponibilité des constituants affectés par la reconfiguration.
- La *phase de modification* réalise les actions souhaitées sur le système comme l'altération de connexions ou la mise à jour d'un composant.
- Si, en phase de préparation, certains constituants sont rendus passifs ou si d'autres ont été recrutés de manière transitoire, il peut être nécessaire de remettre les premiers à l'état actif et de supprimer les seconds de l'architecture. C'est le rôle de la *phase de remise en route* ou de *nettoyage*.

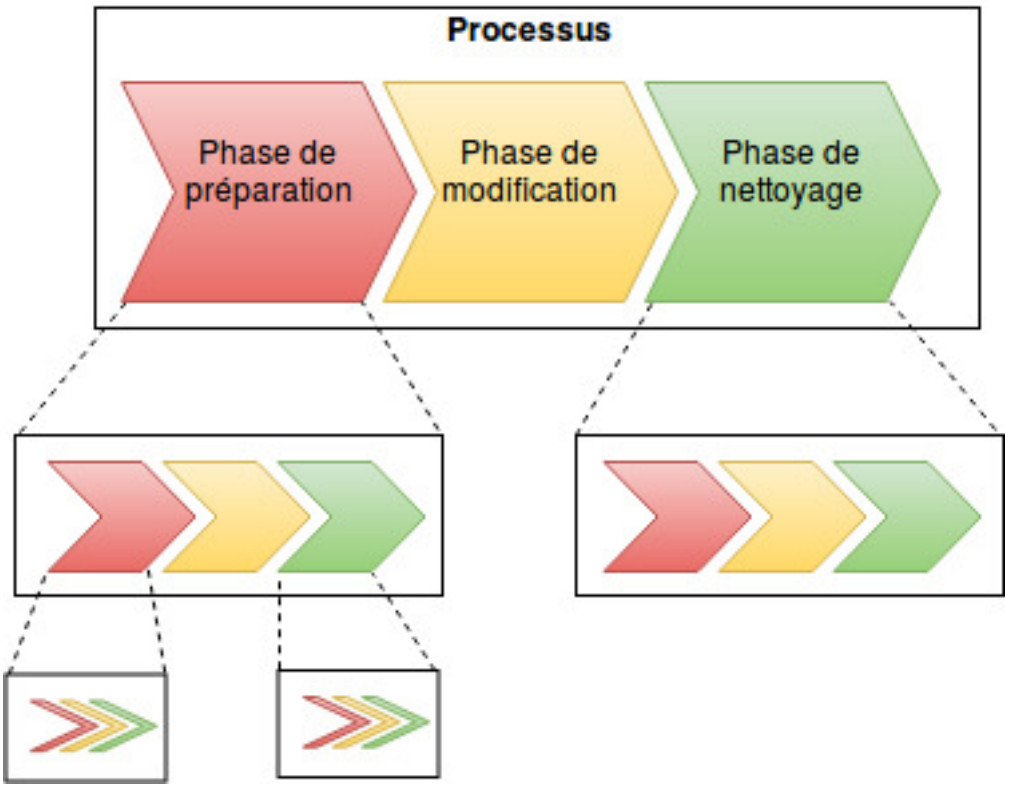

FIGURE 7.2: Phase générique du processus de conception d'une reconfiguration

Dans la mesure où les phases de préparation et de remise en route peuvent elles-mêmes être assimilées à des reconfigurations, le processus de conception proposé est récursif. L'idée générale est la suivante. Étant donnée une reconfiguration à concevoir, l'architecte applique le processus que nous proposons. Il en résulte le besoin de concevoir une phase de préparation et une phase de remise en route, dont les spécifications dérivent de la spécification de la reconfiguration à réaliser ainsi que des préconditions de la phase de modification. Comme le montre la figure 7.2, le processus de conception est alors ré-appliqué de manière récursive pour concevoir les reconfigurations correspondant à ces deux phases de préparation et de remise en route, qui, à leur tour, peuvent chacune nécessiter leurs propres phases de préparation et de remise en route, et ainsi de suite. La récursion s'arrête lorsque la reconfiguration à concevoir est suffisamment simple pour ne nécessiter ni préparation ni remise en route.

À chaque fois que le processus de conception est ré-appliqué, il est constitué des étapes suivantes :

1. Analyse : Identification des modifications à apporter
   Cette étape permet d'identifier les constituants que l'architecte souhaite recruter pour le système de systèmes, et ceux qu'il ne souhaite plus utiliser. L'architecte identifie également les modifications apportées aux liaisons entre les constituants. L'analyse consiste à identifier les différences entre l'architecture source et l'architecture à obtenir.

2. Décision : Choix d'une stratégie de reconfiguration

   Lorsque l'architecte a identifié les modifications à apporter à l'architecture, il confronte celles-ci à son catalogue de patrons de reconfiguration. En étudiant les propriétés préservées (ou non) par chaque patron au regard de la spécification de transition, l'architecte sélectionne et combine un ou plusieurs patrons, dont les solutions lui permettent d'élaborer la reconfiguration. Si aucun patron ne correspond aux modifications à réaliser, l'architecte peut également élaborer une reconfiguration originale.

3. Récursion : Spécification des phases préparatoires et de remise en route

   Le contexte des patrons de reconfiguration indique les éventuelles hypothèses faites par la solution de ces patrons et les préconditions à satisfaire. Si ces hypothèses et préconditions ne sont pas satisfaites dans l'architecture initiale, il convient d'introduire une phase de préparation, chargée d'établir ces hypothèses et préconditions préalablement à l'application des solutions des patrons sélectionnés à l'étape précédente. Une phase de remise en route correspondante annule les modifications transitoires réalisées par la phase de préparation. Les phases de préparation et de remise en route sont des reconfigurations. Sur la base des spécifications produites à l'étape précédentes, l'architecte réapplique le processus de conception pour les concevoir.

Pour rendre le processus plus concret, considérons le cas du déploiement d'un chef de groupe. Cet exemple sera plus largement détaillé dans le chapitre 9. Pour cette reconfiguration qu'on appellera Q, voici les phases simplifiées du processus de conception :

Q.1 Analyse : Identification des modifications à apporter

La figure 7.3 représente l'architecture ciblée. En plus du recrutement d'un chef de groupe, un canal tactique est créé et la liaison du chef d'agrès est modifiée pour connecter ce dernier au canal tactique nouvellement créé, à la place du canal opérationnel.

Q.2 Décision : Choix d'une stratégie de reconfiguration

Le déploiement du chef de groupe ne pose pas de difficulté particulière.

En revanche, la mise en place du canal tactique avec la modification des liaisons du chef d'agrès est plus difficile. La bascule de la liaison doit en effet assurer la continuité du suivi de l'opération réalisée par le chef d'agrès. À la fin de la reconfiguration, le chef de groupe doit être dans un état tel qu'il dispose des mêmes informations que l'opérateur CODIS au sujet de l'opération en cours, et les mises à jour pour prendre en compte les actions réalisées par le chef d'agrès pendant la reconfiguration. Comme il n'est pas possible de suspendre l'activité du chef d'agrès le temps de transférer l'état de l'opérateur CODIS vers le chef de groupe, une solution plus sophistiquée doit être mise en œuvre.

Une possibilité consiste à utiliser le patron *coévolution*, décrit dans la section 6.2.2 et qui résout ce problème.

Q.3 Récursion : Spécification des phases préparatoires et de remise en route

Une fois la stratégie de reconfiguration établie, l'architecte doit définir les mécanismes nécessaires à sa mise en œuvre. Comme indiqué dans le patron *coévolution*, il faut :

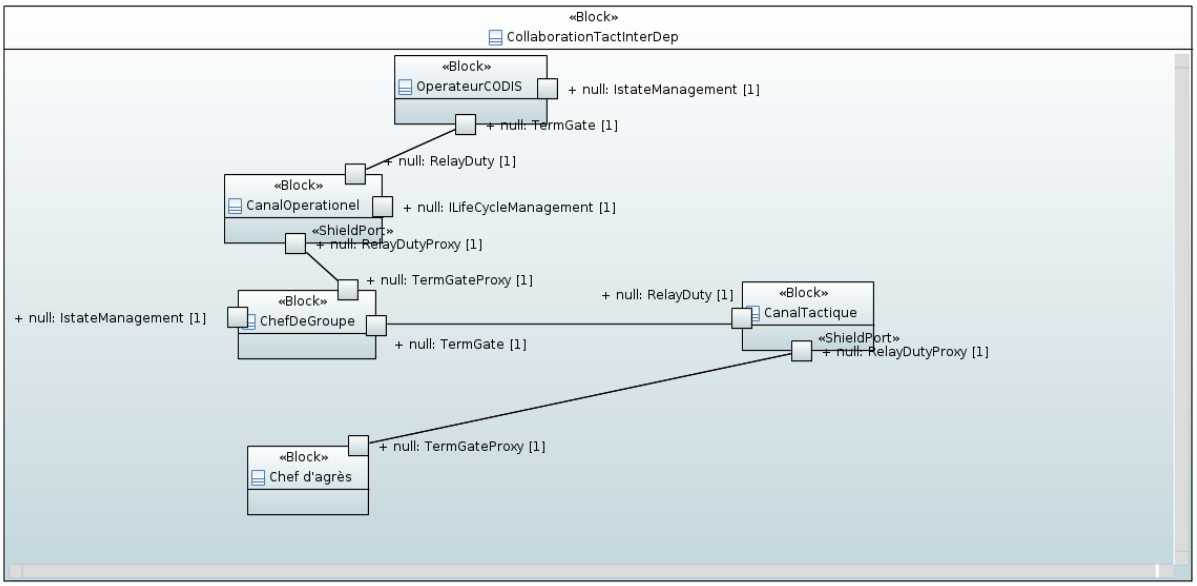

FIGURE 7.3: Architecture ciblée

— transférer l'état de la mission stocké par l'opérateur CODIS vers le chef de groupe ;
— surveiller les communications du chef d'agrès pour mettre à jour les informations du chef de groupe ;
— suspendre les communications du chef d'agrès ;
— détecter quand la reconfiguration est terminée.

Or l'architecture du système de systèmes ne permet pas de faire cela. Il convient donc d'introduire une phase de préparation et la phase de remise en route correspondante. La figure 7.4 montre les deux constituants qui doivent être déployés dans le système de systèmes par la phase de préparation, tel que préconisé par le patron *coévolution* :

— Le canal de communication opérationnel doit être remplacé par une variante dotée d'un tampon afin de retarder de manière transitoire certains messages.
— Un composant doit être inséré afin de réaliser la bonne synchronisation des informations détenues par l'opérateur CODIS et celles connues par le chef de groupe.

Cette architecture correspond donc à l'architecture cible de la phase de préparation. Dans la mesure où des constituants ont été insérés dans l'architecture pour le besoin de la reconfiguration, une phase de remise en route est également requise afin de supprimer ceux-ci. Le processus de concepion est réappliqué aux deux phases de préparation et de remise en route.

7.3 Synthèse

Dans ce chapitre, nous expliquons que la conception d'une reconfiguration requiert la définition de contraintes par l'architecte pour la reconfiguration. Ces contraintes évoluent

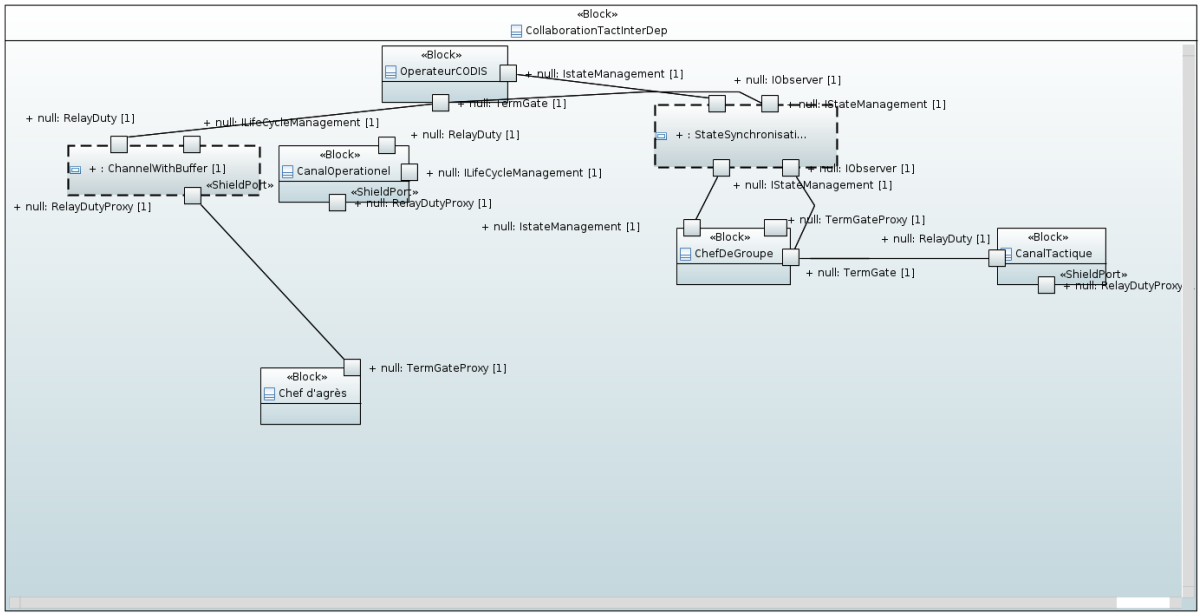

FIGURE 7.4: Architecture pour la phase de préparation à la reconfiguration

pendant la reconfiguration et elles sont conceptualisées par une architecture de transition. Cette vision de la reconfiguration nous a mené à l'élaboration d'un processus de reconfiguration récursif. Ce processus se compose de trois phases (préparation, modification et nettoyage), la phase de préparation et nettoyage peuvent se décomposer en trois phases également.

Le processus de reconfiguration dans des approches automatisées se fait seulement par la description de l'architecture ciblée. Dans le cas de reconfiguration non anticipée comme avec les SdSs, ce genre d'approche n'est pas adaptée, car les contraintes de reconfiguration sont également amenées à évoluer. Sans aller jusqu'à proposer une architecture de transition, des travaux comme ceux de Zhang and Cheng (2005, 2006) énumèrent quelques comportements possibles pendant la reconfiguration. Par rapport aux travaux existants nous avons donc intégré le fait que l'architecte base sa reconfiguration par rapport à une architecture de transition qui fait évoluer des contraintes à respecter pendant la reconfiguration. Ces contraintes nous ont permis d'intégrer des aspects de qualité des services fournis par le système pendant la reconfiguration. Par exemple, dans le cas du patron co-évolution présenté en section 6.2.2, la section conséquences indique que la fonctionnalité de C2 est maintenue sans interruption de service, et précise que la fiabilité est partiellement dégradée. Les contraintes nous ont permis de décrire cela.

