Contributions à la validation des systèmes à composants adaptatifs par génération de tests

Jean-Philippe Gros

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE
FRANCHE-COMTÉ

THÈSE DE DOCTORAT

DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

JEAN-PHILIPPE GROS

Contributions à la validation de systèmes à composants adaptatifs par génération de tests

Thèse présentée et soutenue à Université de Bourgogne Franche-Comté, le 15 Décembre 2021

Composition du Jury :

CHRISTIAN ATTIOGBÉ	Professeur à l'Université de Nantes	Rapporteur
ANTOINE ROLLET	Maître de conférence à l'Université de Bordeaux	Rapporteur
PIERRE-CYRILLE HEAM	Professeur à l'Université de Bourgogne Franche-Comté	Président
JEAN-FRANÇOIS WEBER	Docteur de l'Université de Bourgogne Franche-Comté	Examinateur
OLGA KOUCHNARENKO	Professeur à l'Université de Bourgogne Franche-Comté	Directeur de thèse
FRÉDÉRIC DADEAU	Maître de conférences à l'Université de Bourgogne Franche-Comté	Codirecteur de thèse

N° | X | X | X | X | X | X | X | X | X | X | X | X |

REMERCIEMENTS

En premier lieu, je tiens à remercier mon directeur de thèse, Olga Kouchnarenko, pour avoir accepté de diriger ma thèse et pour ses précieuses remarques me permettant de l'améliorer.
Je tiens également à remercier mes rapporteurs, les professeurs Antoine Rollet et Christian Attiogbé, pour avoir accepté d'évaluer ma thèse. Leurs remarques m'ont permis d'améliorer la qualité de ce manuscrit.
Je remercie également le professeur Pierre-Cyrille Héam pour avoir accepté de faire partie du jury de cette thèse.
Je voudrais également remercier le docteur Jean-François Weber, de l'université de Franche-Comté, pour s'être intéressé à mes travaux et avoir accepté de faire partie du jury.
Je tiens à remercier, tout particulièrement, mes encadrants, Olga Kouchnarenko et Frédéric Dadeau, pour leurs conseils et leur aide tout au long de la thèse. En particulier, je tiens à les remercier pour leur soutien et leur compréhension.
Durant cette thèse j'ai également partagé de bons moments avec les collègues du laboratoire dont certains sont devenus des amis.
Karla merci pour ton accueil à mon arrivée, tu m'as aidé à m'intégrer dans cette ville où je ne connaissais personne.
Je n'oublierai pas Arthur et sa bienveillance ainsi qu'un certain attrait pour un humour que j'apprécie particulièrement.
Antoine, le bon vivant de l'équipe, toujours prêt à se sacrifier pour aider les plus faibles.
Éric, content d'avoir pu discuter avec toi ces derniers temps, cela m'a aidé à évacuer la pression. Le réel finit toujours par l'emporter et dans les moments difficiles il suffit de s'accrocher et tenir.
Fabien et Maria les piliers du laboratoire, toujours présents même au péril de vos vies, c'était bien sympa de discuter avec vous et de prendre des pauses à l'extérieur.
Partager le même lieu finit par créer des liens et c'est avec une certaine émotion que j'adresse ces mots à mes voisins de bureau.
Vahana, ta bonne humeur constante était très agréable et ton point de vue différent de notre vision des européens que nous étions était très intéressant.
Sire Henr**i** de Boutray, à partir de ton arrivée tu as été mon acolyte de thèse et d'autant plus au moment de nos rédactions respectives de manuscrits.
Nous avons eu l'occasion de débattre énormément et avec le temps de respecter nos

origines ainsi que nos points de vue différents.
Pascal, encore une très belle rencontre lors des évènements scientifiques, merci pour ces bons moments que nous avons partagé que ce soit dans le domaine scientifique ou non. J'en profite également pour passer le bonjour à Ana.
J'aimerais également remercier les étudiants rencontrés au fil des séances de cours que j'ai pu encadrer. Votre fraîcheur m'a changé du quotidien de la recherche qui n'est pas toujours simple. J'espère avoir été à la hauteur et être parvenu à vous transmettre le maximum de savoir pour votre vie future.

L'avenir décidera quels contacts nous maintiendront, sachez que quoiqu'il arrive j'ai été très heureux de vous connaître.

Ces années de thèses ont parfois été longues mais j'ai pu compter sur mes amis de longue date pour décompresser et m'aérer l'esprit. Je ne pourrai pas tous vous citer, mais sachez que ces petits week-ends ou congés à Grenoble, Londres ou en Haute-Savoie étaient de véritables bouffées d'oxygène.
Cédric et Anthony, je tiens à vous remercier d'avoir été présents lors de ce qui a été, jusqu'à maintenant la pire période de ma vie. Sans vous je me serais retrouvé seul dans une société où la violence et la torture morale sont devenues la norme. Si vous n'aviez pas été là je n'aurais sans doute pas eu la force de continuer à vivre.
Je vous dois donc beaucoup et je ferais tout mon possible pour vous aider ainsi que vos familles respectives. Et courage à Luis, on se battra jusqu'au bout pour te protéger de la sédentarité forcée.

Sarah, Pierre-Baptiste, Jean-Pierre et Sarah vous vous êtes déplacés en pleine semaine spécialement pour assister à ma soutenance et je souhaite vous remercier pour cela.

Je n'oublie pas mon bon ami Quentin, depuis plus d'une dizaine d'années que nous nous sommes croisées sur les bancs de la fac nous avons partagé beaucoup de bons moments ensemble et cela risque de continuer ainsi avec nos projets communs. Toi et Marie vous êtes un peu de la famille maintenant.

En parlant de famille, j'ai une pensée toute particulière pour mon papa. Tu as toujours été présent même dans les moments difficiles. Je mesure les sacrifices que tu as fait pour que je puisse aspirer à une vie normale et je t'en suis infiniment reconnaissant. Plus que des remerciements je souhaite te partager ma volonté de continuer à avancer pour être digne de ces sacrifices ainsi que de notre famille les De Molette de Morangiès.
Pour finir, maman, parfois j'aimerais te revoir pour discuter de notre société mais je pense que tu ne supporterais pas le monde d'après déjà que le monde d'avant manquait trop de libertés à ton goût.
En attendant de te rejoindre je sais que tu veilles sur moi d'où tu es et je ne t'oublie pas.

SOMMAIRE

1 Introduction **1**

1.1 Problématique et contexte scientifique 3

1.1.1 Validation des systèmes adaptatifs 3

1.1.2 Modélisation des systèmes à composants adaptatifs 4

1.1.3 Politiques d'adaptation 5

1.1.4 Génération de tests 6

1.2 Contributions 7

1.2.1 Questions de recherche 7

1.2.2 Processus de test 9

1.3 Présentation du plan 10

2 Contexte **11**

2.1 Systèmes adaptatifs et boucles de contrôle 12

2.1.1 Systèmes adaptatifs et politiques d'adaptation 12

2.1.2 Reconfigurations dynamiques 13

2.1.3 Boucles de contrôle 14

2.2 Architecture à composants 16

2.2.1 Le concept de composants 16

2.2.1.1 Composants primitifs et composites 16

2.2.1.2 Interfaces 17

2.2.1.3 Assemblages de composants 18

2.2.1.4 Choix du modèle de configuration 19

2.2.2 Le modèle à composants Fractal 22

2.2.3 Le modèle à composants Grid Component Model (GCM) 24

2.3 Validation de systèmes adaptatifs avec des tests 26

2.3.1 Évaluation de la qualité des séquences de test 26
2.3.1.1 Critères statiques 27
2.3.1.2 Critères dynamiques 28
2.3.2 Construction de tests 30
2.3.3 Verdict de tests 33
2.4 Conclusion 36

3 Préliminaires 37

3.1 Ensembles relations et fonctions 37
3.1.1 Ensembles 37
3.1.2 Relations et fonctions 38
3.1.3 Systèmes de transition 40
3.1.4 logique du premier ordre 41
3.1.4.1 Syntaxe 42
3.1.4.2 Sémantique 43
3.2 Exemple fil rouge : le convoi de véhicules VANet 45
3.2.1 VANet 45
3.3 Configurations, logiques temporelles et politiques d'adaptation 47
3.3.1 Modèle et chemin de reconfiguration 48
3.3.2 Logiques Temporelles 51
3.4 La logique FTPL 51
3.4.1 Syntaxe de FTPL 51
3.4.2 Sémantique de FTPL 51
3.4.2.1 Les propriétés de configuration 52
3.4.2.2 Les évènements 52
3.4.2.3 Les propriétés de trace 53
3.4.2.4 Les propriétés temporelles 53
3.5 Politiques d'adaptation 54

4 Proposition de modèle pour les systèmes à composants 59

4.1 Modèle à composants 59

4.1.1 Élements architecturaux 61
4.1.2 Relations architecturales 63
4.1.3 Instances architecturales 66
4.2 Contraintes de cohérence 68
4.2.1 Contraintes de cohérence pour les systèmes à composants 69
4.2.1.1 Contraintes liées au typage 69
4.2.1.2 Contraintes liées aux interfaces 70
4.2.1.3 Contraintes liées aux constructions hiérarchiques 72
4.3 Conclusion 73

5 Génération de tests en ligne pour les systèmes adaptatifs 75
5.1 Problématiques du test de systèmes adaptatifs 76
5.2 Générateur de configurations initiales 77
5.2.1 Algorithme de génération de configurations initiales 78
5.2.2 Échantillonnage des configurations initiales 81
5.2.3 Génération des valeurs pour les paramètres des composants 85
5.2.4 Présentation de la Beta-distribution 86
5.2.5 Exemples de variables selon la Beta-distribution 88
5.3 Processus de génération d'évènements de test en ligne 89
5.3.1 Modèle d'usage pour le test en ligne 89
5.3.2 Génération de tests à partir de modèles d'usage 93
5.4 Conclusion 99

6 Validation de systèmes adaptatifs 101
6.1 Validité du système vis à vis des propriétés FTPL 102
6.1.1 Critères de Couverture des propriétés FTPL 102
6.1.2 Verdict de test en rapport avec les propriétés FTPL 106
6.2 Validité du système vis à vis des politiques d'adaptation 107
6.2.1 Couverture de politique d'adaptation 107
6.2.2 Verdict de test basé sur une politique d'adaptation 109
6.2.3 Mesure de couverture pour les politiques d'adaptation 110

6.2.3.1 Nombre d'exécutions d'une règle . . . 111
6.2.3.2 Éligibilité d'une règle . . . 112
6.2.3.3 Fréquence d'une règle . . . 113
6.2.4 Conformité avec les politiques d'adaptation . . . 113
6.3 Conclusion . . . 114

7 Expérimentations 117

7.1 Implémentation . . . 118
7.1.1 Système sous test . . . 118
7.1.2 Propriétés FTPL . . . 119
7.1.3 Politiques d'adaptation . . . 120
7.1.4 Modèle d'usage . . . 122
7.2 Expérimentation sur les critères de couverture . . . 124
7.2.1 Protocole d'expérimentation . . . 124
7.2.2 Résultats . . . 125
7.2.3 Discussion . . . 129
7.3 Évaluation de la pertinence des configurations initiales . . . 129
7.4 Expérimentations sur les fréquences de déclenchement . . . 133
7.4.1 Protocole d'expérimentation . . . 133
7.4.2 Discussion . . . 135
7.5 Conclusion . . . 136

8 Conclusions et perspectives 137

8.1 Synthèse et bilan . . . 137
8.2 Perspectives . . . 139

1

INTRODUCTION

Sommaire

1.1 Problématique et contexte scientifique 3
1.1.1 Validation des systèmes adaptatifs 3
1.1.2 Modélisation des systèmes à composants adaptatifs 4
1.1.3 Politiques d'adaptation 5
1.1.4 Génération de tests 6
1.2 Contributions 7
1.2.1 Questions de recherche 7
1.2.2 Processus de test 9
1.3 Présentation du plan 10

Les systèmes adaptatifs sont au coeur des développements technologiques [126] et de l'industrie 4.0 dans laquelle on retrouve l'internet des objets (IoT), les systèmes réactifs, les nouvelles applications en robotique, les systèmes cyber-physiques [112] etc. Les systèmes complexes hétérogènes matériels ou logiciels peuvent être vus comme des systèmes à composants et dans ce cas, leur architecture se prête naturellement aux modifications de certaines parties. En effet, les systèmes à composants peuvent supporter des reconfigurations dynamiques, y compris des reconfigurations non anticipées [128]. Cependant, cela ne doit pas se faire au prix de la fiabilité du système : dans un monde de services interconnectés, les utilisateurs (humains ou non) s'attendent à ce que les services soient toujours disponibles. Dans la mesure du possible, les reconfigurations doivent se faire au moment de l'exécution et d'une manière qui ne met pas en péril la fiabilité du système [36].

De manière générale, les systèmes adaptatifs se reconfigurent, le plus souvent par le biais de déclencheurs en fonction des évènements internes et externes renseignés par des capteurs. Ceci peut être réalisé en guidant le système par le biais de règles de reconfiguration [40], qui peuvent être écrites en utilisant une structure SI [signal x est présent] ALORS [exécuter action y] [82], ou des objectifs d'adaptation [152].

Dans sa thèse [157], Jean-François Weber propose de faire évoluer les systèmes adaptatifs via des opérations de reconfiguration regroupées en politiques d'adaptation. D'autres travaux similaires sont décris [52, 53] grâce à l'outil DR-BIP. Lorsque le système peut déclencher des opérations de reconfiguration sans avoir à s'arrêter, on parle alors de reconfigurations dynamiques. Ainsi, il est possible de modifier le système en cours d'exécution afin de lui permettre de s'adapter à son environnement.

La fiabilité du comportement du système doit quant à elle permettre de garantir que les reconfigurations soient effectuées comme prévu. De plus, certaines propriétés de sûreté doivent être maintenues en cas de problème. La sécurité doit aussi être prise en compte afin que les reconfigurations ne soient déclenchées que par des entités autorisées. Il convient également de tenir compte du respect et du confort des usagers en ajoutant des propriétés extra-fonctionnelles. Enfin, face à l'urgence climatique, il est crucial que les reconfigurations réduisent la consommation d'énergie du système en stoppant les composants lorsqu'ils ne sont pas utiles au fonctionnement du système ou en mettant le système dans une configuration physique moins coûteuse en énergie.

Les règles régissant les systèmes adaptatifs sont regroupées en politiques d'adaptation [149], externes au système. Les politiques d'adaptation ont la possibilité de guider de manière non-prescriptive le système adaptatif dans le choix de ses actions à effectuer. Les politiques d'adaptation et le système peuvent être implémentés de différentes manières. En effet, tous deux dépendent de l'environnement dans lequel le système évolue et il y a autant d'alternatives d'implémentations que d'environnements possibles. Les politiques d'adaptation sont implémentées pour guider le système en toutes circonstances. En conséquence, ce n'est pas une implémentation particulière qui est admissible mais un ensemble d'implémentations.

C'est pourquoi, concevoir un modèle d'une implémentation d'un système sous politique d'adaptation en vue de valider une implémentation particulière serait trop restrictif. En conséquence, nous nous intéressons à l'évaluation de la cohérence entre l'implémentation d'une politique d'adaptation et l'exécution du système.

Actuellement, des méthodes de validation [96] existent afin d'établir qu'un système respecte les propriétés (temporelles) spécifiées en utilisant des traces d'exécution du système. L'analyse de traces d'exécution [129, 133] est un bon moyen pour s'assurer que le système et les politiques d'adaptation sont bien implémentés. En effet, l'analyse des traces permet de se focaliser sur le déclenchement des règles de reconfiguration et éviter de devoir gérer l'ensemble des états du système.

1.1/ PROBLÉMATIQUE ET CONTEXTE SCIENTIFIQUE

Dans cette section, nous introduisons les concepts et idées qui nous permettrons de pouvoir exposer la problématique liée au travail présenté dans ce document.

La problématique que nous souhaitons adresser dans cette thèse est :

Comment valider les systèmes adaptatifs à base de composants soumis à des politiques d'adaptation ?

Nous commençons par présenter le concept d'analyse et de validation de ces systèmes. Ensuite, nous nous intéressons à la modélisation de ces systèmes. Enfin, nous présentons les notions de politique d'adaptation et de test pour les systèmes adaptatifs.

1.1.1/ VALIDATION DES SYSTÈMES ADAPTATIFS

Plusieurs approches sont possibles quand il s'agit d'approuver le comportement du système. Ainsi, la notion de Vérification et Validation (V&V) revient souvent. Nous donnons ici un premier aperçu des approches (V&V) en citant quelques exemples. La vérification regroupe des méthodes diverses et variées, qui consistent à prouver que les propriétés sont assurées ou des relations sont établies, en explorant les modèles, les propriétés ou les relations entre composants du système en totalité.
La validation a pour but de montrer que le comportement du système s'est conformé à son objectif. Elle se fait en vérifiant la conformité du produit du comportement du système à ses exigences de besoin.

Les travaux [134] présentent la vérification et la validation pour des modèles de simulation. Plusieurs techniques de validations existent ; la comparaison à d'autres modèles dans laquelle les résultats du modèle à valider sont comparés aux résultats d'autres modèles valides ; le test en conditions extrêmes qui consiste à évaluer si le système est robuste face à des situations extrême et plausible ; l'observation du comportement du modèle obtenu sous forme de trace permet de déterminer si la logique du modèle est correcte.

Comme le précise [6], l'objectif des processus de vérification est d'évaluer la différence entre les résultats produits par le modèle de calcul et le modèle mathématique. Les types d'erreurs peuvent venir du fait que le code n'est pas une implémentation fidèle et précise du modèle discrétisé ou le modèle discrétisé peut ne pas être une représentation exacte du modèle mathématique. Ainsi, la vérification se divise en deux catégories correspondantes : la vérification des codes, qui relève du génie logiciel, et la vérification des solutions, qui implique l'estimation des erreurs. La vérification du code implique d'exercer le programme informatique développé pour mettre en œuvre le modèle de calcul afin de

déterminer et de corriger les erreurs de codage (bugs) ou d'autres déficiences qui nuisent à l'efficacité et à la qualité du modèle. La vérification de la solution d'un modèle de calcul doit, en général, être basée sur des estimations a posteriori de l'erreur. Par définition, les méthodes d'estimation d'erreur a posteriori sont des techniques de post-traitement qui tentent de déterminer des informations quantitatives sur l'erreur numérique réelle.

L'intégration des techniques de vérification et validation dans le cycle de vie des systèmes adaptatifs est un défi important [40]. Le test en ligne (ou test à l'exécution) est une méthode qui consiste à envoyer les séquences de test au système au moment de son exécution. Cette méthode est intéressante notamment pour les systèmes adaptatifs dont les configurations changent au cours de l'exécution.

Dans [131], les auteurs appliquent leurs mécanismes de vérification et validation sur un système adaptatif simulé dans lequel les auteurs ont procédé à une réduction drastique de l'espace des états.

Nous allons nous intéresser à la validation de systèmes adaptatifs par génération automatique de tests.

1.1.2/ MODÉLISATION DES SYSTÈMES À COMPOSANTS ADAPTATIFS

Comme leur nom l'indique, l'utilisation de systèmes à composants adaptatifs [146] permet l'exécution de reconfigurations dynamiques sur les composants. En effet, un système à composants peut évoluer d'une configuration à une autre au moyen d'opérations de reconfiguration, comme l'ajout ou la suppression d'un composant ou d'un lien entre deux composants.

Ainsi, les composants peuvent être activés ou désactivés à la volée (c'est-à-dire pendant l'exécution du système) afin d'adapter les systèmes à l'évolution de leur contexte d'exécution, mesuré par des capteurs. Par exemple, une voiture connectée peut choisir de s'appuyer sur un signal WiFi plutôt que sur une connexion GPS pour économiser la batterie.

Un composant est généralement décrit comme un élément d'une structure plus complexe, pouvant être intégré dans différentes applications. Les composants exposent les fonctionnalités qu'ils fournissent ainsi que les services dont ils ont besoin pour s'exécuter. La modélisation par composants permet de mettre en oeuvre différents concepts : la modularité, la réutilisabilité, les reconfigurations dynamiques et l'adaptabilité [61].
La modularité et la réutilisabilité permettent de structurer au mieux le système pour faciliter son développement. La modularité cherche à extraire les différentes fonctionnalités d'un système et les encapsuler en briques logicielles (ou composants). L'adaptabilité est

un concept permettant à un système de modifier son exécution ou sa structure en fonction de son environnement grâce, par exemple, à des politiques d'adaptation. Parmi les composants, on distingue les composants primitifs qui encapsulent du code métier et les composants composites qui contiennent des sous-composants. Il est possible de lier ces composants au moyen de liens logiques pour former une configuration du système à composants.

Considérons un ensemble de configurations, on peut ainsi, comme dans la thèse de Jean-François Weber [157], définir un modèle pour les systèmes à composants pouvant être assemblés pour construire des applications. L'ensemble des séquences de configurations débute par une configuration choisie et est construit de configuration en configuration au moyen des reconfigurations qu'il est possible d'effectuer. Afin de garantir son bon fonctionnement, le système est gardé par des propriétés de sûreté.

Les travaux de [96] répondent aux questions de cohérence d'une configuration après une reconfiguration en utilisant le langage à composant Fractal [110], un modèle à composants.
Fractal [19, 20], FraSCAti [138, 139] et BIP [7] (grâce à son extension DR-BIP [52, 53]) sont des modèles à composants supportant l'utilisation de reconfigurations dynamiques, qui permettent de reconfigurer un système à composants durant son exécution.

1.1.3/ POLITIQUES D'ADAPTATION

Dans le cadre des systèmes adaptatifs, une politique d'adaptation guide les reconfigurations en explicitant comment celles-ci peuvent être déclenchées en fonction de l'état actuel du système. Chaque politique est composée de reconfigurations et de règles qui spécifient les priorités des reconfigurations, qui sont gardées par des séquences d'événements spécifiques qui peuvent soit exploiter une propriété d'état, soit impliquer des propriétés de logique temporelle.
En effet, afin d'appréhender des comportements du système ne se limitant pas à un instant donné, mais prenant en compte son évolution au cours du temps, l'usage d'une logique temporelle pour exprimer les propriétés utilisées par les politiques d'adaptation est un choix qui s'impose. En se basant, au cours de l'exécution du système, sur l'évaluation de ces propriétés on peut alors estimer comment le système devrait évoluer. Cependant, la nature non-prescriptive des politiques d'adaptations rend impossible de garantir de manière absolue qu'une reconfiguration soit déclenchée.
Un modèle de système à composants pouvant évoluer sur la base de politiques d'adaptation utilisant des schémas temporels prenant en compte des évènements externes à été introduit dans [45].

Comme nous l'avons déjà mentionné, des informations sur l'environnement du système

peuvent être obtenues en utilisant les capteurs externes du système de façon à détecter des changements. Ces changements de l'environnement correspondent à des évènements externes pouvant être intégrés dans les propriétés utilisées par les règles incluses dans les politiques d'adaptation.

Les évènements externes mis bout à bout constituent des séquences de test que nous souhaitons générer automatiquement.

1.1.4/ GÉNÉRATION DE TESTS

Les premiers travaux théoriques sur le test logiciel datent des années 70 avec l'apparition des notions de tests fiables [85] et idéaux [73] qui permettent d'établir que le test ne peut que montrer la présence d'erreur, mais jamais leur absence [43].
Le test d'intégration de systèmes à composants, qui a émergé à la fin des années 1990, permet de s'assurer que de nouveaux assemblages de composants se comportent correctement dans leur nouvel environnement [158]. En effet, bien que chaque composant ait pu être testé indépendamment avec succès, leur combinaison peut engendrer des comportements imprévus.

Le test à partir de modèles est une technique de génération de tests à partir d'une abstraction du système, un modèle, qui est une représentation formelle du système. Dans un processus de test à partir de modèles, le développement du système sous test (System Under Test : SUT), que l'on souhaite tester et la conception d'un modèle pour la génération de test sont mis en parallèle. Cette séparation permet d'avoir une bonne assurance que le modèle n'est pas influencé par des détails provenant du développement ce qui nous donne une plus grande confiance dans les tests produits [125].

Le développement du modèle est une activité nécessairement manuelle qui est une représentation abstraite de la spécification. Ce modèle peut être décrit dans une multitude de formalismes dont le choix dépend de l'aspect du système que l'ingénieur de test souhaite représenter et des outils de génération de tests utilisés. La validation peut être établie via une relation de conformité entre le système sous test et le modèle qui le représente [148].

Dans le cas où il n'est pas possible d'établir un modèle du système, il est possible de considérer le système comme une boîte noire et de focaliser notre attention sur l'environnement, décrit par un modèle d'usage [154]. Cette approche peut être exploitée pour générer des séquences d'évènements externes qui peuvent être envoyées au système pour le stimuler et le valider comme dans [159]. Il est également possible d'utiliser des outils de génération de tests, qu'il est possible d'automatiser comme dans [24], permettant ainsi de générer de longues séquences de tests sans effort.

L'initialisation des variables du système est une étape importante dans la génération des tests. Dans [113], les auteurs procèdent en deux étapes, tout d'abord l'utilisateur renseigne le modèle architectural et les valeurs initiales avec un degré de liberté (ou de précision) et ensuite le solveur numérique initialise les valeurs si une solution existe. Si la solution n'est pas satisfaisante, l'utilisateur peut entrer de nouvelles valeurs pour les variables. Leur approche cible les redondances dans les variables afin de réduire le nombre de valeurs à entrer par l'utilisateur ce qui est particulièrement utile pour les systèmes complexes.

1.2/ CONTRIBUTIONS

Dans cette section, nous présentons nos contributions permettant de répondre à cette problématique. Les systèmes cyber-physiques sont des systèmes complexes et possèdent une infinité d'états qu'il serait impossible de vérifier exhaustivement. C'est pourquoi notre approche est une méthode de validation utilisant des tests sur le système à partir d'une configuration initiale et d'évènement externes. Afin de simuler au mieux le caractère imprédictible de ces évènements, nous avons mis en place un modèle de génération d'évènements aléatoires.

La validation repose sur des contraintes du système. Dans notre cas, nous souhaitons exprimer des contraintes architecturales et temporelles sur des systèmes à composants grâce à la logique FTPL qui a été introduite dans [47]. Cette logique se base sur les travaux de Dwyer sur les patrons de propriétés [49]. Par ailleurs, on notera que la structure complexe des systèmes adaptatifs se prête à une approche décentralisée ; l'évaluation décentralisée de formules temporelles présentée dans [11] ainsi que dans les travaux de thèse de Kalou Cabrera Castillos [23] en est un exemple.

Nous avons présenté le contexte scientifique dans lequel se situe notre problématique. Nous détaillons à présent les questions de recherche en expliquant, pour chacune d'elles, les différentes contributions apportées pour y répondre.

1.2.1/ QUESTIONS DE RECHERCHE

RQ1. **Quels objectifs de test définir pour valider les systèmes adaptatifs ?**
Les systèmes adaptatifs possèdent un nombre d'états fini et trop grand ou infini ne permettant pas d'appliquer directement des méthodes de couverture exhaustives. De plus, nous considérons le système comme une boîte noire et n'avons accès ni a son code ni a sa configuration. C'est pourquoi notre approche d'évaluation de tests repose sur des artefacts externes au système :

Dans un premier temps, nous proposons un critère de couverture sur les propriétés temporelles afin de nous assurer que le système est suffisamment stimulé.
Dans un second temps, nous proposons des critères de couverture sur les politiques d'adaptation du système. En effet, les politiques d'adaptation contiennent des règles de reconfiguration qui se basent sur des configurations et enchaînements temporels particuliers du système. Nous proposons d'utiliser le parcours de ces règles comme critère pour la qualité des tests envoyés au système.

RQ2. **Comment construire les tests ?**

Les tests (ou cas de tests) sont les éléments, actions ou données envoyés au système, nous considérons deux sous-parties dans le test :
1) Les états initiaux qui peuvent être générés à partir du modèle à composants qui servent de configurations de départ pour le test du système.
2) Les actions (ou évènements externes) qui sont décrites par le modèle d'usage, permettant de guider l'exécution.
Nous souhaitons créer les états initiaux par énumération de toutes les configurations possibles, jusqu'à une certaine taille donnée par une borne fixée à priori. Les valeurs des paramètres sont initialisées à part grâce à une loi de distribution probabiliste (loi Beta [75]). Les états initiaux sont ensuite classés en fonction de leur capacité à couvrir le système.
Pour les actions, nous choisissons d'utiliser des automates probabilistes pour simuler l'apparition imprédictible des évènements sur le système.

RQ3. **Comment établir le verdict de test ?**

Une fois que les tests ont été construits et exécutés, la question de comment établir un verdict associé à ces tests se pose.
Comme pour la RQ1, nous allons utiliser les artefacts du système pour répondre à cette question de recherche.
Le premier artefact est un ensemble de propriétés temporelles qui assurent que le système respecte le cadre prédéfini. Il apparaît naturel d'établir un verdict de test par rapport au respect de ces propriétés temporelles.
Le deuxième artefact est la politique d'adaptation du système qui permet de guider le système et l'assister sur le déclenchement de ses opérations de reconfigurations au bon moment. Là encore, un verdict de test est possible en comparant les choix faits par le système en comparaison à la description des règles de reconfigurations présentes dans la politique d'adaptation.
En effet, il est possible d'identifier si une reconfiguration non souhaitée a été activée ou si la priorité de déclenchement des règles n'est pas respectée par le système.

1.2.2/ PROCESSUS DE TEST

Le processus visible en figure 1.1 résume nos contributions. (1) Le système sous test est utilisé pour spécifier le modèle à composants (*Component Model*) permettant de décrire les configurations possibles. (2) Ce modèle rend possible la génération automatique des configurations initiales qui servent d'état initial (*Initial Configuration(s)*) lors de l'exécution du système (3). Un cas de test est composé d'une configuration initiale ainsi que d'une séquence d'évènements contrôlables qu'il faut générer.

Ces évènements contrôlables et externes au système sont décrits à l'exécution grâce à des modèles d'usage (*Usage Models*). Il existe un modèle d'usage pour chaque composant du système. Le générateur de test (*Test Generator*) sélectionne un modèle d'usage parmi l'ensemble des modèles d'usages disponibles (4) et envoie l'évènement contrôlable correspondant au système (5). En recevant ces évènements externes, le système (*Self adaptive System*) peut se reconfigurer (6) en accord avec les politiques d'adaptation (*Adaptation Policies*) qui le guident et les propriétés temporelles (*Temporal Properties*) qui spécifient son comportement tel que souhaité. En se reconfigurant, le système alimente sa trace d'exécution (7). Ces traces d'exécution servent à mesurer l'implémentation des artefacts du processus comme les politiques d'adaptation et le taux de déclenchement des règles de reconfiguration. De plus, les traces servent à mesurer le respect des propriétés extra-fonctionnelles (*Extra-functional Properties*) du système tel que le coût, la durée, le confort de l'utilisateur. Enfin, les traces de reconfiguration servent au générateur de test (8) pour mettre à jour l'état des modèles d'usage (9).

Afin de décrire les évènements adaptés à la configuration du système, l'état des modèles d'usage est mis à jour grâce à un générateur de test . Le générateur de test sert également à choisir un modèle d'usage parmi ceux disponibles et déclencher l'un des évènements correspondant à une des transitions partant de son état courant.

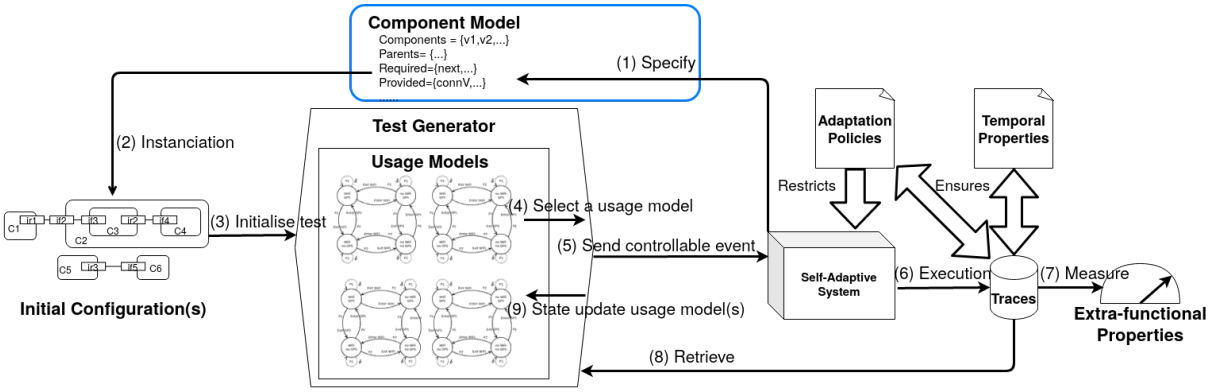

FIGURE 1.1 – Processus de génération de tests en ligne

1.3/ PRÉSENTATION DU PLAN

Cette thèse se découpe en 8 chapitres que nous présentons brièvement.
Les 3 premiers chapitres présentent le contexte de ces travaux :

- Le **Chapitre 1** présente le contexte et les problématiques de cette thèse. Il présente succinctement le processus mis en place pour adresser les problèmes de recherche posés.
- Le **Chapitre 2** présente le contexte scientifique global des différents domaines auxquels cette thèse se rattache. Ce chapitre donne une vue élargie sur les techniques de modélisation des systèmes adaptatifs ainsi que sur leur validation.
- Le **Chapitre 3** est consacré aux définitions et notations utilisées dans la suite du document. Il s'agit notamment de la spécification et la validation des systèmes ainsi qu'à l'expression des propriétés temporelles et politiques d'adaptation.