Le processus développé est adapté à l'exploitation des patrons de reconfiguration. Ceux-ci interviennent à plusieurs niveaux dans le processus de conception. Dans un premier temps, les patrons assistent l'architecte dans la description de l'architecture de transition en fournissant des descriptions précises des contraintes à intégrer pour garantir des propriétés pendant la reconfiguration. D'autre part, les patrons assistent l'architecte en décrivant comment la reconfiguration affecte le système. Cela aide l'architecte à maîtriser

la dégradation pendant la reconfiguration. Pour finir, le processus que nous avons proposé permet de guider l'architecte dans la conception des scripts de reconfiguration en structurant ses activités et maîtrisant les effets de la reconfiguration sur le SdS reconfiguré.

Troisième partie

Framework d'expérimentation et résultats

Chapitre 8

Framework de reconfiguration

Ce chapitre documente le framework Java qui nous sert à évaluer la faisabilité d'une approche par patron de reconfiguration. Ce framework permet de simuler le déploiement d'un système de systèmes et sa reconfiguration. En plus des mécanismes de reconfiguration, le framework instrumente l'architecture déployée afin d'évaluer les contraintes architecturales pendant la reconfiguration et l'exécution du système de systèmes.

La section 8.1 explique les principales abstractions, les décisions de conception et l'avantage pour la reconfiguration dynamique. La section 8.2 présente comment sont spécifiées les reconfigurations. La section 8.3 présente la partie du framework qui montre comment sont vérifiées les propriétés de reconfiguration. Y sont expliqués le type des contraintes supportées, et le résultat pour l'architecte qui veut évaluer les contraintes qu'il a définies. Pour finir la section 8.4 explique comment l'architecte peut interagir avec le framework pour déployer ses reconfigurations et tester si les propriétés sont bien respectées.

8.1 Architecture du framework

Le framework adopte le style architectural composant/connecteur. Comme expliqué par Taylor et al. (2009), ce style architectual est bien adapté à la reconfiguration dynamique : d'une part, le concept de composant explicite les parties du système susceptibles d'être modifiées ; et, d'autre part, le concept de connecteur impose de découpler les composants de manière à en contrôler les interactions. Les connecteurs sont également le lieu pour l'interception des messages par le système de systèmes, un des instruments de contrôle des interactions entre composants.

Le framework s'inspire largement de Fractal (Bruneton et al., 2006), ainsi que de SosADL (Oquendo, 2017), un langage de description spécifiquement conçu pour les systèmes de systèmes.

Adoptant la terminologie de SosADL, les composants sont dénomés CS (système constituant). Ils modélisent les ressources qui implémentent les parties fonctionnelles du SdS (système de systèmes). Des interfaces (au sens Java du terme) décrivent les fonctions

implémentées par les composants, de sorte que le code qui implémente ces fonctions est complètement encapsulé. Le framework conserve la distinction entre interfaces requises, signifiant que l'implémentation du composant dépend d'autres composants, et les interfaces fournies, qui indiquent les fonctions que le composant expose à son environnement. Pour injecter les dépendances, le framework suppose que l'implémentation du composant définit un accesseur pour chaque interface requise (une méthode dont le nom débute par `set`, par exemple la méthode `setEmitterReceptor` dans le listing de la figure 8.1), ainsi qu'un champ mémorisant la référence vers l'objet du connecteur (le champ `dRelay` dans cet exemple). La classe qui correspond à l'implémentation d'un composant implémente l'ensemble des interfaces fournies (dans la figure 8.1, l'interfaces `IStateManagement`).

La figure 8.1, utilisée pour illustrer ce modèle de programmation des composants, donne le code source Java du composant opérateur de la coalition formée par le CODIS56. L'interface `IEmitterReceptorModificator`, non mentionnée jusqu'à présent, déclare l'accesseur `setEmitterReceptor`.

Dans la terminologie SosADL, les connecteurs sont appelés *médiateurs*. Ils modélisent les parties communication entre les CSs du SdS. Nous utilisons le même modèle de programmation que pour les CSs, excepté que la classe spécialise `Mediator` plutôt que `CS`.

Pour s'adapter au contexte des systèmes de systèmes, on considère que les CSs sont des systèmes disposant de leur indépendance managériale et opérationnelle, alors que les médiateurs sont placés sous l'autorité du système de systèmes.

Les configurations sont appelées des *coalitions*. Elles modélisent la collaboration d'un ensemble de CSs pour fournir une fonctionnalité. Comme le montre la figure 8.2, une coalition est constituée de CSs et de médiateurs. La coalition contient également une contrainte encodant le style architectural souhaité par l'architecte et, éventuellement, d'autres coalitions. La figure 8.3 montre l'interface listant les opérations d'introspection et d'intercession d'une coalition, inspirées du contrôleur de contenu de Fractal. La méthode `accept` prend en paramètre des opérations de reconfiguration, qui sont appelées sur tous les composants de la coalition. Toutes ces capacités sont utilisées pour les opérations de reconfiguration décrites dans la section suivante.

Dans la terminologie SosADL, les *gates* et *duties* désignent les ports des CSs et médiateurs, respectivement. Le framework utilise l'implémentation des gates et duties comme objet d'indirection afin de surveiller les envois de message entre les composants pendant l'exécution. La figure 8.4 montre les principales abstractions qui participent à la connexion entre un CS et un médiateur, c'est-à-dire les objets qui réifient le gate et le duty, avec les références Java entre les objets.

8.2 Programmation des reconfigurations

L'architecte dispose d'un ensemble d'opérations de reconfiguration qui lui permettent de reconfigurer l'architecture du SdS. Les opérations de reconfiguration sont catégorisées

```
public class ImplOperator extends CS
    implements IStateManagement, IEmitterReceptorModificator {

    private IEmitterReceptor dRelay;

    public ImplOperator(DradioEmitterReceptor dCanalOp) {
        this.dRelay=dCanalOp;
    }

    @Override public void run() {
        String msgRecu = dRelay.reception();
        // ...
    }

    public void setEmitterReceptor(IEmitterReceptor dOpCodis) {
        this.dRelay=dOpCodis;
    }

    public IEmitterReceptor getEmitterReceptor(){
        return this.dRelay;
    }

    @Override public void importState(String state) {
        // ...
    }

    @Override public String exportState() {
        // ...
    }

    @Override public
        void setEmitterReceptor(IEmitterReceptor provider) {
        // ...
    }
}
```

FIGURE 8.1: Déscription d'un CS

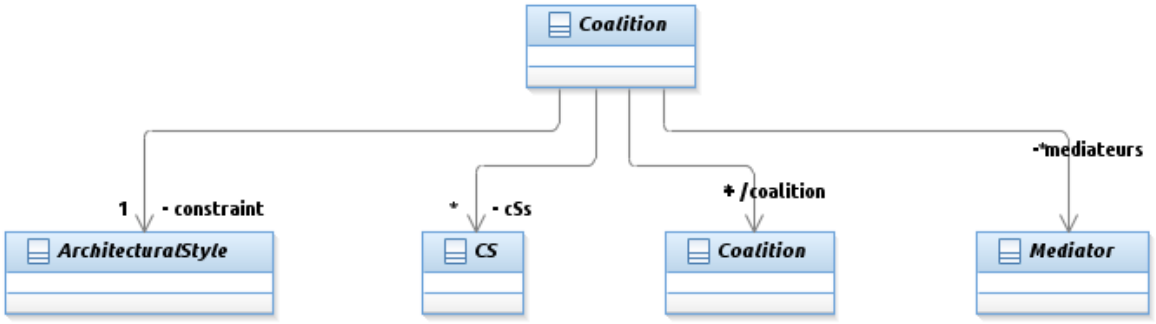

FIGURE 8.2: Structure d'une coalition

```
interface ContentController {
    public void addFcSubComponent(Mediator subComponent);
    public void addFcSubComponent(CS subComponent);
    public void addFcSubComponent(Coalition subComponent);
    public void removeFcSubComponent(Mediator subComponent);
    public void removeFcSubComponent(CS subComponent);
    public void removeFcSubComponent(Coalition subComponent);
    public CS[] getCS();
    public Mediator[] getMediator();
    public Coalition[] getCoalition();
    public void accept(Operation reconf);
}
```

FIGURE 8.3: Interface d'introspection implémentée par les coalitions

FIGURE 8.4: Structure des connexions

en opérations primitives et composites. La figure 8.5 fait la synthèse de la taxonomie des opérations fournies par le framework.

Les opérations primitives sont :
- l'opération de recrutement (`Recruitment`) qui ajoute un composant dans une coalition. Le type des composants sont des CSs, des médiateurs et coalitions.
- l'opération de désengagement (`Unrecruitment`) libère les composants intégrés par l'opération de recrutement.
- l'opération de connexion (`Binding`) connecte les composants d'une coalition.
- l'opération de déconnexion (`Unbinding`) réalise l'opération inverse de la précédente.

Les opérations composites servent à l'orchestration des opérations de reconfiguration primitives. Les opérations composites sont :
- la jonction (`Join`) attend la terminaison d'un ensemble d'opérations.
- l'embranchement (`Fork`) exécute les opérations de reconfiguration de manière concurrente.
- le séquencement (`Sequence`) exécute séquentiellement plusieurs opérations de reconfiguration.

Les opérations composites sont définies comme des classes de premier ordre pour expliciter les structures d'orchestration et donc faciliter la compréhension du script. La définition d'une hiérarchie de classes primitives et composites facilite la conception de façon non anticipée de scripts de reconfiguration complexes, par spécialisation des classes. La figure 8.6 illustre une spécialisation de l'opération `Binding` de connexion spécifiquement pour la connexion d'une interface `IEmitterRecepter`, dans ce cas pour restreindre l'opération à un contexte spécifique.

La figure 8.7 montre la construction d'un script de reconfiguration. A la ligne 1, une collaboration opérationnelle `operCollab` est instanciée. Il s'agit de la coalition destinée à recruter les composants `codis56`, `operator` et `canalOp` instanciés aux lignes 2 à 4. La raison pour laquelle les composants sont instanciés à l'extérieur de la reconfiguration est l'indépendance managériale : les CSs sont gérés et déployés par l'autorité de gestion du constituant, indépendamment du système de systèmes. C'est ce que simulent ces lignes 1 à 4. Puis, la construction de la reconfiguration débute à la ligne 6 : il s'agit d'un script exécutant des opérations séquentiellement. A la ligne 8, la reconfiguration demande à la coalition `operCollab` de recruter le composant `codis56`. La reconfiguration demande ensuite à ce composant `codis56` de recruter à son tour, lignes 9 et 10, les composants `canalOp` et `operator`. Enfin, à la ligne 12, la reconfiguration demande la connexion de `canalOp` et `operator`. La ligne 15 lance l'exécution de la reconfiguration ainsi construite.

8.3 Vérification des propriétés de reconfiguration

Les contraintes évaluées par le framework peuvent être structurelles ou comportementales. Dans notre implémentation, c'est un gestionnaire de modèle qui embarque ces

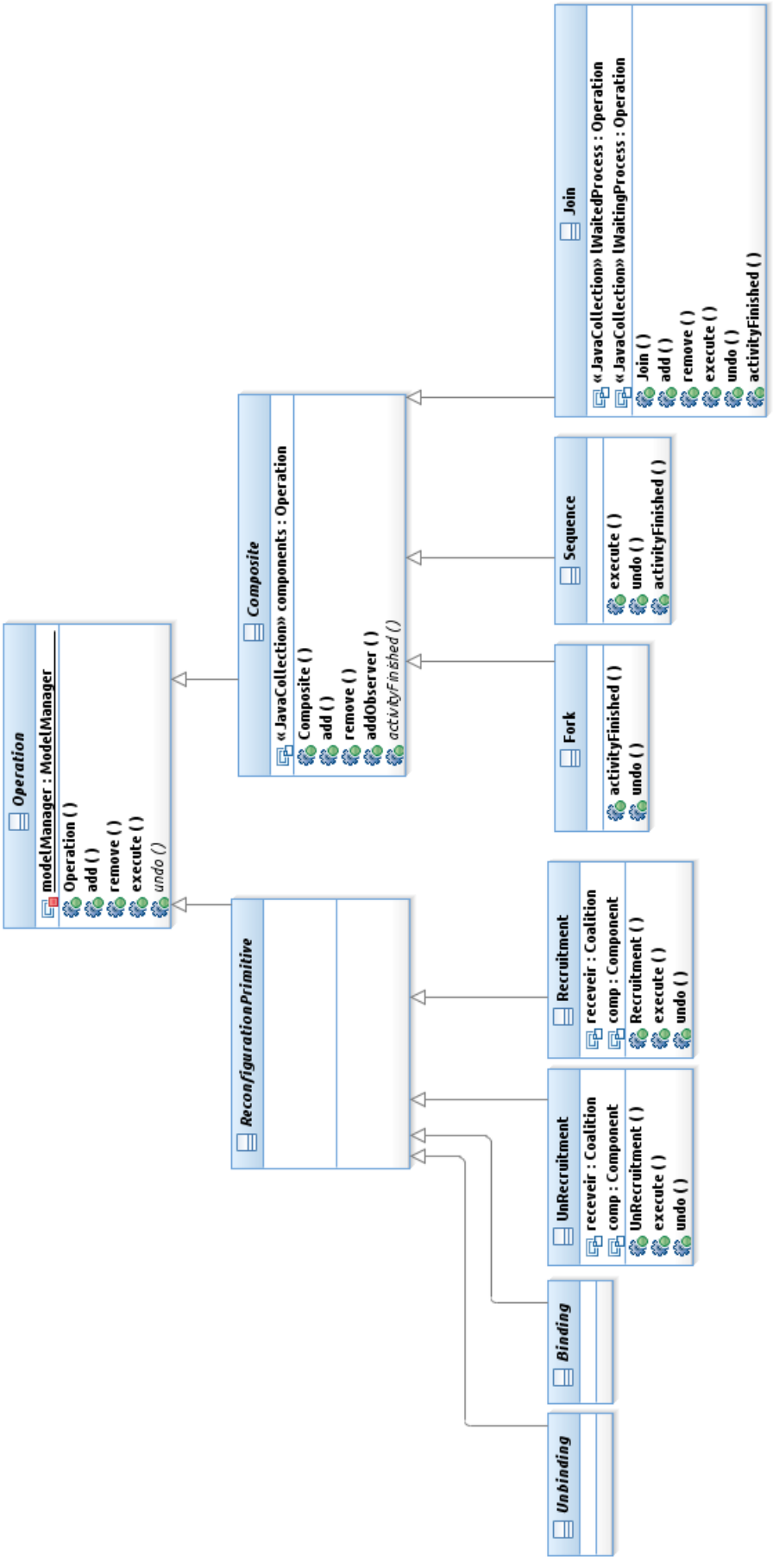

FIGURE 8.5: Opérations de reconfiguration de base

```
public class BindingEmitterReceptorProxy extends Binding {
    private IEmitterReceptorProxy provider;
    private IEmitterReceptorModificatorProxy requirer;

    public BindingEmitterReceptorProxy(
                IEmitterReceptorProxy provider,
                IEmitterReceptorModificatorProxy requirer) {
        this.provider=provider;
        this.requirer=requirer;
    }

    @Override public void execute() {
        requirer.setEmitterReceptor(provider);
        super.execute();
    }

    @Override public void undo() {
    }
}
```