Les chapitres 4, 5 et 6 traitent les contributions de cette thèse :

- Le **Chapitre 4** décrit le modèle à composant pour les systèmes adaptatifs ainsi que les contraintes de cohérence permettant de garantir que le système est dans un état permettant son bon fonctionnement.
- Le **Chapitre 5** décrit notre approche de génération de tests avec en première partie la génération des configurations initiales et en deuxième partie la génération des évènements externes.
- Le **Chapitre 6** présente les méthodes de validation des systèmes adaptatifs avec la définition de critères basés sur les propriétés temporelles en première partie et les politiques d'adaptation en seconde partie.
- Le **Chapitre 7** porte sur l'évaluation expérimentale de cette approche et positionne ce travail dans le cadre de l'étude de cas VANet. Ce chapitre présente le prototype logiciel et la validation des contributions décrites dans les chapitres précédents.
- Enfin, le **Chapitre 8** présente les conclusions ainsi que les perspectives issues de ces travaux.

2

CONTEXTE

Sommaire

2.1 Systèmes adaptatifs et boucles de contrôle 12
2.1.1 Systèmes adaptatifs et politiques d'adaptation 12
2.1.2 Reconfigurations dynamiques 13
2.1.3 Boucles de contrôle 14
2.2 Architecture à composants 16
2.2.1 Le concept de composants 16
2.2.2 Le modèle à composants Fractal 22
2.2.3 Le modèle à composants Grid Component Model (GCM) 24
2.3 Validation de systèmes adaptatifs avec des tests 26
2.3.1 Évaluation de la qualité des séquences de test 26
2.3.2 Construction de tests 30
2.3.3 Verdict de tests 33
2.4 Conclusion 36

Les systèmes adaptatifs sont généralement ouverts et doivent être capables de répondre à de nouvelles exigences et de nouveaux besoins. En même temps, ces systèmes sont déployés dans des environnements hétérogènes et en constante évolution (parfois même hostiles) et doivent donc réagir rapidement à ces changements.
Les politiques d'adaptation permettent de répondre au besoin d'expression de l'adaptativité de tels systèmes en gardant le contrôle sur leur complexité et sont décrites sur deux niveaux :

— le niveau architectural contenant la configuration du système et,

— le niveau fonctionnel décrivant ses actions.

L'auto-adaptation, ou, dans notre cas l'adaptation, est un domaine de recherche contenant de nombreux défis. La feuille de route de 2013 [40] souligne qu'un défi important consiste à faire le lien entre la conception et l'implémentation des systèmes adaptatifs. Les systèmes cyber-physiques sont un très bon exemple de systèmes adaptatifs. En effet, l'adaptation à diverses situations et environnements est devenue une nécessité pour

les systèmes cyber-physiques [93] permettant de construire des infrastructure logicielles plus intelligentes. Pour modéliser de tels systèmes, l'architecture à composants permet à la fois de détailler la configuration du système et les opérations de reconfigurations. Enfin, pour avoir une meilleure confiance dans le comportement du système adaptatif, la validation par le test est nécessaire [142, 162].

Dans ce chapitre, nous commençons par faire un tour d'horizon des systèmes adaptatifs pour ensuite nous focaliser sur les systèmes à composants puis, nous aborderons la question de la validation des systèmes adaptatifs en utilisant des séquences de tests.

2.1/ SYSTÈMES ADAPTATIFS ET BOUCLES DE CONTRÔLE

Dans cette section, nous explicitons comment les systèmes adaptatifs font pour évoluer au cours du temps.
Dans un premier temps, nous présentons les systèmes adaptatifs pour comprendre les enjeux de tels systèmes. Ensuite, nous décrivons le concept de reconfiguration dynamique permettant au système de modifier son architecture au cours de l'exécution. Enfin, nous définissons les boucles de contrôle permettant la prise de décision de certaines reconfigurations dynamiques.

2.1.1/ SYSTÈMES ADAPTATIFS ET POLITIQUES D'ADAPTATION

Un système adaptatif est un système qui évolue en réponse à un changement de son contexte d'exécution afin de s'adapter pour répondre au mieux à l'environnement dans lequel il se trouve. Pour pouvoir considérer une opération de reconfiguration, il faut que cette opération ait un intérêt vis-à-vis des politiques d'adaptation qui guident le système. Un système adaptatif nécessite les éléments suivants :

— un ensemble d'opérations pour modifier et adapter le système,

— un contexte qui regroupe des éléments pertinents de l'environnement,

— une fonction d'adéquation permettant d'évaluer le système par rapport à son contexte et,

— une stratégie d'adaptation.

Dans le domaine de la modélisation dynamique pour systèmes à composants adaptatifs, des ensembles de règles avec priorités sont utilisés dans [79].
L'approche proposée dans [70] suggère d'utiliser un schéma d'adaptation basé sur des règles et des utilités pour optimiser les décisions d'adaptation. Dans [130, 149], les politiques d'adaptation contiennent des règles avec des valeurs sur les priorités alors que dans [91], les fonctions d'utilité sont utilisées pour définir des objectifs pour le système.

Par contre, ces travaux ne valident pas que des ensembles de règles soient fidèlement implémentés. Dans [124], les auteurs améliorent la réactivité des systèmes adaptatifs grâce à des algorithmes prédictifs.
Les reconfigurations dynamiques peuvent permettre de répondre au besoin d'opération pour modifier et adapter le système. Nous présentons dans la section suivante les modèles de boucle de contrôle MAPE et MAPE-K utilisés dans le cadre de systèmes logiciels adaptatifs.

2.1.2/ RECONFIGURATIONS DYNAMIQUES

On parle de reconfiguration pour décrire le changement d'un modèle dont l'architecture évolue d'une configuration donnée vers une autre. Les modèles dynamiques et architectures dynamiques sont caractérisés par la possibilité qu'ils ont d'évoluer lors de leur exécution sans nécessiter l'arrêt ou la réinitialisation du système complet pour que celui-ci puisse être modifié. Les reconfigurations permettant de telles évolutions sont appelées reconfigurations dynamiques. Bien sûr, les reconfigurations dynamiques peuvent être utilisées sur les architectures à base de composants [119]. Dans le cadre des modèles à composants, les reconfigurations dynamiques peuvent prendre en compte les aspects de modularité et d'encapsulation des composants pour mesurer leurs impacts sur le bon fonctionnement du système. L'utilisation de ces reconfigurations est fortement liée à l'architecture du système, c'est pourquoi les composants assemblés doivent êtres choisis avec soin pour que les reconfigurations dynamiques puissent être mises en place. Les opérations de reconfiguration généralement utilisées dans les modèles à composants correspondent à l'ajout et la suppression de composants ou de liens entre les interfaces de composants.
Plusieurs problématiques devant être adressées lors de la mise en application de reconfigurations dynamiques ont été recensées dans [118] ; il s'agit des suivantes :

— les reconfigurations dynamiques doivent assurer la cohérence du système au niveau de sa structure, mais aussi de son comportement,

— certaines propriétés non-fonctionnelles telles que l'intéropérabilité et la performance doivent être aussi prises en compte,

— le modèle et l'implémentation doivent rester cohérents au cours de l'exécution des reconfigurations du système,

— l'exécution des reconfigurations ne doit pas se faire à n'importe quel moment et doit donc être synchronisée avec l'exécution fonctionnelle du système pour que le système ne soit pas amené dans un état incohérent et

— la possibilité d'exécuter plusieurs reconfigurations simultanément doit être gérée dans un système pour éviter des conflits entre les reconfigurations.

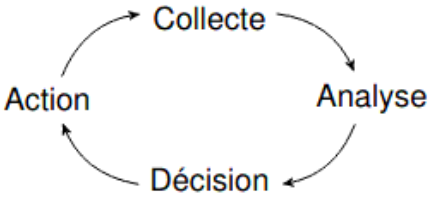

FIGURE 2.1 – Activités d'une boucle de contrôle MAPE

2.1.3/ BOUCLES DE CONTRÔLE

Une particularité des systèmes adaptatifs réside dans le fait que la prise de certaines décisions, habituellement effectuées au moment de la conception dans le cadre du développement de logiciels classiques, est faite au moment de l'exécution. Le raisonnement menant à cette prise de décision utilise souvent un processus de rétroaction ($feedback$) composé de quatre activités (collecte, analyse, décision et action) appelé boucle de contrôle MAPE (acronyme de l'anglais pour $Monitor$, $Analyzer$, $Planner$, $Executor$).

Le modèle de boucle de contrôle [44, 27, 38] représenté en figure 2.1 est composé de quatre activités détaillées ci-dessous :

— **Collecte :** Cette activité ($monitor$) consiste à collecter des informations pertinentes provenant du système adaptatif lui-même, de capteurs externes, de ressources distantes accessibles par le réseau ou encore du contexte "utilisateur" ($usercontext$) de l'application.

— **Analyse :** Les informations collectées sont ensuite analysées ($analyzer$) en utilisant des raisonnements basés sur l'inférence, mais aussi sur des méthodes moins certaines (e.g., probabilité ou logique floue). Bien sûr, cette analyse doit aussi prendre en compte l'architecture du système adaptatif, tant son état courant (et possiblement, dans une certaine limite, ses états passés) que ses possibilités d'évolution.

— **Décision :** La phase de décision ($Planner$) utilise les résultats produits par l'activité d'analyse afin de déterminer si une opération de reconfiguration est souhaitable et, si oui, quelle opération parmi celles envisageables doit être privilégiée. Pour ce faire, des hypothèses sont générées et des techniques issues de la théorie de la décision ou des méthodes d'analyse de risque sont utilisées pour produire une décision.

— **Action :** cette activité ($Executor$) consiste à mettre en oeuvre la décision prise au cours de la phase précédente en s'appuyant sur des mécanismes propres au système adaptatif lui-même. Cette action et son résultat doivent être enregistrés dans un journal (log) et, suivant l'impact de la reconfiguration, les utilisateurs ou administrateurs de l'application peuvent être notifiés.

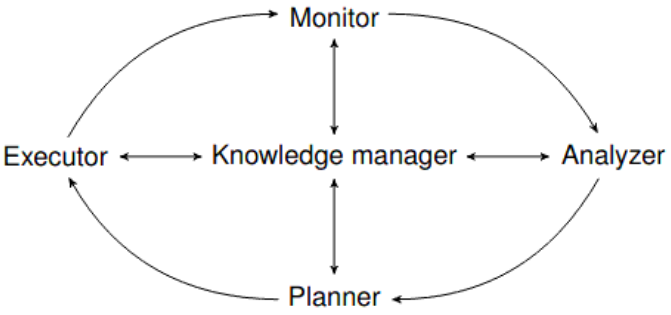

FIGURE 2.2 – Activités d'une boucle de contrôle MAPE-K

Le modèle de référence de boucle MAPE-K [31] (MAPE-K loop reference model) a été proposé par IBM pour guider la conception de systèmes adaptatifs [90].
D'autres approches [65] utilisent la boucle MAPE-T pour évaluer l'applicabilité et l'utilité de cas de test pendant l'exécution.
Le modèle MAPE-K peut être utilisé dans divers domaines tels que dans les travaux de [76] où l'approche permet de résoudre les pannes d'exécution en adaptant dynamiquement l'architecture du système. Dans [137], les auteurs atteignent un certain niveau d'auto-réparation pour les systèmes cyber-physiques. Si une incohérence entre le monde physique détecté et le monde virtuel supposé peut être détectée, une stratégie de compensation est choisie et le processus adapté est exécuté.

Ce modèle, inspiré (comme celui de la boucle de contrôle) par celui de la boucle rétroactive ($feedback - loop$) utilisé dans la théorie du contrôle, est composé des cinq éléments ($Monitor$, $Analyzer$, $Planner$, $Executor$, $Knowledge\ manager$) représentés dans la figure 2.2. À chacun de ces éléments sont associés des fonctionnalités spécifiques ainsi qu'un flux d'informations permettant le contrôle autonome de l'application logicielle gérée par la boucle MAPE. Intuitivement, la différence fondamentale entre la boucle MAPE et la boucle MAPE-K réside dans le rôle du gestionnaire de connaissance ($Knowledge\ manager$) qui est un élément contenant (a minima) les informations collectées par le moniteur. Ces informations peuvent être consultées, mises à jour et complétées par tous les éléments de la boucle MAPE. L'autre différence réside dans le fait que les activités de la boucle MAPE sont effectuées par des éléments distincts, ce qui n'est pas forcément le cas de la boucle MAPE-K.

Nous avons explicité le fonctionnement des systèmes adaptatifs à présent, nous abordons la modélisation des systèmes à composants.

2.2/ ARCHITECTURE À COMPOSANTS

Dans cette section, nous introduisons le concept de composant en explicitant la notion d'architectures orientées composants. Ensuite, nous présentons le modèle à composants Fractal servant de base à de nombreux modèles à composants. Enfin, nous présentons GCM (Grid Component Model) dont nous avons utilisé certaines fonctionnalités. GCM est un modèle dérivé de Fractal.

2.2.1/ LE CONCEPT DE COMPOSANTS

Les systèmes adaptatifs sont capables de s'adapter à leur environnement [3, 18, 27]. Afin d'appréhender la complexité de ces systèmes, la question de leur modélisation se pose.

Il n'y a pas de définition unique de composant, et il existe de nombreux modèles à composants qui différent les uns des autres en fonction de leur domaine d'application.
Nous avons choisi un modèle architectural proche de ceux définis dans [97, 48], qui s'inspirent de la représentation basée sur les graphes dans [102], concernant le modèle à composants Fractal.
Certains concepts que nous présentons dans les prochaines sections sont communs à la plupart des modèles à composants. Nous commençons par présenter les notions de composants primitifs et composites, puis nous nous intéressons à la façon dont les composants sont liés entres-eux en explicitant le concept d'interface. Enfin, nous montrerons comment s'effectue l'assemblage d'un système à composants en utilisant la notion de composition.

2.2.1.1/ COMPOSANTS PRIMITIFS ET COMPOSITES

La plupart des définitions s'accordent sur le fait qu'un composant soit une entité autonome capable de fournir un service. De plus, le développement et l'assemblage des composants étant indépendants, un même composant peut être utilisé dans différents contextes, au sein de diverses applications. C'est pourquoi, dans les approches à base de composants, deux types de composants existent :

— les composants composites, qui sont des composants qui contiennent d'autres composants appelés sous-composants, et,

— les composants primitifs, qui sont des composants qui ne contiennent pas de sous-composants et qui, par exemple, peuvent exécuter du code ou être liés à un objet.

La figure 2.3 illustre ces deux notions. Les composants $C3$, $C4$, $C5$ sont des composants primitifs. Le composant C2 est un composant composite qui contient $C3$ et $C4$. Le composant C1 est un composant composite qui contient le composant composite $C2$ et

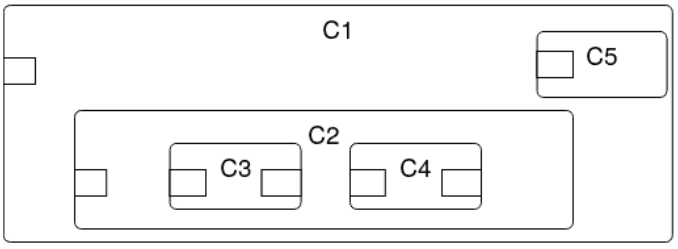

FIGURE 2.3 – Exemple de composant composite

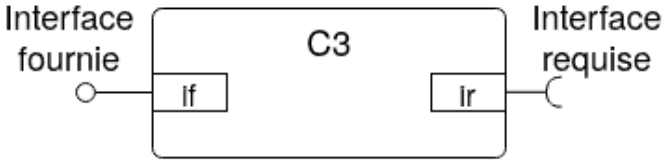

FIGURE 2.4 – Interfaces fournie et requise d'un composant

le composant primitif $C5$.

Ainsi, les composants composites peuvent encapsuler des composants primitifs ou d'autres composants composites, ce qui permet de :

— masquer certains détails dans une architecture logicielle,

— faciliter la compréhension du système,

— permet de considérer le système à un haut niveau d'abstraction et,

— réutiliser directement des assemblages de composants.

2.2.1.2/ INTERFACES

Dans une architecture à composants, les interfaces d'un composant sont les seuls points d'accès depuis ou vers l'extérieur. Les seules informations nécessaires à l'utilisation du composant sont les spécifications de ses interfaces ; celles-ci peuvent prendre la forme d'une description des propriétés et dépendances fonctionnelles ou des services fournis par le composant [13]. Ainsi, il n'est pas nécessaire de connaître le contenu d'un composant (qu'il s'agisse de code dans le cas d'un composant primitif ou d'autres composants dans le cadre d'un composant composite) pour pouvoir l'utiliser.

Lorsqu'une interface est utilisée par un composant pour fournir un service, on parlera d'interface fournie (provided interface) ou d'interface serveur (par analogie à une interaction de type client-serveur). Si, en revanche, une interface est utilisée pour satisfaire une dépendance, on parlera d'interface requise (required interface) ou d'interface client. La figure 2.4 illustre cette notion d'interface, où if est une interface fournie et ir une interface

requise.
D'après [121], une interface est composée de deux éléments :

— les ports, qui sont des points de connexion entre le composant et son environnement, et,

— les services, qui décrivent le comportement fonctionnel du composant en exprimant la sémantique des fonctionnalités (requises ou fournies).

Les ports requis permettent de recevoir des services en provenance de l'environnement alors que les ports fournis permettent de fournir un service aux autres composants. Il est possible que plusieurs ports soient utilisés par un même service.
Des formalismes liés aux contrats sont souvent utilisés dans le cadre des composants pour spécifier plus précisément le comportement d'un composant. Différents niveaux de contrats existent et sont classifiés dans les travaux proposés dans [15]. Quatre niveaux sont distingués :

— le niveau structurel porte sur les signatures et correspond à la spécification d'interfaces des objets ;

— le niveau fonctionnel utilise des invariants ou des pré-conditions et post-conditions [80] pour exprimer des propriétés sur les données qui transitent par les interfaces ;

— le niveau comportemental permet d'exprimer des propriétés sur le comportement entre les différents composants qui interagissent ensembles qui peuvent être spécifiées à l'aide de formalismes tels que CSP [81], et ;

— le niveau non-fonctionnel permet d'exprimer des propriétés non-fonctionnelles telles que, par exemple, des propriétés liées à la qualité de service.

2.2.1.3/ ASSEMBLAGES DE COMPOSANTS

Afin de construire un système à composants donné, nous effectuons la composition des interfaces des divers composants au moyen de liens logiques. Ces liens connectent les composants et synchronisent les services requis et fournis d'un composant avec les services requis et fournis par d'autres composants. Il est nécessaire que tous les connecteurs entre les composants d'un système satisfassent les différents contrats exprimés au niveau de leurs interfaces ; c'est pourquoi le mécanisme d'assemblage se doit d'intégrer la vérification de cette compatibilité. Nous considérons deux types de liens :

— des liens de type "client-serveur" où l'interface requise d'un composant "client" est connectée à l'interface fournie d'un composant "serveur", et

— des liens représentant une relation de délégation qui permet de lier une interface d'un composant composite avec une interface d'un de ses sous-composants ; on

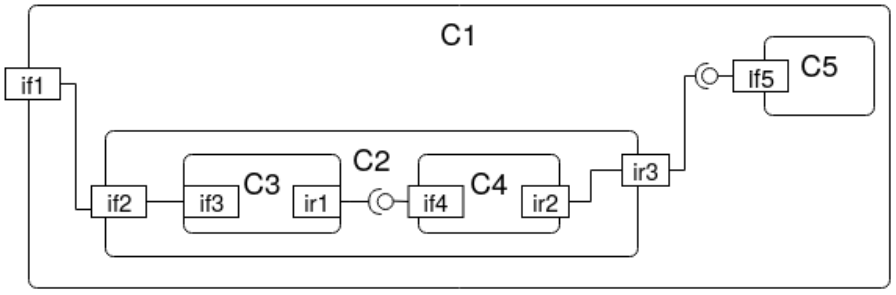

FIGURE 2.5 – Assemblage de composants

notera que les interfaces impliquées dans une relation de délégation sont toutes deux :

— soit fournies (cas d'un service exposé par un composant composite qui est fourni par un de ses sous-composants),

— soit requises (cas d'une dépendance d'un sous-composant qui peut être satisfaite directement, via un lien de type "client-serveur", ou non, via un lien de type délégation, par le composant composite qui le contient).

La figure 2.5 illustre l'assemblage des différents composants de la figure 1.2. Les liens entres les composants C3 et C4, d'une part, et C2 et C5, d'autre part, sont des liens "classiques" de type "client-serveur". Les liens entres C1 et C2, d'une part, et C2 et C3, d'autre part, sont des liens de type délégation n'impliquant que des interfaces fournies. Enfin, le lien entre C4 et C2 est de type délégation impliquant des interfaces requises. L'opération d'assemblage permet de créer un système complet ou un sous-composant d'un système plus complexe.

2.2.1.4/ CHOIX DU MODÈLE DE CONFIGURATION

Il existe plusieurs modèles à composants différents. Dans [78], un composant est défini comme un élément logiciel conforme à un modèle à composants qui définit des standards de composition et d'interaction.
Avant de présenter le modèle à composant Fractal que nous avons utilisé dans nos travaux, nous mentionnons ci-dessous quelques autres approches de modèles à composants.

EJB : Les EJB [42] (Entreprise JAVA Beans) ont des objets JAVA dédiés à la construction d'application J2EE (JAVA 2 platform, Entreprise Edition). Ils permettent de définir une architecture à composants où les composants sont exécutés sur un serveur et appelés par un client distant. Les EJB ont une partie métier contenant le code des composants logiciels appelés *Beans* et une partie conteneur (*container*) qui peut être vue comme un composant composite qui implémente les interfaces nécessaires à la communication entre

les EJB et les différents services disponibles au sein d'un serveur J2EE.

CCM : Le modèle CCM [155] (CORBA Component Model) est un modèle à composants standardisé par l'OMG (Object Management Group). CCM supporte les applications distribuées orientées objet et développées selon le modèle client-serveur en se basant sur l'appel de méthodes distantes (RPC). Les composants CCM possèdent des attributs configurables pour permettre leur réutilisation et leur configuration ; ils exposent à leur environnement des services synchrones fournis et requis (facettes et receptacles) ainsi que des sources et des puits d'évènements pour le support de services asynchrones.

Les services Web : L'architecture SOA [122] (Service Oriented Architecture) utilise les technologies Web afin d'établir et de gérer les communications entre ses composants appelés services Web. SOAP [1] (Simple Object Access Protocol) est le protocole de communication qui permet de définir les formats des messages échangés entre les composants. Plusieurs standards comme WSDL [2] (Web Service Description Language) ; BPEL (Business Process Execution Language) et WSCL [3] (Web Services Language) sont utilisés pour décrire les caractéristiques de services Web.

.NET : La plateforme .NET [4] de Microsoft se compose d'un ensemble de produits de développement et d'exécution. Cette plateforme hétérogène supporte différents langages de programmation tels que C++, C# ou VB.NET. Les composants .NET sont appelés *assembly* et sont des bibliothèques ou des exécutables qui sont composés de fichiers d'implémentation contenant du code exécutable.

OSGi : Le modèle à composants OSGi [5], basé sur l'exécution de services JAVA, peut être exécuté sur des systèmes dont la mémoire est limitée et permet de télécharger, mettre à jour et de supprimer dynamiquement les composants sans stopper le système.

BIP : Le langage appelé Behavior Interaction Priority (BIP) [7], comme son nom l'indique, résulte de trois couches : les comportements spécifiés sous la forme d'ensembles de transitions, les interactions entre les comportements qui sont possibles grâce aux ports et connecteurs de différents types, et les priorités qui permettent de choisir entre différents comportements. BIP propose également des composants primitifs (appelés *atomic components*) et des composants composites (appelés *compound components*). Les connecteurs entre les ports peuvent être en broadcast ou en rendez-vous. Le langage propose des mécanismes pour structurer les systèmes adaptatifs tels que des priorités dans les interactions entre composants. Ce langage supporte également les reconfigurations dynamiques grâce à son extension DR-BIP [52, 53].

Frascati Le modèle Frascati [138, 139] est une implémentation open source de la spéci-

1. https ://www.w3.org/TR/soap
2. https ://www.w3.org/TR/wsdl
3. https ://www.w3.org/TR/wscl10/
4. https ://dotnet.microsoft.com/
5. https ://www.osgi.org/

fication SCA (Service Component Architecture). Le coeur de la spécification SCA décrit un langage d'assemblage de composants (component assembly language) indépendant des IDL (Interface Definition Languages), des protocoles de communication ou d'autres propriétés non-fonctionnelles [12]. Ainsi, comme le suggère [145], SCA permet de tirer partie d'un large spectre de technologies permettant l'implémentation de composants logiciels et de leurs services (e.g., JAVA, Python, Scala, PHP, etc.) afin de les connecter (au moyen de Web services, SOAP, REST, RPC, RMI, etc).

Event-B Comme en attestent les travaux [5, 4], il est possible d'utiliser la modélisation d'Event-B [6] pour les systèmes cyber-physiques ou l'internet des objets. Cette approche propose une structure générique pour la modélisation formelle de ces systèmes. Les propriétés d'invariance permettent de garantir la cohérence architecturale du système. Afin de prendre en compte l'aspect évolutif de ces systèmes les auteurs proposent une application de contrôle du comportement du logiciel envoyant des ordres en fonction des données reçues par les capteurs du système.

UML La modélisation reposant sur UML (Unified Modeling Language) possède plusieurs avantages. Premièrement, le langage UML (Unified Modeling Language) est représenté et utilisé dans un grand nombre de projets ce qui en fait un langage compréhensible par une large majorité des utilisateurs dans le domaine. Deuxièmement, le langage UML est facile à utiliser [62] de par son aspect graphique, et l'utiliser permet de concevoir rapidement un modèle du système. Cependant, la grande capacité d'expressivité du langage le rend plus complexe à utiliser qu'un modèle à composants dédié pour la modélisation de système cyber-physiques adaptatifs. Il est à noter que le langage UML possède des extensions et des intégrations permettant la modélisation des systèmes cyber-physiques adaptatifs, sans avoir besoin d'utiliser un langage complémentaire.

> **ModelicaML** est une tentative d'intégrer UML et Modelica [7] (un langage non propriétaire, orienté objet, qui permet de modéliser des systèmes physiques complexes) pour la modélisation et la simulation des exigences et de la conception des systèmes [135].
>
> **SysML** est un profil UML 2.0, spécialisé dans les applications d'ingénierie des systèmes [66]. Un effort est en cours pour intégrer Modelica avec SysML pour la modélisation du domaine physique [68].

SysWeaver [41] est un outil de développement à base de modèles qui comprend un schéma de génération de code flexible pour les systèmes distribués en temps réel. Cet outil utilise Simulink [89], un outil de conception à partir de modèle permettant de simuler les systèmes modélisés. Les aspects fonctionnels du système sont spécifiés dans

6. http://www.event-b.org/
7. https://modelica.org/

Simulink et traduits dans un mode SysWeaver.

Comme nous pouvons le voir, chaque modèle propose des définitions différentes permettant de s'appliquer dans des domaines propres à chacun. Cependant, chaque modèle présente des faiblesses lorsqu'il doit exprimer des cas en dehors de son domaine. Pour ces raisons, nous avons décidé de proposer un modèle à composants plus générique laissant ainsi le maximum de libertés en terme de choix de domaines d'application. Ce modèle s'inspire des concepts de Fractal et de GCM que nous présentons dans les sous-sections suivantes.

2.2.2/ LE MODÈLE À COMPOSANTS FRACTAL

Nous présentons dans cette section le modèle à composant Fractal qui a servi de base pour notre modèle. Le modèle Fractal est très utilisé pour la conception de modèles à composants et il existe de nombreuses versions qui enrichissent le modèle de base. Le modèle à composants Fractal permet classiquement la définition d'un assemblage de composants grâce à la liaison entre des interfaces. Cependant, les composants sont constitués d'une membrane et d'un contenu. Les composants sont constitués de deux types d'interfaces :

— Les interfaces métier qui correspondent aux points d'accès au composant. Elles correspondent aux interfaces requises et fournies dont nous avons parlé précédemment.

— Les interfaces de contrôle qui prennent en charge les capacités non-fonctionnelles de Fractal, c'est-à-dire l'introspection et la configuration du composant. Cela correspond par exemple au démarrage ou à l'arrêt des composants, au paramétrage d'attributs de composants, mais aussi à la reconfiguration dynamique par l'ajout et la suppression de composants ou de liaisons entre les composants.

Une liaison Fractal est définie classiquement comme une connexion entre des interfaces de deux composants. Notons que le type [8] d'une interface fournie doit être obligatoirement du type ou du sous-type de l'interface requise à laquelle elle est reliée.

La figure 2.6, issue du tutoriel "HelloWorld with Fraclet and Fractal ADL" [9], représente l'exemple du composant *Helloworld* souvent utilisé dans le cadre de Fractal. Cet exemple montre les différentes interfaces et liaisons existantes pour décrire l'assemblage d'une architecture à composants avec un composant Client et un composant Server.

Fractal met à disposition un ensemble d'interfaces de contrôle permettant l'accès à des contrôleurs. Ces contrôleurs proposés par Fractal permettent de gérer le

8. Par exemple, dans le cadre de l'implémentation de Fractal utilisant JAVA, le type d'une interface correspond à une interface JAVA

9. https ://fractal.ow2.io/fractal-distribution/helloworld-julia-fraclet/user-doc.html

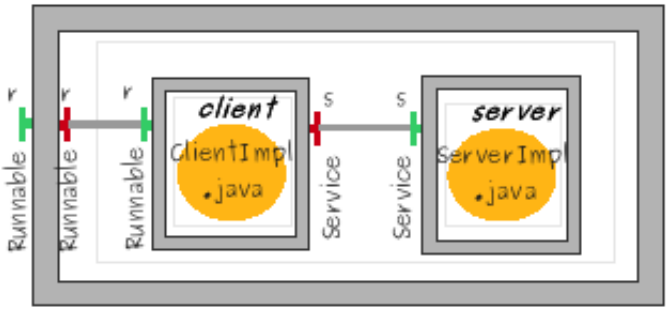

FIGURE 2.6 – Composant *Helloworld* en Fractal

cycle de vie (*LifeCycleController*), les liaisons (*BindingController*) ainsi que des attributs (*AttributeController*) pour les composants primitifs, ou des sous-composants (*ContentController*) pour les composants composites. Grâce à ces contrôleurs, le caractère dynamique des composants Fractal et les reconfigurations dynamiques de l'architecture d'une application en cours d'exécution sont possibles. Les interfaces de contrôle standard fournies par les composants Fractal permettent ainsi de créer de nouveaux composants, de modifier le contenu des composites en y ajoutant ou en retirant des sous-composants, et de créer ou supprimer des connexions entre interfaces.
Il est également possible de définir, au moyen d'un fichier au format XML, la façon dont un composant composite est assemblé en utilisant le langage Fractal ADL (Architecture Definition Language). La figure 2.7 (issue, comme la figure 2.6, du tutoriel "HelloWorld with Fraclet and Fractal ADL") montre la syntaxe utilisée pour spécifier un lien de type "client-serveur" et un lien de délégation en utilisant Fractal ADL. On notera que même pour le lien de délégation, la syntaxe utilise les termes client et server. Comme ce modèle est extensible, un aspect intéressant est que l'utilisateur a la possibilité de définir ses propres interfaces de contrôle. Il peut donc mettre en place ses propres contrôleurs qui vont pouvoir, par exemple, observer le système ou même agir sur son exécution.