FIGURE 8.6: Exemple d'opération de connexion d'une interface

```
1  operCollab = new OperationalCollaboration();
2  codis56 = new Codis56();
3  operator = new ImplOperator();
4  canalOp = new ImplOperationalCanal();
5
6  Operation op = new Sequence();
7
8  op.add(new Recrutment(operCollab, codis56));
9  op.add(new Recrutment(codis56, canalOp));
10 op.add(new Recrutment(codis56, operator));
11
12 op.add(new BindingCodisER(new RelayDuty(canalOp),
13                           new TermGate(operator)));
14
15 op.run();
```

FIGURE 8.7: Exemple de script de reconfiguration

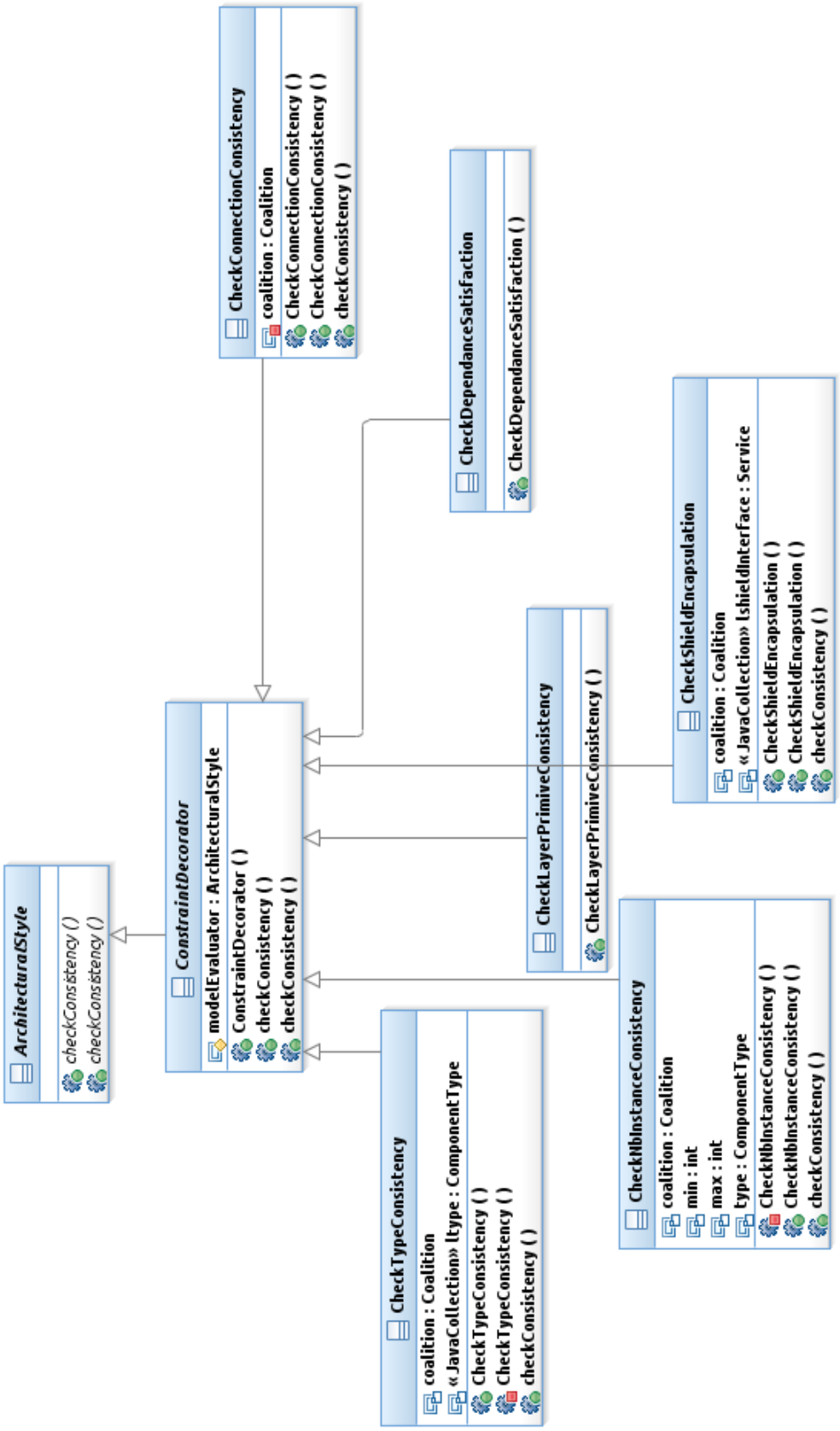

FIGURE 8.8: Contraintes architecturales du framework

contraintes. De plus, l'ensemble de contraintes est dynamiquement modifiable pour permettre la prise en compte des contraintes de l'architecture de transition, présentées en section 7.1.

Selon les besoins, l'architecte peut paramétrer la politique de vérification et faire évoluer dynamiquement les contraintes que doit satisfaire la configuration. Comme déjà indiqué, c'est la coalition qui contient les contraintes.

Pour les besoins de l'évaluation des contraintes, les primitives de reconfiguration embarquent un mécanisme de notification. Lorsqu'il est notifié à la fin de chaque opération primitive, le gestionnaire de modèle réévalue les contraintes pour déterminer si elles sont bien toujours satisfaites. Pour ce faire, le manager de modèle parcourt récursivement les coalitions présentes dans le modèle.

La figure 8.8 liste les contraintes que nous avons définies :
- `CheckTypeConsistency` vérifie le type des composants.
- `CheckNbInstanceConsistency` vérifie que le nombre des instances est conforme à ce qui est dans la coalition.
- `CheckShieldEncapsulation` vérifie que la visibilité des champs déclarées par les interfaces est bien limitée à une coalition.
- `CheckLayerPrimiveConsistency` vérifie que les connexions entre les composants respectent le patron architectural en couches.

8.4 Interaction de l'architecte avec le framework

Le framework de reconfiguration repose sur un ensemble d'activités et d'outils.

La première activité consiste à modéliser l'architecture du SdS. Pour cela l'architecte peut utiliser *Papyrus* et développer les modèles architecturaux conformément au framework de modélisation défini dans le chapitre 5.

Puis les modèles architecturaux sont transformés vers une représentation compatible avec le framework de reconfiguration. Cette étape est manuelle mais facilement automatisable.

L'architecture se concentre ensuite sur la conception du script de reconfiguration en utilisant le processus décrit au chapitre 7. Cette étape peut inclure la définition de primitives de reconfiguration, de mécanismes d'évolution et de scénarios de test.

L'architecte configure le framework, c'est-à-dire, il fixe les paramètres de la politique de vérification des contraintes de reconfiguration. Il fournit également le script de reconfiguration et des scénarios de test.

L'activité suivante consiste à exécuter la reconfiguration.

Pour finir, l'architecte consulte le résultat de la reconfiguration. Le framework affiche sur la sortie standard quelles sont les contraintes qui ont été violées pendant l'exécution de la reconfiguration.

8.5 Synthèse

Ce chapitre a présenté le framework de reconfiguration que nous avons développé dans le but d'évaluer la faisabilité de notre approche par patrons de reconfiguration.

Le framework outille l'architecte pour décrire un script de reconfiguration. Une fois le script de reconfiguration décidé, il peut simuler l'exécution d'un SdS ainsi que le déploiement de son script de reconfiguration.

Nous avons vu en particulier le choix d'une description composant/connecteur. Du point de vue de la reconfiguration dynamique, cela permet de capturer les parties modulaires du SdS et de contrôler leurs points d'interaction avec les autres parties du SdS. La mise en œuvre des opérations de reconfiguration est aussi facilitée. Le framework permet une description des concepts inhérents au domaine des SdSs qui sont les coalitions et les médiateurs.

L'architecte de la reconfiguration peut définir une reconfiguration par un ensemble de primitives de reconfiguration. Il est possible de simuler plusieurs types d'exécution des opérations de reconfiguration : séquence, parallèle et point de rendez-vous. Cela permet de prendre en compte différents types de contrainte dans l'application des opérations de reconfiguration. Par exemple, lorsque des opérations de reconfiguration ne peuvent pas être anticipées.

Le framework définit un ensemble de primitives architecturales qui permet à l'architecte de vérifier des propriétés sur l'architecture pendant la reconfiguration. L'architecte peut décrire les contraintes avant l'exécution de la simulation et les faire évoluer pendant son exécution. Nous pouvons ainsi prendre en compte le fait que l'architecte peut ajouter ou supprimer des contraintes pendant l'application de façon momentanée pendant l'exécution de la reconfiguration.

Pour finir nous avons vu que l'architecte est assisté par l'outil Papyrus pour modéliser l'architecture du SdS. Une fois les architectures source et cible définies, elles sont générées dans le framework de reconfiguration. L'architecte doit ensuite définir des scénarios de test qui consistent à décrire le comportement des médiateurs et des CSs. Puis, il définit le script souhaité. Une fois l'exécution terminée, l'architecte récupère un résultat qui indique si les contraintes ont été respectées.

Chapitre 9

Expérimentation

Une des contributions de cette thèse est le concept de patron de reconfiguration et comment il peut être mis en œuvre. L'objet de l'expérimentation est donc l'approche par patron de reconfiguration. Il s'agit d'évaluer la faisabilité de cette approche pour reconfigurer une architecture de SdS (système de systèmes). Les objets contextuels sont l'étude de cas présentée dans le chapitre 4 et les configurations architecturales produites par le processus de modélisation du chapitre 5. Dans la section 9.1, nous présentons le plan de l'expérimentation. Il consiste à donner le détail de la sélection du contexte en présentant ses principales caractéristiques à savoir l'environnement de l'expérimentation, le sujet, le problème de reconfiguration adressé et sa généralisation possible à d'autres domaines. Ensuite, nous présentons les hypothèses formulées dans l'expérimentation. Puis, nous voyons la selection des variables de l'expérimentation. La section décrit également la conception de l'expérimentation et montre l'instrumentation. La section 9.2 présente l'exécution de l'expérimentation. Pour finir la section 9.3 fournit l'interprétation et l'analyse des résultats.

9.1 Planification de l'expérimentation

9.1.1 Sélection du contexte

L'expérimentation est réalisée hors-ligne. Ce choix nous permet de contrôler l'environnement de l'expérimentation. D'autre part, il s'impose du fait de l'absence d'accès aux infrastructures des SdSs de service d'urgence. Le sujet de l'expérimentation est le doctorant qui a réalisé cette thèse.

Le problème de reconfiguration proposé repose sur la projection d'un problème de reconfiguration étudié dans l'état de l'art sur une étude de cas réelle. Le cas étudié est celui de l'organisation des services de secours. Il met en avant des évolutions qui surviennent lorsque un SDIS (Service Départemental d'Incendie et de Secours) requiert des effectifs qu'il n'a pas. Le problème de reconfiguration est celui de remplacer un composant de commandement supervisant un ensemble de composants opérationnels. Ce problème est bien un problème réel, mais il implique des simplifications. Ces simplifications nous permettent de réaliser des expérimentations en vue d'une première évaluation des patrons de

reconfiguration.

Le contexte de l'expérimentation est réalisé dans un contexte spécifique aux SdSs de service de secours. Cependant les problèmes de reconfiguration abordés ne sont pas spécifiques aux domaines et sont généralisables à d'autres domaines de SdS et d'autres types de système.

9.1.2 Formulation des hypothèses

Nous voulons démontrer l'hypothèse générale que les patrons de reconfiguration assistent l'architecte. Pour cela on propose de décomposer cette hypothèse en plusieurs sous hypothèses :

H1 : l'exécution de la reconfiguration ne viole pas de règle de reconfiguration.
H2 : la conception des reconfigurations est facilitée par les patrons de reconfiguration.
H3 : l'exécution de la reconfiguration termine la reconfiguration

L'hypothèse H1 s'intéresse à l'utilité des patrons de reconfiguration pour définir les objectifs de la reconfiguration et l'identification des problèmes de reconfiguration. L'hypothèse H2 s'intéresse à l'utilité des patrons de reconfiguration pour identifier des solutions possibles et sélectionner une solution. L'hypothèse H3 s'intéresse à l'utilité des patrons de reconfiguration pour aider l'architecte à mettre en place les principes d'ingénierie nécessaires à la réalisation de la reconfiguration.

9.1.3 Sélection des variables

Nous distinguons deux types de variables de l'expérimentation : les dépendantes et les indépendantes.

Les variables indépendantes

— Le framework de reconfiguration qui permet à l'architecte de mettre en œuvre des opérations de reconfiguration de base et fournit des composants
— le catalogue de reconfiguration qui est composé de quatre patrons de reconfiguration : tranquillité, quiescence, co-evolution et re-routage.

Les variables dépendantes

— Le niveau d'assistance des patrons de reconfiguration. La variable prend la valeur *bon* quand l'architecte estime que les patrons lui ont fourni une aide sinon *mauvais*.
— Les contraintes de reconfiguration qui sont la disponibilité des parties reconfigurées, la cohérence d'état de la partie de l'architecture, la compatibilité de l'architecture reconfigurée avec l'architecture cible.
— La terminaison de la reconfiguration.

9.1.4 Choix de conception du type d'expérimentation

L'expérimentation consiste à fournir à l'architecte une architecture cible, source et un catalogue de reconfigurations. Sur la base du processus de reconfiguration standardisé, l'architecte indique les itérations réalisées en expliquant l'architecture ciblée par l'itération et si c'est le cas la raison d'une nouvelle itération. À la fin de la conception de la reconfiguration l'architecte indique si les patrons ont été utiles où non. Après exécution de la reconfiguration le résultat d'exécution est consulté pour déterminer si la reconfiguration s'est terminée et si elle a violé des contraintes de reconfiguration.

9.1.5 Instrumentation

Les instruments sont les consignes données au sujet et les mesures réalisées. Les consignes sont essentiellement de suivre le processus de reconfiguration expliqué dans le chapitre 7. Les mesures sont le résultat de l'évaluation des contraintes de reconfiguration. Ces évaluations sont fournies par le framework de reconfiguration.

9.2 Exploitation de l'expérimentation

9.2.1 Préparation

Architecture source et cible

La figure 9.1 montre l'architecture courante du SdS et la figure 9.2 montre l'architecture ciblée par l'architecte. La reconfiguration consiste à déléguer le suivi de la mission au chef de groupe. Le chef de groupe supervise le commandement de l'intervention pour l'inondation. Le canal tactique permet de relayer les communications entre le chef de groupe et le chef d'équipe.

Architecture de transition

L'architecture de transition vérifie que les propriétés de la reconfiguration sont respectées. Nous avons vu que les propriétés de la reconfiguration sont que le chef d'équipe doit être continuellement supervisé et que l'état de la mission doit rester cohérent ; pour cela les messages émis doivent tous être reçus dans l'ordre d'émission. De plus on vérifiera qu'à la fin de la reconfiguration le SdS est bien conforme au style architectural ciblé.

Comme expliqué dans l'introduction, la reconfiguration doit préserver la disponibilité du service qui supervise le chef d'équipe et la cohérence de l'état de la mission. Dans ce cas, le patron de reconfiguration assiste l'architecte en décrivant quel type de propriété il peut modéliser. Ici, il s'agit d'un invariant structurel et comportemental. La propriété structurelle est que, pendant la reconfiguration, l'opérateur est toujours connecté au canal opérationnel et le chef de groupe est toujours connecté au canal tactique. La propriété comportementale est que l'état de la mission est systématiquement synchronisé entre l'opérateur et le chef de groupe. Dans le contexte cette propriété peut être vérifiée à la fin de la reconfiguration. En effet, la reconfiguration est terminée quand le chef d'équipe

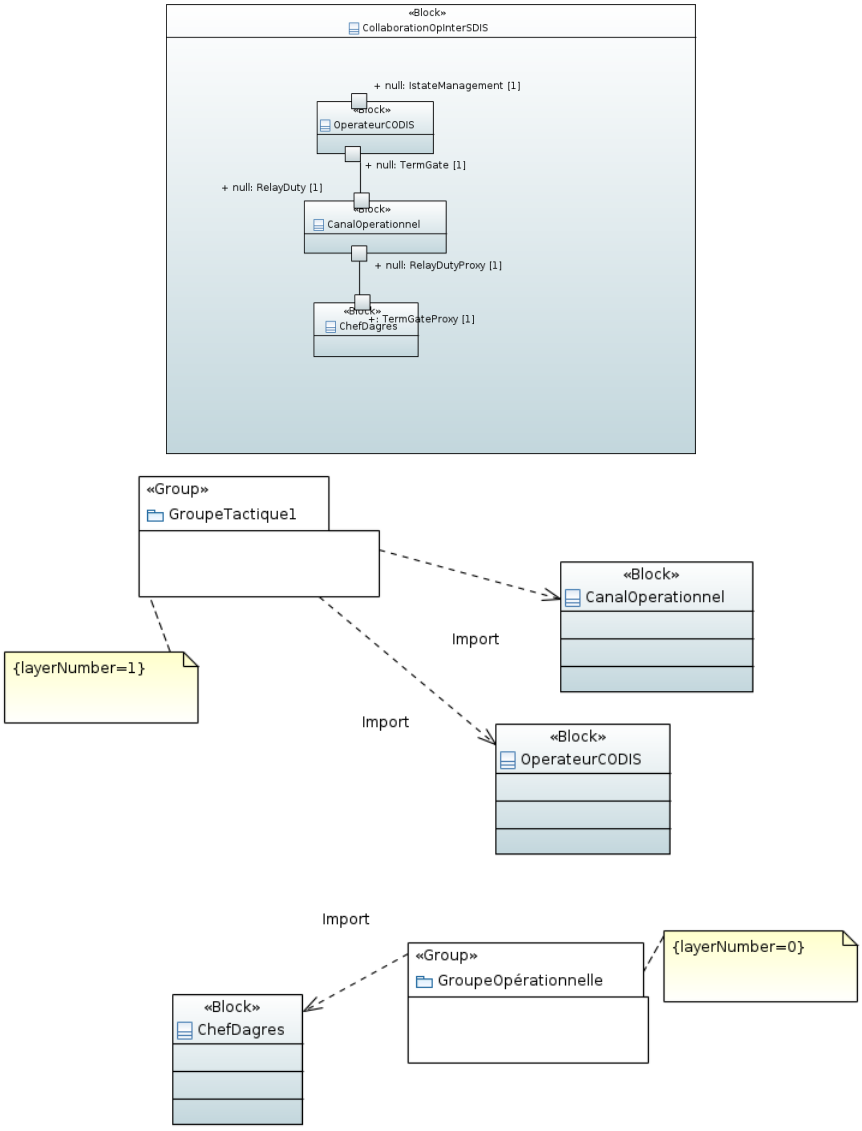

FIGURE 9.1: Architecture source

change de canal de communication. Dans le cas où plusieurs chefs d'équipe étaient impliqués, la vérification aurait dû se faire après chaque échange de messages entre un chef de groupe et l'opérateur.

En résultat, l'architecte définit un style architectural qui capture la propriété structurelle précédente. Elle sera donc évaluée à chaque application d'une opération de reconfiguration. Une contrainte comportementale sera aussi définie et appelée à la fin du script de reconfiguration.

Script de test

Un script de test embarque les contraintes mentionnées dans le style architectural de transition. Ces contraintes sont vérifiées à chaque opération de reconfiguration effectuée.

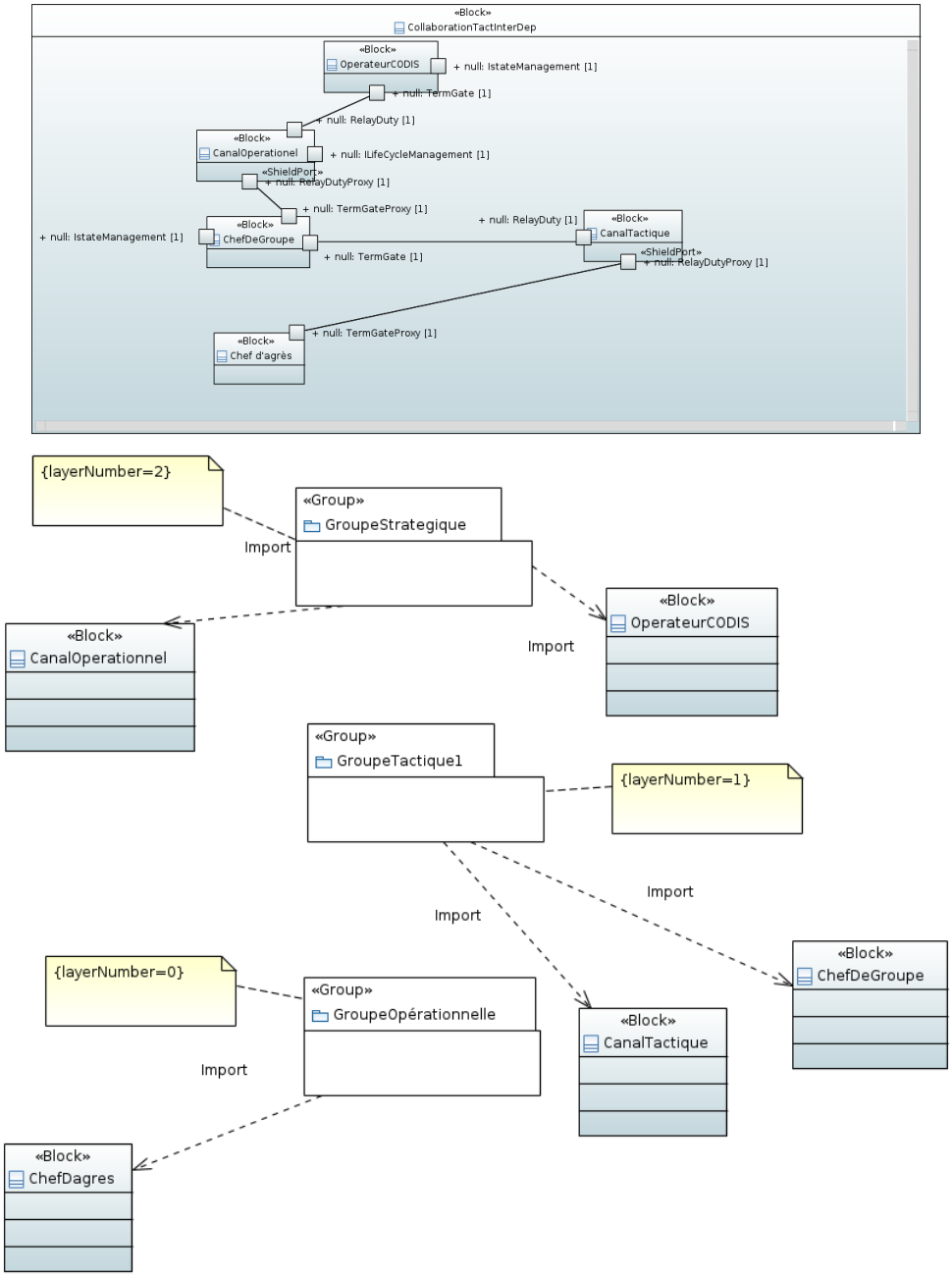

FIGURE 9.2: Architecture ciblée par l'itération 1

```
GroupLeader groupLeader= new GroupLeader(); CanalStrategic
canalStrategic = new canalStrategic();

Recrutment recru1 = new Recrutment(groupeTactique1, groupLeader);
Recrutment recru2 = new Recrutment(groupeTactique1,
                                   canalStrategic);

BindingEmitterReceptor binding1 = new
BindingEmitterReceptorProxy(canalStrategic, groupeTactique1);
```

FIGURE 9.3: Script itération 1 phase de préparation

Une autre partie du script demande au *teamleader* d'envoyer périodiquement des messages à son supérieur dans le but de vérifier que les contraintes ne sont pas violées.

9.2.2 Exécution

L'exécution de l'expérimentation est décomposée suivant le processus de reconfiguration identifié dans le chapitre 7. Chaque itération est donc divisée en trois phases : préparation, modification et nettoyage. Chaque itération sera décrite par l'architecture qu'elle cible et la stratégie suivie. Chaque phase sera décrite par un résultat qui est un script de reconfiguration.

Iteration 1 : phase préparation

Architecture ciblée : cf. figure 9.2
Stratégie : patron co-évolution
Résultat : décision d'une nouvelle itération et définition d'une nouvelle architecture (cf. figure 9.5) et script 9.3

Commentaire La figure 9.5 montre une architecture qui intègre les exigences pour mettre en œvre le patron de reconfiguration. La nouvelle architecture intègre un composant qui permet la synchronisation des états entre l'opérateur et le chef de groupe. Le canal opérationnel avec tampon est déployé pour suspendre les communications du chef de groupe. Le composant de synchronisation a pour rôle de surveiller les changements d'état de l'opérateur. Dès qu'un changement d'état est détecté, le composant met à jour l'état du chef de groupe.

Itération 2 : phase de préparation

Architecture ciblée : cf. figure 9.5
Stratégie : patron tranquillité
Résultat : décision d'une nouvelle itération et définition d'une nouvelle architecture (cf. figure 9.5, script de reconfiguration (cf. figure 9.4).

```
Synchronization synch = new Synchronization(null, null);
ChannelWithBuffer chBuff = new ChannelWithBuffer();

Recrutment recru4 = new Recrutment(groupStrategic, chBuff);
Recrutment recru3 = new Recrutment(groupStrategic, synch);

BindingStateManagement binding2 = new BindingStateManagement(
                                          synch, groupLeader);

BindingStateManagement binding3 = new
BindingStateManagement(synch, operator);
BindingObserver binding4 = new BindingObserver(operator, synch);
BindingObserver binding5 = new BindingObserver(
                                          groupLeader, synch);
```

FIGURE 9.4: Script itération 2 phase de préparation

Commentaire L'intention de la reconfiguration est de déployer les mécanismes d'évolution nécessaires à l'application du patron co-évolution. Le déploiement du composant de synchronisation n'altère pas les propriétés de reconfiguration. 7.4, on voit les opérations de reconfiguration nécessaires comme l'ajout du composant de synchronisation, la temporisation et les opérations de connexion associées. Le déploiement du nouveau canal opérationnel implique le risque de l'incohérence de l'état du SdS. Par exemple, le canal opérationnel peut collecter des messages du chef d'équipe et être déconnecté avant d'avoir transmis ces messages. La seconde partie de la reconfiguration peut être réalisée en suivant le patron tranquillité. Le patron de tranquillité garantie que la reconfiguration pourra être appliquée en garantissant un arrêt sûr du composant et donc que tous les messages seront transmis avant de connecter le chef d'équipe au canal opérationnel équipé d'un service de temporisation. L'inconvénient de cette stratégie de reconfiguration est que le critère de tranquillité ne soit jamais atteint et donc que la reconfiguration ne termine pas. Dans ce contexte, les communications sont sans état entre le canal opérationnel et le chef d'équipe ce qui implique que le critère de tranquillité est atteint dès que le canal a transmis le message reçu. En effet, quand le message est transmis le canal a terminé son exécution et donc ses transactions en cours avec ses composants voisins. Le chef d'équipe peut être connecté avec le nouveau canal de communication opérationnel, étant donné que le canal opérationnel termine son exécution.

Commentaire Le patron de reconfiguration tranquillité implique de pouvoir détecter les états de passivité du canal opérationnel et de réaliser les opérations de reconfiguration. Dans ce contexte, la passivité du composant est atteinte quand il a émis le message qu'il a reçu. La mise en œuvre du patron implique de pouvoir observer les messages émis par le chef d'équipe et lus par l'opérateur ainsi qu'un processus capable d'appliquer des opérations de déconnexion du canal opérationnel et connexion du canal opérationnel avec tampon

Commentaire La figure 9.6 montre la nouvelle architecture. Le composant tranquillité a pour fonction de surveiller les messages émis par le chef d'équipe et reçus par l'opérateur ainsi que de réaliser des opérations de déconnexion et connexion des dépendances requises par ces derniers.

Itération 3 : phase préparation

Architecture ciblée : cf. figure 9.6
Stratégie : aucune
Résultat : aucune.

Itération 3 : phase modification

Architecture ciblée : cf. figure 9.6
Stratégie : aucune
Résultat : script de reconfiguration (cf. figure 9.7).

Itération 2 : phase modification

Architecture ciblée : cf. figure 9.5
Stratégie : patron tranquilité
Résultat : script de reconfiguration (cf. figure 9.8).

Itération 3 : phase nettoyage

Architecture ciblée : cf. figure 9.6
Stratégie : aucune
Résultat : script de reconfiguration (cf. figure 9.9).

Itération 2 : phase nettoyage

Architecture ciblée : cf. figure 9.5
Stratégie : patron tranquilité
Résultat : script de reconfiguration (cf. figure 9.10).

Itération 1 : phase modification

Architecture ciblée : cf. figure 9.2
Stratégie : patron de co-évolution
Résultat : script de reconfiguration (cf. figure 9.11).

Commentaire La figure 9.2 montre l'architecture ciblée par l'itération. Elle correspond à l'objectif de la reconfiguration.

9.2.3 Chargement et exécution

La figure 9.12 montre le résultat de l'exécution du script. Nous voyons que les contraintes de reconfiguration n'ont pas été violées pendant la reconfiguration.

9.3 Synthèse

L'objectif de cette expérimentation était de valider la faisabilité d'une approche de reconfiguration par patron de reconfiguration. Pour cela, nous avons fourni à un architecte : le framework de reconfiguration proposé dans le chapitre précédent, fournit une architecture initiale et ciblée ainsi qu'un processus de reconfiguration. L'architecte de la reconfiguration a réalisé trois itérations impliquant huit phases. L'architecte a répondu que le niveau d'assistance était bon. Si nous regardons la terminaison de la reconfiguration, nous avons vu qu'elle est effective mais aussi que les contraintes imposées n'ont pas été violées.