Implémentations Fractal est un modèle à composants indépendant des langages de programmation. Plusieurs plateformes sont ainsi disponibles dans différents langages de programmation. Elles permettent d'étendre le modèle à composants Fractal et répondent à un besoin particulier. L'implémentation de référence du modèle est la plateforme Julia [20] écrite en Java. L'objectif principal de cette implémentation est la définition de contrôleurs qui sont présents dans la membrane. Les contrôleurs notés C, BC, CC et $ServiceAttributes$ sur la figure 2.7 correspondent respectivement au $LifeCycleController$, $BindingController$, $ContentController$ et $AttributeController$. Julia permet aussi l'ajout de contrôleurs particuliers qui sont des intercepteurs. Ces intercepteurs permettent de réagir à l'appel des méthodes sur les interfaces métiers des composants. D'autres implémenta-

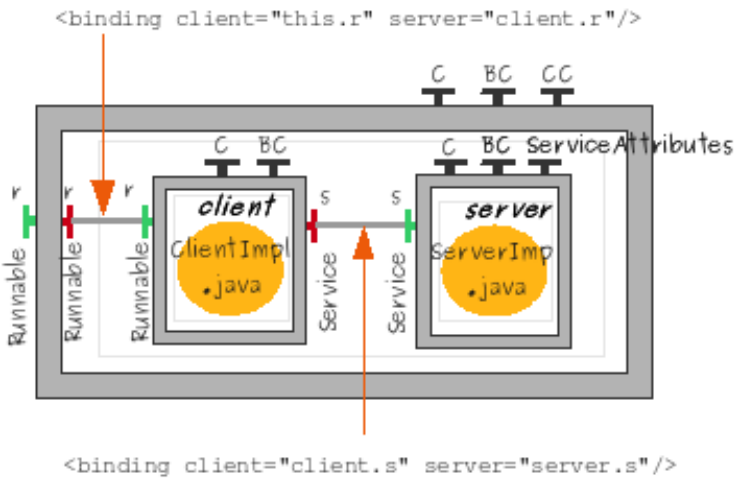

FIGURE 2.7 – Spécification de liens entre interfaces en utilisant Fractal ADL

tions existent en Java telles que AOKell [10] ou ProActive [11]. AOKell est une implémentation de Fractal utilisant la programmation par aspects. ProActive a pour but de permettre le développement de composants actifs sur les grilles de calcul. D'autres langages sont aussi utilisés pour implémenter Fractal : le langage C avec Think [56] et Cécilia [12] pour l'implémentation de systèmes embarqués et de systèmes d'exploitation, .NET (FracNet) ou encore smalltalk (FracTalk).

Nous avons choisi le modèle le plus à même de répondre à notre besoin de générer automatiquement des instances du même type de composant à savoir GCM qui est une extension de Fractal.

2.2.3/ LE MODÈLE À COMPOSANTS GRID COMPONENT MODEL (GCM)

Le modèle GCM [9, 8, 10] (basé sur la librairie Java ProActive) a été défini par le réseau d'excellence européen CoreGRID, qui rassemble des chercheurs dans le domaine des technologies de grille et de pair à pair. Il s'appuie sur les aspects suivants, hérités des travaux de Fractal comme base de l'architecture des composants :

— hiérarchie (les composants composites peuvent contenir des sous-composants), pour avoir une vue uniforme des applications à différents niveaux d'abstraction ;

— capacités d'introspection, pour surveiller et contrôler l'exécution d'un système en cours d'exécution ;

— capacités de reconfiguration, pour configurer dynamiquement un système.

10. https ://fractal.ow2.io/aokell/
11. http ://proactive.inria.fr/
12. https ://mvnrepository.com/artifact/org.objectweb.fractal.cecilia

Parmi les caractéristiques centrales du GCM, nous présentons brièvement les plus innovantes :

- support du déploiement : les composants distribués ont la capacité être déployés sur divers systèmes hétérogènes ;
- prise en charge des communications un à plusieurs (*multicast*), plusieurs à un (*gathercast*) et plusieurs à plusieurs (*broadcast*) : souvent, les applications Grid se composent d'un grand nombre d'instances de composants de mêmes types qui peuvent être adressés en tant que groupe et qui peuvent communiquer ensemble de manière très structurée ;
- support pour l'adaptabilité non-fonctionnelle et le calcul autonome : Grid est un environnement hautement évolutif et les applications Grid sont capables de s'adapter à ces conditions d'exécution changeantes ;
- GCM permet également aux interfaces non fonctionnelles de supporter la composition d'aspects extra-fonctionnels.

L'expressivité du modèle GCM est moins fine que celle du modèle Fractal, mais elle permet une extensibilité, une base théorique solide et une aptitude à l'optimisation et aux implémentations compétitives ce qui a pour effet de réduire la complexité du système. Pour nos travaux, l'ajout majeur de GCM est qu'il permet de créer plusieurs instances du même type de composant contrairement à Fractal dans lequel chaque composant possède son type.
En plus de gérer les instances de composants, GCM gère les interfaces *Gathercast*, *Multicast* et *BroadCast* :

- Les interfaces *Gathercast* rassemblent les invocations provenant de plusieurs composants sources.
  Elles doivent coordonner les invocations entrantes en définissant des barrières de synchronisation.
  Pour la synchronisation, nous avons besoin de connaître les participants, donc les invocations aux interfaces *Gathercast* sont des liens bidirectionnels.
  La barrière de synchronisation peut être définie sur des invocations spécifiques pour l'instance avec un délai d'attente possible.
- Les interfaces *Multicast* envoient leur invocation à plusieurs composants destination.
- Les interfaces *Broadcast* rassemblent les fonctionnalités des interfaces *Gathercast* et *Multicast* et peuvent donc à la fois rassembler et envoyer à plusieurs composants des invocations.

Une illustration des interfaces de GCM est visible dans la figure 2.8.

Une fonctionnalité manquante à GCM est la souplesse dans la création et la suppression des composants à l'exécution. Il aurait été possible de garder ce modèle en créant à

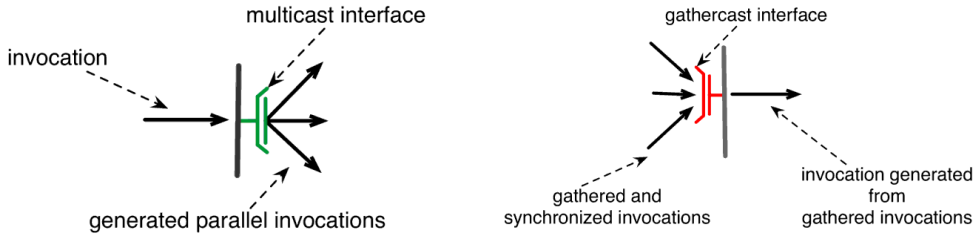

FIGURE 2.8 – Interfaces multicast et gathercast

l'avance une grande quantité de composants en attente à lier au moment de les utiliser. Cependant, nous pensons que cette approche aurait réduit l'efficacité de GCM.

Nous allons à présent détailler les méthodes de validation de ces systèmes à composants.

2.3/ VALIDATION DE SYSTÈMES ADAPTATIFS AVEC DES TESTS

Dans la section précédente, nous avons indiqué comment définir une architecture à composants.

La validation et la vérification des logiciels (V&V) concernent l'évaluation de la qualité des systèmes logiciels tout au long de leur cycle de vie [40]. L'objectif est de s'assurer que le système logiciel satisfait à ses exigences fonctionnelles et répond aux critères de qualité attendus.

Cependant, les systèmes adaptatifs en ligne en général, ne peuvent pas être validés à l'aide des techniques traditionnelles de vérification et de validation, car ils évoluent dans le temps [111].

Cette problématique est l'objet de cette thèse. Ainsi, nous présentons dans cette section le contexte scientifique permettant de répondre aux questions de recherches présentées en section 1.2.1.
Nous commençons par le contexte associé à la RQ1 : "Quels objectifs de test définir pour valider les systèmes adaptatifs ?". Ensuite, nous passons au contexte associé à la RQ2 : "Comment construire les tests ?". Nous finissons avec le contexte associé à la RQ3 : "Comment établir le verdict de test ?".

2.3.1/ ÉVALUATION DE LA QUALITÉ DES SÉQUENCES DE TEST

Malgré des décennies de travail par les chercheurs et les praticiens sur de nombreuses techniques d'assurance qualité des logiciels, les tests restent l'une des approches les

plus largement pratiquées et étudiées pour évaluer et améliorer la qualité des logiciels [120]. C'est pourquoi il est important d'évaluer la qualité des séquences de tests car c'est sur celles-ci que repose l'évaluation de la qualité du système en lui-même.
Dans un premier temps, nous présentons les critères statiques puis nous présentons les critères dynamiques.

2.3.1.1/ CRITÈRES STATIQUES

Les critères de sélection statiques sont des critères qui reposent sur la structure du modèle. Leur objectif est d'assurer un besoin de couverture de certains éléments du modèle sans s'appuyer sur des besoins de tests spécifiques.

Les critères statiques peuvent s'appliquer à différents endroits du processus en mesurant la couverture d'éléments spécifiques. Nous nous intéressons principalement aux critères de couverture des états/transitions. Ces critères sont principalement utilisés pour les modèles à base de transitions tels que les systèmes de transitions étiquetés ou les statecharts UML. Il est également possible que le système ait plusieurs états actifs en même temps, un par automate mis en parallèle.
Les critères de couverture état/transition sont les suivants :

- Le **all-states coverage** qui nécessite que chaque état du système soit parcouru au moins une fois.
- Le **all-transition coverage** nécessite que chaque transition du système doit être couverte au moins une fois.
- Le **all-transition-pairs coverage** Chaque paire adjacente de transitions doit être couverte au moins une fois. Il s'agit de l'ensemble des produits cartésiens entre les transitions entrantes et sortantes à chaque état.
- Le **all-loop-free coverage** nécessite que tous les chemins qui ne repassent pas par un état déjà traversé doivent être couverts.
- Les critères **all-loop-free coverage et all-one-loop coverage** qui nécessitent également la couverture des chemins avec au plus une boucle. Le premier critère requiert en plus la couverture des préfixes de chaque boucle.
- Pour finir le **all-path coverage** est le critère le plus exhaustif puisqu'il nécessite la couverture de tous les chemins. Ce critère donne facilement lieu à des chemins infinis et donc non-couvrables dans la pratique. En effet, en ajoutant une transition dont l'état de départ est l'état d'arrivée un nouveau chemin est créé à chaque passage de cette transition. Cela a pour effet de créer des chemins infinis qui ne sont pas couvrables dans la pratique.

  Pour palier à ce problème le critère **all-k-path coverage** permet de limiter le passage de boucles à k fois et demande la couverture des itérations de 0 à k.

Nous nous inspirons des travaux [151], où les auteurs décrivent une technique de génération de données de test à partir de statecharts UML. Ils proposent d'appliquer les critères $all - transition\ coverage$ et $all - transition\ pairs\ coverage$ pour la couverture des transitions des statecharts, et le critère $full - predicate\ coverage$ sur les gardes de transitions pour plus de finesse.

2.3.1.2/ CRITÈRES DYNAMIQUES

Les critères dynamiques, par opposition aux critères statiques, sont des critères sur l'aspect temporel du système et représentent des enchaînements particuliers d'opérations ou d'états du système. Nous distinguons deux catégories de critères dynamiques : les scénarios et les propriétés du système. Les scénarios sont l'expression de séquences particulières, d'opération ou d'états, sur le système définis par l'ingénieur de test. Les propriétés sont des exigences au niveau de la spécification que le système doit respecter. Nous décrivons maintenant les critères dynamiques sur les propriétés du système.

Les travaux [55] présentent la vérification en ligne comme une technique de validation complémentaire pour les systèmes à base de composants adaptatifs écrits dans le cadre de BIP [7]. Cette technique est basée sur un cadre de vérification d'exécution général et expressif. Il construit dynamiquement une abstraction minimale de l'état d'exécution actuel du système afin de réduire l'impact sur les performances. En générant des moniteurs directement en tant que composants BIP, l'approche décrite est en mesure de générer des programmes contrôlés réels.

Les travaux [127] présentent une approche itérative pour construire et analyser des modèles de conception UML pour les systèmes adaptatifs. Cette approche vérifie trois types de propriétés clés. Les propriétés locales qui doivent être valables dans un domaine particulier. Les propriétés transitoires qui doivent être respectées pendant l'adaptation. Les propriétés globales ou invariants qui doivent être présents à tout moment. Ainsi, en séparant les préoccupations au niveau du modèle, la complexité de la vérification de l'exactitude du modèle des programmes adaptatifs est réduite.

Il existe de nombreuses technologies d'assurance spécifiques au systèmes adaptatifs qui tirent parti des informations obtenues lors de d'exécution. Les tests continus [71, 117] peuvent être exécutés sans surveillance, pendant une longue période (par exemple, pendant la nuit). Ainsi, le système peut être mis à l'épreuve dans un grand nombre de scénarios et de conditions différents, ce qui serait impossible à réaliser par des tests manuels. Pour les fautes impliquant de longues séquences de messages et des données d'entrée spécifiques, le test continu semble particulièrement adapté pour explorer ces états.

Les travaux de [64] introduisent MAPE-T, une boucle de rétroaction pour compléter les

stratégies de test avec des capacités d'exécution basées sur la surveillance, l'analyse, la planification et l'exécution pour les systèmes adaptatifs.

Les propriétés temporelles d'un système sont d'autres éléments de la spécification permettant d'évaluer les tests. Ici, nous considérons une propriété temporelle au sens large, c'est-à-dire qu'il s'agit de la représentation d'un sous-ensemble d'exécutions du système qui respectent certains enchaînements d'opérations.

Il est possible de procéder par une spécification logique à l'aide des propriétés temporelles décrites avec des opérateurs temporels tels que la LTL ou la CTL, ou par une spécification utilisant des automates.
Dans [83], les auteurs décrivent une approche de couverture de flot de données sur des systèmes à base de transitions à partir de propriétés temporelles décrites en CTL [54] et en utilisant le model-checker symbolique SMV [108].
Dans [49], les auteurs proposent un langage de patrons de propriétés temporelles. Cette étude part du principe que la majorité des propriétés temporelles considérées dans le test logiciel découlent d'un ensemble restreint de patrons paramétrables. Ainsi, pour les auteurs, les propriétés temporelles sont définies comme étant la composition d'une portée (scope), qui représente la portion d'exécution du système considérée pour la propriété et d'un motif (pattern) qui est une propriété devant être continuellement satisfaite dans une portée donnée.
Dans [63], les auteurs présentent une technique permettant de caractériser la pertinence d'un cas de test vis-à-vis d'une propriété temporelle par le biais d'un critère de pertinence (property relevance criterion). Les auteurs utilisent un modèle sous la forme d'une structure de Kripke et des propriétés temporelles en LTL, et partitionnent les cas de test selon deux catégories : les cas de test positifs, qui détectent une faute si leur exécution échoue, et les cas de tests négatifs, qui détectent une faute si leur exécution passe avec succès. La pertinence d'un cas de test négatif est établie si le cas de test est un contre-exemple de la propriété sur le modèle (tel que l'on pourrait obtenir avec un model-checker). La pertinence d'un cas de test positif est établie si le cas de tests peut échouer et produit une violation de la propriété. Ainsi, si l'exécution d'un test échoue mais ne provoque pas la violation de la propriété considérée, alors il n'est pas considéré comme pertinent. En s'appuyant sur ces définitions, les auteurs proposent un critère de pertinence pour un test vis-à-vis d'une propriété, qui permettra ensuite d'évaluer une suite entière vis-à-vis de la propriété.
Dans [109] les auteurs représentent le système comme un graphique de flux sur lesquels ils établissent des mesures pour vérifier que les transitions des propriétés du système et des politiques d'adaptation sont parcourues. Les traces utilisées visent à satisfaire des objectifs de tests également appelés critères de couverture. Ces traces sont obtenues en stimulant le système à l'aide de données de tests. Les traces d'exécution sont alors analysées : les propriétés, sont validées pendant l'exécution du système sous test. L'ana-

lyse à l'exécution ou à posteriori des traces permet également de détecter de potentielles erreurs sur le respect des règles d'adaptation.
Dans le domaine des systèmes à composants, des tests en ligne sont utilisés dans [74] pour détecter des violations de propriétés pour éventuellement revenir à une configuration normale si la reconfiguration effectuée viole une des propriétés. Plus particulièrement, plusieurs articles utilisent les tests intégrés pour les systèmes à composants [153, 72], avec un ciblage sur l'architecture des systèmes à composants.
Dans [21], les auteurs utilisent une approche par test à partir de modèles dans laquelle le système sous test est stimulé par un automate probabiliste (un MDP : Markov Decision Process). Les états du modèle sont liés au comportement du système dans le but de générer des actions qui conduisent au comportement ciblé.

Notre approche utilise les critères dynamiques en ligne basés sur la couverture des propriétés temporelles ainsi que des opérations de reconfiguration.

2.3.2/ CONSTRUCTION DE TESTS

Cette section se décompose en deux parties, la première détaille l'état de l'art pour le calcul d'états initiaux et la deuxième détaille l'état de l'art sur la génération d'évènements externes pour le système.

Le test fonctionnel est une étape importante du cycle de vie des logiciels [14]. La nécessité de tester un produit est liée au besoin de valider que les fonctionnalités décrites dans les spécifications sont mises en œuvre exactement comme prévu. Les tests sont réalisés en appliquant des cas de test à une implémentation testée, en faisant des observations pendant l'exécution des tests et en attribuant ensuite un verdict sur l'exécution concrète de l'implémentation. La conception des tests en boîte noire dans l'industrie est rarement soutenue par des outils et repose sur l'expertise des ingénieurs de validation. Par conséquent, la subjectivité est considérable dans la création des ensembles de tests. La conception manuelle des tests fonctionnels est généralement basée sur un examen informel de la spécification du système.

L'utilisation de tests en ligne est plus appropriée que les tests hors ligne pour les systèmes adaptatifs qui se reconfigurent au moment de l'exécution. Les tests en ligne combinent la génération et l'exécution des tests : le cas de test est généré à un moment donné, puis exécuté immédiatement [100].

Initialisation d'un système Dans le test en ligne, les configurations sont souvent générées automatiquement à partir d'un modèle à composants en utilisant les méthodes de test aux limites pour générer les valeurs du système comme dans [101]. Les auteurs présentent une génération de cas de test basée sur des objectifs de domaines de défini-

tion avec un retour possible pour raffiner les critères des limites pour les objectifs qui ne sont pas atteignables. Dans [86], les délimitations des paramètres sont définies dans le modèle.

Dans notre approche, les limites des paramètres sont définies dans le modèle de la même façon que dans [86], qui consiste à générer une diversité de combinaisons de paramètres pour le système. Notre contribution consiste à proposer de générer ces combinaisons selon plusieurs distributions probabilistes afin de diversifier les configurations initiales ainsi que les cas de tests qui en résultent.

Dans [163], les auteurs proposent un compilateur architectural pour les systèmes adaptatifs permettant de générer des configurations architecturales. Ce compilateur utilise KAOS [35, 104], un modèle fonctionnant à partir de buts et d'exigences. Dans la même approche les travaux de [114] permettent de générer des dizaines de configurations ayant chacune des centaines de buts générés aléatoirement. Il est donc possible de générer de manière automatique des configurations pour systèmes adaptatifs.

Les travaux de [107] proposent l'initialisation de variables à partir de domaines. Pour cela, ils utilisent des contraintes qui sont vues comme des relations logiques entre plusieurs inconnues (ou variables), chacune prenant une valeur dans un domaine donné de valeurs possibles, où un domaine est un ensemble de valeurs possibles qu'une variable peut prendre. La programmation par contraintes est un paradigme de programmation dans lequel les contraintes entre les variables sont définies de manière déclarative, et où les valeurs des variables sont déterminées à l'aide d'un solveur. La programmation par contraintes classique traite des domaines finis pour les variables, qui sont généralement mis en correspondance avec les domaines de l'environnement. La résolution des contraintes implique d'abord la réduction des domaines de variables par des techniques de propagation [136] qui éliminent les valeurs incohérentes à l'intérieur des domaines, puis à trouver des valeurs pour chaque variable contrainte. Grâce à cette méthode, la vérification et l'évaluation du modèle transformé sont très efficaces.

Les travaux dans [70], visent à améliorer la mise à l'échelle pour les systèmes avec reconfigurations dynamiques en procédant par génération automatique de configurations initiales.

Dans [67], les auteurs proposent de formaliser les configurations de systèmes cyber-physiques. Cependant, leur approche se focalise sur la capacité du système à continuer de fonctionner en cas de problème car arrêter un système cyber-physique peut s'avérer coûteux. Notre approche est focalisée sur la simulation des systèmes, pour cela nous formalisons les configurations de systèmes pour automatiser la génération de configurations afin de favoriser l'analyse du comportement du système.
L'automatisation de génération [144], présente une méthode pour générer une configuration de blocs (équivalents à des composants dans notre approche) et de matrices de

connexion (équivalents à des interfaces dans notre approche). Cette méthode procède en associant les blocs et les matrices grâce à un algorithme basé sur la théorie des graphes et s'applique aux FPGA (Field Programmable Gate Arrays).

Nous proposons une initialisation automatique du système fonctionnant à partir d'un modèle architectural et d'un ensemble de contraintes.

Génération de séquences de tests La génération de séquences de tests à partir d'une formalisation des exigences d'un système, s'intègre bien dans le contexte des systèmes adaptatifs. L'objectif d'une telle approche est de décharger par automatisation le travail de génération de test et de déplacer l'effort vers l'activité de spécification [115].

Dans [50], les auteurs présentent une procédure de génération de cas de test de manière automatique à l'aide d'algorithmes génétiques.
Le test d'accessibilité (en particulier, le travail de Lei et Carver [103]) est une approche qui combine à la fois le test non déterministe [13] et le test déterministe [14] pour générer automatiquement toutes les séquences d'événements de synchronisation sans construire un modèle statique.
Dans [17], les auteurs implémentent une application appelée *Providentia* qui capture les différentes combinaisons du modèle basé sur des objectifs et détermine l'attribution optimale de poids pour chaque reconfiguration en utilisant un algorithme génétique. Le modèle est ensuite appliqué à l'exécution au système.

L'approche [101], propose un critère de parcours du graphe de flot de contrôle, la génération de tests permet de s'assurer que les variables d'état ont été initialisées en utilisant des paramètres d'opération aux bornes de leur domaine pour limiter l'explosion combinatoire. Dans un premier temps, à partir des préconditions et postconditions, BZ-TT [2] calcule les valeurs aux limites des variables qui constituent les buts à atteindre. Le moteur symbolique CLPS, fondé sur la théorie des ensembles, tente de trouver une séquence pour atteindre un état permettant à la variable de prendre la valeur limite considérée dans le but. Dans [16], les auteurs utilisent les annotations JML comme modèle pour déterminer les données de test pertinentes.

Dans [161], les auteurs utilisent une approche pour générer des cas de tests automatiquement pour les systèmes adaptatifs et valident à la fois les propriétés fonctionnelles et extra-fonctionnelles. Leur approche se base sur le fait que certaines étapes de validation peuvent dépendre d'informations qui ne sont disponibles qu'au moment de l'exécution du test et qui doivent être recueillies à l'aide de capteurs. Pour de telles situations, le

13. Le résultat n'est pas déterministe : chaque exécution de la même suite de tests a la possibilité de produire un résultat différent. Cette méthode de test consiste à exécuter le programme avec la même entrée plusieurs fois en espérant que les fautes seront révélées par au moins une des exécutions.
14. Dans ce cas la même suite de teste donnera le même résultat. Cette méthode consiste à modifier la suite de tests pour espérer produire un résultat différent et révéler les fautes.

mécanisme de validation doit garder en mémoire un modèle de décision afin d'ajuster la validation. En conséquence, le modèle de test est exécuté et ajusté au moment de l'exécution du test pour être utilisé comme entrée de simulation et son état est validé par rapport au système réel pendant l'exécution.

Dans [132], les auteurs spécifient un système adaptatif avec un modèle de buts à atteindre en fonction des choix du système (solutions alternatives). Un modèle de but est utilisé pour capturer et analyser les besoins et les intentions des parties prenantes, et pour représenter les exigences fonctionnelles et non fonctionnelles. Les solutions alternatives sont appelées points de variation ($variationpoint$) et sont utilisées pour savoir quels choix sont applicables dans une situation donnée. Cette approche requiert de pouvoir contrôler quelle reconfiguration doit être déclenchée à chaque instant, ce qui demande de pouvoir modifier directement le code du système sous test.

Les systèmes adaptatifs basés sur de multiples entités concurrentes, doivent décider comment allouer les ressources rares et comment définir les paramètres de qualité de chaque application pour satisfaire au mieux l'utilisateur. Dans [124], les auteurs proposent une approche d'adaptation, appelée configuration anticipative, qui exploite les prédictions de la disponibilité future des ressources pour améliorer l'utilité de l'opération pour l'utilisateur.

Les travaux [60] présentent une génération de test pour des systèmes à composants en temps-réel en modélisant chaque composant par un automate temporisé à entrées/sorties (timed input/output automaton). Des modifications sont effectuées sur les automates pour obtenir des séquences de tests contenant des dangers pour le système permettant d'évaluer sa robustesse.

Nous proposons également une génération automatique de tests en ligne à partir d'automates modélisant les évènement des composants. Dans notre approche, nous ne modifions pas les automates mais préférons que l'automate décrive les évènements en prenant en compte les reconfiguration du système. Pour cela, nous définissons des automates dont l'état courant peut être mis à jour grâce aux observations des changements de configuration du système.

2.3.3/ VERDICT DE TESTS

Le test et la vérification sont deux techniques complémentaires pour l'analyse et l'assurance de l'exactitude des systèmes. La vérification fournit des propriétés sur les systèmes basée sur des modèles mathématiques du système alors que le test s'effectue par stimulation du système réel. La vérification donne donc un verdict certain sur la validité de ses propriétés tandis que le test observe une partie du comportement du système. C'est pour cela que lorsque le modèle mathématique du système est trop complexe à concevoir les

verdicts de tests permettent de valider le comportement du système [148].

Dans [51], les auteurs cherchent à tester séparément les mécanismes d'adaptation. Dans cette approche, il est possible d'examiner des situations particulières, plus précisément quand les opérations de reconfigurations sont effectuées permettant ainsi de se conformer avec les politiques d'adaptation.

En utilisant les spécificités du modèle à reconfigurations [95] pour générer des données à envoyer au système, l'approche dans [156] permet au testeur de se focaliser sur des parties spécifiques de la séquence de configuration qui permet aux politiques d'adaptation d'être validées.

Les travaux [164] utilisent les réseaux de Petri et la logique temporelle LTL et se concentrent sur la validation des invariants, de la tolérance et de l'adaptation du système.

L'évaluation proposée dans [152] a pour but de tester la qualité des logiciels. Cette structure est basée sur un ensemble de propriétés d'adaptation liées à des attributs pour la qualité. Ces mesures peuvent être réutilisées pour évaluer des propriétés d'adaptation. Il est cependant possible que les structures temporelles ne soient pas directement liées aux métriques pour la qualité. L'approche dans [29] propose des moyens pour calculer ces métriques grâce à la mesure de couverture sur propriétés.

Les travaux de [28] présentent Rainbow qui est un framework pour l'ingénierie d'un système avec des capacités auto-adaptatives en temps réel pour surveiller, détecter, décider et agir sur les opportunités d'amélioration du système. Cette approche de l'auto-adaptation utilise des techniques basées sur l'architecture combinées à des théories de contrôle et d'utilité. Les propriétés surveillées d'un système en cours d'exécution sont reflétées dans un modèle d'architecture qui permet de raisonner automatiquement sur les changements appropriés pour améliorer la qualité de service dans le système cible. La définition des utilités est utilisée pour analyser les compromis entre les dimensions de la qualité et sélectionner l'adaptation appropriée. Les changements sont ensuite effectués dans le système, qui est réobservé dans une forme de contrôle en boucle fermée. Cette observation est faite grâce à Znn.com, un système fonctionnant en ligne qui imite les systèmes du monde réel avec un environnement expérimental pour faciliter l'évaluation. Znn.com permet de :

— surveiller et effectuer des changements sur le système,

— fournir des interfaces pour que Rainbow puisse s'y connecter,

— produire un environnement dans lequel il est possible d'injecter des problèmes à résoudre,

— enregistrer le comportement du système.

L'évaluation du système est faite sur les capacités d'adaptation du système qui peuvent s'apparenter à des propriétés extra-fonctionnelles. Les résultats sont mis sous forme de

graphique pour être lisibles et que l'ingénieur de validation puisse donner facilement son verdict.

Comme cela est expliqué dans [57], le test de propriétés extra-fonctionnelles est souvent négligé. Les auteurs contribuent en apportant une analyse des propriétés extra-fonctionnelles et visent à prédire, allouer des ressources de tests et comparer différentes approches en observant les résultats des décisions prises sur l'aspect extra-fonctionnel.

La vérification de systèmes adaptatifs peut être réalisée en amenant les modèles logiciels et la vérification de modèles au moment de l'exécution, afin de supporter le raisonnement automatique perpétuel sur les changements. Une fois qu'un changement est détecté, le système peut lui-même prédire si des violations des exigences peuvent se produire et activer les contre-actions appropriées. Cependant, les techniques et les outils de vérification de modèles existants n'ont pas été conçus pour une utilisation en temps réel ; ils ne répondent donc guère aux contraintes imposées par l'analyse à la volée en termes de temps d'exécution et d'utilisation de la mémoire. Les travaux [58, 59] abordent ce problème et se concentrent sur la satisfaction perpétuelle des exigences non fonctionnelles. Leur principale contribution est la description d'un cadre mathématique pour un contrôle de modèle probabiliste efficace à l'exécution en temps réel. Cette approche génère statiquement un ensemble de conditions de vérification qui peuvent être évaluées efficacement à l'exécution dès que des changements sont apportés au modèle. L'approche proposée supporte également l'analyse de sensibilité, qui permet de raisonner sur les effets des changements et peut conduire à des stratégies d'adaptation efficaces.

Dans [99] les auteurs effectuent un suivi du système de différentes manières. Si un objectif lié aux exigences du modèle est attribué à un système existant ou à un composant, il peut être possible d'interroger ce système ou composant pour obtenir le statut de l'objectif. Sinon, des capteurs dans l'environnement peuvent être utilisés pour déterminer si l'objectif a été atteint sans interroger le ou les composants concernés. Étant donné un modèle d'objectifs caractérisant diverses manières d'atteindre un certain objectif, il est possible de classer ces alternatives par rapport à leur satisfaction de l'ensemble de critères de qualité représentés dans le modèle par des objectifs logiciels (softgoals : exigences extra-fonctionnelles).

Comme mentionné dans [39, 37], les systèmes adaptatifs et auto-adaptatifs sont un défi en termes d'assurance à plusieurs niveaux tels que l'adaptation, l'automatisation, les dépendances entre les règles. Pour répondre à cela, les auteurs proposent un contrôle par rétroaction qui mesure le comportement du système et le modifie si nécessaire. Nous nous sommes inspirés de ces travaux pour contrôler le système mais nous préférons informer l'utilisateur d'une erreur potentielle plutôt que de changer directement le comportement du système. Les deux raisons sont que notre approche est orientée boîte noire donc nous n'avons pas de contrôle direct du système et nous considérons que l'erreur

peut venir de la phase de développement plus que de l'environnement lui-même.

2.4/ CONCLUSION

Nous avons présenté dans ce chapitre le contexte scientifique des systèmes adaptatifs, nous allons maintenant présenter les notions qui nous serons utiles dans les contributions.

Nous fondons notre approche sur la modélisation de systèmes à composants et plus particulièrement sur notre formalisme que nous introduisons dans le chapitre 4 inspiré par l'outil Fractal.

Nous avons décrit les mécanismes de validation pour les systèmes adaptatifs. Nous présentons nos contributions pour répondre à la problématique de la construction des tests. Notre approche que nous présentons dans le chapitre 5 vise à automatiser la construction des tests.

Nous avons décrit les critères de couverture à la fois statiques et dynamiques et avons pointé le manque de travaux en lien avec les politiques d'adaptation. L'approche que nous présentons dans le chapitre 6 défini des critères de couverture et sur les propriétés temporelles ainsi que les politiques d'adaptation. Le chapitre 6 a également pour objectif de présenter nos travaux sur la validité du système vis-à-vis des propriétés temporelles mais aussi une approche originale impliquant les politiques d'adaptation.

3

PRÉLIMINAIRES

Sommaire

3.1 Ensembles relations et fonctions 37
3.1.1 Ensembles 37
3.1.2 Relations et fonctions 38
3.1.3 Systèmes de transition 40
3.1.4 logique du premier ordre 41
3.2 Exemple fil rouge : le convoi de véhicules VANet 45
3.2.1 VANet 45
3.3 Configurations, logiques temporelles et politiques d'adaptation . . 47
3.3.1 Modèle et chemin de reconfiguration 48
3.3.2 Logiques Temporelles 51
3.4 La logique FTPL 51
3.4.1 Syntaxe de FTPL 51
3.4.2 Sémantique de FTPL 51
3.5 Politiques d'adaptation 54

Ce chapitre est consacré aux définitions et notations utilisées dans la suite du document. Nous nous intéressons à la spécification et la validation des systèmes adaptatifs ainsi qu'à l'expression de leurs propriétés.