Les différentes hypothèses proposées nous permettent de fournir une première évaluation de l'approche par l'analyse des variables dépendantes. Dans un premier temps, la facilité de la reconfiguration nous indique que l'architecte a pu facilement identifier un ensemble de solutions et choisir la plus pertinente. Ici l'architecte a pu au moins envisager autant de solutions que de patrons de reconfiguration disponibles. Cela montre aussi que la forme de la documentation est suffisamment expressive pour l'architecte. Ensuite, la terminaison de la reconfiguration nous indique que l'architecte a pu mettre en œuvre les principes d'ingénierie requis. Cela montre que la documentation aide l'architecte à implémenter sa reconfiguration. Enfin, la satisfaction des contraintes de reconfiguration indique que l'architecte a choisi les bons patrons de reconfiguration. Cela montre également que les patrons sont réutilisables.

L'expérimentation montre cependant des limites à prendre en compte. Le choix du sujet pose un problème puisqu'il a participé à la documentation des reconfigurations. Cela influence deux variables dépendantes : la facilité à comprendre les solutions des patrons est plus importante mais aussi la capacité à choisir un patron de reconfiguration adapté au problème défini. Le contexte de reconfiguration de l'expérimentation ne donne pas de contrainte particulière sur l'environnement de reconfiguration. Toutes les opérations de base sont fournies et ne requièrent pas d'être développées. La conséquence est que la terminaison de la reconfiguration est également facilitée. Nous noterons aussi que le cas d'étude est un modèle simplifié du SdS.

En conclusion, l'expérimentation a montré la faisabilité d'une approche par patrons de reconfiguration pour assister l'architecte dans la conception d'une reconfiguration de SdS. Nous voyons que plusieurs points peuvent être améliorés pour prendre en compte les limites de l'expérimentation. Le point principal est le sujet utilisé. L'expérimentation doit prendre une population d'architectes non expérimentés pour confirmer l'évaluation. Ensuite, des efforts de modélisation doivent être faits pour prendre en compte plus de détails dans le cas d'étude. D'autres variables dépendantes peuvent également être prises

en compte, comme par exemple le temps de conception des reconfigurations. En effet, dans un contexte critique, le temps de conception d'une reconfiguration a de l'importance.

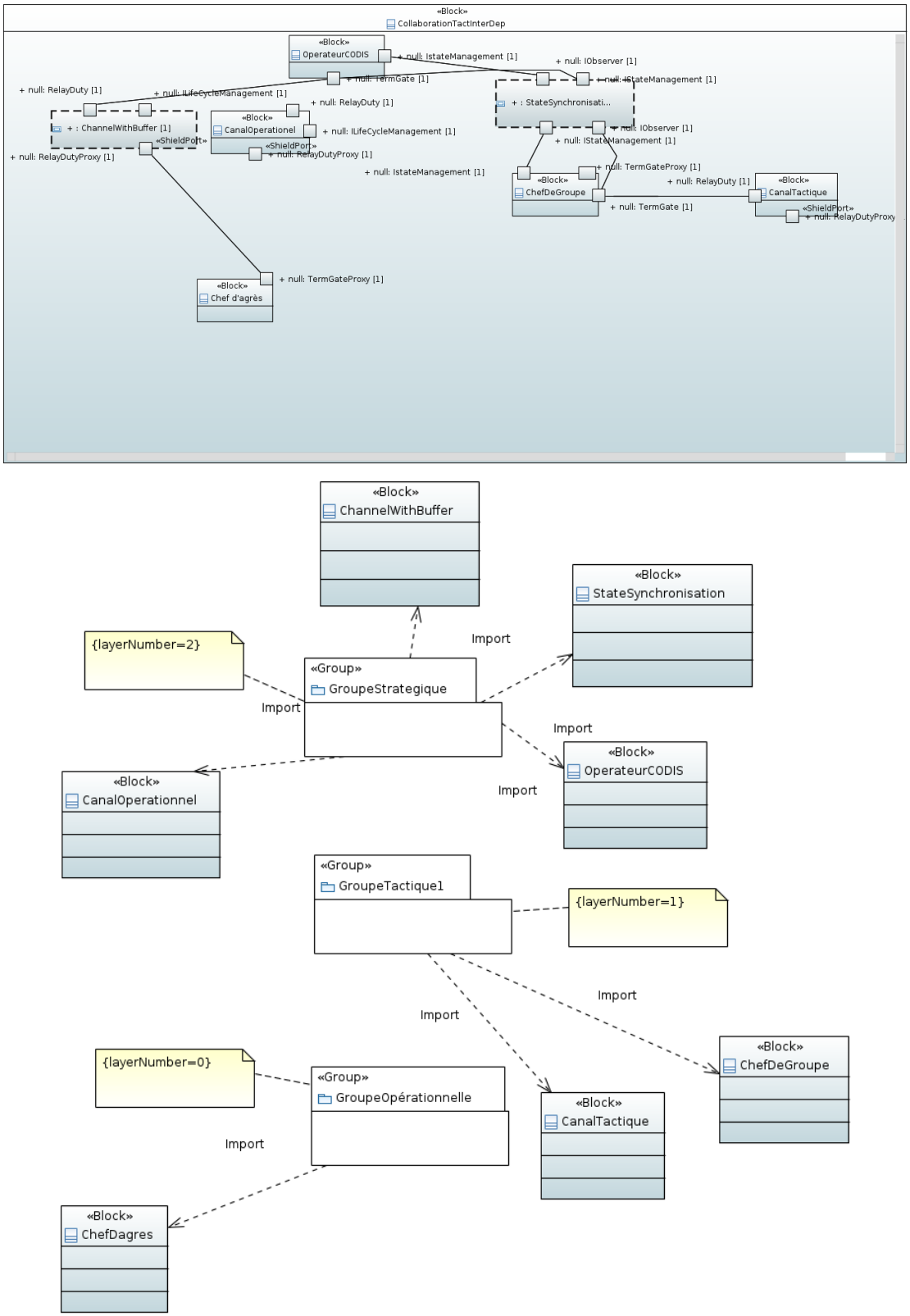

FIGURE 9.5: Architecture ciblée par l'itération 2

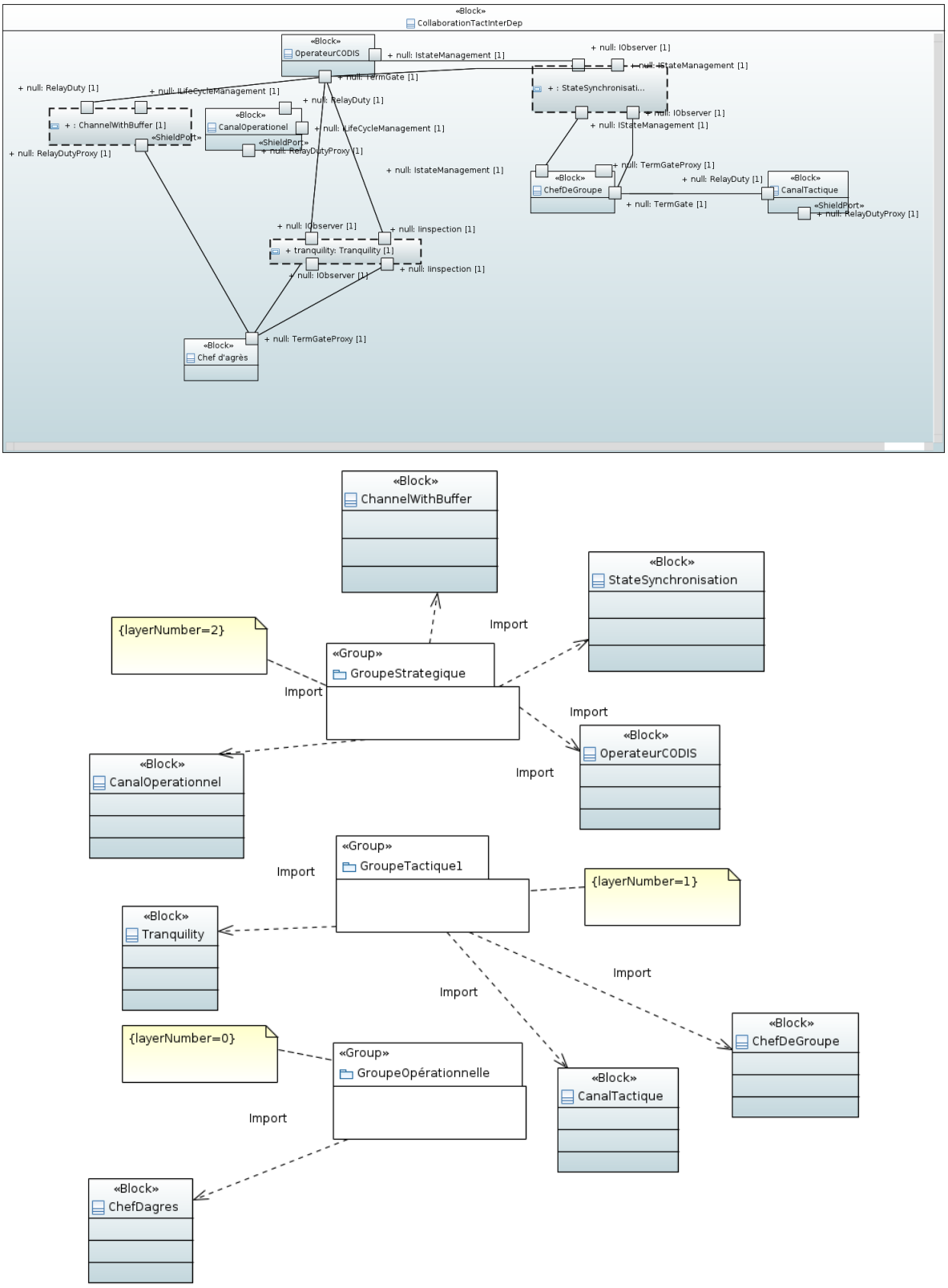

FIGURE 9.6: Architecture ciblée par l'itération 3

```
Tranquility tranq = new Tranquility();

Recrutment recru5 = new Recrutment(groupeTactique1,tranq);
BindingObserver binding6 = new BindingObserver(tranq, operator);
BindingObserver binding5 = new BindingObserver(tranq,
                                                teamLeader);
```

FIGURE 9.7: Script itération 3 phase de modification

```
BindingEmitterReceptor binding7 =
    new BindingEmitterReceptorProxy(operator, chBuff);
```

FIGURE 9.8: Script itération 2 phase de modification

```
UnBindingObserver unbinding6 = new UnBindingObserver(tranq,
                                                     operator);
UnBindingObserver unbinding5 = new UnBindingObserver(tranq,
                                                   teamLeader);
UnRecrutment unrecru5 =
    new UnRecrutment(groupeTactique1,tranq);
```

FIGURE 9.9: Script itération 3 phase de nettoyage

```
UnBindingStateManagement unbinding2 =
    new UnBindingStateManagement(synch, groupLeader);
UnBindingStateManagement unbinding3 =
    new UnBindingStateManagement(synch, operator);

UnBindingObserver unbinding4 =
    new UnBindingObserver(operator, synch);
UnBindingObserver unbinding5 =
    new UnBindingObserver(groupLeader, synch);

UnRecrutment unrecru4 =
    new UnRecrutment(groupStrategic, chBuff);
UnRecrutment unrecru3 =
    new UnRecrutment(groupStrategic, synch);
```

FIGURE 9.10: Script itération 2 phase de nettoyage

```
UnBindingEmitterReceptor unbinding8 =
    new UnBindingEmitterReceptorProxy(
                            canalOperationel, teamLeader);
BindingEmitterReceptor binding9 =
    new BindingEmitterReceptorProxy(
                            canalOperationel, groupLeader);
```

FIGURE 9.11: Script itération 1 phase de modification

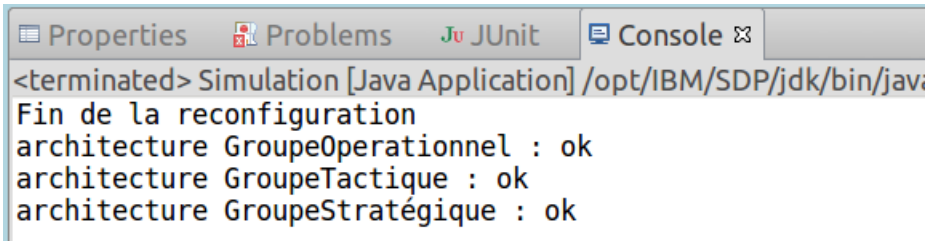

FIGURE 9.12: Résultat de la simulation

Quatrième partie

Conclusion - Perspectives

Chapitre 10

Conclusion et perspectives

Dans ce dernier chapitre nous allons synthétiser nos réponses aux questions posées dans l'introduction de ce mémoire, puis nous dresserons des perspectives faisant suite aux différents aspects de nos travaux.

10.1 Synthèse

La problématique que nous avons identifiée en introduction, la manière dont l'architecte doit prendre en compte le développement évolutionnaire d'un SdS (système de systèmes), a été déclinée en trois questions de recherche. Dans la section 10.1.1 nous allons tout d'abord synthétiser notre réponse à la question de la modélisation de la configuration du SdS. Puis la section 10.1.2 présente une synthèse de notre réponse à la question du processus de conception des reconfigurations. La section 10.1.3 donne une synthèse de notre réponse à base de patron de reconfiguration, afin de répondre à la question de la documentation des choix de conception des reconfigurations. Enfin, la section 10.1.4 fait la synthèse de l'expérimentation que nous avons conduite afin de valider les propositions.

10.1.1 Modélisation de la configuration du système de systèmes

Pour répondre à cette première question, nous avons étudié au chapitre 3 les principaux travaux concernant la modélisation des SdSs. Après avoir étudié SysML (System Modeling Language) et UPDM (Unified Profile for DoDAF and MODAF), nous nous sommes intéressés aux deux projets DANSE (Designing for Adaptability and evolutioN in System of systems Engineering) et COMPASS (Comprehensive Modelling for Advanced Systems of Systems), et plus précisément aux vues et aux langages de modélisations que ces deux projets défendent dans le cadre de la conception d'un SdS. Alors que le projet DANSE réutilise le framework UPDM, ce n'est pas le cas du projet COMPASS. Nous avons donc dû aligner les frameworks les uns avec les autres, afin d'élaborer un cadre commun pour analyser les approches de modélisation de ces précédents travaux.

Cette analyse de l'existant a mis en évidence que ces travaux précédents ne permettent pas de modéliser l'architecture d'un SdS, et plus particulièrement sa configuration, avec

le niveau d'exactitude et de précision requis pour la reconfiguration dynamique, c'est-à-dire de sorte que le modèle reflète le contexte réel de reconfiguration. Nous avons donc développé au chapitre 5 un framework de modélisation, s'inspirant des travaux des projets DANSE et COMPASS, mieux adapté à notre étude. Nous avons sélectionné un sous-ensemble des vues définies par le framework UPDM et utilisé SysML. Nous avons adapté certains choix, par exemple, en préconisant le diagramme de cas d'utilisation pour modéliser les objectifs d'interaction et les ressources. Pour répondre au besoin de précision des modèles produits, nous avons proposé d'enrichir ces modèles à l'aide de contraintes adaptées du catalogue de primitives architecturales de Zdun and Avgeriou (2008). Ces contraintes sont exprimées en OCL dans d'un profil qui définit des stéréotypes pour préciser (par annotation) les éléments de l'architecture. Pour assurer l'exactitude, nous imposons de plus que le processus de modélisation soit descendant.

Nous avons appliqué ce processus pour le scénario décrit au chapitre 4 afin de modéliser les configurations du cas d'étude nous servant à la validation de ce travail.

10.1.2 Processus de reconfiguration

En réponse à la seconde question de recherche nous avons proposé, au chapitre 7, un processus pour la conception des reconfigurations. Notre analyse de l'existant au chapitre 2 ne nous a pas permis de mettre en évidence un tel processus, même si Malabarba et al. (2000); Zhang and Cheng (2006); Boyer et al. (2013, 2017), par exemple, mentionnent la présence de plusieurs phases dans une reconfiguration.

Nous avons donc identifié les étapes qui permettent à l'architecte de produire une reconfiguration. En nous appuyant sur les précédents travaux sus-cités, nous avons proposé qu'une reconfiguration se décompose en trois phases : préparation, modification et nettoyage. Concrètement, la phase de préparation consiste à adapter le SdS pour permettre les modifications désirées. Dans un système distribué classique, cette phase de préparation aurait été, par exemple, le passage de certains composants vers leur état passif pour assurer la quiescence, et la phase de nettoyage remet les composants dans leur état actif. Dans le cas d'un SdS, ces phases peuvent être plus sophistiquées : elles peuvent inclure, par exemple, le déploiement de composants additionnels de manière transitoire, le temps de la reconfiguration, dans l'objectif d'établir tous les pré-requis de la modification tout en respectant les contraintes imposées par les indépendances managériales et opérationnelles. Nous avons proposé, dans le cas général, de considérer les phases de préparation et de nettoyage comme des reconfigurations à part entière. Le processus de conception proposé se ré-applique donc récursivement jusqu'à obtenir des reconfigurations simples.

Comme pour la modélisation des configurations, nous avons illustré le processus en l'appliquant concrètement dans le cadre du scénario de développement évolutionnaire décrit au chapitre 4.

10.1.3 Documentation à l'aide de patrons de reconfiguration

Pour répondre à la troisième question de recherche, nous avons décidé de reprendre le concept de patron, déjà bien connu pour documenter les choix de conception architecturale. En étudiant l'état de l'art de la reconfiguration au chapitre 2, nous avons constaté que des travaux précédents tels que ceux de Gomaa and Hussein (2004) ont déjà envisagé de transposer le concept de patron au domaine de la reconfiguration. D'autres travaux ont également abordé cette question de la documentation sans avoir recours au concept de patron. Néanmoins, nous avons estimé que la question méritait d'être davantage approfondie, notamment pour compléter les éléments de documentations à inclure afin de mieux prendre en compte les spécificités des SdS.

Au chapitre 6, nous avons donc précisé notre propre concept de patron de reconfiguration, en adoptant des notations semi-formelles. Au regard des travaux existants, nous améliorons la réutilisation des patrons en explicitant davantage les dépendances et prérequis. Il s'agit, notamment, d'une manière de prendre en compte les indépendances managériales et opérationnelles, en rendant explicite dans chaque patron les capacités de gestion et de contrôle que le patron suppose vis-à-vis des systèmes constituant le SdS.

Le chapitre 6 contient la description de deux patrons, nommés *quiescence* et *co-évolution*. Un troisième patron *re-routing* est donné en annexe A.

10.1.4 Expérimentation

Au chapitre 8, nous avons décrit un framework programmé en Java dans le but de réaliser une expérimentation décrite au chapitre 9 sur la base du cas d'étude décrit au chapitre 4. Il s'agit d'un système de systèmes pour une mission de réponse à une inondation par les services de secours français. Dans le scénario que nous avons modélisé, nous avons défini trois configurations, correspondant à l'évolution plausible d'une mission de secours devant être adaptée à l'aggravation de la situation. Cela nous a conduit, au cours de l'expérimentation, à concevoir deux reconfigurations en appliquant le processus de conception que nous avons défini. Dans cette étape, nous avons également utilisé les patrons que nous avons définis, qui documentent des techniques de reconfiguration que nous avons identifiées dans l'état de l'art, et qui sont adaptés aux systèmes de systèmes. Notre approche à base de patrons favorise la réutilisation d'éléments de conception, notamment grâce à la description du contexte, qui indique quand un patron est applicable, et les conséquences et forces qui précisent les effets obtenus lorsque le patron est appliqué.

L'expérimentation que nous avons conduite valide la faisabilité de l'approche proposée. Nous allons voir dans les perspective qui suivent comment nous envisageons de pallier certaines limites de notre approche et de sa validation.

10.2 Perspectives

Nous allons dresser quelques perspectives de notre travail tant au niveau du processus, de l'outillage des patrons que du domaine et de la population pour l'expérimentation.

Même si la notion de patron de reconfiguration telle que nous l'avons définie a été pensée avec, à l'esprit, la volonté de pouvoir composer plusieurs patrons les uns avec les autres lors de l'élaboration d'une reconfiguration, il ne s'agissait pas d'un objectif de notre travail. Dans les patrons que nous avons définis, nous avons l'intuition que tous ne sont pas au même niveau. Alors que le patron *co-évolution* nous semble répondre à un problème qui peut tout à fait être l'objectif global de la reconfiguration, propre à des situations spécifiques. Au contraire, nous pensons que le patron *quiescence* apporte une solution à un problème qui n'est qu'une étape de reconfiguration, utile dans plusieurs situations. Approfondir ce sujet pourrait apporter des éléments de réponse à la question de la composition des patrons de reconfiguration.

Au-delà de la simple question de la possibilité de composer les patrons les uns avec les autres, il s'agit de produire des outils pour faciliter l'utilisation des patrons de reconfiguration. Sur la base d'un outil de modélisation des reconfigurations, les éléments du modèle de la reconfiguration peuvent être annotés pour indiquer l'utilisation d'un patron similairement aux stéréotypes définis par Zdun and Avgeriou (2008) pour indiquer l'utilisation d'une primitive architecturale. Il s'agit d'une piste pour répondre au problème de la traçabilité entre les patrons et les reconfigurations.

Notre expérimentation nous a permis de valider la faisabilité de l'approche en considérant un cas d'étude. Il est nécessaire de poursuivre l'étude de validation de nos travaux en étendant l'expérimentation à d'autres cas réels. Dans un premier temps, ces expériences additionnelles peuvent conserver le cas de l'organisation des secours. S'il semble difficilement envisageable de tester notre approche dans le cadre d'une véritable opération, deux expériences peuvent toutefois être menées. D'une part, les retours d'expérience d'une opération pourraient être exploités pour élaborer un scénario d'une opération réelle. Même sans expérimenter en temps réel, une telle expérimentation nous permettrait de confronter notre approche pour l'évolution du dispositif à la réalité de l'opération de secours. D'autre part, mettre en place une expérimentation à l'occasion d'un exercice d'entraînement nous permettrait de tester notre approche avec les contraintes temporelles d'une opération réelle. En effet, la temporalité d'une opération réelle de secours nécessite de concevoir puis appliquer les reconfigurations du dispositif dans des temps contraints, compatibles avec la catastrophe (naturelle, technologique) à laquelle l'opération doit répondre.

Par ailleurs, un second ensemble d'expérimentations devrait permettre de qualifier et quantifier l'aide effectivement fournie à l'architecte pour la conception de la reconfiguration. À cet effet, une première expérimentation pourrait utiliser des étudiants, par exemple en fin de Master, pour fournir une population d'architectes significatives. Une telle expérience peut être mise en place en scindant la population en deux sous-groupes randomisés : l'un utilisant nos propositions, l'autre servant de groupe témoin. Des métriques telles que le temps nécessaire à la conception de la reconfiguration, et un questionnaire permettraient de réaliser une comparaison tant sur des critères quantitatifs que quali-

tatifs. Une population d'étudiants conduirait nécessairement à un biais causé par leur manque d'expérience. C'est pourquoi nous proposons qu'une expérience soit également menée avec des architectes expérimentés, mais vraisemblablement moins nombreux. Ces deux expériences se complèteraient ainsi, compensant les biais incontournables l'une de l'autre, aboutissant à un meilleur degré de validation de l'approche.

Le problème de la vérification d'une reconfiguration est un problème qui a été peu traité par les précédents travaux. C'est un problème que nous n'avons pas abordé dans cette thèse. Nos propositions peuvent contribuer au traitement de ce problème. En effet, les patrons de reconfiguration pourraient être vus non seulement comme des éléments de documentation et de conception, mais aussi comme des éléments de vérification réutilisables. Une telle perspective soulève la question de savoir comment apporter l'assurance requise dans la description des patrons, c'est-à-dire comment un patron peut être validé ou vérifié, puis de combiner les éléments de vérification issus des patrons dans un processus de vérification de la reconfiguration complète.

Sans nous prononcer pour défendre spécifiquement une approche, nous pouvons noter que vérifier une reconfiguration par tests soulève une question de méthodologie et, par exemple, de définition de critères de couverture. Les méthodes formelles basées sur la preuve de théorème permettent certes d'utiliser les quantificateurs pour éviter d'avoir à énumérer les constituants du système de systèmes, mais risquent de poser des problèmes de taille des preuves. Les méthodes basées sur le *model-checking*, enfin, certes automatiques, posent habituellement des problèmes de temps de calcul et requièrent l'énumération exhaustive de tous les constituants.

Dans cette thèse, le système de systèmes pris comme cas d'étude est en réalité un système de systèmes socio-technique, auquel participent des acteurs humains en plus des systèmes techniques. Nous avons décidé de faire abstraction de cela, et de considérer que les acteurs humains peuvent être modélisés de la même manière que les systèmes techniques. Outre les spécificités des acteurs humains qui mériteraient d'être pris en compte dans la modélisation d'un tel système de systèmes socio-technique, le thème de la conduite du changement a également été étudié dans les domaines du management et de la sociologie des organisations afin d'identifier les freins susceptibles de s'opposer, par exemple, aux modifications organisationnelles. Dans le cas d'un système de systèmes socio-technique, les freins peuvent également être un obstacle à la reconfiguration ou au développement évolutionnaire, ou exprimer des contraintes additionnelles à prendre en compte dans la conception de la reconfiguration. Se pose donc la question de savoir comment l'approche que nous avons proposée peut être étendue pour intégrer de telles préoccupations.

Annexe A

Patron re-routing

Intention

Le patron vise à réaliser des opérations de reconnexion tout en assurant que les messages envoyés pendant la reconfiguration sont correctement transmis.

Ce genre de reconfiguration intervient lorsque l'architecte souhaite réorganiser les connexions entre les composants du système. Une approche pour réaliser cette reconfiguration est de simplement déconnecter puis connecter la connexion ciblée de façon ad-hoc. Un problème qui pourrait survenir est la perte des messages envoyés par les composants. La raison serait que les composants ne mettent pas dans une zone tampon les messages lorsqu'ils n'ont pas la capacité de les envoyer. Une autre approche est de prendre en compte dans la reconfiguration qu'il faut utiliser une zone tampon pour les messages lorsque l'architecte reconfigure les connexions. L'inconvénient est que la zone tampon ne fait pas partie du fonctionnement normal des composants. Par conséquent, la reconfiguration doit attendre que le tampon soit vide pour finaliser la reconfiguration.

Ce type de patron est utile dans des SdSs (systèmes de systèmes) comme des systèmes de surveillance d'inondation tels que celui de Hughes et al. (2011). Ce SdS se compose de systèmes autonomes fournis par des organisations gouvernementales et civiles. Ces systèmes autonomes sont, par exemple, des drones ou des sondes fixes. Ils sont équipés de capteurs produisant des données environnementales qui sont collectées par des stations fixes qui donnent des prévisions sur le risque d'inondation à partir des données. Les exigences de ce type de système sont un compromis entre la gestion de l'énergie des systèmes et le temps de collecte des données par la station. Lorsque les systèmes sont géographiquement proches, il est possible d'économiser l'énergie en adoptant une configuration dans laquelle chaque système ne communique qu'avec ses voisins, permettant ainsi de réduire la puissance des signaux radio ; mais la réception des messages peut être plus longue par la station, le temps que ces messages se propagent de proche en proche dans le réseau. Si on souhaite plutôt que les informations soient obtenues plus rapidement, la configuration doit privilégier les plus courts chemins entre chaque système et la station de collecte. Dans ce type de système, les ressources des systèmes autonomes sont limitées (batterie, mémoire, processeur), les messages ne sont pas dans une zone tampon par défaut

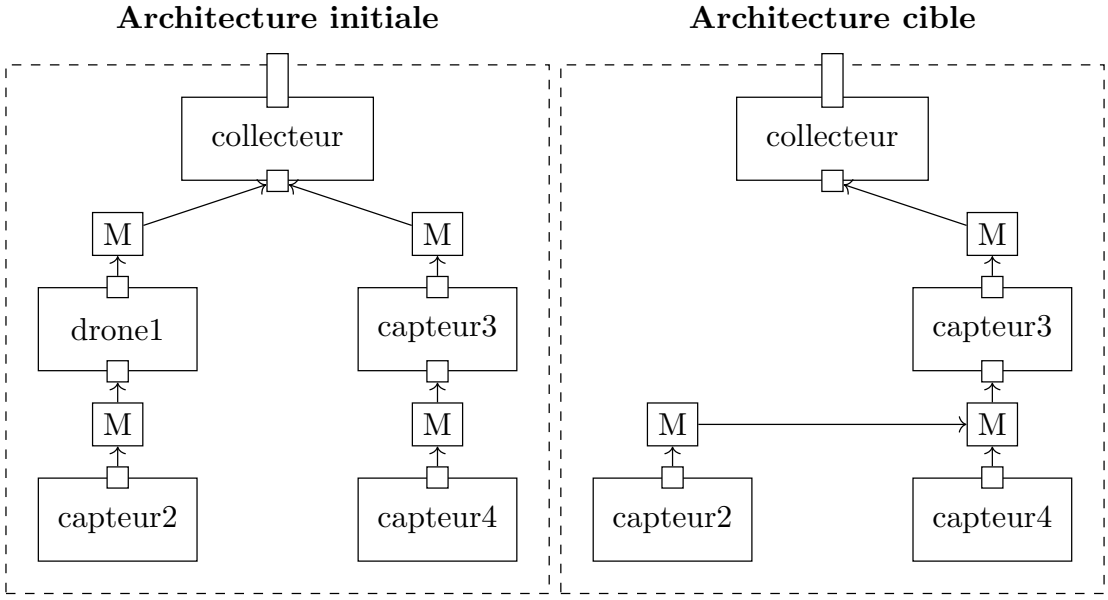

FIGURE A.1: Architecture initiale et finale

et la perte de message implique une ré-émission coûteuse. L'architecte de la reconfiguration doit donc éviter la perte de message. L'architecte peut avoir alors recours à ce patron pour préserver l'énergie et maîtriser la mémoire de ses systèmes. La figure A.1 montre un cas de reconfiguration extrait de cet exemple que nous avons exposé. Dans cette figure, la reconfiguration consiste à déconnecter le *capteur2* du *drone1* et de le reconnecter au *capteur3*.