3.1/ ENSEMBLES RELATIONS ET FONCTIONS

Cette section a pour but de fixer le vocabulaire sur les ensembles, relations et fonctions utilisés au cours de ce mémoire de thèse.

3.1.1/ ENSEMBLES

Pour tout ensemble E, nous notons :

— $card(E)$, le nombre d'éléments de E, i.e, le cardinal de E,

— $\emptyset$, l'ensemble vide,

— $\mathbb{B} = \{vrai, faux\}$ le type booléen.

Pour deux ensembles E et F, nous notons :

— $E \setminus F$, l'ensemble des éléments de E n'appartenant pas à F

— $F \subseteq E$, l'inclusion de F dans E, i.e, lorsque F est un sous ensemble de E

— $E \cup F$, l'union des ensembles E et F

— E^n, l'ensemble des n-uplets d'éléments de E

3.1.2/ RELATIONS ET FONCTIONS

Une relation (binaire) entre deux ensembles E et F est un sous-ensemble $\mathcal{R}$ du produit cartésien $E \times F$. Soient $e \in E$, $f \in F$ d'une part et $\mathcal{R} \subseteq (E \times F)$, une relation entre E et F d'autre part. Nous notons $e\mathcal{R}f$ ou $(e, f) \in \mathcal{R}$ pour signifier que e est en relation avec f par la relation $\mathcal{R}$. Dans ce cas, E est l'ensemble de départ et F est l'ensemble d'arrivée par $\mathcal{R}$. La relation réciproque $\mathcal{R}^{-1}$ est définie par l'ensemble $\{(f, e) \in F \times E | (e, f) \in \mathcal{R}\}$.

Définition 1 : Domaine et codomaine d'une relation

Soit $\mathcal{R} \subseteq E \times F$ une relation entre E et F.
Nous appelons le domaine de $\mathcal{R}$, noté $dom(\mathcal{R})$, l'ensemble des éléments de E qui ont une image dans F par $\mathcal{R}$ défini par :

$$dom(\mathcal{R}) = \{e \in E | \exists f \in F.(e, f) \in \mathcal{R}\}$$

Nous appelons le codomaine de $\mathcal{R}$, noté $ran(\mathcal{R})$, l'ensemble des images par $\mathcal{R}$ des éléments de E défini par :

$$ran(\mathcal{R}) = \{f \in F | \exists e \in E.(e, f) \in \mathcal{R}\}$$

Définition 2 : Relation réflexive, symétrique, antisymétrique et transitive

Soit $\mathcal{R} \subseteq E \times E$ une relation de E dans E.

— $\mathcal{R}$ est réflexive si $\forall e.(e \in E \Rightarrow e\mathcal{R}e)$

— $\mathcal{R}$ est symétrique si $\forall e1, e2.(e1, e2 \in E \wedge e1\mathcal{R}e2 \Rightarrow e2\mathcal{R}e1)$

— $\mathcal{R}$ est antisymétrique si $\forall e1, e2.(e1, e2 \in E \wedge e1\mathcal{R}e2 \wedge e2\mathcal{R}e1 \Rightarrow e1 = e2)$

— $\mathcal{R}$ est transitive si $\forall e1, e2, e3.(e1, e2, e3 \in E \wedge e1\mathcal{R}e2 \wedge e2\mathcal{R}e3 \Rightarrow e1\mathcal{R}e3)$

Définition 3 : Fermeture transitive d'une relation et relation acyclique

Soit $\mathcal{R} \subseteq E \times E$ une relation définie sur un ensemble E. La fermeture transitive de $\mathcal{R}$, notée $\mathcal{R}^+$, est la plus petite (au sens de l'inclusion) relation transitive définie sur E contenant $\mathcal{R}$. Autrement dit, $\mathcal{R}^+$ est la relation binaire définie sur l'ensemble E telle que :

— $\mathcal{R}^+$ est transitive,

— $\mathcal{R} \subset \mathcal{R}^+$,

— Pour toute relation binaire $S \subseteq E \times E$ transitive définie sur l'ensemble E, si $\mathcal{R} \subset S$ alors $\mathcal{R}^+ \subset S$.

Une relation est dite acyclique si

$$\forall e, e'.(e, e' \in E \wedge e\mathcal{R}^+e' \wedge e'\mathcal{R}^+e \Rightarrow e = e')$$

Définition 4 : Relation fonctionnelle

Une relation est dite fonctionnelle (ou une fonction est une relation) $\mathcal{R} \subseteq E \times F$ si tout élément de E a au plus une image dans F par $\mathcal{R}$. Lorsqu'une relation φ est une fonction, nous notons $\varphi : E \to F$ à la place de $\varphi \subseteq E \times F$, et $\varphi(e) = f$ à la place de $(e, f) \in \varphi$.

Une fonction totale $\varphi : E \to F$ est une relation fonctionnelle telle que pour tout $e \in E$, il existe exactement un $f \in F$ tel que $\varphi(e) = f$. Une fonction totale est appelée application. Réciproquement, une fonction partielle $\varphi : E \to F$ est une relation fonctionnelle telle que pour tout $e \in E$, il existe au plus un $f \in F$ tel que $\varphi(e) = f$.

Définition 5 : Ensemble des parties

Soit E un ensemble. On appelle ensemble des parties de E et on note $\mathcal{P}(E)$, l'ensemble de tous les sous-ensembles de E. On a alors :

$$A \in \mathcal{P}(E) \Leftrightarrow A \subset E$$

$$\mathcal{P}(E) = \{A \mid A \subset E\}$$

Exemple 1 (*Ensemble des parties*) : *Soit E un ensemble. L'ensemble des parties de E est l'ensemble, généralement noté $\mathcal{P}(E)$, dont les éléments sont les sous-ensembles de E :*

$E = \emptyset \Rightarrow \mathcal{P}(E) = \{\emptyset\}$

$E = \{a\} \Rightarrow \mathcal{P}(E) = \{\emptyset; E\}$

$E = \{a; b\} \Rightarrow \mathcal{P}(E) = \{\emptyset; \{a\}; \{b\}; E\}$

$E = \{a; b; c\} \Rightarrow \mathcal{P}(E) = \{\emptyset; \{a\}; \{b\}; \{a; b\}; \{a; c\}; \{b; c\}; E\}$

Définition 6 : Filtrage de fonction multivaluées

Soit $\mathcal{A}$ un sous-ensemble non vide d'un ensemble E, $\mathcal{A} \in P(E)\backslash\emptyset$. L'ensemble $\mathcal{F}l_A = \{X \in P(E) | \mathcal{A} \subseteq X\}$ noté $\triangleleft E$ (ou $E \triangleright$) est un filtre.

Exemple 2 (*Filtrage d'un ensemble*) : *Soit un ensemble* $E = \{\{a,b\},\{b,c\},\{a,c\}\}$
Le résultat du filtrage $\triangleleft E$ *par* $\{a\}$ *est :* $\{a\} \triangleleft E = \{\{a,b\},\{a,c\}\}$

Définition 7 : Fonction injective, surjective et bijective

Soit $\varphi : E \to F$ une fonction.

- φ est injective si $\forall e, f.(e, f \in E \wedge \varphi(e) = \varphi(f) \Rightarrow e = f)$,
- φ est surjective si $\forall f.(f \in F \Rightarrow \exists e.(e \in E \wedge \varphi(e) = f))$,
- φ est bijective si elle est à la fois injective et surjective.

3.1.3/ SYSTÈMES DE TRANSITION

Cette section a pour but de présenter des modèles sémantiques souvent utilisés par la suite pour décrire le comportement du système.
Les systèmes de transitions vont nous permettre de spécifier le comportement de systèmes adaptatifs. Nous les définissons ci-dessous.

Définition 8 : Système de transitions ST

Un système de transitions est un n-uplet $\langle Q, Q_0, \to \rangle$ où :

- Q est un ensemble d'états,
- Q_0 est un ensemble d'états initiaux tel que $Q_0 \subseteq Q$,
- $\to \subseteq Q \times Q$ est une relation de transitions.

Il peut être intéressant de différencier les actions d'un système modélisé en étiquetant les transitions avec des noms d'actions comme c'est le cas pour les systèmes de transitions étiquetés :

Définition 9 : Système de transitions étiqueté STE

Un système de transitions étiqueté est un n-uplet $S = \langle Q, Q_0, E, \rightarrow, F \rangle$ où :

- Q est un ensemble d'états,
- Q_0 est un ensemble d'états initiaux tel que $Q_0 \subseteq Q$,
- E est un ensemble de noms d'actions,
- $\rightarrow \subseteq Q \times E \times Q$ est une relation de transition étiquetée,
- $F \subseteq Q$ est l'ensemble des états finaux.

À noter que l'état final n'est pas obligatoire dans un STE.
Un STE est fini si les ensembles Q et E sont finis.
Dans le cas contraire, le système est infini.

Nous présentons dans l'exemple ci-dessous un STE visible en figure 3.1.

Exemple 3 (*Système de transitions étiqueté*) : *Dans cet exemple, le STE contient 3 états $q0$, $q1$ et $q2$ pouvant effectuer 4 actions $e1$, $e2$, $e3$ et $e4$. Les cercles représentent les états du système, les doubles cercles représentent les états finaux du système (ici $q2$) et les flèches orientées d'un état source vers un état destination représentent les transitions. Une transition sans état source permet de pointer les états initiaux du système.*

Nous notons $q' \xrightarrow{e} c'$ un élément $(q, e, q') \in \rightarrow$ représentant la transition étiquetée par e entre les états q et q'.

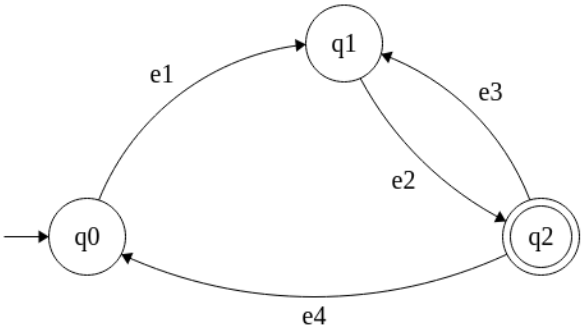

FIGURE 3.1 – Exemple d'un système de transitions étiqueté

3.1.4/ LOGIQUE DU PREMIER ORDRE

Nous nous intéressons, dans cette section, à la logique du premier ordre qui permet de spécifier des propriétés sur les ensembles et les relations.

3.1.4.1/ SYNTAXE

Nous nous intéressons tout d'abord à la syntaxe de cette logique en présentant le langage utilisé pour exprimer des formules de premier ordre.

Définition 10 : Langage du premier ordre

Le langage de la logique du premier ordre est formé avec un ensemble de symboles spécifiés par :

- Un ensemble infini dénombrable de variables $V = \{v_1, v_2, v_3, ...\}$,
- La négation $\neg$ où $\neg A$ est vraie si et seulement si A est fausse,
- La déclaration inconditionnellement vraie $\top$, et la déclaration inconditionnellement fausse $\bot$,
- Les connecteurs logiques $\wedge, \vee, \neg, \Rightarrow$,
- Les quantificateurs $\exists, \forall$,
- Un ensemble dénombrable éventuellement vide de symboles de relations $\mathcal{S}_\mathcal{R} = \{r_1, r_2, r_3, ...\}$ ainsi qu'une fonction $ar : \mathcal{S}_\mathcal{R} \to \mathbb{N}^*$ spécifiant le nombre d'arguments de chaque symbole de relation,
- Un ensemble dénombrable éventuellement vide de symboles de fonctions $\mathcal{S}_\mathcal{F} = \{f_1, f_2, f_3, ...\}$ ainsi qu'une fonction $ar : \mathcal{S}_\mathcal{F} \to \mathbb{N}^*$ spécifiant le nombre d'arguments de chaque symbole de fonction,
- Un ensemble dénombrable éventuellement vide de symboles de constantes $\mathcal{S}_C = \{c_1, c_2, c_3, ...\}$,
- Un ensemble dénombrable éventuellement vide de symboles de prédicats $\mathcal{S}_\mathcal{P} = \{P, Q, R, ...\}$ ainsi qu'une fonction $ar : \mathcal{S}_\mathcal{P} \to \mathbb{N}^*$ spécifiant le nombre d'arguments de chaque symbole de prédicat.

Nous définissons à présent la notion de termes qui sont construits à partir des variables et des fonctions.

Définition 11 : Termes

L'ensemble des termes, noté $\mathcal{T} = \{t_1, ..., t_n\}$, est défini inductivement par :

- Les variables et les constantes sont des termes,
- Si $t_1, ..., t_n$ sont des termes et si f est un symbole de fonction tel que $ar(f) = n$ alors $f(t_1, ..., t_n)$ est un terme.

Nous nous intéressons maintenant aux formules qui sont obtenues avec des prédicats et des connecteurs logiques.

Définition 12 : Formules

Soient $t_1, ..., t_n$ des termes dans $\mathcal{T}$ et P un symbole de prédicat dans $\mathcal{S_P}$ tel que $ar(P) = n$, $P(t_1, ..., t_n)$ est une formule atomique. L'ensemble $\mathcal{EF} = \{A, B, C, ...\}$ des formules du premier ordre est défini inductivement par :

- Les formules atomiques sont des formules ;
- Soient A et B des formules. $A \wedge B, A \vee B, A \Rightarrow B$ et $\neg A$ sont des formules,
- Soient A une formule et x une variable, alors $\forall x.A$ et $\exists x.A$ sont des formules.

3.1.4.2/ SÉMANTIQUE

Nous nous intéressons maintenant à la sémantique de la logique du premier ordre. Pour cela, il faut donner un sens aux fonctions et aux prédicats en interprétant le langage du premier ordre.

Définition 13 : structure d'interprétation

Une structure d'interprétation $\mathcal{SI} = (E, I)$ pour un langage du premier ordre est la donnée d'un ensemble E appelé domaine et d'une fonction d'interprétation I associant des fonctions, des relations et des prédicats sur E aux symboles de fonctions et de prédicats de ce langage tels que :

- Pour chaque fonction f d'arité n, $I(f) : E^n \rightarrow E$,
- Pour chaque relation r d'arité n, $I(r) \subseteq E^n$,
- Pour chaque constante c $I(c) \in E$,
- Pour chaque symbole de prédicat P d'arité $n, I(P) : E^n \rightarrow \mathbb{B}$.

Nous introduisons l'ensemble CP de propositions de configuration sur les composants et leurs relations. Les propositions de configurations sont utilisées pour définir des configurations cohérentes (par exemple pour satisfaire les propriétés d'invariance au niveau architectural).
La fonction d'interpretation $l : C \rightarrow CP$ donne la plus grande conjonction de $cp \in CP$ évaluée à vraie sur $c \in C$, qui caractérise la configuration courante de la manière la plus précise.
Maintenant que l'interprétation I définie, nous définissons une affectation qui donne la valeur des variables libres dans les formules.

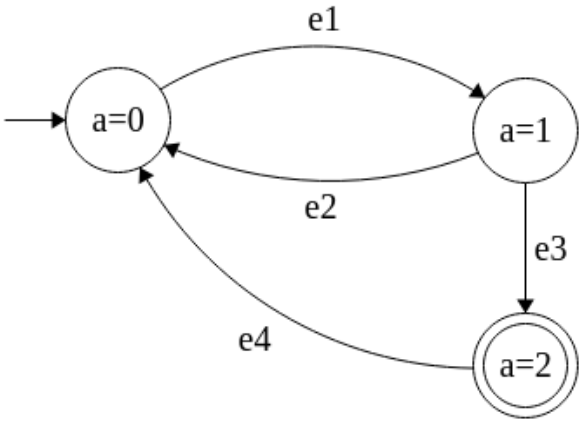

FIGURE 3.2 – Exemple d'un système de transitions étiqueté interprété

Définition 14 : Affectation

Une affectation de valeurs aux variables est une fonction $p : \mathcal{V} \rightarrow E$. Nous notons $\rho[v := e](v) = e$ l'affectation de valeurs aux variables définie par :

— $\rho[v := e](v') = \rho(v'), si\ v \neq v'$,

— $\rho[v := e](v) = e$.

Il peut être intéressant d'utiliser des variables comme c'est le cas pour les systèmes de transitions étiquetés interprétés :

Définition 15 : Système de transitions étiqueté interprété $STEI$

Un système de transitions étiqueté interprété est un n-uplet $S = \langle Q, Q_0, E, \rightarrow, V, l, F \rangle$ où :

— Q est un ensemble d'états,

— Q_0 est un ensemble d'états initiaux tel que $Q_0 \subseteq Q$,

— E est un ensemble de noms d'actions,

— $\rightarrow \subseteq Q \times E \times Q$ est une relation de transition étiquetée,

— V est un ensemble de variables,

— $l : C \rightarrow CP$ est la fonction d'interprétation totale qui donne la plus grande conjection de $cp \in CP$ évaluée à vrai dans $c \in C$ qui est utilisée pour caractériser l'état courant de la manière la plus précise possible,

— $F \subseteq Q$ est l'ensemble des états finaux.

Nous présentons dans l'exemple ci-dessous un $STEI$ visible en figure 3.2.

Exemple 4 (*Système de transition étiqueté interprété*) : *Dans cet exemple, le $STEI$ contient 3 états pouvant effectuer 4 actions $e1, e2, e3, e4$. Les cercles représentent les*

états du système avec la valeur des variables clés, les doubles cercles représentent les états finaux du système et les flèches orientées d'un état source vers un état destination représentent les transitions. Une transition sans état source permet de pointer les états initiaux du système.
Nous notons $q' \xrightarrow{e} c'$ *un élément* $(q, e, q') \in \rightarrow$ *représentant la transition étiquetée par* e *entre les états* q *et* q'*.*

3.2/ EXEMPLE FIL ROUGE : LE CONVOI DE VÉHICULES VANET

Afin d'illustrer les différents concepts introduits dans ce document, nous présentons une application de convois de véhicules appelée VANet (Vehicular Ad-Hoc Network).

3.2.1/ VANET

Dans ce système, les véhicules sur la route sont connectés en réseau et peuvent être organisés en convoi ou individuellement. Dans chaque peloton, on trouve un leader qui se situe à l'avant. Un véhicule seul peut demander à rejoindre un peloton ou créer un nouveau peloton en se groupant avec un autre véhicule seul. Lorsque les véhicules sont dans un convoi, la conduite passe en mode automatique ce qui donne de nombreux avantages. En effet, grâce à l'automatisation de la conduite, les distances de sécurité peuvent être réduites ce qui permet un gain en terme de consommation d'énergie [150]. Le convoi roule de manière plus fluide qu'un automobiliste ce qui permet d'améliorer la circulation ainsi que le confort de conduite [84]. Certains travaux soulignent le fait que les convois automatiques améliorent la sécurité pour l'utilisateur [87]. Le fait d'automatiser la conduite augmente également le confort pour le conducteur même si les systèmes ne sont pas complètement autonomes et que le conducteur doit garder un oeil sur la route.
La recherche dans le domaine des convois de véhicules existe depuis plusieurs années, en Août 1997 le National Automated Highway System Consortium (NAHSC) a présenté une preuve sur la faisabilité d'une conduite autonome [147].

Dans notre exemple, le but est de faire en sorte que les véhicules roulent le plus possible en convoi.

Un ou plusieurs véhicules peuvent être instanciés. Les véhicules peuvent évoluer librement (mode $solo$) ou dans un convoi (mode $platoon$). Dans le cas de véhicules en mode $platoon$, ils sont dans un convoi avec un véhicule à l'avant (appelé $leader$) suivi par les autres véhicules. Le système VANet peut être vu comme un système à composants dans lequel les véhicules sont connectés ensemble, pour constituer le convoi, et connectés à un composant principal, la route. Les dépendances entre véhicules permettent également de faire passer les informations sur les variables, par exemple, pour une recherche

du nouveau $leader$ basée sur les niveaux de batterie et sur la distance restante à parcourir.

Nous définissons ici les caractéristiques de notre convoi de véhicules. Les véhicules circulent sur une voie rectiligne de type autoroute, ils sont tous sur la même route mais à des positions différentes. Si deux véhicules veulent se rejoindre, ils doivent adapter leur vitesse afin de se retrouver au même endroit. L'autonomie des véhicules descend avec le temps et un véhicule seul ou $leader$ perd plus vite son énergie qu'un véhicule dans un convoi (non $leader$). Chaque véhicule possède son niveau d'énergie. Lorsque ce niveau est bas le véhicule devra s'arrêter à une station pour se recharger. Si le véhicule est dans un convoi, il doit tout d'abord sortir du peloton avant de pouvoir s'arrêter à la station. La distance restante avec l'objectif d'arrivée du véhicule indique le nombre de kilomètres restants à parcourir. La position du véhicule sur la route permet de savoir quand deux véhicules peuvent se rejoindre et également de savoir quand un véhicule s'approche d'une station. La position est en relation avec la variable de vitesse qui est ajustable.
Un exemple est visible dans la figure 3.3. Dans cette figure, il y a 3 véhicules $solo$ dont un véhicule arrêté pour recharger sa batterie ainsi qu'un convoi constitué de 3 véhicules.

Lorsque deux véhicules se rapprochent, une demande de regroupement est effectuée. La demande va être analysée par les véhicules qui vont décider le choix idéal entre plusieurs paramètres tel que le niveau d'énergie, la distance restante. Si la demande est acceptée les véhicules forment alors un convoi. De la même façon, un convoi peut accepter d'autres véhicules jusqu'à un certain nombre de véhicules maximum. Cette limitation du nombre de véhicules sert à garder de bonnes conditions de sécurité au sein du convoi. En effet, un trop grand nombre de véhicules dans un convoi donnerait lieu à des allées et venues au sein des convois.

Les raisons qui vont amener un véhicule à quitter un convoi sont multiples. Les véhicules peuvent sortir du convoi soit pour se recharger à une station, soit pour quitter la route car ils ont atteint leur destination. De plus, le véhicule $leader$ peut changer, soit parce qu'il doit quitter le convoi, soit parce qu'un autre véhicule du convoi est plus adapté (meilleure autonomie ou distance à la destination plus élevée).

Le conducteur du véhicule peut également décider à tout instant de repasser en mode conduite manuelle. Pour cela, il devra d'abord sortir du convoi sachant qu'au sein d'un peloton une seule manœuvre est autorisée à la fois et donc, si des manœuvres sont déjà engagées, le conducteur devra attendre avant de pouvoir sortir du convoi. Nous utilisons ce cas d'étude VANet car il est représentatif des systèmes adaptatifs [88].

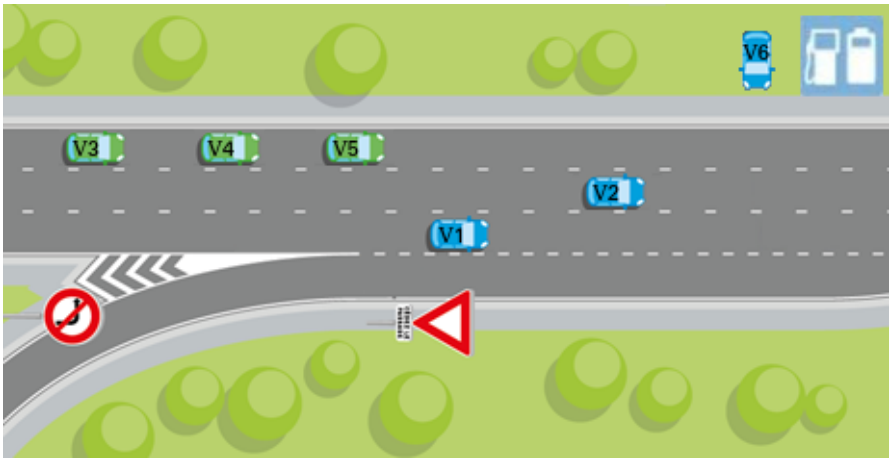

FIGURE 3.3 – Un exemple configuration VANet

3.3/ CONFIGURATIONS, LOGIQUES TEMPORELLES ET POLITIQUES D'ADAPTATION

Afin de pouvoir représenter des contraintes architecturales qui correspondent aux différentes relations entre les éléments architecturaux, nous définissons les propriétés de configuration en utilisant la logique du premier ordre.

Soit $C = \{c_0, c_1, c_2...\}$ un ensemble de configurations définies par le langage du premier ordre (cf définition 10).
Nous rappelons que CP est l'ensemble de propositions de configuration sur les composants et leurs relations.
Une fonction d'interprétation $l : C \rightarrow CP$ donne la plus grande conjonction de $cp \in CP$ évaluée à vraie sur $c \in C$, qui caractérise la configuration courante de la manière la plus précise.
Soit $\mathcal{R}$, l'ensemble des opérations de reconfiguration qui permettent au système d'évoluer dynamiquement. Il y a des combinaisons d'opérations telles que l'instantiation ou la destruction de composants ; la mise en marche ou à l'arrêt des composants ; lier ou enlever les liens entre les interfaces de composants, etc.

— Soit $\mathcal{R}_{run} = \mathcal{R} \bigcup \Theta \bigcup \{run\}$, l'ensemble des opérations d'évolution (y compris les actions internes au système qui n'ont pas déclenché de reconfiguration),

— dans lequel $\mathcal{R}$ est un ensemble fini d'opérations de reconfiguration,

— Θ est l'ensemble des opérations déclenchées par des évènements externes ($\mathcal{R} \bigcap \Theta = \emptyset$) et,

— run est un nom générique permettant de représenter toutes les opérations d'exécution du système.

Nous considérons comme dans [92], que les évènements externes sont capturés par le système et traités immédiatement en déclenchant une opération de Θ.

3.3.1/ MODÈLE ET CHEMIN DE RECONFIGURATION

Définition 16 : Modèle de reconfiguration

La sémantique des systèmes cyber-physiques auto-adaptatifs est définie par le système de transitions étiquetées : $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ où
C est un ensemble de configurations,
$C^0 \subseteq C$ est un ensemble de configurations initiales,
$\mathcal{R}_{run}$ est l'ensemble des opérations d'évolution,
$\rightarrow \subseteq C \times \mathcal{R}_{run} \times C$ est la relation de reconfiguration,
$l : C \rightarrow CP$ est la fonction d'interprétation totale définie précédemment (cf définition 15).

Les comportements d'un système adaptatif peuvent être décrits par un chemin de reconfiguration décrivant l'ensemble des reconfigurations qui ont été exécutées.

Définition 17 : Chemin de reconfiguration

Soit le modèle S, un chemin de reconfiguration σ de S est une séquence de configurations $c_0, c_1, c_2, \ldots$ telle que $\forall i \geq 0,\ \exists\ ope_i \in \mathcal{R}_{run}.\ (c_i, ope_i, c_{i+1}) \in \rightarrow$.

Le chemin de reconfiguration est une séquence de configurations du système ce qui est différent de la trace d'exécution qui est une séquence des opérations observées du système.

Définition 18 : Trace d'exécution

La trace d'exécution de σ, notée $tr(\sigma)$, est le mot $ope_0 ope_1 \ldots ope_i \ldots$ composé des opérations observées $ope_0, ope_1, \ldots, ope_i, \ldots$.
L'ensemble des chemins de reconfiguration est noté Σ, et l'ensemble des chemins de reconfiguration finis Σ^f ($\subseteq \Sigma$).

Définition 19 : Exécution

Une exécution est un chemin de reconfiguration σ dans Σ tel que $\sigma(0) \in C^0$ dans lequel C^0 est l'ensemble des configurations initiales. Nous écrivons c_i ou $\sigma(i)$ pour noter la i-ème configuration de σ. Pour le suffixe du chemin, nous utilisons la notation $\sigma_i : \sigma(i), \sigma(i+1), ...,$.
Le segment de chemin est noté $\sigma_i^j : \sigma(i), \sigma(i+1), ..., \sigma(j-1), \sigma(j)$.

Exemple 5 (*Évolution de la configuration*) : *Cet exemple s'appuie sur la figure 3.4 dans laquelle sont illustrés les configurations du système, les évènements externes reçus ainsi que les reconfigurations déclenchées.*
En deuxième partie de la figure se trouve le même exemple sous forme de séquence. L'exemple commence avec un système à la configuration $C1$ constitué d'un composant r de type $road$ et de deux composants $v1$ et $v2$ de type $vehicle$.
Ensuite, un nouveau composant $v3$ de type $vehicle$ souhaite entrer avec l'évènement $v3.enter(r)$. Cet évènement n'a pas besoin d'être validé par le système qui inscrit run dans sa trace d'exécution et met à jour ses variables internes. À la configuration $C2$, le système possède donc un composant de type $vehicle$ supplémentaire $v3$. L'évènement $v2.join(v3)$ indique que $v2$ souhaite former un convoi avec $v3$. Le système réagit à cet évènement par un $v3.acceptJoin(v2)$ indiquant que le système a accepté cette reconfiguration.
En configuration $C3$, est formé un composant de type convoi p à partir de $v3$ et $v2$. Dans cette configuration $C3$, $v3$ souhaite sortir de p comme le montre l'évènement $v3.forceQuit(p)$. Le système accepte cette demande comme le montre l'évènement $p.acceptQuit(v3)$.
D'un point de vue configuration, $C4$ est similaire à $C2$. Pour le reste, aucun évènement n'est envoyé au système dans la configuration $C4$, le système réagit avec un run.
À nouveau les configurations $C5$ et $C4$ sont similaires et aucun évènement externe. Sauf que cette fois le système réagit avec un $v3.leave(r)$. Cette reconfiguration est due au fait que le système met à jour ses paramètres entre chaque configuration. La configuration ne change pas d'un point de vue architectural mais il est possible que certains paramètres déclenchent des reconfigurations. Dans le cas présent, $v3$ a atteint sa destination et doit quitter la route. La dernière configuration de notre exemple nommée $C6$ est similaire à $C1$ après le départ de $v3$.

Nous définissons à présent l'atteignabilité d'une configuration.

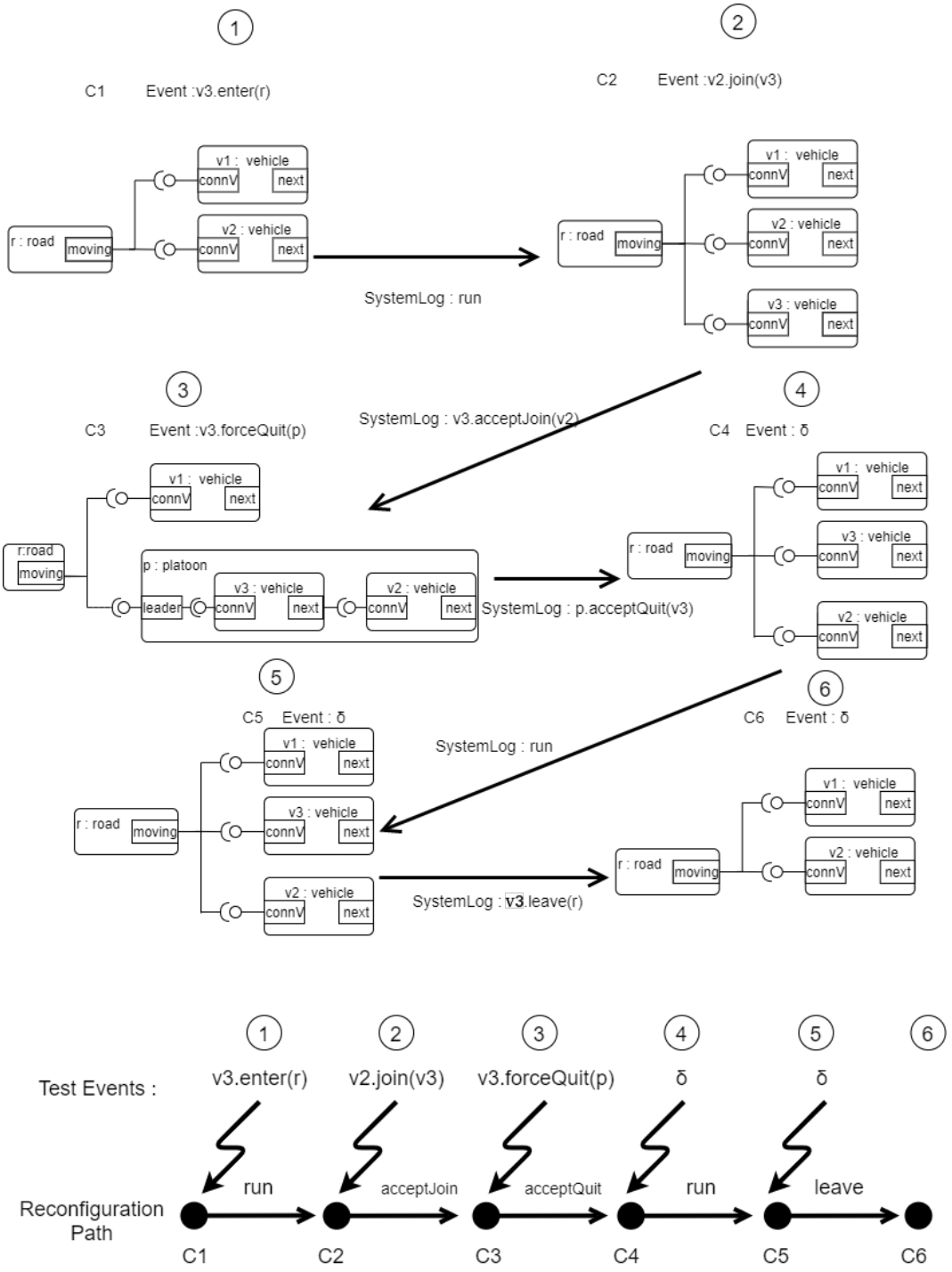

FIGURE 3.4 – Évolution de la configuration d'un système adaptatif

Définition 20 : Atteignabilité d'une configuration

Soit Σ l'ensemble des chemins de reconfiguration, et Σ^f ($\subseteq \Sigma$) l'ensemble des chemins de reconfiguration finis.
Une configuration c' est dite atteignable depuis c lorsqu'il existe un chemin de reconfiguration $\sigma = c_0, c_1, \ldots, c_n$ dans Σ^f de telle sorte que $c = c_0$ et $c' = c_n$.

3.3.2/ LOGIQUES TEMPORELLES

Les invariants et les formules de la logique du premier ordre présentés jusqu'à présent ne sont pas suffisants pour exprimer les propriétés dites dynamiques sur les systèmes. Pour cela, les logiques temporelles sont des formalismes davantage adaptés pour ce type de propriétés. il existe de nombreuses logiques temporelles dont une classification peut être trouvée, par exemple, dans [30]. Nous présentons dans la section suivante suivante la logique FTPL que nous avons utilisée dans nos travaux.

3.4/ LA LOGIQUE FTPL

La logique FTPL (pour First order logic Temporal Pattern Language) introduite dans [47], qui se base sur les travaux de Dwyers sur les schémas de spécification [49] était utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permettait pas la prise en compte d'évènements externes. C'est pourquoi les travaux de [95] ont étendu la logique FTPL avec des évènements externes afin de pouvoir utiliser la même logique pour exprimer des politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité. Dans cette section, nous décrivons d'abord la syntaxe de la logique FTPL étendue aux évènements externes, puis nous décrivons sa sémantique.

3.4.1/ SYNTAXE DE FTPL

La figure 3.5 donne la syntaxe du langage FTPL, il est composé de différentes couches :

- Les propriétés temporelles ($temp$)
- Les propriétés de trace ($trace$)
- Les évènements liés aux reconfigurations ($event$)
- les propriétés de configuration (cp $\in CP$)

Cette logique prend en compte les évènements externes (ext), ainsi que les évènements obtenus des opérations de reconfiguration.

3.4.2/ SÉMANTIQUE DE FTPL

La sémantique FTPL définie dans [95] est résumée dans les définitions de cette sous-section. Les évènements externes (comme les évènements dans [92]) interviennent de manière instantanée et peuvent être vus comme des invocations de méthodes effectuées par des capteurs (externes) lorsqu'un changement est détecté dans leur environ-

<*FTPL*>	::=	<*temp*>|<*events*>
<*temp*>	::=	**after** <*events*> <*temp*>
	|	**before** <*events*> <*trace*>
	|	<*trace*> **until** <*events*>
<*trace*>	::=	**always** cp
	|	**eventually** cp
	|	<*trace*> $\wedge$ <*trace*>
	|	<*trace*> $\vee$ <*trace*>
<*events*>	::=	<*event*>,<*events*>|<*event*>
<*event*>	::=	ope **normal**
	|	ope **exceptional**
	|	ope **terminates**
	|	ext

FIGURE 3.5 – Syntaxe FTPL

nement. Soit $Prop_{FTPL}$ l'ensemble des formules FTPL obéissant à la grammaire FTPL de la Fig. 3.5.

3.4.2.1/ LES PROPRIÉTÉS DE CONFIGURATION

Les propriétés de configuration permettent de s'assurer que le système est dans un état cohérent.

Définition 21 : Satisfaction d'une propriété de configuration

Soient $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ un système à reconfiguration, $cp \in CP$ et $\sigma(i)$ une configuration de C. La satisfaction de cp sur $\sigma(i)$ est définie comme suit :

$$\sigma(i) \models cp \; if \; l(\sigma(i)) \Rightarrow cp$$

3.4.2.2/ LES ÉVÈNEMENTS

Les évènements internes permettent la détection des effets d'une reconfiguration. Ceci nous permet d'exprimer des propriétés en prenant en compte les résultats des différentes reconfigurations du système pouvant survenir. Étant donné une opération de reconfiguration $ope \in \mathcal{R}$, nous considérons les évènements internes suivants :

— ope **normal** signifie que la reconfiguration ope s'est terminée normalement,

— ope **exceptional** signifie que la reconfiguration ope s'est terminée anormalement c'est-à-dire qu'une exception a été détectée,

— ope **terminates** signifie que la reconfiguration ope a été exécutée et qu'elle s'est terminée normalement ou anormalement.

Définition 22 : Satisfaction d'un évènement sur une configuration

Soient σ un chemin de reconfiguration de $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l \rangle$, $ope \in \mathcal{R}$ une opération de reconfiguration et ext un évènement externe. Étant donné un évènement e décrit par <*event*> dans la syntaxe de la logique FTPL, sa satisfaction sur la i^{eme} configuration de σ, notée $\sigma(i) \models e$ est définie par : $\sigma(i) \models ope$ **normal** if $i > 0 \wedge \sigma(i-1) \neq \sigma(i) \wedge \sigma(i-1) \xrightarrow{ope} \sigma(i) \in \rightarrow$
$\sigma(i) \models ope$ **exceptional** if $i > 0 \wedge \sigma(i-1) = \sigma(i) \wedge \sigma(i-1) \xrightarrow{ope} \sigma(i) \in \rightarrow$
$\sigma(i) \models ope$ **terminates** if $\sigma(i) \models ope$ **normal** $\vee\ \sigma(i) \models ope$ **exceptional**
$\sigma(i) \models ext$ if $eval_\sigma(cp_{ext}, i) = \top$
$\sigma(i) \models event, events$ if $\sigma(i) \models event \vee \sigma(i) \models events$

3.4.2.3/ LES PROPRIÉTÉS DE TRACE

Une propriété de trace permet de spécifier une contrainte qui doit être satisfaite sur un chemin de reconfiguration.

Définition 23 : Satisfaction d'une propriété de trace sur un chemin de reconfiguration

Soient σ un chemin de reconfiguration de $S = \langle C, C^0, \mathcal{R}un, \rightarrow, l \rangle$ et cp une propriété de configuration. Étant donné une propriété de trace $trace$ décrite par <*trace*> dans la syntaxe de la logique FTPL, sa satisfaction sur le chemin σ, notée $\sigma \models trace$, est définie inductivement par :
$\sigma \models$ **always** cp if $\forall i.(i \geqslant 0 \ \Rightarrow \sigma(i) \models cp)$
$\sigma \models$ **eventually** cp if $\exists i.(i \geqslant 0 \wedge \sigma(i) \models cp)$
$\sigma \models trace_1 \wedge trace_2$ if $\sigma \models trace_1 \wedge \sigma \models trace_2$
$\sigma \models trace_1 \vee trace_2$ if $\sigma \models trace_1 \vee \sigma \models trace_2$

Intuitivement,

— **always** cp est satisfaite sur un chemin de reconfiguration σ si et seulement si cp est satisfaite sur chaque configuration de σ,

— **eventually** cp est satisfaite sur un chemin de reconfiguration σ si et seulement si cp est satisfaite sur au moins une configuration de σ, et

— les sémantiques de la conjonction et de la disjonction sont classiques.

3.4.2.4/ LES PROPRIÉTÉS TEMPORELLES

Les propriétés temporelles sont basées sur toutes les propriétés précédentes, c'est-à-dire qu'elles utilisent les propriétés de configuration, les évènements et les propriétés de

trace.
Nous rappelons que pour un chemin de reconfiguration $\sigma \in \sum, \sigma_i$ décrit le suffixe du chemin contenant $\sigma(i), \sigma(i+1), ...,$ et σ_i^j décrit le segment du chemin contenant $\sigma(i), \sigma(i+1), ..., \sigma(j-1), \sigma(j)$.

Définition 24 : Satisfaction d'une propriété temporelle sur un chemin de reconfiguration

Soient σ un chemin de reconfiguration de $S = \langle C, C^0, \mathcal{R}un, \rightarrow, l \rangle$, $events$ une liste d'évènements, $trace$ une propriété de trace et $temp$ une propriété temporelle. Étant donnée une propriété temporelle $temp1$ décrite par $<temp>$ dans la syntaxe de la logique FTPL, sa satisfaction sur la configuration σ, notée $\sigma \models temp1$, est définie inductivement par :
$\sigma \models$ **after** $event\ temp$ if $\forall i.(i \geqslant 0 \wedge \sigma(i) \models event \Rightarrow \sigma_i \models temp)$
$\sigma \models$ **before** $event\ trace$ if $\forall i.(i > 0 \wedge \sigma(i) \models event \Rightarrow \sigma_0^{i-1} \models trace)$
$\sigma \models trace$ **until** $event$ if $\exists i.(i > 0 \wedge \sigma(i) \models event \wedge \sigma_0^{i-1} \models trace)$

Intuitivement,

— la propriété **after** $event\ temp$ est satisfaite sur un chemin de reconfiguration σ si la satisfaction d'un évènement de $event$ sur une configuration de σ implique la satisfaction de la propriété temporelle $temp$ sur le suffixe de σ débutant à cette configuration,

— **before** $event\ trace$ est satisfaite sur un chemin d'évolution σ si pour chaque configuration de σ, la satisfaction d'un évènement de $event$ sur celle-ci signifie que la propriété de trace $trace$ est satisfaite sur le préfixe de σ finissant à la configuration précédente,

— $trace$ **until** $event$ est satisfaite sur un chemin d'évolution σ s'il y a une confguration de σ, qui satisfait un évènement de $event$ et la propriété de $trace$ est satisfaite sur le préfixe de σ finissant avant l'occurence de l'évènement.

Dans la section suivante, nous présentons les politiques d'adaptation nécessaires pour guider le système adaptatif.

3.5/ POLITIQUES D'ADAPTATION

Les politiques d'adaptation sont utilisées afin de prendre en compte les contraintes au niveau des ressources du système, des évènements de son environnement ou même des propriétés temporelles basées sur des séquences de configurations.
Les politiques d'adaptation sont définies par :

— des opérations de reconfigurations architecturales pour spécifier les possibles modifications de l'architecture ;

— des règles d'adaptation, utilisées pour relier les propriétés au système et définir le besoin [1] d'activer une reconfiguration.

Les politiques d'adaptation sont liées aux utilités comme dans [69] qui proposent de combiner les fonctions d'utilité et les règles d'adaptation au niveau architectural. Les travaux effectués dans la thèse de Jean-François Weber [157] ont étendu la définition des politiques d'adaptation en intégrant la logique FTPL basée sur les travaux de Dwyer [49].
Dans notre approche, les reconfigurations des politiques d'adaptation sont gardées par des séquences d'évènements spécifiques qui peuvent à la fois décrire une configuration du système ou impliquer des propriétés temporelles. Nous présentons les définitions en suivant [25, 95].

Définition 25 : Politiques d'adaptation

Soit S un STE (système de transition d'états ou LTS pour labeled transition system en anglais) et ses reconfigurations $\mathcal{R} \cup \Theta$ et $Ftype$ un ensemble de types flous. Soit $\sigma(i) \in C$, une politique d'adaptation de $\sigma(i)$ est définie comme $A = \langle R_N, R_R \rangle$, dans lequel :

— $R_N \subseteq \mathcal{R}$ est un ensemble fini (non-vide) de noms de reconfiguration,

— R_R est un ensemble (non-vide) de règles de reconfiguration de la forme $\langle F, B, G, I \rangle$ dans lequel :

  — $F \in Ftype$ est un type flou,

  — $B \subseteq \{\phi_{\sigma(i)} = value \mid \phi \in Prop_{FTPL} \wedge value \in \mathbb{B}_2\}$ est un ensemble de propriétés dans $Prop_{FTPL}$ évaluées dans $\mathbb{B}_2$ sur $\sigma(i)$, dites $triggers$,

  — $G \subseteq \{cp_{\sigma(i)} = value \mid cp \in CP \wedge value \in \mathbb{B}_2\}$ est un ensemble de propriétés de configuration dans CP évalué dans $\mathbb{B}_2$ sur $\sigma(i)$, dites $guards$,

  — $I \subseteq R_N \times F$ est une relation entre les reconfiguration et les valeurs floues, définissant leur utilité.

Les règles de reconfiguration et en particulier B et G dépendent de plusieurs éléments tels que l'environnement, la configuration du système et ses paramètres. Les valeurs floues définies dans [105, 123] permettent de considérer ces différents éléments à la fois, comme dans [26, 45].
Dans les faits, on associe à un besoin une valeur (qu'on appelle valeur floue) appartenant à un ensemble qu'on appelle un type flou (par exemple low,medium,high).

1. Comme dans les travaux de [25, 46], les valeurs floues (e.g. dans {`low`, `medium`, `high`}) sont utilisées pour exprimer ce besoin.

Les règles de reconfiguration stipulent, dans un premier temps, la propriété FTPL à valider, décrite après l'utilisation du mot-clé `when` qui délimite l'intervalle d'évènements durant lequel la propriété est évaluée à 'vrai'. Dans un second temps, les règles de reconfiguration contiennent une garde sur la configuration du système, qui est décrite après l'utilisation du mot-clé `if` qui définit la configuration courante du système qui peut entraîner une reconfiguration. Pour finir, le nom de la reconfiguration R_N et son utilité F est renseignée sous forme de valeur floue (`high, medium, low`) après le mot clé *utility* afin d'associer une utilité avec un nom d'opération de reconfiguration.
Une règle de reconfiguration s'écrit selon le modèle suivant :

when *trigger b*
  if *guard g*
  then utility of *reconfiguration R* is *p*

Dans lequel : *trigger* $b = \wedge_{b_j \in B} b_j$ est une condition temporelle pour le déclenchement de la reconfiguration sur la valeur de la propriété FTPL. *guard* $g = \wedge_{g_l \in G} g_l$ est une condition de logique du premier ordre sur la configuration du système pour lequel la reconfiguration peut avoir lieu ; *reconfiguration* $R \in R_N$ est le nom de la reconfiguration concernée, avec le niveau de priorité indiqué par p. Reprenons l'exemple VANet, afin de garantir les propriétés FTPL, nous devons réfléchir aux reconfigurations à effectuer. C'est le rôle des politiques d'adaptation et c'est au niveau de leurs règles de reconfiguration que ces reconfigurations sont spécifiées. Deux exemples de règles de reconfiguration sont données en figure 3.6.
Ces deux exemples s'appliquent aux véhicules et servent à déterminer les moments de passage de relais entre le véhicule *leader* et un autre véhicule du convoi.
Dans le premier cas, la reconfiguration `PassRelay` se déclenche, pour un véhicule au sein d'un convoi *(after Join before acceptQuit*) lorsqu'il n'a plus assez de batterie (VehicleId.battery < 33) et qu'il est $leader(state = leader)$. Dans ce cas, le véhicule doit être rétrogradé.
Dans le second cas, la reconfiguration `GetRelay` se déclenche, pour un véhicule au sein d'un convoi (*after Join before acceptQuit*) lorsqu'il a plus de batterie que le véhicule leader *(VehicleId.battery > Leader.battery*) et que son état est bien au sein du convoi (*state = platooned*). Dans ce cas, le véhicule devrait remplacer le *leader* actuel.

```
when (after Join before acceptQuit and VehicleId.battery < 33)
   if (state = leader) then
        utility of PassRelay is high

when (after Join before acceptQuit and VehicleId.battery > Leader.battery)
   if (state = platooned) then
        utility of GetRelay is medium
```

FIGURE 3.6 – Deux règles de reconfiguration du convoi de véhicules autonomes

Les définitions et notations des systèmes adaptatifs étant établies, nous présentons à partir du chapitre suivant, les contributions de cette thèse.

Cette partie à pour but de détailler chacune des contributions faites autour du SUT : Critères de couverture, génération de tests et de configurations initiales, détection d'erreurs/incohérences, méthodes d'analyse (politiques d'adaptation et respect des propriétés extra-fonctionnelles).

4

PROPOSITION DE MODÈLE POUR LES SYSTÈMES À COMPOSANTS

Sommaire

4.1 Modèle à composants 59
4.1.1 Élements architecturaux 61
4.1.2 Relations architecturales 63
4.1.3 Instances architecturales 66
4.2 Contraintes de cohérence 68
4.2.1 Contraintes de cohérence pour les systèmes à composants . . . 69
4.3 Conclusion 73

Dans ce chapitre, nous abordons la définition d'un modèle pouvant décrire les systèmes à composants et qui est également adapté pour les systèmes adaptatifs. Ce modèle se base sur les travaux de [97] dans lequel nous avons ajouté des contributions que nous détaillons au fur et à mesure de ce chapitre. Dans un premier temps, nous définissons l'architecture du modèle à composants permettant de décrire formellement les systèmes à composants formés d'éléments, relations ainsi que notre contribution sur les instances des configurations. Ensuite, nous proposons de décrire les contraintes de cohérence du modèle à composants pour les systèmes à composants qui expriment les conditions que nous souhaitons préserver pour la cohérence et le bon fonctionnement d'un système à composants.

4.1/ MODÈLE À COMPOSANTS

Nous avons choisi un modèle architectural permettant de définir les contraintes du système que nous explicitons en section 4.2.1 afin d'avoir les conditions nécessaires à la cohérence et au bon fonctionnement du système. Le modèle choisi (partie encadrée en bleu dans la figure 4.1 est un modèle à composants qui permet de répondre aux exigences

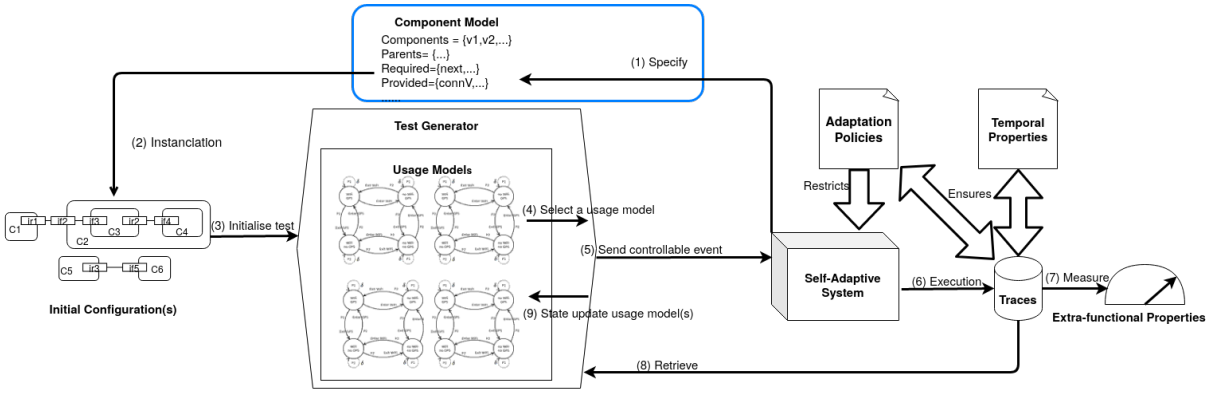

FIGURE 4.1 – Partie du processus étudiée dans cette section

.

suivantes : i) Les composants constituent la configuration du système. ii) Les composants sont liés entre eux par le biais d'interfaces fournies et requises. iii) La hiérarchisation des composants est nécessaire en composants composites et primitifs. iv) Certains composants peuvent avoir des variables que l'on appellera paramètres qui ont des valeurs.
La principale contribution de ce chapitre est la possibilité de générer plusieurs instances du même type de composant. Ainsi, nous faisons la distinction entre les types d'éléments architecturaux et les instances d'éléments architecturaux ainsi qu'entre les types de relations architecturales et les instances de relations architecturales. Notre modèle permet une gestion dynamique de ces instances qui peuvent être ajoutées ou retirées lors de l'exécution.

Nous commençons par définir la notion de configuration contenant à la fois les éléments architecturaux, les relations architecturales et les instanciations des éléments et relations.

Définition 26 : Configuration

Une configuration c est un triplet $\langle Elem, Rel, Inst \rangle$ où :

- $Elem$ est un ensemble d'éléments architecturaux,
- Rel est un ensemble de symboles de relations fonctionnelles,
- $Inst$ est un ensemble d'instances des éléments $Elem$ et Rel.

Les éléments architecturaux correspondent aux types de composants, interfaces et paramètres. Les relations architecturales correspondent aux liens entre composants d'un certain type, liens de hiérarchie (parents/enfants) et délégations. Les instances correspondent aux occurrences architecturales des éléments architecturaux et des relations fonctionnelles.

4.1.1/ ÉLEMENTS ARCHITECTURAUX

Notre modèle à composants définit les instances à part. En conséquence, les éléments architecturaux se focalisent sur les types des composants comme nous pouvons le voir en définition 27 qui reprend la définition faite dans [97].

Définition 27 : Éléments architecturaux

L'ensemble des éléments architecturaux $Elem$ est défini par :
$Elem = \{CType, IProvided, IRequired, IType, Parameter, PType, Contingency, State\}$

Nous explicitons ces éléments ci-dessous.

— $CType$ est un ensemble de types pour les composants,

— $IProvided$ est un ensemble d'interfaces fournies par les différents composants,

— $IRequired$ est un ensemble d'interfaces requises par les différents composants,

— $IType$ est un ensemble fini de types d'interfaces,

— $Parameter$ est un ensemble de paramètres qui sont les variables des différents composants,

— $PType$ est un ensemble fini de types de paramètres,

— $Contingency$ est un ensemble fini contenant à la fois les valeurs possibles de la contingence ainsi que la cardinalité permettant d'indiquer si un ou plusieurs liens peuvent s'effectuer sur l'interface,

— $State$ est un ensemble fini contenant les valeurs possibles de l'état d'un type de composant.

L'exemple 6 illustre un ensemble $Elem$ de la définition 27 pour l'exemple VANet.

Exemple 6 (*Ensemble* $Elem$ *pour VANet*) : *Cet exemple est accompagné de la figure 4.2.*

— $CType = \{Vehicle, Platoon, Station, Road\}$
Les types de composants utilisés représentent des véhicules $(Vehicle)$ *sur la route* $(Road)$ *et vont se recharger aux stations* $(Station)$*. Si ces véhicules se regroupent ils forment des convois* $(Platoon)$*.*

— $IProvided = \{connV, leader, docks\}$
Les interfaces fournies utilisées permettent aux véhicules de se connecter à la route ou à un convoi $(connV)$*, mais également à une station* $(docks)$ *et à un convoi de se connecter à la route* $(leader)$*.*

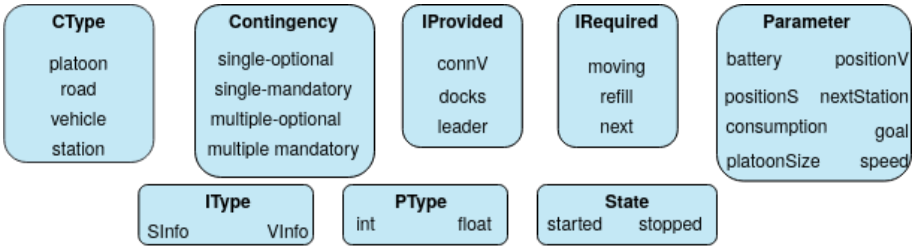

FIGURE 4.2 – Éléments de configuration

— $IRequired = \{moving, next, refill\}$
Les interfaces requises utilisées permettent à la route d'être connectée à des véhicules ou convois $(moving)$ *; à un véhicule d'être connecté à un autre véhicule* $(next)$ *; à une station d'être connectée à un véhicule* $(refill)$.

— $IType = \{VInfo, SInfo\}$
Les types d'interfaces utilisés sont $VInfo$ *qui permettent de partager respectivement les informations du type* $Vehicle$ *et* $Station$.

— $Parameter = \{battery, consumption, positionS, positionV, speed, goal, nextStation, platoonSize\}$
Les paramètres sont utilisés pour apporter des mesures au système et indiquer le niveau de batterie $(battery)$, *la consommation* $(consumption)$, *la position* $(positionV)$, *la vitesse* $(speed)$ *et la destination* $(goal)$ *d'un véhicule. Le paramètre* $platoonSize$ *indique le nombre de véhicules présents dans le convoi. Le paramètre* $positionS$ *est utilisé pour indiquer la position d'une station sur la route et* $nextStation$ *la distance avec la station suivante la plus proche.*

— $PType = \{int, float\}$
Le type choisi pour représenter les paramètres est soit un nombre entier (int) *soit un nombre flottant* $(float)$.

— $Contingency = \{single - optional, single - mandatory, multiple - optional, multiple - mandatory\}$
La contingence est liée aux interfaces, elle prend en compte le caractère obligatoire ou facultatif de ce lien $(optional/mandatory)$. *Nous avons ajouté par rapport au travaux de [97] la prise en compte du nombre d'interfaces auxquelles la lier* $(single/multiple)$.

— $State = \{started, stopped\}$
L'état d'un composant indique si le composant est en marche $(started)$ *ou à l'arrêt* $(stopped)$.

Les éléments de l'exemple 6 sont visibles dans la figure 4.2. Comme nous pouvons le voir, aucune relation n'existe pour le moment entre ces éléments.

4.1.2/ RELATIONS ARCHITECTURALES

Dans cette section, nous définissons les relations entre éléments qui, pour rappel, correspondent à l'élément Rel de la configuration $c = \langle Elem, Rel, Inst \rangle$. Pour définir les relations architecturales, nous reprenons la définition faite dans [97].

Définition 28 : Relations architecturales

L'ensemble de symboles de relations architecturales Rel est défini par :
$Rel = \{IProvidedType, IRequiredType, Provider, Requirer, Binding, IContingency, ParameterType, Definer, ParentType, DelegateProv, DelegateReq, InitState\}$

Nous explicitons ces relations ci-dessous :

— $IProvidedType : IProvided \rightarrow IType$ est une fonction totale qui associe à chaque interface fournie un type d'interface,

— $IRequiredType : IRequired \rightarrow IType$ est une fonction totale qui associe à chaque interface requise un type d'interface,

— $Provider : IProvided \rightarrow CType$ est une fonction totale surjective qui associe à chaque interface fournie le type de composant auquel elle appartient,

— $Requirer : IRequired \rightarrow CType$ est une fonction totale surjective qui associe à chaque interface requise le type de composant auquel elle appartient,

— $Binding : (CType \times IProvided) \nrightarrow (CType \times IRequired)$ est une fonction partielle qui spécifie les liens entre des types de composants via leurs interfaces fournies et requises,

— $IContingency : (IRequired \cup IProvided) \rightarrow Contingency$ est une fonction totale qui associe à chaque interface requise ou fournie la contingence correspondante,

— $ParameterType : Parameter \rightarrow PType$ est une fonction totale qui associe à chaque paramètre son type,

— $Definer : Parameter \rightarrow CType$ est une fonction totale qui associe à chaque paramètre le type de composant auquel il appartient,

— $ParentType : CType \rightarrow \{CType \cup \emptyset\}$ est une fonction qui lie un type de sous composant au type de composant qui peut le contenir.

— $DelegateProv : \{IProvided \nrightarrow IProvided\} \cup \emptyset$ est une fonction partielle et injective qui spécifie la délégation entre une interface fournie d'un type de sous-composant et une interface fournie du type de composant qui le contient.

— $DelegateReq : \{IRequired \nrightarrow IRequired\} \cup \emptyset$ est une fonction partielle et injective qui spécifie la délégation entre une interface requise d'un type de sous-composant et une interface requise du type de composant qui le contient.

— $InitState : CType \rightarrow State$ est une fonction totale qui lie un type de composant à son état

Il est à noter que nous avons modifié la relations $Binding$ par rapport aux travaux de [97] pour prendre en compte la différence entre les types et les instances.

L'exemple 7 illustre l'ensemble Rel de la définition 28 pour l'exemple VANet détaillé précédemment.

Exemple 7 (*Relations architecturales de l'exemple VANet*) : *Cet exemple est accompagné de la figure 4.3.*

— $IProvidedType = \{(connV \mapsto VInfo), (docks \mapsto SInfo), (leader \mapsto VInfo)\}$
Le type d'interface fournie $connV$ qui appartient aux véhicules est mis en relation avec le type $VInfo$. Le type d'interface fournie $docks$ qui appartient aux stations est mis en relation avec le type $SInfo$. Le type d'interface fournie $leader$ qui appartient aux convois est mis en relation avec le type $VInfo$.

— $IRequiredType = \{(moving \mapsto VInfo), (next \mapsto VInfo), (refill \mapsto SInfo)\}$,
Les types d'interfaces requises $moving$ et $next$ sont mis en relation avec le type $VInfo$ pour partager les informations du type $Vehicle$. Le type d'interface requise $refill$ est mis en relation avec le type $SInfo$ pour partager les informations du type $Station$.

— $Provider = \{(connV \mapsto Vehicle), (leader \mapsto Platoon), (refill \mapsto Station)\ \}$
La fonction $Provider$ spécifie que l'interface fournie $connV$ est liée au type de composant $Vehicle$; que $leader$ appartient au type de composant $Platoon$; que l'interface fournie $refill$ appartient au type de composant $Station$.

— $Requirer = \{(moving \mapsto Road), (next \mapsto Vehicle), (docks \mapsto Vehicle)$
La fonction $Requirer$ spécifie que l'interface requise $moving$ appartient au type de composant $Road$, et que $next$ et $docks$ appartiennent à $Vehicle$.

— $Binding = \{((Vehicle, connV) \mapsto (Vehicle, next)), ((Vehicle, connV) \mapsto (Road, moving)), ((Station, refill) \mapsto (Vehicle, docks)), ((Vehicle, connV) \mapsto (Platoon, leader)), ((Platoon, leader) \mapsto (Road, moving))\ \}$
La fonction $Binding$ spécifie que les types de composants $Vehicle$ sont liés au travers de leurs interfaces $connV$ et $next$; les types $Vehicle$ et $Road$ sont liés au travers de leurs interfaces $connV$ et $moving$; les types $Station$ et $Vehicle$ sont liés au travers de leurs interfaces $refill$ et $docks$; les types $Vehicle$ et $Platoon$ sont liés au travers de leurs interfaces $connV$ et $leader$; les types $Platoon$ et $Road$ sont liés au travers de leurs interfaces $leader$ et $moving$.

— $IContingency = \{connV \mapsto (single-mandatory), leader \mapsto (single-mandatory), refill \mapsto (multiple-optional), moving \mapsto (multiple-optional), next \mapsto (single-optional), leader \mapsto$

$(single - mandatory), docks \mapsto (multiple - optional)\}$

Chaque interface possède obligatoirement une contingence. Les contingences $mandatory$ *obligent les interfaces* $connV$ *et* $leader$ *à être liées à une interface requise. Comme ces interfaces appartiennent aux composants respectifs* $Vehicle$ *et* $Platoon$*, ces composants possèdent au moins une relation* $Binding$ *en rapport avec ces interfaces. Les autres interfaces sont optionnelles. La cardinalité des interfaces peut être* $single\ (connV,\ next,\ leader)$ *pour restreindre l'interface à un unique lien ou* $multiple\ (refill,\ moving,\ docks)$ *pour ne pas restreindre le nombre de liens de l'interface.*

— $ParameterType = \{battery \mapsto int,\ positionS \mapsto int,\ positionV \mapsto int,\ nextStation \mapsto int,\ speed \mapsto int,\ goal \mapsto int,\ platoonSize \mapsto int,\ consumption \mapsto float\ \}$

Les types assignés à chaque paramètre dépendent du niveau de précision de la spécification. La distance restante à accomplir $(goal)$ *en kilomètres est exprimée en nombres entiers, il en est de même pour la vitesse* $(speed)$ *en kilomètres/heure, les positions des véhicules* $(positionV)$ *et stations* $(positionS,\ nextStatopn)$ *et le niveau de batterie* $(battery)$ *exprimé en pourcentage. Pour le paramètre* $consumption$*, nous avons choisi de l'exprimer en nombres flottants.*

— $Definer = \{battery \mapsto Vehicle,\ positionV \mapsto Vehicle,\ positionS \mapsto Station,\ nextStation \mapsto Station,\ speed \mapsto Vehicle,\ goal \mapsto Vehicle,\ consumption \mapsto Vehicle,\ platoonSize \mapsto Platoon\ \},$

Les paramètres $battery,\ positionV,\ speed,\ goal$ *et* $consumption$ *sont associés au type* $Vehicle$*, les paramètres* $positionS$ *et* $nextStation$ *sont associés au type* $Station$*, et le paramètre* $platoonSize$ *est associé au type* $Platoon$*.*

— $ParentType = \{Road \mapsto \emptyset,\ Platoon \mapsto \emptyset,\ Vehicle \mapsto \emptyset,\ Vehicle \mapsto Platoon,\ Station \mapsto \emptyset\ \}$

Les types de composants $Road,\ Platoon$ *et* $Station$ *ne sont en relation avec aucun composant parent. Le type de composant* $Vehicle$ *peut être en relation avec son composant parent* $Platoon$ *lorsqu'il est dans un convoi ou peut ne pas être en relation avec un composant parent lorsqu'il est seul.*

— $DelegateProv = \{connV \mapsto leader\}$

La fonction partielle $DelegateProv$ *permet de déléguer l'interface fournie* $connV$ *du véhicule leader inclus dans le convoi avec l'interface fournie* $leader$ *de son convoi.*

— $DelegateReq = \emptyset$

Il n'y a pas de délégation entre interfaces requises dans l'exemple VANet.