Contexte

Ce patron de reconfiguration est supporté par des styles architecturaux comme le *Pipe&Filter* (Shaw and Garlan, 1996) et *C2* (Taylor et al., 1996). Dans ce type d'architecture, il y a un couplage faible entre les composants du système, et les communications sont unidirectionnelles. Le découplage est assuré par des médiateurs qui jouent, principalement, un rôle de routage des messages. Les médiateurs doivent être en mesure de fournir un service de mémoire tampon. Le patron de reconfiguration considère que les communications sont sans état.

Ce patron considère que le SdS est plutôt un SdS dirigé. En effet, le patron requiert un fort niveau d'ingérence des médiateurs.

Problème

Suite à une reconnexion ad-hoc, des messages peuvent être perdus si la déconnexion intervient avant qu'un composant n'ait pu envoyer son message. Des messages peuvent être également dupliqués, par exemple si l'émetteur a deux connexions simultanées. Comme

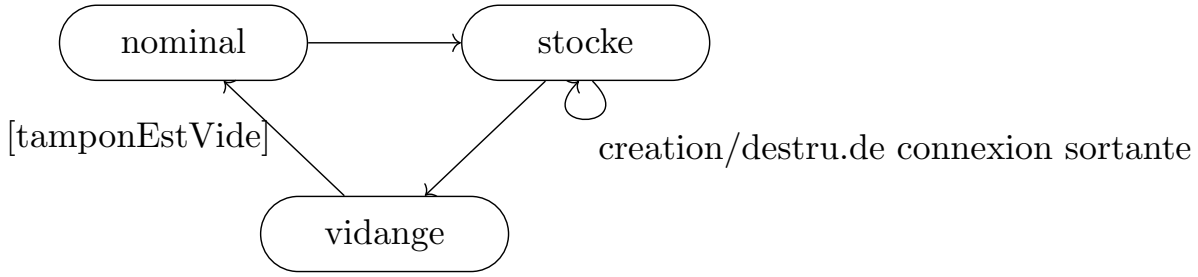

FIGURE A.2: Grammaire de reconfiguration

des effets négatifs peuvent survenir à cause de la duplication ou de la perte de message due aux opérations de reconnexion, l'architecte de la reconfiguration doit alors préserver deux invariants qui sont :

— un émetteur doit continuellement être connecté à un récepteur capable de conserver le message reçu et le transmettre plus tard.
— le récepteur de l'architecture initiale et celui de l'architecture ciblée ne doivent jamais être connectés simultanément au même émetteur.

Pour résoudre ces invariants, l'architecte de la reconfiguration doit mettre en œuvre un service de temporisation.

Solution

Dans la solution proposée, le médiateur fournit un service de mémoire tampon. La figure A.2 montre la grammaire de reconfiguration implémentée par le médiateur. La grammaire se compose des états suivants :

— un état nominal : le médiateur reçoit un message puis le transmet directement à son destinataire.
— un état stocké : le médiateur reçoit un message et le stocke dans un tampon.
— un état vidange : tous les messages stockés dans le tampon sont transmis aux composants récepteurs. Si de nouveaux messages arrivent, ils sont ajoutés à la fin du tampon et sont transmis dès que possible.

La reconfiguration suit les étapes suivantes :

- étape 1 : le médiateur passe de l'état nominal à stocké. À ce stade, le tampon devient le nouveau récepteur valide. Il stocke les messages.
- étape 2 : dans cet état, plusieurs actions sont possibles. Les connexions avec les composants receveurs sont manipulables, elles peuvent être ajoutées ou supprimées. C'est à ce moment que l'architecte applique les opérations de reconnexion ciblées. Les invariants sont toujours vérifiés : l'émetteur est connecté à un récepteur valide (stocke les messages reçus) et les récepteurs de l'architecture source et cible ne sont pas connectés à l'émetteur.
- étape 3 : une fois les opérations de reconnexion réalisées, le médiateur passe dans l'état de vidange. Tous les messages reçus dans l'étape 1 sont renvoyés. Ainsi aucun message n'est perdu.

- étape 4 : dès que tout le tampon est vidé, le médiateur revient dans son état nominal. La reconfiguration est terminée.

Conséquence

L'application du patron a plusieurs conséquences :
— les messages envoyés pendant la reconfiguration sont préservés.
— les messages ne sont pas dupliqués.

Annexe B

Contraintes OCL des modèles architecturaux

Dans cette annexe, nous résumons le style architectural suivi par le SdS de service d'urgence, puis la façon dont nous l'avons modélisé. Nous expliquons pour cela pourquoi SysML n'a pas suffit à sa modélisation et comment nous réutilisons les primitives architecturales développées par Zdun and Avgeriou (2008) pour modéliser le style architectural du SdS avec OCL (Object Constraint Language).

Pour définir le style architectural, nous avons analysé le cas d'étude pour en définir quelques principes. Le premier principe est que les composants du SdS sont regroupés selon leur niveau de décision : stratégique, tactique et opérationnel. Ensuite, les communications suivent une politique spécifique qui restreint les communications intergroupes. Les groupes sont ordonnés et les communications ne sont autorisées qu'avec les composants des niveau n+1 ou n-1 : par exemple, les composants du groupe opérationnel ne communiquent pas avec les composants du groupe stratégique. Nous devons donc étendre le concept de groupe avec celui de couche. Un troisième principe est que les communications des composants entre les couches ne peuvent se faire que via des canaux de communication dédiés. Les composants du groupe tactique ne peuvent être contactés que via le canal opérationnel. On utilise pour cela le concept de proxy.

L'expressivité des diagrammes UML et SysML présente quelques limites pour modéliser ces concepts. Le principe du style architectural repose sur la capacité à modéliser des groupes virtuels de composants formant des stratifications. Les relations d'association, agrégation et composition impliquent d'expliciter des structures. Cela ne nous convient pas : nous devrions utiliser une notion de *block* alors qu'il ne ferait référence à aucune structure physique. Comme solution, il est possible de modéliser ces groupes virtuels par des packages. L'inconvénient est qu'un élément ne peut pas être possédé par plusieurs packages. Cela limite les capacités de modélisation. Pour finir, il n'y a pas d'élément de modélisation SysML (System Modeling Language) pour modéliser le concept de bouclier, c'est-à-dire, qu'un élément encapsule l'accès à des composants d'une couche. Il faudra donc développer des contraintes particulières pour modéliser que le bouclier restreint l'accès au composant du groupe qu'il protège.

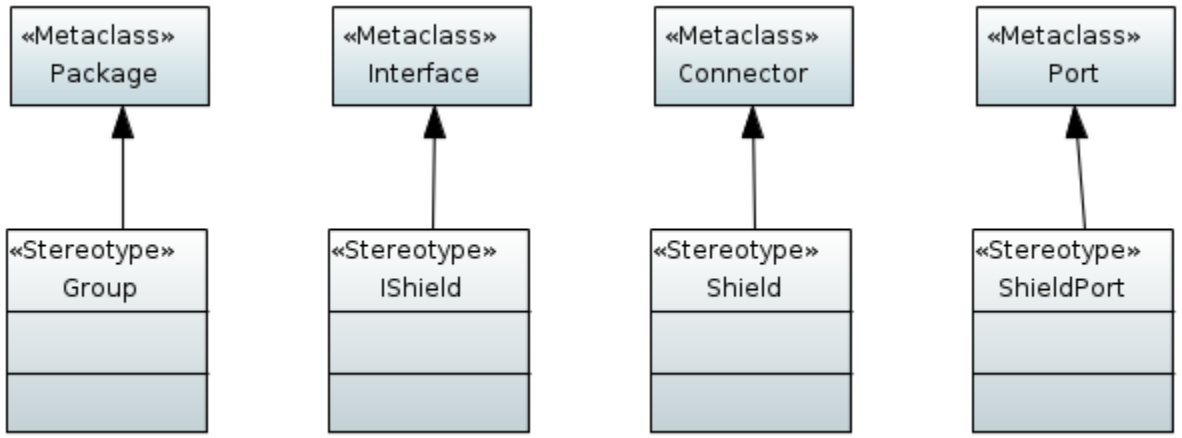

FIGURE B.1: Stéréotypes utilisés pour modéliser le style architectural du SdS de service d'urgence

```
-- A Group does not own any members
inv: self.basePackage.ownedMember->size()=0

-- All the imported members of a group are Components
inv: self.basePackage.importedMember->forAll(
        oclIsTypeOf(Core::Component))
```

FIGURE B.2: Stéréotype Group

```
-- A Layer member can only be part of one layer and not multiple layers
inv: self.basePackage->forAll(p1,p2:Core::Package|
 p1<>p2 implies
 p1.importedMember->intersection(
       p2.importedMember)->isEmpty())

-- Components in Layer X may only be connected to components in the same
-- Layer and Layer X-1 but not other Layers.
inv: self->forAll(l1,l2:Layer| l1<>l2 implies
 ((l1.layerNumber-l2.layerNumber).abs()>1 implies
  not l1.basePackage.ownedMember->forAll(c:Core::Component|
    l2.basePackage.ownedMember->exists(connects(c)))))

-- Check whether a Component is connected directly or indirectly to another
-- component through connectors
def: connects(target: Component): Boolean =
  self.ownedPort.opposite.class->includes(target) or
  self.ownedPort.opposite.class->exists(connects(target))
```

FIGURE B.3: Stéréotype Group étendu

```
-- The visibility of the methods of IShield are declared public so that
-- any client can access it directly
inv: self.baseInterface.feature->forAll(f |
 f.visibility = Core::VisibilityKind::public)

-- IShield interfaces are provided by a member of a group
inv: self.baseInterface->forAll(i|
 Core::Package.importedMember.oclAsType(Core::Component)
   .provided->includes(i))
```

FIGURE B.4: Stéréotype IShield

```
-- There is always an association that types the Shield and that association is navigable
-- in both ways so the classes own the association ends (preconditions so that
-- Property.opposite is not empty)
inv: self.baseConnector.type->size()=1
inv: self.baseConnector.type.ownedEnd->isEmpty()

-- A Shield Connector matches the provided IShield interface of a shield component
-- to the matching required interface of a client component.
inv: self.baseConnector.end->forAll(e1,e2:Core::ConnectorEnd|
  e1<>e2 implies
  (e1.role->notEmpty() and e2.role->notEmpty()) and
  (((e1.role.oclAsType (Core::Port).required=
        e2.role.oclAsType (Core::Port).provided) and
     e1.role.oclAsType(Core::Port).required->
        forAll(i| IShield.baseInterface->exists(j| j=i)))
   or
   ((e1.role.oclAsType (Core::Port).provided=
        e2.role.oclAsType (Core::Port).required) and
     e1.role.oclAsType(Core::Port).provided->
        forAll(i| IShield.baseInterface->exists(j| j=i)))))
```

FIGURE B.5: Stéréotype Shield

```
-- A shield port provides one or more interfaces, and one of them is an IShield interface
inv: self.basePort.provided->size()>=1 and
 self.basePort.provided->forAll(i:Core::Interface|
   IShield.baseInterface->exists(j| j=i))

-- All components connected to this port who are not client components (require the
-- same IShield that self provides) are members of the same group and that group has
-- the same name as the tagged value "shieldGroup"
inv:
let ShieldedComponents:Bag(Core::Component) =
 self.basePort.opposite->reject(p:Core::Port|
   p.required->includes(self.basePort.provided))
    .class.oclAsType(Core::Component)
in
 Groupings::Group.basePackage->one(importedMember->
   includesAll(ShieldedComponents) and
   name=self.shieldGroup)
 and
-- for each such component c, who does not provide an IShield interface, all provided
-- interfaces of c are of visibility "package"
 ShieldedComponents.ownedPort->reject(p:Core::Port|
     Shields::IShield.baseInterface->includesAll(p.provided))
   ->forAll(p:Core::Port|
     p.provided.feature->forAll(f|
       f.visibility=Core::VisibilityKind::package))
```

FIGURE B.6: Stéréotype ShieldPort

Au lieu d'utiliser la relation d'appartenance pour inclure un composant dans le groupe, on peut utiliser une relation *import*. La figure B.1, montre que l'on utilise un stéréotype *Group* qui étend la métaclasse *package*. La figure B.2 montre les contraintes associées. Ces contraintes modélisent qu'un groupe se compose d'au moins un élément et que tous les éléments sont des composants. Ensuite, nous avons besoin de la notion de couche pour modéliser que les groupes sont ordonnés. Afin de préciser l'ordre de chaque groupe nous précisons un attribut d'ordre du groupe dans la strate formée par les couches. La figure B.3 présente les stéréotypes ajoutés à la notion de groupe. Une fois la notion couche exprimée nous devons pouvoir vérifier qu'un composant n'appartient pas à plusieurs groupes, puis qu'un composant n'est pas connecté à la couche n-2. Une fois les contraintes de stratification exprimées, il reste à exprimer la notion de bouclier. Pour cela nous intégrons la primitive *Shield*. Elle se compose de plusieurs stéréotypes qui permettent de restreindre l'accès des composants appartenant à un groupe à la faveur du ou des composants boucliers de ce groupe :

— le stéréotype *IShield* étend la métaclasse interface ;
— le stéréotype *Shield* étend la métaclasse *Connector* ;
— le stéréotype *ShieldPort* étend la métaclasse port.

La figure B.4 présente les contraintes associées à *IShield*. Elles imposent que toutes les caractéristiques d'une interface soient publiques, et que le composant qui réalise cette interface appartienne à un groupe. Le stéréotype *Shield* impose que seuls les composants clients du composant bouclier puissent s'y connecter. Les interfaces doivent donc être compatibles. La figure B.5 décrit les contraintes utilisées. Pour finir le stéréotype *ShieldPort* impose que les composants qui ne sont pas clients et qui sont connectés au composant bouclier, fassent partie du même groupe que lui. Finalement, les composants qui appartiennent à cet ensemble doivent avoir toutes leurs interfaces fournies avec la visibilité package. La figure B.6 reprend ces contraintes en OCL.

Bibliographie

Allen, R., Douence, R., and Garlan, D. (1998). Specifying and Analyzing Dynamic Software Architectures. In *Colloquium on Formal Approaches in Software Engineering*, page 21.