— $InitState = \{Road \mapsto started, Platoon \mapsto started, Vehicle \mapsto started\}$

La fonction $InitState$ *permet de donner l'état par défaut de chaque type de composant.*

Les relations de l'exemple 7 sont visibles dans la figure 4.3. Comme nous pouvons le voir, ces relations permettent d'organiser ensemble les éléments de la figure 4.2. Tous ces

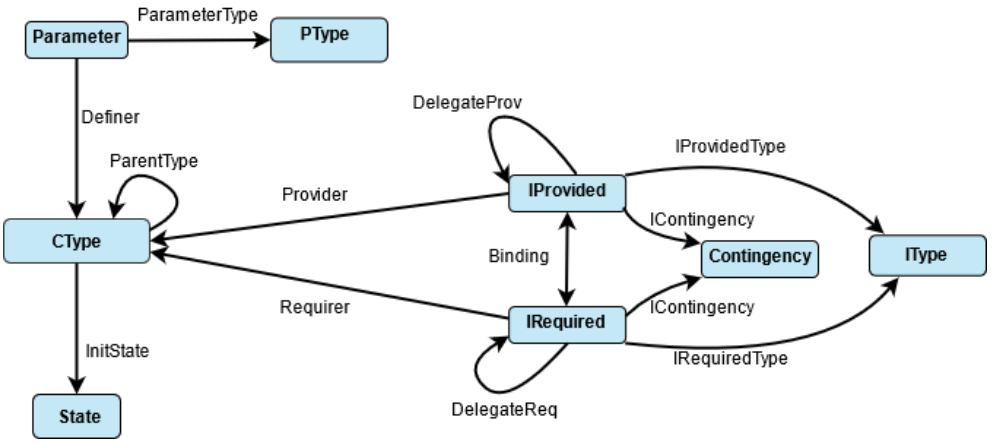

FIGURE 4.3 – Relations architecturales

éléments et relations sont organisés en fonction de leurs types et non de leurs instances que nous nous proposons de décrire dans la section suivante.

4.1.3/ INSTANCES ARCHITECTURALES

Dans cette section, nous présentons notre contribution sur la définition d'instances architecturales des éléments et relations précédemment définies, ces instances correspondent à l'élément $Inst$ de la configuration $c = \langle Elem, Rel, Inst \rangle$. Les travaux de [10] présentent également une méthode de définition d'instances, cependant ces instances sont définies à l'avance de manière statique. Cette vision est similaire à la nôtre pour ce chapitre sur la modélisation de configurations mais ne permet pas de faire apparaître ou disparaître des éléments lors des reconfigurations au moment de l'exécution du système.

Définition 29 : Instances architecturales

L'ensemble des instances architecturales $Inst$ est défini par :
$Inst = \{Components, ComponentTypes, ParentInstances, Binds, DelProvs, DelReqs, States, Params\}$

Ces instances sont explicitées ci-dessous :

— $Components$ est un ensemble fini non vide d'instances de composants de $CType$.

— $ComponentTypes : Components \rightarrow CType$ est une fonction totale qui associe chaque instance de composant à son type.

— $ParentInstances : Components \nrightarrow (Components \cup \emptyset)$ est une fonction partielle qui associe un sous composant au composant qui le contient.

— $Binds : (Components \times IProvided) \nrightarrow (Components \times IRequired)$ est une fonction partielle qui est utilisée pour lier entre-elles des interfaces fournies et requises.

— $DelProvs : \{(Components \times IProvided) \nrightarrow (Components \times IProvided)\} \cup \emptyset$ est une fonction partielle qui spécifie les délégations entre deux interfaces fournies.

— $DelReqs : \{(Components \times IRequired) \nrightarrow (Components \times IRequired)\} \cup \emptyset$ est une fonction partielle qui spécifie les délégations entre deux interfaces requises.

— $States : Components \rightarrow State$ est une fonction totale qui associe à chaque instance de composant son état.

— $Params : Components \nrightarrow (Parameter \nrightarrow PType)$ est une fonction partielle qui, pour un composant donné, associe la fonction partielle qui associe un type à un paramètre donné.

Maintenant que les instances ont été définies, nous présentons l'exemple 8 qui illustre cet ensemble $Inst$ de la définition 29.

Exemple 8 (*Instances architecturales de l'exemple VANet*) : *Cet exemple reprend les instances de la configuration architecturale visible dans la figure 4.4. Afin de ne pas surcharger la figure, nous avons choisi de représenter uniquement les instances d'interfaces liées. Cette configuration est la version architecturale de l'exemple du convoi de véhicules VANet.*

— $Components = \{v1.1, v1.2, v1.3, v2.1, v2.2, v3, v4, v5, r, p1, p2\}$
Chaque composant de l'exemple est défini ici. Le choix du nom est libre. Nous avons choisi pour la compréhension de l'exemple, de nommer les composants avec l'initiale de leur type et un nombre permettant de les identifier. Pour les véhicules présents dans un convoi, leur nom est composé du numéro du convoi les incluant ainsi qu'un nombre indiquant leur place au sein de ce convoi.

— $ComponentTypes = \{v1.1 \mapsto Vehicle, v1.2 \mapsto Vehicle, v1.3 \mapsto Vehicle, v2.1 \mapsto Vehicle, v2.2 \mapsto Vehicle, v3 \mapsto Vehicle, v4 \mapsto Vehicle, v5 \mapsto Vehicle, p1 \mapsto Platoon, p2 \mapsto Platoon, s \mapsto Station, r \mapsto Road\}$
Cette fonction nous permet d'associer les instances de composant à leur type de composant.

— $ParentInstances = \{v1.1 \mapsto p1, v1.2 \mapsto p1, v1.3 \mapsto p1, v2.1 \mapsto p2, v2.2 \mapsto p2, v3 \mapsto \emptyset, ..., p \mapsto \emptyset, r \mapsto \emptyset, s \mapsto \emptyset\}$
Le composant $p1$ est parent des véhicules v1.1, v1.2 et v1.3 et p2 est parent des véhicules $v2.1$ et $v2.1$.

— $Binds = \{ ((v3, connV) \mapsto (r, moving)), ((v4, connV) \mapsto (r, moving)),$
$((v5, connV) \mapsto (r, moving)), ((v5, connV) \mapsto (r, moving)), ((p1, leader) \mapsto (r, moving)),$
$((p2, leader) \mapsto (r, moving)), ((v1.2, connV) \mapsto (v1.1, next)),$
$((v1.3, connV) \mapsto (v1.2, next)), ((v2.2, connV) \mapsto (v2.1, next)), (s, refill) \mapsto (v5, docks) \}$
Cet ensemble liste les liens entre tous les composants de l'exemple. Si le véhicule $v5$ sort de la station et est accepté dans le convoi $p2$, les liens $(v5, connV), (r, moving)$ et $(s, refill)(v5, docks)$ seront supprimés au profit de $(v2.2, connV), (v2.2, next)$

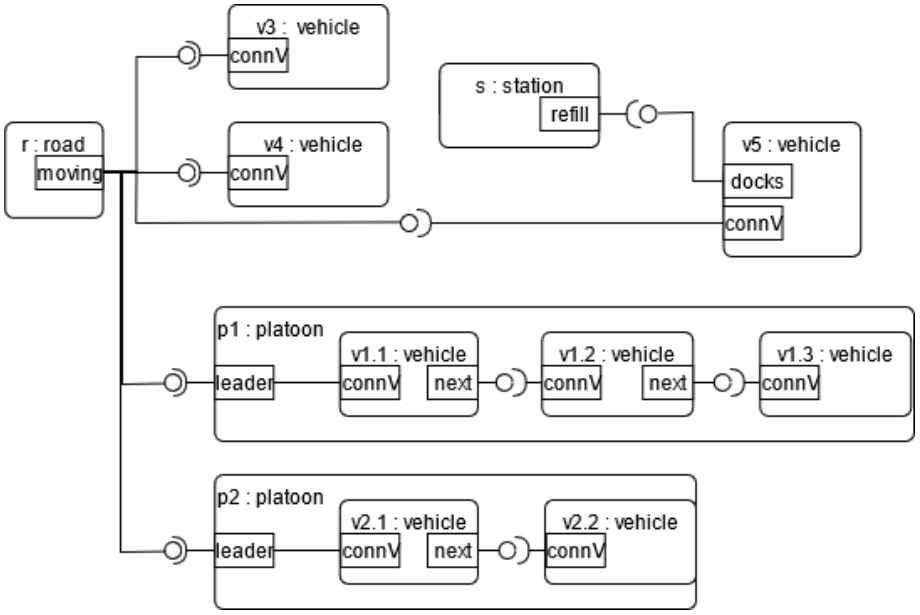

FIGURE 4.4 – Une instance d'architecture à composants de l'exemple VANet

— $DelProvs = \{ ((v1.1, connV) \mapsto (p1, leader)) , ((v2.1, connV) \mapsto (p2, leader)) \}$
Les relations de délégation fournies qui permettent aux instances de composants $v1.1$ *et* $v1.2$ *d'être* $leader$ *de leur convoi.*

— $DelReqs = \emptyset$
Dans notre cas il n'y a pas de relation de délégation requise.

— $States : \{v1.1 \mapsto started, v1.2 \mapsto started, v1.3 \mapsto started, v2.1 \mapsto started, v2.2 \mapsto started, v3 \mapsto started, v4 \mapsto started, v5 \mapsto started, p1 \mapsto started, p2 \mapsto started, s \mapsto started\}$
Dans notre exemple, tous les composants sont en état de marche $started$*, lorsqu'un véhicule aura atteint sa destination il sera arrêté* $stopped$*.*

— $Params = \{v1.1 \mapsto \{(battery \mapsto 31), (consumption \mapsto 1.2), (positionV \mapsto 253), (speed \mapsto 91), (destination \mapsto 331)\}, v1.2 \mapsto \{...\}, ..., s \mapsto \{(positionS \mapsto 291), (nextStation \mapsto 42)\}\}$
Les valeurs de chaque paramètre sont présentes ici. Ces valeurs peuvent être changées à chaque reconfiguration.

Maintenant que la configuration du modèle est définie, nous pouvons présenter les contraintes de cohérence du modèle.

4.2/ CONTRAINTES DE COHÉRENCE

Dans la section précédente, nous avons présenté le modèle à composants permettant de décrire l'architecture des systèmes à composants.
Dans cette section, nous présentons les contraintes de cohérence inspirées de [97] qui

ont été adaptées pour s'appliquer à la refonte du modèle et assurer que les différentes configurations du système soient cohérentes sur le plan architectural. Les contraintes de cohérence permettent de garantir que le système est dans une configuration permettant son bon fonctionnement et sont applicables pour tout système compatible avec une configuration à composants.

4.2.1/ CONTRAINTES DE COHÉRENCE POUR LES SYSTÈMES À COMPOSANTS

Nous présentons dans cette section les contraintes de cohérence. Pour faciliter la lisibilité nous avons choisi de définir, quand c'est nécessaire, les éléments suivants :

— $Interface = IProvided \cup IRequired$ est l'union des ensembles $IProvided$ et $IRequired$,

— $Delegate = DelegateProv \cup DelegateReq$ est l'union des ensembles $DelegateProv$ et $DelegateReq$,

— $Dels = DelProvs \cup DelReqs$ est l'union des ensembles $DelProvs$ et $DelReqs$,

— $InterfaceType = IProvidedType \cup IRequiredType$ est l'union des ensembles $ProvidedType$ et $IRequiredType$.

— $Containers : Interface \mapsto Components$ est une fonction partielle qui associe une interface à l'ensemble des instances de composants qui les contiennent.

Les contraintes sont présentées dans trois sous-sections. Dans un premier temps, nous présentons les contraintes liées au typage, puis les contraintes liées aux constructions hiérarchiques, et enfin les contraintes liées aux interfaces.

4.2.1.1/ CONTRAINTES LIÉES AU TYPAGE

Les contraintes liées au typage sont les suivantes :

— Soient deux types de composants, si leurs interfaces sont liées, alors elles doivent être de même type et appartenir à des composants ayant un parent commun ($\emptyset$ est un parent possible).

$\zeta_1 : \forall c1, c2 \in CType, \exists ip \in IProvided, \exists ir \in IRequired.\{(c1, ip) \mapsto (c2, ir)\} \triangleleft Binding) \neq \emptyset \Rightarrow IProvidedType(ip) = IRequiredType(ir) \wedge (\exists p \in CType.ParentType(Provider(ip)) = p \wedge ParentType(Requirer(ir)) = p)$

— Le typage des éléments de l'ensemble $Binds$ doit être cohérent avec les éléments des ensembles $Provided$, $Requirer$ et $Binding$.

$\zeta_2 :\forall c1, c2 \in Components, \forall i1, i2 \in Interface$. Binds(c1,i1)=(c2,i2) $\Rightarrow ComponentTypes(c1) = Provider(i1) \wedge ComponentTypes(c2) = Requirer(i2) \wedge (card(componentType(c1), i1) \mapsto (componentType(c2), i2) \triangleleft Binding) = 1)$

Cela signifie que si un couple d'instances $(composant, interface)$ est présent dans $Binds$ alors ce couple est également présent dans $Provider$ ou $Requirer$ et $Binding$.

— L'ensemble des paramètres du codomaine de $Params$ doit être égal à l'ensemble des paramètres du type de composant du domaine de $Definer$.
$\zeta_3 : \forall$ c $\in Components, dom(ran(Params(c))) = dom(Definer \triangleright \{ComponentTypes(c)\})$

Nous détaillons maintenant la contrainte ζ_3 dans l'exemple 9 ci-dessous :

Exemple 9 (*Détail de la contrainte* ζ_3) : *La contrainte de cohérence* $\zeta_3 : \forall c \in Components, dom(ran(Params(c))) = (dom(Definer) \triangleright \{ComponentTypes(c)\})$ *exprime le fait que l'ensemble des paramètres* $Parameter$ *associés à* $Params$ *doivent appartenir à l'ensemble des paramètres* $Parameter$ *associés à* $Definer$ *et réciproquement.*
Dans notre exemple, où une illustration est visible en figure 4.5, nous allons comparer deux configurations : la configuration A visible à gauche et la configuration B visible à droite.
La fonction $Params$ *associe à chaque composant* $(Components)$ *la fonction qui associe chacun des paramètres* $(Parameter)$ *du composant à leur type* $PType$. $Definer$ *est une fonction qui associe à chaque paramètre* $(Parameter)$ *le type de composant* $(CType)$ *auquel il appartient.*
Soit $ParamInst = dom(ran(Params(c)))$.
Soit $ParamDefiner = dom(Definer) \triangleright \{ComponentTypes(c)\}$.
Pour que la contrainte soit valide, il faut que l'ensemble $ParamInst$ *soit égal à l'ensemble* $ParamDefiner$.
Pour la Configuration A, l'ensemble $ParamInst = \{speed, goal, battery, positionS\}$ *et l'ensemble* $ParamDefiner = \{speed, positionV, battery, positionS\}$.
Or, d'une part le paramètre $positionV$ *n'appartient pas à l'ensemble* $ParamInst$ *et d'une autre part le paramètre* $goal$ *n'appartient pas à l'ensemble* $ParamDefiner$. *La contrainte de consistance n'est donc pas respectée.*
En revanche, dans la configuration B, l'ensemble $ParamInst = ParamDefiner = \{speed, positionV, goal, battery, positionS\}$. *Ce qui implique que la contrainte de cohérence* ζ_3 *est valide pour la configuration B.*

4.2.1.2/ CONTRAINTES LIÉES AUX INTERFACES

Les contraintes liées aux interfaces sont les suivantes :

— Un composant contient au moins une interface fournie.
$\zeta_4 \ \forall c \in Components \Rightarrow (\exists ip \in IProvided.Containers(ip) = c)$

— Soient deux interfaces liées par une instance de relation $Binds$, elles ne peuvent être liées à une délégation et à une interface d'un de leur parent en même temps.
$\zeta_5 : \forall ip \in IProvided, \forall ir \in IRequired, \forall id \in Interface, \forall c1, c2 \in Components.$

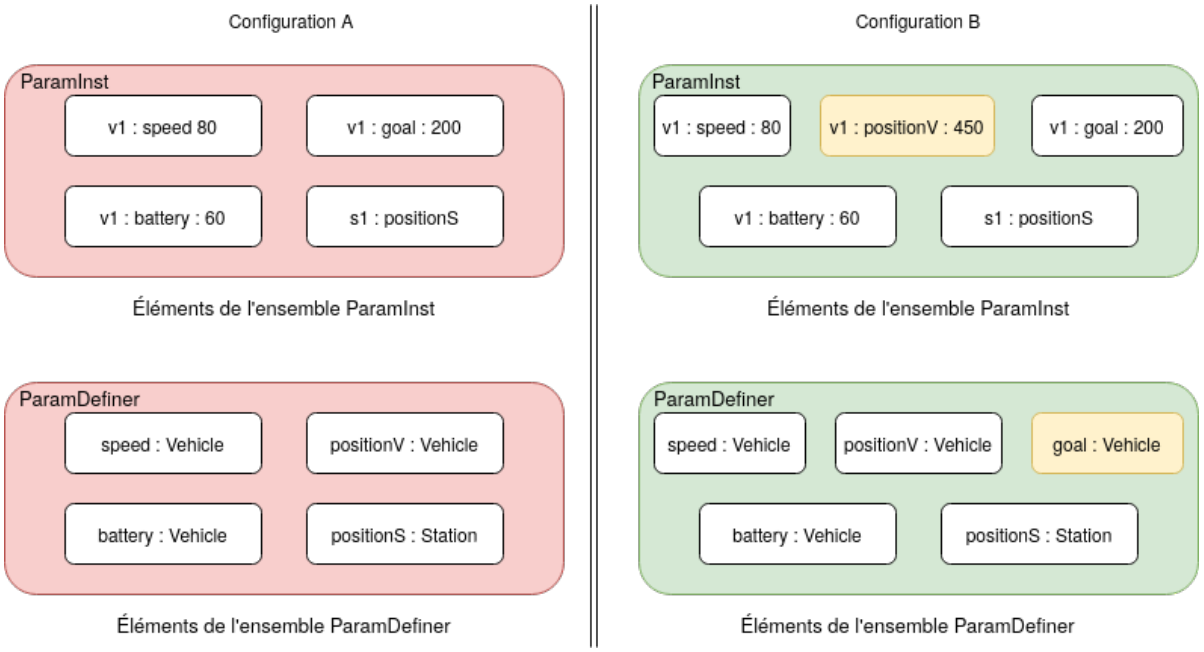

FIGURE 4.5 – Instance de paramètres dans $Params$ et $Definer$

$(Binds(c1, ip) = (c2, ir) \Rightarrow (\{c1, ip\} \triangleleft DelProvs = \emptyset) \wedge (\{c2, ir\} \triangleleft DelReqs = \emptyset))$

— Réciproquement, si deux interfaces sont liées par une relation $Dels$, elles ne peuvent être liées en même temps par une relation $Binds$.
$\zeta_6 : \forall i, i' \in Interface, \exists c1, c2 \in Component.Dels(c1, i) = (c2, i') \Rightarrow (\{c1, i\} \triangleleft Binds = \emptyset) \wedge (\{c2, i'\} \triangleleft Binds = \emptyset)$

— Étant donné un sous-composant et un de ses composants parent, une interface fournie (resp. requise) appartenant au sous-composant est déléguée à une seule interface fournie (resp. requise) (ζ_{10}), de même type (ζ_7, ζ_8), appartenant au même composant parent (ζ_9).
$\zeta_7 : \forall i, i' \in Interface.(Delegate(i) = i' \wedge i \in IProvided \Rightarrow i' \in IProvided)$
Soient deux interfaces reliées par une délégation, si l'interface qui délègue appartient à l'ensemble $IProvided$ alors l'autre interface aussi.
$\zeta_8 : \forall i, i' \in Interface.(Delegate(i) = i' \wedge i \in IRequired \Rightarrow i' \in IRequired)$
Soient deux interfaces reliées par une délégation, si l'interface déléguée appartient à l'ensemble $IRequired$ alors l'autre interface aussi.
$\zeta 9 : \forall i1 \in IProvided, \forall ir \in IRequired.(Delegate(i1) = i2) \Rightarrow (InterfaceType(i1) = InterfaceType(i2)) \wedge ParentType(Containers(i1) = Containers(i2))$
Soient deux interfaces reliées par une délégation, elles sont de même type et l'interface déléguée appartient à un composant parent du composant contenant l'interface qui délègue.

— Une interface fournie est déléguée à une seule interface.
$\zeta_{1O} : \forall i \in IProvided \Rightarrow card(\{i\} \triangleleft Delegate) = 1$
Si une interface est déléguée à deux interfaces à la fois alors ces interfaces sont une seule et même interface.

— La contigence *single* d'une interface restreint son nombre maximal de liens à 1.
$\zeta_{11} : \forall i1 \in Interface.ran(IContingency(i1)) \in \{single - mandatory, single - optional\} \Rightarrow \forall c1 \in Components.(c1 \in Containers(i1) \wedge card(\{c1, i1\} \triangleleft ran(Binds)) \leq 1)$
Si la contingence d'une interface est *single* alors le nombre d'occurrences de cette interface pour un composant donné est inférieur ou égal à un.

4.2.1.3/ CONTRAINTES LIÉES AUX CONSTRUCTIONS HIÉRARCHIQUES

Les contraintes liées aux constructions hiérarchiques sont les suivantes :

— Aucun type de composant ne peut descendre de son type.
$\zeta_{12} : \forall c, c' \in CType.(c = c') \Rightarrow (ParentType(c) \neq c' \wedge ParentType(c') \neq c)$

— Un composant peut être en état démarré (état *started*) si et seulement si ses interfaces requises avec une contigence *mandatory* sont liées par une relation *Binding* ou *Delegate* :
$\zeta_{13} : \forall ir \in IRequired, \exists c \in Components\ .\ (IContingency(ir) = (single - mandatory \vee multiple - mandatory) \wedge c \in Containers(ir) \wedge States(c) = started \Rightarrow (\{c, ir\} \triangleleft Binds \neq \emptyset) \vee (\exists i2 \in Interface\ .\ DelegateProv(i2) = ir \vee DelegateReq(ir) = i2)))$
Pour toute instance de composant avec un état *started* contenant un interface requise de contingence *mandatory* alors il existe une autre instance de composant contenant une interface tel que ces deux instances de composants soient liées par *Binds* ou *Dels*.

— La relation *ParentInstances* est cohérente avec la relation *ParentType*.
$\zeta_{14} : \forall c1 \in Components, \exists p \in Components\ .\ ParentInstances(c1) = p \Rightarrow ParentType(ComponentTypes(c1)) = ComponentTypes(p)$

Pour toute instance de composant associé à une relation *ParentInstances* il existe le type du composant dans la relation *ParentType* et l'instance du composant est dans *ParentInstances*.

4.3/ CONCLUSION

Dans ce chapitre, nous avons formalisé la notion de configuration en définissant les différents éléments architecturaux, les relations entre eux ainsi que notre contribution sur les instances de ces composants et relations rendant possible l'instantiation multiple du même type de composant.
Nous avons également décrit les contraintes de cohérence sur les configurations de systèmes à composants pour qu'elles soient applicables sur les éléments, les relations et les instances afin de garantir que le système soit dans un état permettant son bon fonctionnement.
Dans le chapitre suivant, nous présentons comment nous générons des configurations initiales et des évènements de test permettant de stimuler le système.

5

GÉNÉRATION DE TESTS EN LIGNE POUR LES SYSTÈMES ADAPTATIFS

Sommaire

5.1 Problématiques du test de systèmes adaptatifs 76

5.2 Générateur de configurations initiales 77

5.2.1 Algorithme de génération de configurations initiales 78

5.2.2 Échantillonnage des configurations initiales 81

5.2.3 Génération des valeurs pour les paramètres des composants . . 85

5.2.4 Présentation de la Beta-distribution 86

5.2.5 Exemples de variables selon la Beta-distribution 88

5.3 Processus de génération d'évènements de test en ligne 89

5.3.1 Modèle d'usage pour le test en ligne 89

5.3.2 Génération de tests à partir de modèles d'usage 93

5.4 Conclusion 99

Dans ce chapitre, nous étudions les méthodes de génération de tests. Dans un premier temps, nous explicitons les problématiques associées à la génération de tests pour les systèmes adaptatifs et les solutions proposées. Dans un second temps, nous détaillons notre algorithme de génération de configurations initiales qui, à partir des éléments architecturaux détaillés dans le chapitre 4 et de différentes distributions probabilistes génère des instances de configurations initiales en respectant les contraintes de cohérence architecturales. Dans un troisième temps, nous présentons notre processus de génération d'évènements contrôlables qui repose sur des modèles d'usage probabilistes, ainsi qu'un générateur de test en ligne.

5.1/ PROBLÉMATIQUES DU TEST DE SYSTÈMES ADAPTATIFS

Les systèmes adaptatifs réagissent aux évènements externes décrivant l'environnement dans lequel ils évoluent en déclenchant des reconfigurations.
La validation des systèmes adaptatifs lève des questions au niveau des politiques d'adaptation et des reconfigurations déclenchées par le système. En effet, en envoyant un évènement spécifique au système, il peut déclencher une reconfiguration qui pourrait être différente des reconfigurations possibles décrites pas les politiques d'adaptation. Le système peut également ne pas réagir du tout si on lui renvoyait le même évènement à différents moments. Ces différences de réactions sont dues à plusieurs facteurs prenant en compte les configurations du système et les politiques d'adaptation. Les politiques d'adaptation régissent le comportement du système et indiquent quelle reconfiguration déclencher grâce à la configuration du système et de valeurs floues sans pour autant être prescriptives. C'est à cause de l'association de ces facteurs qu'il existe une infinité de possibilités de reconfigurations des systèmes adaptatifs. Il serait très difficile, voire impossible de créer un modèle permettant d'établir toutes les mises en oeuvre possibles des politiques d'adaptation afin de générer toutes les possibilités de reconfigurations du système sans être prescriptif.

Nous proposons dans ce chapitre une approche permettant de générer des cas de tests de manière automatique et diversifiée. Les cas de tests sont constitués de configurations initiales et d'évènements contrôlables qui, associés, permettent de stimuler le système qui génère des logs utilisés pour effectuer la vérification du système que nous aborderons dans le chapitre 6. Nous proposons une approche pour la génération de configurations initiales que nous effectuons en nous basant sur une partie du modèle à composants ainsi que la génération des évènements contrôlables basée sur des modèles d'usage. Nous proposons également une méthode pour la génération automatique d'évènements contrôlables afin de produire des tests de manière systématique. Grâce à cette automatisation, il est possible de générer de nombreux tests de longueur importante permettant d'effectuer des analyses statistiques sur les traces d'exécution obtenues.
L'envoi d'une commande au système ne provoque ni systématiquement ni immédiatement une reconfiguration de sa part. Pour ces raisons, nous choisissons de stimuler le système par l'intermédiaire d'évènements externes (au système) sans faire prévaloir le choix de l'implémentation à la manière d'une boîte noire.
En conséquence, les informations du système nécessaires pour la génération de test sont obtenues via des données fournies par le système telles que les traces d'exécution qui offrent une vision globale du système.
La génération des tests peut s'effectuer en ligne ou hors ligne ; nous avons choisi de générer les tests en ligne. Bien qu'ils soient plus coûteux en temps de génération, les

tests en ligne ont la possibilité de s'adapter aux reconfigurations du système ce qui est indispensable pour la génération d'évènements de test sur les systèmes adaptatifs. Cette capacité d'adaptation au système est primordiale car les tests générés doivent être pertinents et cohérents avec la configuration du système. Sinon, le système pourrait réagir de manière incontrôlée. En effet, il n'est pas possible de savoir à l'avance comment ni quand les systèmes adaptatifs réagissent.
Avec la génération de test en ligne, il nous est donc possible d'observer les reconfigurations du système par l'intermédiaire des traces produites et de générer des tests en fonction de ces observations.

Nous proposons maintenant de définir les étapes de génération de tests abordées dans ce chapitre qui sont visibles dans la figure 5.1 dans la partie bleue. La première étape visible sous l'appellation $Initial\ Configuration(s)$ consiste à générer automatiquement des configurations initiales ce qui permet d'avoir une diversité de configurations. En effet, il nous est possible de faire varier le nombre de composants, les relations avec les autres composants ainsi que les domaines des paramètres initiaux (variables) de manière automatique.
Les configurations initiales respectent les contraintes du système sous test du modèle architectural pour le système à composants visible sous l'appellation $Component\ model$ et décrit dans le chapitre 4. Une fois la configuration générée, elle est envoyée au système via l'étape $(3)\ Initialise\ test$.
Ensuite, chaque composant possède un modèle d'usage spécifique visible sous l'appellation $Usage\ model$ dans la figure 5.1 qui est un modèle probabiliste permettant d'automatiser l'environnement du système sous test.
À chaque pas de test, le générateur de test ($Test\ generator$) sélectionne de manière aléatoire un modèle d'usage ($(4)\ Select\ a\ usage\ model$) parmi ceux disponibles et déclenche la transition du modèle d'usage sélectionné. Cette transition est un évènement contrôlable que le générateur de test envoie au système ($(5)\ Send\ controllable\ event$). En réagissant à ces évènements, le système produit des traces d'exécution ($(6)\ Execution$) qui, dans ce chapitre servent d'information au générateur de test ($(8)\ Retrieve$) pour lui permettre de mettre à jour les états des modèles d'usage concernés ($(9)\ State\ update\ usage\ model(s)$).

Nous présentons dans ce chapitre, comment nous avons mis en place les différents éléments de cette approche en commençant par la génération des configurations initiales.

5.2/ GÉNÉRATEUR DE CONFIGURATIONS INITIALES

Cette section décrit un algorithme de génération de configurations à partir du modèle à composants $\langle Elem, Rel, Inst\rangle$ pour notre système adaptatif [33]. Cet algorithme est utilisé pour énumérer toutes les possibilités définies par le modèle à composants afin de générer

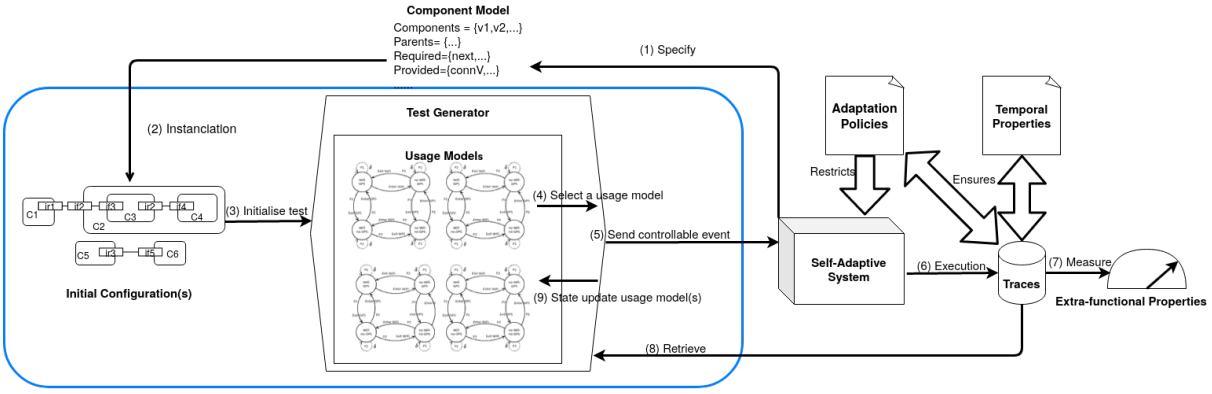

FIGURE 5.1 – Processus de génération de tests en ligne

des configurations et optimisé pour éliminer les configurations symétriques équivalentes. Le but de cet algorithme combinatoire est de construire un ensemble de configurations qui soient correctes par construction.
Une fois cet ensemble construit, il est possible d'échantillonner un sous-ensemble de configurations le plus hétérogène possible. Ces configurations peuvent alors être utilisées comme configurations initiales dans le processus de test visible en figure 5.1.

5.2.1/ ALGORITHME DE GÉNÉRATION DE CONFIGURATIONS INITIALES

L'algorithme de génération de test est résumé dans l'algorithme 1.
Cet algorithme prend en entrée les parties du modèle à composants $Elem$ et Rel afin de produire toutes les instantiations $Inst$ jusqu'à une taille donnée (exprimée en nombre de composants). Afin d'éliminer les configurations non pertinentes vis-à-vis des contraintes spécifiques au système (e.g restreindre le nombre maximal de véhicules dans un convoi) une fonction d'invariants peut être ajoutée pour définir les configurations valides.

L'algorithme 1 est paramétré par : le nombre total de composants (N), une fonction d'invariants ($invariant$) qui permet de fournir les contraintes spécifiques au système, et une fonction d'instantiation de paramètres ($genParameters$) qui détermine les valeurs des paramètres. Les paramètres de la fonction $genParameters$ sont : $Comp$, une liste de composants, CT, une fonction totale qui associe à chaque composant son type, $Definer$, une fonction partielle qui associe une valeur à un paramètre donné et $Elem$ et Rel. L'algorithme combinatoire procède en étapes successives.

Chaque étape consiste à identifier toutes les solutions possibles une par une et de les explorer. Une fois que toutes possibilités ont été explorées dans une étape, l'algorithme retourne à l'étape précédente pour passer à la solution suivante et explorer toutes les possibilités avec cette nouvelle solution. Les différentes étapes sont les suivantes :

Étape 1. L'algorithme commence avec l'étape 1 (ligne 11) en prenant en compte toutes les partitions possibles des composants en se basant sur les bornes maximales de cardinalité. Cette étape se base sur la description de $CTypes$ constituant $Elem$ (cf définition 27). Chaque paire dans $Comps \times CT$ (les instances de composants $Comps$ constituent $Inst$) est prise en compte dans l'étape suivante.

Étape 2. L'étape 2 (ligne 12) consiste à produire, à partir d'un ensemble d'instances de composants, une relation de parenté qui satisfait les contraintes suivantes : (i) chaque composant composite a au moins deux composants enfants, et (ii) aucune boucle ne doit exister dans la relation de parenté. À ce moment, la fonction $isFresh$ (ligne 13) est utilisée pour détecter si la solution obtenue a déjà été rencontrée et prend en compte les équivalences symétriques déjà calculées dans une précédente itération de la boucle. Un exemple d'équivalence symétrique est illustré dans la figure 5.2 où il y a deux véhicules dans un convoi et un véhicule seul dans les deux cas. Si tel est le cas, la solution équivalente n'est pas retenue pour l'étape suivante et l'algorithme continue avec l'élément

```
 1: Inputs
 2:    Elem
 3:    Rel
 4:    N : int
 5:    invariant : Inst → 𝔹
 6:    genParameters : Comp, CT, Definer → Values, Elem, Rel
 7: Output
 8:    SInst // the set of possible instantiations
 9: Begin
10: SInst ← ∅
11: for all Comp, CT from genComponents(N, Elem, Rel) do
12:    for all Parents from genParenting(Comp, CT, Elem, Rel) do
13:       if not isFresh(Parents) then
14:          proceed to the next value of Parents
15:       end if
16:       for all Delegations from genDelegations(Comp, Parents, Elem, Rel) do
17:          if not isFresh(Delegations) then
18:             proceed to the next value of Delegations
19:          end if
20:          for all Binds from genBindings(Comp, Parents, Delegations, Elem, Rel) do
21:             if not isFresh(Binds) then
22:                proceed to the next value of Binds
23:             end if
24:             Values = genParameters(Comp, CT, Definer, Elem, Rel)
25:             Inst = ⟨ Comp, CT, Parents, Delegations, Binds, Values ⟩
26:             if invariant(Inst) then
27:                SInst ← SInst ∪ Inst
28:             end if
29:          end for
30:       end for
31:    end for
32: end for
33: End
```

Algorithme 1 : Algorithme de génération de configurations initiales

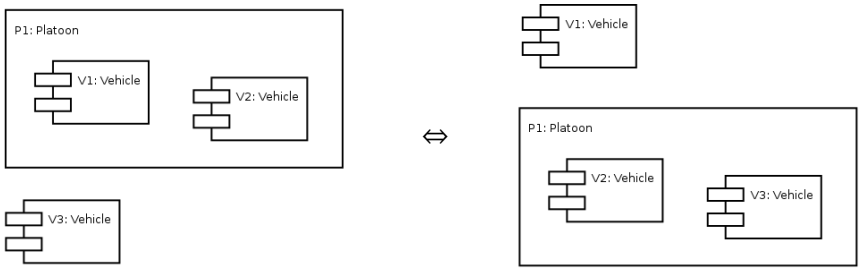

FIGURE 5.2 – Symétries dans les relations de parenté

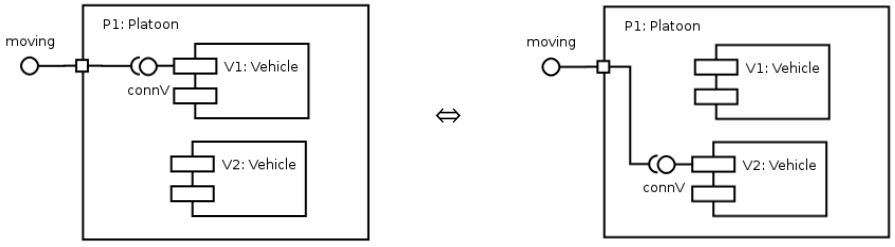

FIGURE 5.3 – Symétries dans les relations de délégation

suivant de l'ensemble $Parents$.

Étape 3. L'étape 3 (ligne 16) consiste à effectuer les délégations fournies $DelProv$ ou requises $DelReq$ des composants composites à un de leurs composants enfants. De la même façon que dans les étapes précédentes, les solutions symétriquement identiques sont écartées comme cela est illustré dans la figure 5.3.

Étape 4. L'étape 4 (ligne 20) consiste à utiliser les relations de parenté et délégation et à produire les liens des interfaces fournies qui satisfont les contraintes architecturales (e.g seuls les composants ayant le même parent peuvent être liés) et les contraintes de contingence (single/multiple, mandatory/optional). Ici encore, les solutions symétriquement identiques sont écartées comme cela est illustré en figure 5.4.

Étape 5. Une fois que la structure du système à composants est générée, l'algorithme peut faire de même pour les valeurs des paramètres des composants. Les paramètres sont générés en fonction du type, grâce à une fonction d'évaluation qui peut être définie par l'utilisateur (lignes 24-25).

Étape 6. Pour finir, la configuration générée est vérifiée sur ses invariants (ligne 26), avant d'être ajoutée à l'ensemble des configurations générées (ligne 27).

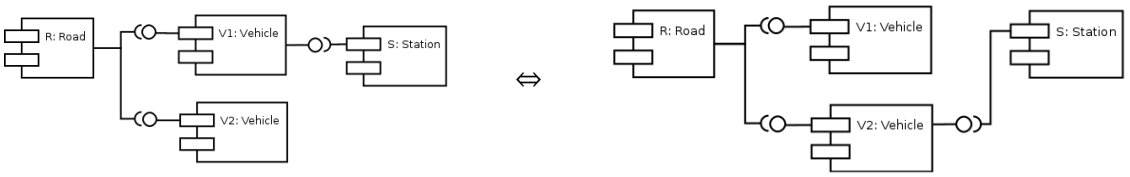

FIGURE 5.4 – Symétries dans les relation de lien

Au final, seules les configurations valides et cohérentes, dont le nombre d'instances de composants est inférieur à N sont conservées. [1].

Proposition 1 : *Soit un nombre d'instances de composants N à générer, l'algorithme 1 termine soit en fournissant un ensemble $SInst$ de configurations cohérentes avec jusqu'à N composants (les composants symétriquement équivalents obtenus par permutation architecturale sont retirés), soit en retournant un ensemble vide si aucune des configurations ne satisfait les contraintes de cohérence ou les contraintes spécifiques au système.*

5.2.2/ ÉCHANTILLONNAGE DES CONFIGURATIONS INITIALES

Une fois que les configurations initiales possibles ont été générées, il serait peu pertinent, voire impossible, de toutes les sélectionner pour tester le système. En effet, d'une part le test de système sur des configurations différentes a pour but de savoir si les configurations initiales ont un poids sur les simulations du système et donc l'utilisation de configurations initiales similaires n'apporterait pas de réponse à cette question. De plus, les simulations prennent du temps et doivent être effectuées plusieurs fois sur chaque configuration, en fonction de la complexité du modèle architectural du système il est donc possible de générer de nombreuses configurations et toutes les tester prendrait trop de temps. C'est pourquoi nous souhaitons nous focaliser sur une partie de ces configurations.

Comparer les configurations. Le calcul de différence entre les configurations est obtenu en les comparant deux par deux. Ce calcul est composé de deux parties : une partie sur l'architecture du système et une autre partie sur ses paramètres et, en particulier, sur ses valeurs.

Tout d'abord, nous comparons les structures de chaque configuration en comptant la différence Δ_{comps} entre le nombre d'instances de composants (composites ou primitifs), la différence $\Delta_{hierachy}$ entre le nombre de tous les ancêtres de composants impliqués et la différence $\Delta_{bindings}$ entre le nombre de liens entre interfaces.

La différence de complexité de configuration est calculée en utilisant ces 3 différences

1. L'implémentation de cet algorithme est visible en suivant ce lien : https://fdadeau.github.io/CSConfigGen/

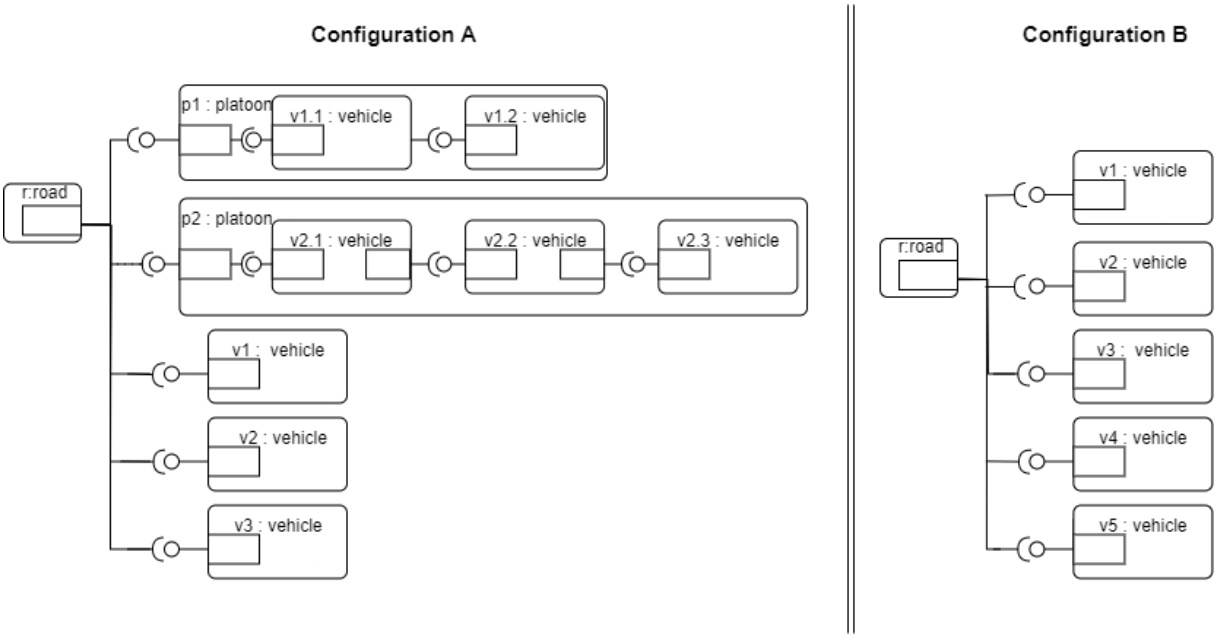

FIGURE 5.5 – Illustration des configurations A et B

grâce à la formule suivante :

$$k = log_{10}(\Delta_{comps}^{\Delta_{hierarchy}+\Delta_{bindings}})$$

Cette formule s'inspire des travaux de [1] qui utilisent une exponentielle inversée pour avoir un score borné. Nous avons conservé la même intuition sans pour autant avoir besoin de borner le score et c'est pour cela qui nous avons utilisé le log_{10} à la place de e^{-x}.

Cette formule donne un coefficient k qui est pensé pour représenter la complexité de l'architecture. Ce coefficient est utilisé pour le calcul du score final que nous détaillons en deuxième partie. Nous proposons de faire ce calcul dans un exemple avec le cas VANet.

Exemple 10 : *Considérons la configuration que nous appelons configuration A contenant 8 véhicules, 2 convois contenant respectivement 3 et 2 véhicules, 1 route, et 10 liens (configuration A). Nous proposons de comparer la configuration A avec la configuration B, composée de 5 véhicules, aucun convoi, 1 route et 5 liens. Les configuration A et B sont visibles en figure 5.5. Nous procédons au calcul de chaque différence : Dans la configuration A, il y a 11 composants (9 primitifs et 2 composites) et 6 dans la B, nous obtenons donc $\Delta_{comps} = 5$. Dans la configuration A, il y a 5 relations de parenté et aucune dans la configuration B, nous obtenons donc $\Delta_{hierarchy} = 5$. Dans la configuration A, il y a 10 liens et 5 dans la configuration B, nous obtenons donc $\Delta_{bindings} = 5$. Cela nous donne pour le calcul du coefficient k : $k = log_{10}(5^{5+5}) = 6.99$.*

La génération de cas de tests implique d'initialiser les valeurs des paramètres des com-

posants. Lors du test des systèmes adaptatifs, les valeurs de ces paramètres peuvent impacter les règles de reconfigurations du système. En effet, les politiques d'adaptation définissent des règles d'adaptation dans lesquelles les paramètres du système sont utilisés pour indiquer à quel moment il peut se reconfigurer. Pour cette raison, l'ingénieur de validation doit être en mesure d'identifier parmi les paramètres lesquels sont utilisés dans les règles d'adaptation.

Nous souhaitons comparer les valeurs des paramètres entre deux configurations deux à deux. Il est donc nécessaire d'avoir le même nombre de valeurs de paramètres pour les deux configurations à comparer. Si une configuration possède, pour un paramètre donné, un nombre de valeurs plus élevé que l'autre configuration, nous procédons à un regroupement des valeurs de la première configuration. Les plus proches valeurs deux à deux sont sélectionnées pour être regroupées et cela jusqu'à que le nombre de valeurs soit le même dans les deux configurations. Le regroupement de 2 valeurs consiste à faire la moyenne arithmétique de celles-ci.

Imaginons un nombre n d'instances de composants du même type, l_1 et l_2 des listes triées de n valeurs du paramètre Par. La différence proportionnelle est calculée par la formule suivante :

$$score_{Par} = \frac{100}{n \times (max_{Par} - min_{Par})} \times \sum_{i=1}^{n} |l_{1_i} - l_{2_i}|$$

où max_{Par} et min_{Par} représentent respectivement les valeurs maximales et minimales du paramètre Par. Les scores de tous les paramètres de $Pars \subseteq Params$ qui impactent la règle d'adaptation sont agrégés dans un score final. Ce score final reprend le calcul de k, le coefficient de la différence d'architecture entre deux configurations.

Le score final est obtenu en appliquant la formule suivante :

$$difScore = k \times \sum_{Par \in Pars} score_{Par}$$

Exemple 11 : *Considérons à nouveau les configurations A et B de l'exemple 10, en nous focalisant sur le paramètre de la batterie ($battery$). Supposons que les niveaux de batterie pour les véhicules de la configuration A soient $(20, 22, 34, 54, 62, 72, 80, 99)$. Comme il y a 3 véhicules de plus dans la configuration A que dans la configuration B, nous devons effectuer 3 regroupements de valeurs. Ces regroupements sont $(20, 22)$, $(55, 62)$ et $(72, 80)$. Après le regroupement effectué (par moyenne arithmétique), les valeurs de batterie pour A sont $l_1 = (21, 34, 58, 76, 99)$. Pour la configuration B, nous prenons $l_2 = \{26, 55, 62, 74, 89\}$. En appliquant la formule de $score_{Par}$ au paramètre $battery$, nous obtenons :*

$$score_{Bat} = \frac{100}{5 \times (100 - 20)} \times \sum_{i=1}^{5} |l_{1_i} - l_{2_i}| = 0.25 \times 42 = 8.4$$

En considérant un score de 5.1 pour le paramètre $distance$ *(se basant sur le même calcul que pour le paramètre* $battery$*), le score de différence aggrégé est alors :* $difScore = 5.59 \times (8.4 + 5.1) = 75.46$.

Échantillonnage de configuration. L'échantillonnage de configurations consiste à réduire l'ensemble des configurations possibles à un sous-ensemble de taille N dans lequel les scores de différence sont maximisés.

Ce problème d'optimisation peut être résolu de diverses manières, tel que le SAT-solving, la programmation linéaire, le regroupement [77] (plus connu sur l'appellation clustering), la programmation génétique [98], etc.
Dans nos travaux, nous avons choisi un algorithme glouton (greedy algorithm en anglais) visible ci-dessous dans l'algorithme 2. En se basant sur un ensemble de m configurations initiales générées, une matrice de score de taille $m \times m$ (nommée $Scores$ ligne 2) est construite, avec $a_{i,j}$ l'élément représentant un $difScore$ entre la configuration i et la configuration j[2]. NS (ligne 2) dénote le nombre de configurations à sélectionner. $Configs$ est l'ensemble contenant les indices des configurations sélectionnées (ligne 4).

L'algorithme commence par sélectionner le score le plus élevé dans la matrice $Scores$ grâce à la fonction $selectHigestScore$ (ligne 6) qui parcourt la matrice et retourne les indices correspondant au plus grand score de différence avec la i-ème et la j-ème configuration.
Puis, l'algorithme marque l'élément correspondant comme sélectionné (ligne 7). L'ensemble $Configs$ est alors mis à jour (lignes 8-9). Ensuite, dans la ligne correspondante i et colonne j, les scores restants les plus élevés sont choisis (ligne 11 et ligne 14).

Les configurations correspondantes sont ajoutées à l'ensemble $Configs$ (ligne 13 et ligne 16).
Ensuite, les fonctions $selectHighestScoreInLine(i)$ (resp. $selectHighestScoreInColumn(j)$) parcourent la i-ème ligne (resp. la j-ème colonne) de la matrice $Scores$, et retournent l'index de la colonne (j') (resp. ligne i') correspondant à leurs scores respectifs les plus élevés. Afin de préparer le prochain pas d'itération, les indices sont mis à jour (ligne 17 et ligne 18). Les étapes de la ligne 11 à 18 sont répétées, jusqu'à ce que la taille de l'ensemble $Configs$ atteigne NS.

Par construction, seules les configurations avec un score de différence élevé sont sélectionnées.

Proposition 2 : *En connaissant* NS*, le nombre de configurations à sélectionner dans l'ensemble* $SInst$ *de taille* m*, l'algorithme 2 termine en fournissant l'ensemble* $Configs$ *de taille* $NS \leq m$ *d'indices de configuration à partir de* $SInst$ *qui ont les différences de scores les plus significatives.*

2. Dans cette matrice $a_{i,i} = 0$ et $a_{i,j} = a_{j,i}$.

Inputs
  $Scores$, NS
Output
  $Configs$
Begin
$i, j \leftarrow$ selectHigestScore($Scores$)
mark $a_{i,j}$ as selected
Configs.add(i)
Configs.add(j)
repeat
  $j' \leftarrow$ selectHighestScoreInLine(i)
  mark $a_{i,j'}$ as selected
  Configs.add(j')
  $i' \leftarrow$ selectHighestScoreInColumn(j)
  mark $a_{i',j}$ as selected
  Configs.add(i')
  $i \leftarrow i'$
  $j \leftarrow j'$
until Configs.size() < NS
return $Configs$
End

Algorithme 2 : Échantillonnage de configuration initiale

5.2.3/ GÉNÉRATION DES VALEURS POUR LES PARAMÈTRES DES COMPOSANTS

Dans la section précédente, nous avons présenté notre algorithme de génération de configurations initiales. Cet algorithme utilise une fonction appelée $genParameters()$ que nous explicitons dans cette section et qui consiste à initialiser les paramètres par distribution probabiliste. Les valeurs sont tirées selon différentes distributions probabilistes permettant à la fois de couvrir la diversité du domaine de définition et d'automatiser le processus.

Nous souhaitons initialiser ces paramètres pour nos configurations initiales. Ces configurations initiales servent de point de départ à nos tests sur le système. L'objectif principal des tests est de déceler de potentielles erreurs dans le système, il faut donc initialiser des variables qui vont solliciter le système pour déterminer s'il réagit correctement mais sans le mettre dans une situation dans lequel tous les choix de reconfiguration mènent à une erreur. Chaque paramètre est lié à un domaine par l'utilisateur qui possède les connaissances pour les définir. Pour se faire, nous procédons par tirage probabiliste selon la Beta-distribution [75] pour modéliser l'initialisation des valeurs de chaque variable qui permet de configurer les coefficients probabilistes.

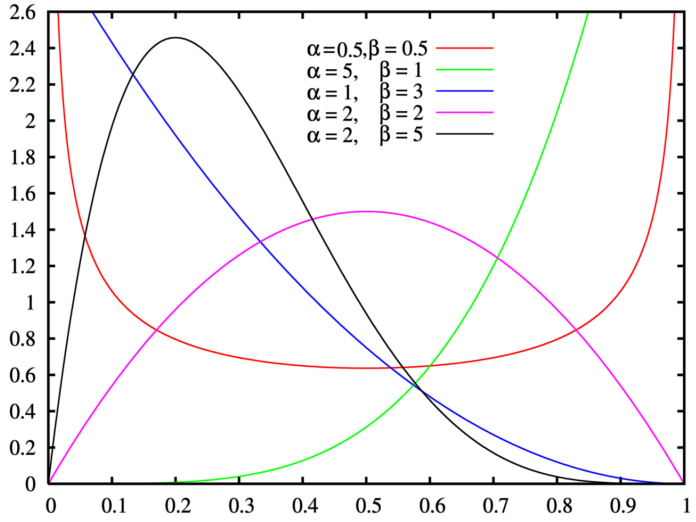

FIGURE 5.6 – Densité de probabilité

5.2.4/ PRÉSENTATION DE LA BETA-DISTRIBUTION

La Beta-distribution appartient à la famille des distributions probabilistes continues et est définie sur le domaine [0,1]. C'est un cas spécial de la loi de Dirichlet [116], avec seulement deux paramètres α et β, deux nombres de $\mathbb{R}$ dans $[0, 10]$. La Beta-distribution admet une infinité de formes dont quelques exemples sont visibles en figure 5.6, elle permet de modéliser de nombreuses distributions à support fini. Ainsi, il est possible de donner des coefficients probabilistes différents pour un domaine donné. Le choix du nombre de distributions est motivé à la fois par la diversité des valeurs obtenues et le temps de calcul nécessaire. En effet, ces distributions permettent de générer les valeurs des paramètres qui est une étape dans la génération de configurations initiales étant elle-même une étape dans le test du système.

Nous avons sélectionné quatre couples de valeurs α et β permettant d'obtenir quatre instances de Beta-distribution. Ces instances indiquent la densité sur le domaine [0,1] ce qui nous permet d'effectuer autant de tirages que l'on souhaite initialiser de valeurs. Les valeurs obtenues sont des nombres compris dans [0,1], il est donc nécessaire d'effectuer une transformation affine. Cette transformation qui s'effectue sur chaque valeur obtenue permet de passer du domaine [0,1] au domaine souhaité.

Il est possible d'évaluer le taux de couverture attendu (expected area coverage) en fonction de la partie du domaine comme dans les travaux de [94]. Dans notre approche, nous générons directement les valeurs donc nous pouvons directement influencer le taux de couverture en modifiant le couple de valeur α et β de la Beta-distribution. Nous présentons à présent ces quatre couples de valeurs α et β de Beta-distribution et quelles valeurs sont obtenues à partir de ces couples.

La Beta-distribution qui a comme paramètres le couple ($\alpha = 0.5$, $\beta = 0.5$) et visible en figure 5.7, a une densité élevée sur les valeurs proches du maximum et du minimum du domaine. Cette distribution permet de faire des tirages avec de grandes variances et des valeurs moyennes comparables aux tirages aléatoires. Intuitivement, cette distribution permet de s'assurer que les bornes des domaines ont bien été choisies.

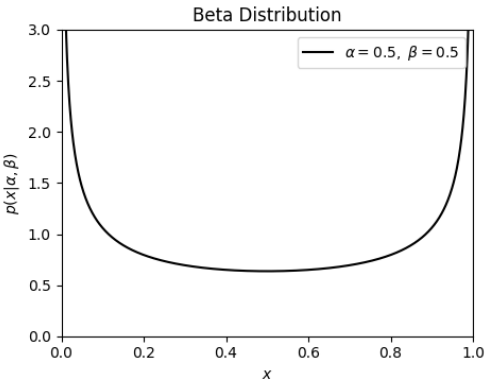

FIGURE 5.7 – Beta-distribution aux extrémités

La Beta-distribution qui a comme paramètres le couple ($\alpha = 0.5, \beta = 2$) et visible en figure 5.8 a une densité élevée sur les valeurs proche du minimum du domaine.

Cette distribution au minimum permet de faire des tirages avec des valeurs moyennes faibles et des valeurs de variances comparables aux tirages aléatoires. Intuitivement, cette distribution permet de focaliser les valeurs des tirages sur la borne basse du domaine.

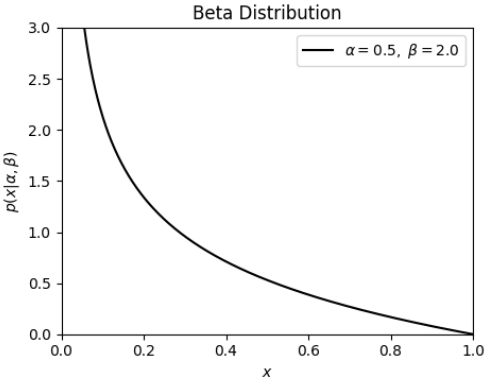

FIGURE 5.8 – Beta-distribution au minimum

La Beta-distribution qui a comme paramètres le couple ($\alpha = 2, \beta = 0.5$) et visible en figure 5.9 a une densité élevée sur les valeurs proches du maximum domaine. Cette distribution au maximum permet de faire des tirages avec des valeurs moyennes élevées et des valeurs de variances comparables aux tirages aléatoires. Intuitivement, cette distribution permet de focaliser les valeurs des tirages sur la borne haute du domaine.

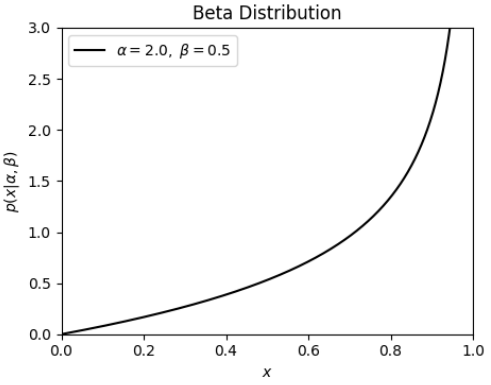

FIGURE 5.9 – Beta-distribution au maximum

La Beta-distribution qui a comme paramètres le couple $(\alpha = 1, \beta = 1)$ et visible en figure 5.10 a une densité équitablement répartie sur le domaine. Cette distribution effectue des tirages aléatoires équiprobables. Intuitivement, cette distribution permet de générer des valeurs sans se focaliser sur une partie du domaine.

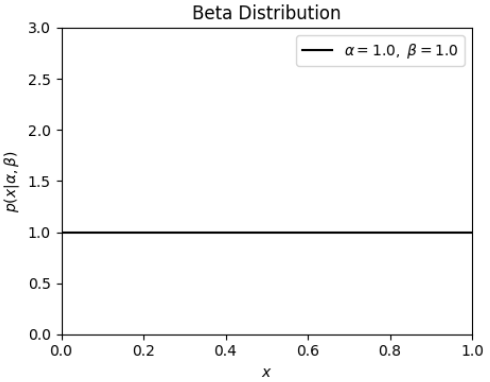

FIGURE 5.10 – Beta-distribution aléatoire

Afin de comprendre comment fonctionne ces variations de Beta-distribution, nous présentons dans la prochaine section les quatre instances différences autour de la batterie des véhicules.

5.2.5/ EXEMPLES DE VARIABLES SELON LA BETA-DISTRIBUTION

Dans cette section, nous utilisons l'exemple VANet pour comparer les Beta-distributions présentées dans la section précédente pour l'initialisation du niveau de batterie des véhicules. La valeur de la batterie est exprimée en pourcentage et donc sa valeur est comprise entre 0 et 100. Cependant, nous avons choisi de générer des valeurs comprises entre 20 et 80. Pour mieux nous rendre compte de la différence entre les instances de distribution, nous allons générer 1000 valeurs de batterie dont les valeurs sont résumées dans la table 5.1.

TABLE 5.1 – Résultats de la génération des instances de la Beta-distribution

	Moyenne	*Variance*	*Minex*	*Maxex*
betaExt	50.5	460.5	227	253
betaMin	28.6	159.8	376	7
betaMax	68.0	158.0	1	498
betaRand	49.7	291.5	127	119

Les lignes *betaExt*, *betaMin*, *betaMax* et *betaRand* représentent respectivement la distribution focalisée aux extrémités visible figure 5.7, la distribution au minimum visible figure 5.8, la distribution au maximum visible figure 5.9 et la distribution aléatoire visible figure 5.10. Pour chaque instance de distribution, nous renseignons respectivement dans le tableau la moyenne des valeurs, la variance, le nombre de valeurs proches de la borne minimale avec une marge de 10% ainsi que le nombre de valeurs proches de la borne maximale avec une marge de 10%.

Pour résumer, la *betaExt* est similaire à la *betaRand* en termes de valeur *Moyenne*, mais la *betaExt* possède la plus grande *Variance* étant donné ses valeurs proches des bornes.
La *betaMin* et la *betaMax* possèdent des valeurs de *Variance* similaires.

La $betaMin$ focalisant sur la borne inférieure, ses valeurs $Moyenne$ et $Maxex$ sont les plus faibles alors que sa valeur $Minex$ est la plus élevée. La $betaMax$ focalisant sur la borne supérieure ses valeurs $Moyenne$ et $Maxex$ sont les plus élevées alors que sa valeur $Minex$ est la plus faible. Ces différentes distributions présentent des résultats différents et peuvent donc être utilisées seules ou en complément entre elles. Par exemple, dans notre système VANet la distribution au minimum semble plus appropriée pour générer des véhicules avec des niveaux de batterie faible et ainsi voir si le système est suffisamment robuste en situation difficile. Dans le domaine des architectures de matériel informatique il est sûrement plus judicieux d'utiliser la $betaMax$ en complément afin de tester les limites de capacité de certains composants tel que le processeur. Dans notre approche, nous avons également utilisé la distribution aux extrémités pour nous assurer que les domaines définis sont valides vis-à-vis du comportement du système. Les résultats obtenus permettent de déterminer quelle(s) distribution(s) sont les plus adaptées à chaque valeur de paramètre du système étudié.
Notre méthode de génération de configurations initiales étant présentée, nous abordons à présent notre processus de génération d'évènements contrôlables.

5.3/ PROCESSUS DE GÉNÉRATION D'ÉVÈNEMENTS DE TEST EN LIGNE

Dans la section précédente, nous avons vu comment générer des instances de configurations initiales. Nous présentons dans cette section les modèles d'usage rendant possible la génération d'évènements contrôlables.
Les systèmes adaptatifs évoluent en recevant les évènements externes. Comme dans les travaux de [140], nous proposons une approche où les choix de reconfiguration du système sont en accord avec les politiques d'adaptation. Ces politiques d'adaptation guident le système, permettant de savoir à quel moment les reconfigurations peuvent être exécutées. Dans cette section, nous présentons le processus mis en place pour la génération d'évènements contrôlables et pour éprouver le système ainsi que les politiques d'adaptation.
Nous utilisons un modèle d'usage qui décrit des évènements contrôlables pertinents par rapport à la configuration du système.