André, F., Daubert, E., Nain, G., Morin, B., and Barais, O. (2012). F4plan : An Approach to Build Efficient Adaptation Plans. In Sénac, P., Ott, M., and Seneviratne, A., editors, *Mobile and Ubiquitous Systems : Computing, Networking, and Services*, pages 386–392, Berlin, Heidelberg. Springer Berlin Heidelberg.

Arshad, Naveed, Heimbigner, and Dennis (2005). A Comparison of Planning Based Models for Component Reconfiguration. Technical report, University of Colorado.

Barot, V., Henshaw, M., Siemieniuch, C., Sinclair, M., Lim, S. L., Henson, S., Jamshidi, M., and De Laurentis, D. (2013). Trans-Atlantic Research and Education Agenda in Systems of Systems. Technical report.

Batista, T., Joolia, A., and Coulson, G. (2005). Managing Dynamic Reconfiguration in Component-Based Systems. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Morrison, R., and Oquendo, F., editors, *Software Architecture*, volume 3527, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg.

Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., Wisniewski, R. W., and Kerr, J. (2005). Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 279–291.

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

Bonsangue, M., Clarke, D., and Silva, A. (2012). A model of context-dependent component connectors. *Science of Computer Programming*, 77(6) :685–706.

Boyer, F., Gruber, O., and Pous, D. (2013). Robust reconfigurations of component assemblies. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 13–22. IEEE Press. 31.

Boyer, F., Gruber, O., and Pous, D. (2017). A robust reconfiguration protocol for the dynamic update of component-based software systems : A RECONFIGURATION PROTOCOL FOR THE DYNAMIC UPDATE OF COMPONENTS. *Software : Practice and Experience*, 47(11) :1729–1753.

Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., and Shaw, M. (2009). Engineering Self-Adaptive Systems through Feedback Loops. In Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The FRACTAL Component Model and Its Support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284.

Bryans, J., Payne, R., Holt, J., and Perry, S. (2013). Semi-formal and formal interface specification for system of systems architecture. In *SysCon*, pages 612–619. IEEE.

Buisson, J., Dagnat, F., Leroux, E., and Martinez, S. (2015). Safe reconfiguration of Coqcots and Pycots components. In *The 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*.

Chen, G., Jin, H., Zou, D., Liang, Z., Zhou, B. B., and Wang, H. (2016). A Framework for Practical Dynamic Software Updating. *IEEE Transactions on Parallel and Distributed Systems*, 27(4) :941–950.

Coleman, J. W., Malmos, A. K., Larsen, P. G., Peleska, J., Hainsz, R., Andrews, Z., Payne, R. J., Foster, S., Miyazawa, A., Bertolinik, C., and others (2012). COMPASS tool vision for a system of systems Collaborative Development Environment. In *SoSE*, pages 451–456.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26(1) :1 :1–1 :42.

da Silva, C. E. and de Lemos, R. (2011). A framework for automatic generation of processes for self-adaptive software systems. *Informatica*, 35(1).

Dahmann, J. and Baldwin, K. (2008). Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering. In *2008 2nd Annual IEEE Systems Conference*, pages 1–7.

Dorn, C. and Taylor, R. N. (2015). Analyzing runtime adaptability of collaboration patterns. *Concurrency and Computation : Practice and Experience*, 27(11) :2725–2750.

Durán, F. and Salaün, G. (2016). Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122 :524 – 537.

Etzien, C., Gezgin, T., Passerone, R., Amold, A., Mangeruca, L., and Shindin, E. (2014). DANSE Modelling Formalism, including Domain Metamodel & Semantics : Focused on support for analysis and optimization. Technical Report D_6.2.3, OFFIS e.V.

Firesmith, D. (2010). Profiling Systems Using the Defining Characteristics of Systems of Systems (SoS). *Software Engineering Institute*.

Forcolin, M., Petrucco, P. F., Previato, R., Lloyd, R., Payne, R., Ingram, C., and Zoe, A. (2013). Accident Response Use Case Engineering Analysis Report Using Current Methods & Tools. Technical report, INSIEL.

Friedenthal, S., Moore, A., and Steiner, R. (2014). *A practical guide to SysML : the systems modeling language*. Morgan Kaufmann.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2015). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garlan, D. and T. Monroe, R. (2000). *Acme : Architectural Description of Component-Based Systems*. Cambridge University Press.

Georgiadis, I., Magee, J., and Kramer, J. (2002). Self-organising software architectures for distributed systems. In *Workshop On Self-healing Systems*, page 33. ACM Press.

Ghafari, M., Heydarnoori, A., and Haghighi, H. (2015). A Safe Stopping Protocol to Enable Reliable Reconfiguration for Component-Based Distributed Systems. In Dastani, M. and Sirjani, M., editors, *Fundamentals of Software Engineering*, volume 9392, pages 100–109. Springer International Publishing, Cham.

Gomaa, H. and Hussein, M. (2004). Software reconfiguration patterns for dynamic evolution of software architectures. In *Fourth Working IEEE/IFIP Conference on Software Architecture, 2004. WICSA 2004. Proceedings*, pages 79–88. 95.

Guessi, M., Oquendo, F., and Nakagawa, E. Y. (2016). Checking the architectural feasibility of Systems-of-Systems using formal descriptions. In *SoSE*, pages 1–6. IEEE.

Hayden, C. M., Saur, K., Smith, E. K., Hicks, M., and Foster, J. S. (2014). Kitsune : Efficient, General-Purpose Dynamic Software Updating for C. *ACM Transactions on Programming Languages and Systems*, 36(4) :1–38. 16.

Hicks, M. and Nettles, S. (2005). Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6) :1049–1096.

Hjálmtýsson, G. and Gray, R. (1998). Dynamic C++ Classes : A Lightweight Mechanism to Update Code in a Running Program. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA. USENIX Association.

Hoare, C. A. R. (2000). *Communicating sequential processes*. Prentice-Hall international series in computer science. Prentice Hall, New York, reprinted edition. OCLC : 249557024.

Hughes, D., Ueyama, J., Mendiondo, E., Matthys, N., Horré, W., Michiels, S., Huygens, C., Joosen, W., Man, K. L., and Guan, S.-U. (2011). A middleware platform to support river monitoring using wireless sensor networks. *Journal of the Brazilian Computer Society*, 17(2) :85–102.

Huynh, N. T. (2017). *A development process for building adaptative software architectures*. PhD Thesis, IMT Atlantique Bretagne-Pays de la Loire.

Jegourel, C., Legay, A., and Sedwards, S. (2012). A Platform for High Performance Statistical Model Checking – PLASMA. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Flanagan, C., and König, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214, pages 498–503. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kalawsky, R. S., Joannou, D., Bhatt, A., Ramalingam, K., Sanduka, I., Masin, M., Shindin, E., and Shani, U. (2014). Report on DANSE Architectural Approaches. Technical Report D5.4, Loughborough University.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1) :41–50.

Kramer, J. and Magee, J. (1990). The evolving philosophers problem : Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11) :1293–1306. 911.

Kruchten, P. (1995). The 4+1 View Model of architecture. *IEEE Software*, 12(6) :42–50.

Kruchten, P. (2004). *The rational unified process : an introduction*. Addison-Wesley Professional.

König, B. (2014). *Analysis and Verification of Systems with Dynamically Evolving Structure*. PhD thesis, Stuttgart.

Lochow, T., Sanduka, I., Bullinga, R., Arnold, A., Kalawsy, R., Cristau, G., Jung, M., Etzien, C., and Honour, E. (2013). Concept Alignment Description. Technical report, EADS DE.

Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., and Lu, J. (2011). Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, page 245. ACM Press. 56.

Maes, P. (1987). Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12) :147–155.

Magee, J. and Kramer, J. (1996). Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 3–14, New York, NY, USA. ACM.

Maier, W. M. (1998). Architecting principles for systems-of-systems. *Systems Engineering*, 1(4) :267–284.

Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J. F. (2000). Runtime Support for Type-Safe Dynamic Java Classes. In Bertino, E., editor, *ECOOP 2000 — Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 337–361. Springer Berlin Heidelberg. 252.

Mangeruca, L., Froschle, S., Etzien, C., Gezgin, T., Legay, A., Boyer, B., Arnold, A., Masin, M., and Shindin, E. (2013). Specification of the goal contracts specification language. Technical Report D_6.3.2, ALES S.r.l.

Medvidovic, N. and Taylor, R. (2010). Software architecture : foundations, theory, and practice. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 471–472.

Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. (2006). Practical dynamic software updating for C. In *Conference on Programming Language Design and Implementation*, page 72. ACM Press.

ODUSD(A&T)SSE (2009). *Systems Engineering Guide for Systems of Systems, V 1.0.*

Oliveira, N. and Barbosa, L. S. (2015). Reasoning about software reconfigurations : The behavioural and structural perspectives. *Science of Computer Programming*, 110 :78–103.

OMG (2013). Unified Profile for DoDAF and MODAF (UPDM). Technical report, OMG.

OMG (2015). OMG Systems Modeling Language (OMG SysML). Technical report, OMG.

Oquendo, F. (2004). pi-ADL : An Architecture Description Language Based on the Higher-order Typed pi-calculus for Specifying Dynamic and Mobile Software Architectures. *SIGSOFT Softw. Eng. Notes*, 29(3) :1–14.

Oquendo, F. (2017). Architecturally describing the emergent behavior of software-intensive system-of-systems with SosADL. In *SoSE*, pages 1–6. IEEE.

Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62.

Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society.

Perez, J., Ali, N., Carsi, J. A., and Ramos, I. (2005). Dynamic Evolution in Aspect-Oriented Architectural Models. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Morrison, R., and Oquendo, F., editors, *Software Architecture*, volume 3527, pages 59–76. Springer Berlin Heidelberg, Berlin, Heidelberg.

Petri, C. A. (1962). Fundamentals of a Theory of Asynchronous Information Flow. In *IFIP Congress*, pages 386–390.

Pissias, P. and Coulson, G. (2008). Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *IET Software*, 2(4) :348–361. 17.

Purtilo, J. and Hofmeister, C. (1991). Dynamic reconfiguration of distributed programs. In *International Conference on Distributed Computing Systems*, pages 560–571. IEEE Comput. Soc. Press.

Rasche, A. and Polze, A. (2005). Dynamic Reconfiguration of Component-based Real-time Software. In *Workshop on Object-Oriented Real-Time Dependable Systems*, pages 347–354. IEEE.

Robert, A. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University.

Rutten, E., Marchand, N., and Simon, D. (2017). Feedback Control as MAPE-K Loop in Autonomic Computing. In de Lemos, R., Garlan, D., Ghezzi, C., and Giese, H., editors, *Software Engineering for Self-Adaptive Systems III. Assurances*, volume 9640, pages 349–373. Springer International Publishing, Cham.

Shani, U., Etzien, C., Gezgin, T., Mangeruca, L., Stramandinoli, F., Senni, V., Marazza, M., Boyer, B., Shindin, E., Kalawsky, R. S., Lebeaupin, Y., and Albert, M. (2015). DANSE Prototype - Final Report. Technical Report D_8.8, IBM.

Shaw, M. and Garlan, D. (1996). *Software architecture : perspectives on an emerging discipline*. An Alan R. Apt book. Prentice Hall, Upper Saddle River, NJ.

Taylor, R., Medvidovic, N., Anderson, K. M., Whitehead Jr., E., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. (1996). A component- and message-based architectural style for GUI software. *Software Engineering, IEEE Transactions on*, 22(6) :390–406.

Taylor, R. N., Medvidovic, N., and Oreizy, P. (2009). Architectural styles for runtime software adaptation. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 171–180. IEEE.

Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. (2007). Tranquility : A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Trans. Softw. Eng.*, 33(12) :856–868. 137.

Vandewoude, Y., Rigole, P., Urting, D., and Berbers, Y. (2003). Draco : An adaptive runtime environment for components. *Appendix of the EMPRESS deliverable for Runtime Evolution and Dynamic (Re) configuration of Components*.

Waewsawangwong, P. (2004). A constraint architectural description approach to self-organising component-based software systems. In *International Conference on Software Engineering*, pages 81–83. IEEE Comput. Soc.

Wernli, E., Lungu, M., and Nierstrasz, O. (2013). Incremental Dynamic Updates with First-class Contexts. *The Journal of Object Technology*, 12(3) :1 :1. 6.

Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., and Perry, S. (2012). Features of CML : A formal modelling language for Systems of Systems. In *7th International Conference on System of Systems Engineering, SoSE*, pages 1–6. IEEE.

Zdun, U. and Avgeriou, P. (2008). A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9) :1003–1034.

Zhang, J. and Cheng, B. H. C. (2005). Specifying adaptation semantics. *ACM SIGSOFT Software Engineering Notes*, 30(4) :1.

Zhang, J. and Cheng, B. H. C. (2006). Model-based Development of Dynamically Adaptive Software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA. ACM.

Titre : Développement évolutionnaire de systèmes de systèmes avec une approche par patron de reconfiguration dynamique

Mots clés : Système de systèmes, Reconfiguration dynamique, patron de reconfiguration

Résumé : La complexité croissante de notre environnement socio-économique se traduit en génie logiciel par une augmentation de la taille des systèmes et par conséquent de leur complexité. Les systèmes actuels sont le plus souvent concurrents, distribués à grande échelle et composés d'autre systèmes. Ils sont alors appelés Systèmes de Systèmes (SdS). La complexité des systèmes de systèmes réside dans leurs cinq caractéristiques intrinsèques qui sont : l'indépendance opérationnelle des systèmes constituants, leur indépendance managériale, la distribution géographique, l'existence de comportements émergents, et enfin un processus de développement évolutionnaire.

Les SdS évoluent dans des environnements non prévisibles et intègrent constamment de nouveaux systèmes. Nous avons traité la problématique du développement évolutionnaire d'un SdS en utilisant la reconfiguration dynamique. Nous avons défini un processus pour élaborer des modèles de configurations et un processus de conception de la reconfiguration intégrant le concept de patron de reconfiguration. Pour la validité et la faisabilité de notre approche, nous avons développé un framework d'expérimentation basé sur notre cas d'étude réel d'organisation des systèmes de secours français.

Title : Evolutionary development of systems of systems with a dynamic reconfiguration pattern approach ....

Keywords : System of systems, dynamic reconfiguration, reconfiguration pattern

Abstract : The growing complexity of our socio-economic environment is reflected in software engineering by an increase of the size of systems and therefore their complexity. Current systems are mostly concurrent, widely distributed and composed of other systems. They are then called Systems of Systems (SoS). The complexity of systems of systems lies in five intrinsic characteristics: the operational independence of the constituent systems, their managerial independence, the geographical distribution, the existence of emerging behaviours, and finally an evolutionary development process.

SoS evolve in unpredictable environment and are constantly integrating new systems. We deal with the problem of the evolutionary development of a SoS by using dynamic reconfiguration. We have defined a process for developing configuration models and a reconfiguration design process incorporating the concept of reconfiguration pattern. For he validity and feasibility of our approach, we have developed an experimental framework based on our real case study of organization of the French emergency service.