5.3.1/ MODÈLE D'USAGE POUR LE TEST EN LIGNE

Les modèles d'usage définis dans les travaux de [154] ont pour but de décrire les différents évènements externes qui peuvent exister dans l'environnement du système sous l'appellation d'évènements contrôlables.

Définition 30 : Évènements contrôlables

Soit C l'ensemble des composants du système : $C = \{c_1, ..., c_i, ..., c_n\}$.
L'ensemble des évènements contrôlables Ec_i pour un composant c_i est défini par : $Ec_i = \{ec_i^1, ec_i^2, ..., ec_i^j\}$,
où un évènement ec_i^j est soit un évènement externe soit une quiescence δ.
La quiescience représente le fait qu'aucune action n'est effectuée dans le système pour une certaine période. L'ensemble des évènements contrôlables Θ pour tous les composants est : $\Theta = \bigcup_{1 \leq i \leq n} Ec_i$.
Une séquence d'évènements contrôlables θ est un élément de l'ensemble de toutes les séquences d'évènements Θ^* : $\theta \in \Theta^*$.

Les évènements externes sont vus comme les évènements envoyés périodiquement au système et définis dans [92]. Ces évènements sont déclenchables à n'importe quel moment et peuvent être considérés comme des notifications provenant des capteurs du système lorsqu'un changement est détecté dans l'environnement. Les évènements contrôlables sont constitués d'évènements externes ou quiescence. Les quiescences (δ) sont ajoutées pour prendre en compte l'aspect discret de nos transitions. En effet, comme nous effectuons nos simulations en tenant compte de valeurs de temps discrètes, les quiescences permettent de simuler de manière réaliste l'environnement ainsi que le temps de réponse du système aux évènements externes. Cette unité de temps est paramétrable par l'utilisateur et est la même pour tous les composants associés au modèle d'usage.

Les travaux de [7] présentent une génération d'évènements contrôlables. Nous propo-

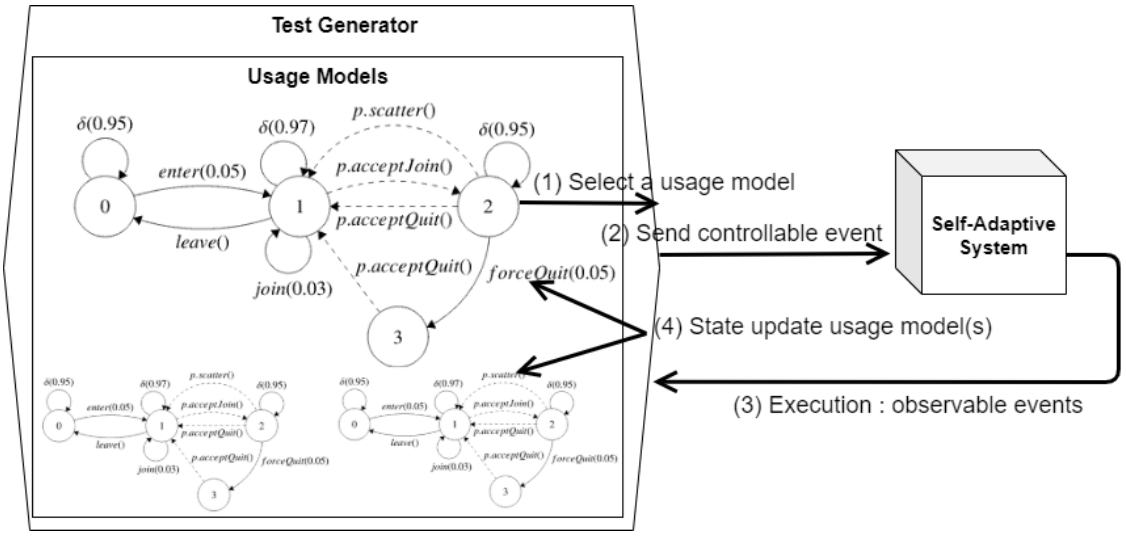

FIGURE 5.11 – Processus de génération d'évènements contrôlables à l'aide d'un générateur de test

sons d'ajouter les transitions d'évènements observables sur les modèles d'usages afin

qu'ils puissent décrire des évènements contrôlables cohérents avec l'état du système. La figure 5.11 montre par quel moyen les modèles d'usages décrivent les évènements contrôlables. Le générateur de test est l'élément qui sert d'intermédiaire en transmettant (2) pour chaque pas de test l'évènement contrôlable du modèle d'usage choisi (1) vers le système. Pour que le modèle d'usage puisse spécifier des transitions cohérentes avec la configuration du système, nous supposons la présence d'évènements observables (3). Cet évènement observable est une transition qui peut être déclenchée par le générateur de test pour le ou les modèle(s) d'usage(s) concerné(s) (4). Chaque modèle d'usage est un automate dans lequel chaque transition représente un évènement $externe$, $observable$ ou $quiescence$, appelé évènement de test, et chaque état représente la configuration du système face à l'évènement.

Définition 31 : Évènements de test

Soit Θ l'ensemble des évènements contrôlables, et O l'ensemble d'évènements observables.
L'ensemble des évènements de test Tev est défini par : $Tev = \Theta \cup O$. Une séquence d'évènements de test Sev est un élément de l'ensemble de toutes les séquences d'évènements $(\Theta \cup O)^*$: $Sev \in (\Theta \cup O)^*$

En recevant chaque évènement de test, le système se reconfigure et complète sa trace d'exécution comme dans les travaux de [160]. Le générateur de tests récupère la trace d'exécution pour en extraire les évènements observables afin de lui permettre de mettre à jour les états des modèles d'usage correspondants.

Définition 32 : Modèle d'usage à partir d'un automate probabiliste

Un modèle d'usage est défini par : $\mathcal{A}_C = \langle Q, q_0, Tev, F, P \rangle$, avec Q est un ensemble d'états, $q_0 \in Q$ est l'état initial, Tev est un ensemble d'évènements de test, F est une relation de transition $F \in Q \times Tev \times Q$, et soit Θ l'ensemble des évènements contrôlables, P est la probabilité [a] d'une transition $P : Q \times \Theta \rightarrow [0; 1]$ telle que $\forall q \in Q \Rightarrow \Sigma_{e \in \Theta} P(q, e) = 1$.

a. Si aucune transition sortante d'un état courant n'est étiquetée par un évènement, sa probabilité est 0.

Notre modèle d'usage est un automate probabiliste dont les transitions correspondent aux évènements externes, la quiescence et des évènements observables.

Il est possible que la modification de certains composants du système influence l'état d'un modèle d'usage différent. C'est pourquoi les évènements observables contiennent si nécessaire l'identifiant du composant parent en plus de l'opération de reconfiguration. Pour cela, nous utilisons le terme $parent$ dans les transitions concernées pour rensei-

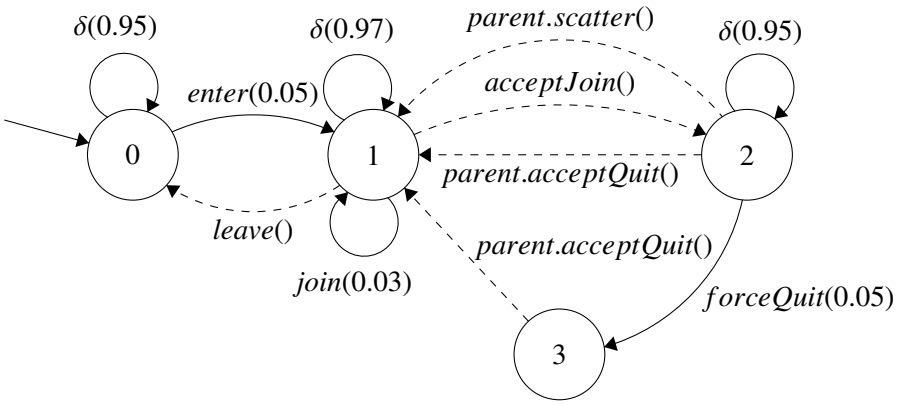

FIGURE 5.12 – Modèle d'usage pour un véhicule

gner l'identifiant du composant parent au composant associé au modèle d'usage. Ainsi, $parent$ renvoie le composant parent conformément à la relation $Parent$ décrite dans la section 4.1.2. De ce fait, le générateur de test peut propager l'évènement du système quand il correspond exactement à un des évènements observables de l'état courant du modèle d'usage. Ces composants, en réagissant individuellement aux cas de tests, vont participer à l'évolution globale du système visible à travers les évènements observables. Nous présentons ici le modèle d'usage d'un véhicule visible en figure 5.12 que nous explicitons au travers de l'exemple 12 pour les différents types de composants : véhicule ou peloton.

Exemple 12 (*Modèle d'usage sur l'exemple VANet*) : *Un véhicule commence par entrer sur la route avec l'évènement $enter$. Chaque évènement possède une probabilité d'être déclenché. Ainsi, à l'état initial de notre modèle d'usage, le véhicule a une probabilité de 0.05 d'entrer sur la route. Les transitions nommées avec δ représentent une unité de temps durant lequel l'état du véhicule ne change pas. La valeur des probabilités de la quiescence est calculée de manière à ce que la somme des probabilités associées aux transitions sortantes d'un état soit égale à 1 (par définition 32). Les nombres entre parenthèses désignent les probabilités associées aux transitions. Si besoin, l'état du modèle d'usage peut être mis à jour afin qu'il décrive des évènements de test cohérents avec l'état du système. Les transitions observables sont représentées par des flèches en pointillés et ne possèdent pas de probabilité de déclenchement puisqu'elles ne sont pas spécifiées par le modèle d'usage. Dans notre exemple de convoi de véhicules, le véhicule peut se joindre à d'autres véhicules, pour cela il faut qu'il entre dans la zone d'autres véhicules ce qui est représenté par la transition $join$. Si les véhicules recevant l'évènement $join$ décident de refuser de se regrouper il ne se passe rien et le modèle d'usage des véhicules reste à l'état 1. Sinon, les véhicules acceptent de se regrouper, la transition interne $acceptJoin()$ est déclenchée et le modèle d'usage du véhicule nouvellement en convoi passe à l'état 2. Ensuite, le véhicule peut sortir du convoi de différentes ma-*

nières. La première intervient lorsque le conducteur décide de repasser en mode manuel avec la transition $forceQuit()$. Dans ce cas le système va finir ses manoeuvres en cours, puis accepter la transition par un $acceptQuit()$. Une autre possibilité pour le véhicule de sortir du convoi est au moment où le véhicule doit se recharger à une station ou qu'il approche de son objectif. Dans ce cas, le système déclenche de lui-même la transition observable $acceptQuit$ qui résulte d'une succession d'évènements et de reconfigurations. Dernière possibilité, le composant renvoyé par $parent$ se disperse et l'état du modèle d'usage associé au véhicule est mis à jour avec la transition $parent.scatter()$. L'état du modèle d'usage associé au convoi est également mis à jour avec la transition $scatter()$.

Le modèle d'usage étant défini, nous pouvons maintenant détailler comment initialiser les états des automates.
Comme nous l'avons vu, chaque modèle d'usage est spécifique à son instance de composant et chaque état représente une configuration pour le composant. Il est donc nécessaire d'initialiser l'état du modèle en accord avec sa configuration. Pour cela, nous supposons qu'il existe une fonction déterminant l'état initial d'un modèle d'usage pour un composant donné. Nous appelons cette fonction $init_{CType}$.

Exemple 13 : *Supposons qu'une instance de composant v de type véhicule apparaisse dans l'ensemble $Inst$ qui est généré par le processus décrit dans la section 5.2.2. L'état initial de l'automate de v est obtenu grâce à la fonction suivante :*

```
function init_Vehicle(v, Inst)
   if ((v, connV) ∉ Inst.Binds) return 0
   else if (Inst.Parents(v) ≠ ∅) return 1
   else return 2
```

L'automate de v est visible en figure 5.12 et cette fonction prévoit les 4 états qui constituent l'automate.

À présent, nous nous intéressons à la manière dont sont générés les tests grâce aux modèles d'usage.

5.3.2/ GÉNÉRATION DE TESTS À PARTIR DE MODÈLES D'USAGE

Dans la section précédente, nous avons vu que les modèles d'usage permettent de modéliser les interactions entre le système et son environnement. Nous allons maintenant développer les méthodes utilisées pour générer ces évènements contrôlables pour le système à partir de l'ensemble des modèles d'usage et d'un générateur de test.
D'une part, le générateur de tests met à jour l'état des modèles d'usage grâce aux observations de la trace d'exécution du système. D'autre part, le générateur de tests sélectionne les modèles d'usage puis produit et envoie au système les évènements corres-

pondant aux transitions décrites par les modèles d'usage.

L'algorithme 3 de génération de tests sélectionne un composant afin de générer un évènement contrôlable décrit par son modèle d'usage. Nous avons choisi de procéder à la sélection du composant aléatoirement, d'autres stratégies de sélection sont possibles telles que des listes d'attente ou des pondérations permettant d'ajouter de l'équité. En effet, le tirage aléatoire ne garanti pas que les composants seront tous sélectionnés alors qu'un tirage équitable empêcherait de tirer plusieurs fois de suite le même composant plusieurs fois de suite tant que les autres composants n'ont pas été sélectionnées. Sur des expérimentations contenant un nombre d'évènements important, l'approche aléatoire permet de sélectionner tous les modèles d'usages et déclencher leurs transitions. Nous détaillons cet algorithme ci-dessous.

```
1  inputs :
2  NumberOfSteps// Number of steps to generate
3  for all A_c do
4    state(A_c) ← q_0(A_c)
5  end for
6  i ← 0
7  trace ← retrieveTrace()
8  uncovered ← updateCoverage()
9  nbStep ← NumberOfSteps
10 while i < nbStep ∨ uncovered
11   trace ← retrieveTrace()
12   comp ← selectComponent()
13   A_comp ← stateOfComponent(comp)
14   for all A_c do
15     state(A_c) ← stateUpdate(A_c, trace)
16   end for
17   e ← pick(state(A_comp))
18   if e ≠ δ then
19     comp.send(e)
20     i ← i + 1
21   end if
22   uncovered ← updateCoverage()
23   await()
24 done
```

Algorithme 3 : Algorithme de génération de tests en ligne

Cet algorithme permet de générer une séquence d'évènements contrôlables.

Le nombre de pas $NumberOfSteps$ à générer est donné en paramètre d'entrée de cet algorithme (ligne 2). Dans un premier temps, l'algorithme initialise chaque automate (lignes 3 à 5). En ligne 7, nous initialisons la trace qui contient les informations sur les opérations de reconfiguration effectuées et qui peuvent être des évènements observables pour le modèle d'usage en cours d'exploration. En ligne 8, nous mesurons le taux de couver-

ture effectué qui est obtenu grâce aux critères de couverture. Ces critères requièrent que le système passe par des configurations spécifiques, nous présenteront ces critères de couverture dans le prochain chapitre. Associés au nombre de pas à générer (ligne 9), les critères de couverture déterminent la condition d'arrêt de la boucle $while$ (en ligne 10) dans laquelle nous générons les évènements.
Dans cette boucle, la trace d'exécution est récupérée en ligne 11 puis, l'instance de composant est sélectionnée (ligne 12) ainsi que l'état de son automate (ligne 13). L'état courant de chaque automate est mis à jour grâce à la fonction $stateUpdate()$ conformément aux traces d'exécutions obtenues (ligne 15). Une fois la mise à jour effectuée, nous utilisons la fonction $pick()$ pour tirer une transition au hasard parmi celles possibles pondérées par des probabilités (ligne 17). L'évènement est envoyé au système par le biais de la fonction $send()$ et du composant considéré en ligne 19. Chaque automate est mis à jour en ligne 21 et la trace mise à jour en ligne 24. Le taux de couverture est mis à jour en ligne 26. Avant de reboucler sur le pas de test suivant, le processus se met en pause (await() ligne 27). Cela permet de simuler le temps écoulé entre les pas de tests et le temps nécessaire au système pour effectuer physiquement les reconfigurations souhaitées. Dans le cas d'une quiescence, le but est de simuler des périodes sans évènement avec le système qui continue d'évoluer en mettant à jours les paramètres de chaque composant.

Terminaison de l'algorithme 3 Maintenant que nous avons présenté l'algorithme 3, nous allons montrer sa terminaison. En voici les arguments informels.
Cet algorithme contient une boucle for, lignes 3 à 5, qui parcourt chaque élément de l'ensemble fini des automates fini et donc se termine. Il en est de même pour les deux autres boucles for, lignes 14 à 16 et lignes 20 à 22.
La boucle while (lignes 10 à 28) possède un choix dans sa condition. En effet, en conservant $i < nbStep \vee uncovered$ nous garantissons qu'à la fin de l'algorithme les critères de couvertures et la longueur de la séquence de test seront conformes à ce qui était attendu. Cependant, il n'est pas possible de garantir que l'algorithme se termine car la couverture peut ne jamais atteindre les critères attendus.
Pour remédier à ce problème, il est possible de changer la condition pour $i < nbStep \wedge uncovered$. Dans ce cas, la terminaison de l'algorithme serait garantie car la taille de la séquence de test finit par atteindre la valeur de $nbSteps$ mais le respect de tous les critères de couverture n'est alors pas garanti lorsque l'algorithme se termine.

Nous présentons un exemple (cf exemple 14) de fonctionnement du générateur de test visible en figure 5.13 pour le cas VANet. Dans cette figure, les modèles d'usage les plus gros sont ceux concernés par l'étape en cours. Tous les modèles d'usages proviennent du même automate visible en figure 5.12.

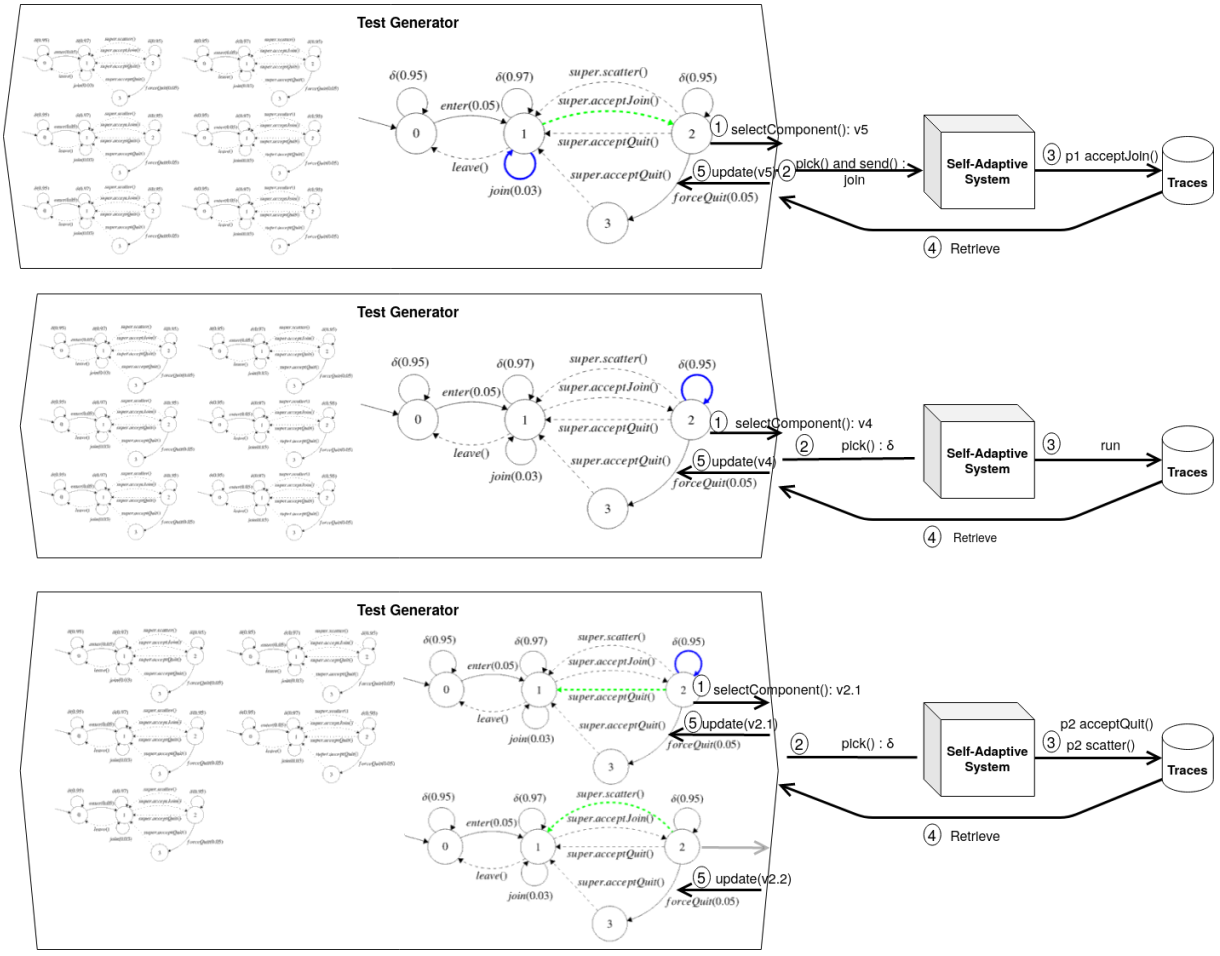

FIGURE 5.13 – Processus de génération d'évènements de test en ligne

Exemple 14 (*Génération de test en ligne*) : *Nous partons de la configuration visible en figure 4.4 composée de deux convois p1 et p2, dans lequel p1 contient les véhicules* $v1.1$*,* $v1.2$ *et* $v1.3$ *; p2 contient les véhicules* $v2.1$ *et* $v2.2$*. Les véhicules* $v4$ *et* $v5$ *sont seuls, la station* s*, la route* r*, les pelotons* $p1$ *et* $p2$ *sont présents mais ne sont pas représentés en figure 5.13 car aucun modèle d'usage ne leur est associé. Dans la figure, nous pouvons voir qu'il y a 3 parties représentant chacune l'exécution d'un pas de test, ces parties contiennent 5 étapes numérotées. Les éléments du processus sont le système* $(Self - Adaptive\ system)$*, les traces d'exécution* $(Traces)$*, le générateur de test* $(Test\ Generator)$ *qui contient les modèles d'usage (cf figure 5.12).*

La première partie de la figure montre l'exemple du véhicule $v5$ *qui demande à joindre le peloton* $p1$ *qui accepte la demande.*

Dans un premier temps, le générateur de test sélectionne un composant ($(1)\ selectComponent() : v5$*). Ensuite, la transition (visible en bleu dans la figure) spécifiée par le modèle d'usage du composant est sélectionnée et il s'agit de l'évènement* $join()$ *qui est envoyé au système adaptatif (*$(2) pickAndSend() : join$*). Le système met à jour ses variables internes et déclenche la reconfiguration pour ajouter le véhicule* $v5$ *au peloton*

$p1$. *Cette reconfiguration est ajoutée dans la trace d'exécution (*(3) $p1$ $acceptJoin()$*) à la suite de toutes les reconfigurations déclenchées depuis de début de l'exécution du système. Ensuite, le générateur de test récupère le contenu de la trace d'exécution pour en déduire la dernière reconfiguration déclenchée (*(4) $Retrieve$*). Le générateur de test met à jour l'état courant de* $v5$ *dont la transition est visible en vert (*(5) $super.acceptJoin()$*). À ce stade,* $super$ *correspond à l'identifiant de* $p1$ *pour* $v5$.

La deuxième partie de la figure montre l'exemple d'un système qui ne déclenche aucune reconfiguration.
Dans un premier temps, le générateur de test sélectionne un composant ((1) $selectComponent()$ $v4$*). Ensuite, la transition (visible en bleu dans la figure) spécifiée par le modèle d'usage du composant est sélectionnée et il s'agit de la quiescence* δ *(*(2) $pick()$: δ*). Le système met à jour ses variables internes mais ne déclenche aucune reconfiguration (*(3) Run*). Le générateur de test récupère la trace d'exécution (*(4) $Retrieve()$*) qui ne demande pas d'effectuer l'étape de mise à jour d'aucun des modèles d'usage.*

Dans la troisième partie de la figure, nous détaillons la sortie du véhicule $v2.2$ *du convoi* $p2$ *qui entraîne également la sortie du véhicule* $v2.1$.
Dans un premier temps, le générateur de test sélectionne le composant $v1.1$ *(*$selectComponent()$ $v2.2$*). Ensuite, la transition (visible en bleu dans la figure) spécifiée par le modèle d'usage du composant est sélectionnée et il s'agit à nouveau de la quiescence* δ *(*(2) $pick()$: δ*). Le système met à jour ses variables internes dont le niveau de batterie de* $v2.1$ *qui devient trop faible et déclenche la reconfiguration faisant sortir* $v2.1$ *de* $p2$*. Seul le véhicule v2.2 est encore présent dans* $p2$ *ce qui déclenche en même temps la dissolution de* $p2$ *et donc la sortie de* $v2.2$*. Le système enregistre la reconfiguration dans la trace d'exécution (*(3)$p2$ $acceptQuit()$ $p2$ $scatter()$*). Le générateur de test récupère la trace d'exécution (*(4) $Retrieve()$*) et met à jour l'état courant de* $v2.1$ *(*(5) $super.acceptQuit()$*) et de* $v2.2$ *(*(5) $super.scatter()$*) dont les transitions sont visibles en vert.*

En association avec les configurations initiales, le générateur nous permet de générer des cas de test. Un cas de test est un couple avec un état initial suivi d'évènements contrôlables. Les cas de tests sont les éléments issus du processus de génération de tests décrit dans ce chapitre.

Définition 33 : Cas de test

Soit c_0 une configuration initiale du système et θ une séquence d'évènements contrôlables.
Un cas de test tc est défini comme $tc = \langle c_0; \theta \rangle$.
Un ensemble non vide de cas de tests est appelé une suite de tests TS :
TS=$\{tc1, tc2, ..., tcn\}$

Un cas de test peut être vu comme un mot, composé d'une configuration initiale suivie d'une séquence d'évènements contrôlables. Voici un exemple de cas de test (où les points virgules sont ajoutés pour plus de visibilité) associé au cas VANet : $S0; \delta; v1.enter(R); \delta; v1.join(v2); \delta; ...; \delta; v2.forceQuit(); \delta; ...; v2.join(v1); \delta; ...; \delta; v2.join(v3); \delta; \delta$
Les cas de test sont obtenus en utilisant le modèle de marche aléatoire de Markov [141]. À la manière d'une marche à l'aveugle, cette méthode consiste à générer des pas sachant que chaque pas n'est pas en corrélation avec le prochain.
Pour avoir une suite de tests, on répète le processus de génération de cas de tests.

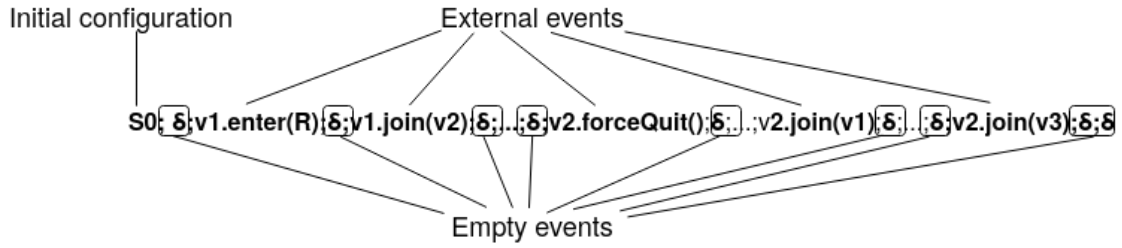

FIGURE 5.14 – Cas de test pour l'exemple VANet

Nous détaillons à présent l'exemple de cas de test associé au cas VANet.

Exemple 15 (*Cas de test pour l'exemple VANet*) : *Afin de rester lisible, le cas de test suivant est focalisé sur les évènements impliquant les véhicules $v1$, $v2$ et $v3$. L'identification des composants dans les évènements est nécessaire car un type de composant peut avoir plusieurs instances, comme le cas des véhicules qui sont tous définis par le même type de composant $Vehicle$ mais avec un identificateur différent (v1,v2,v3). Les instances de composants et leurs automates sont sélectionnés conformément à l'algorithme 3. Il en est de même pour le tirage aléatoire des transitions qui sont des évènements contrôlables. La figure ne représente pas les évènements observables car ils ne font pas partie des cas de tests mais sont utiles à leur génération. Ces évènements observables seront mentionnés dans l'exemple pour comprendre les choix du système.*
Dans un premier temps, $v1$ entre sur la route suite à l'évènement $v1.enter$. Ensuite, $v1$ essaye de se joindre à un convoi en envoyant une demande au véhicule $v2$ avec l'évènement $v1.join(v2)$. Nous considérons que le véhicule $v2$ accepte avec l'évènement observable $v2.acceptJoin(v1)$. Plus loin dans la séquence, $v2$ décide de quitter le convoi avec

l'évènement $v2.forceQuit()$*, ce qui est reconnu par le peloton avec un évènement observable* $super.acceptQuit(v2)$*.*
Plus tard, $v2$ *essaye de se joindre à nouveau dans le convoi (*$v2.join(v1)$ *sans que le système n'accepte avec un évènement observable* $super.acceptQuit(v2)$*. Pour finir,* $v2$ *essaye une nouvelle fois de rejoindre un convoi avec l'évènement* $v2.join(v3)$ *mais le système ne réagit toujours pas sûrement car* $v2$ *ne satisfait plus les conditions pour rejoindre le convoi.*

5.4/ CONCLUSION

Dans ce chapitre, nous avons présenté un algorithme de génération de configurations initiales qui s'appuie sur l'architecture à composants définie dans le chapitre 4 ainsi qu'une méthode d'initialisation de variables. Cette méthode d'initialisation des variables associées à des domaines, applique des distributions probabilistes paramétrables pour faire varier la densité de tirage sur ces domaines de définition.
Nous avons également développé une méthode pour la génération d'évènements permettant de stimuler le système. Cette méthode utilise des modèles d'usage qui décrivent des évènements, tout en étant cohérent avec les évènements de reconfiguration que le système a effectué. De ce fait, nous avons présenté un processus de génération de cas de test. Dans le prochain chapitre, nous nous focalisons sur les méthodes d'évaluation du comportement du système pour nous assurer que le système satisfait les propriétés temporelles et obéit aux politiques d'adaptation. En effet, le système stimulé doit se reconfigurer de manière conforme aux reconfigurations attendues. La conformité du système vis-à-vis de ses traces d'exécution permet de faire le lien entre les évènements de test et les chemins de reconfiguration.
Maintenant que nous avons présenté comment des cas de tests sont créés, il est possible de lancer des simulations du système et d'évaluer son comportement.

6

VALIDATION DE SYSTÈMES ADAPTATIFS

Sommaire

6.1 Validité du système vis à vis des propriétés FTPL 102

6.1.1 Critères de Couverture des propriétés FTPL 102

6.1.2 Verdict de test en rapport avec les propriétés FTPL 106

6.2 Validité du système vis à vis des politiques d'adaptation 107

6.2.1 Couverture de politique d'adaptation 107

6.2.2 Verdict de test basé sur une politique d'adaptation 109

6.2.3 Mesure de couverture pour les politiques d'adaptation 110

6.2.4 Conformité avec les politiques d'adaptation 113

6.3 Conclusion 114

Les systèmes adaptatifs sont guidés, contrôlés par des artefacts externes qui sont distincts du système. Parmi ces artefacts, on retrouve les propriétés temporelles et les politiques d'adaptation. Ces systèmes adaptatifs ont par définition la capacité de s'adapter continuellement en réponse aux évènements qui surviennent lors de leur exécution. Dans notre cas, cette capacité d'adaptation est pilotée par les politiques d'adaptation qui définissent des règles de reconfiguration en fonction des évènements externes et internes du système. Les évènements internes sont dépendants du système qui n'est donc pas directement contrôlable et peut avoir une multitude de configurations différentes. Pour répondre à ces défis, nous proposons une solution de validation en rapport avec les propriétés temporelles et politiques d'adaptation. Pour cela, nous définissons des critères de couverture qui fournissent un objectif de test et peuvent être utilisés comme moyen d'évaluer une suite de tests, en mesurant la part de l'artefact considéré que la suite de tests couvre. Nous détaillons la mise en place des critères de couverture puis nous proposons une méthode d'analyse du système permettant de vérifier son comportement que nous proposons de positionner, dans un premier temps au niveau des propriétés temporelles. Dans un deuxième temps, nous proposons la même mise en place mais cette fois pour les politiques d'adaptation.

6.1/ VALIDITÉ DU SYSTÈME VIS À VIS DES PROPRIÉTÉS FTPL

Dans cette section, nous nous focalisons sur la validité des propriétés temporelles. Nous commençons par définir des critères de couverture pour ensuite établir des verdicts de tests basés sur le respect des propriétés FTPL.

6.1.1/ CRITÈRES DE COUVERTURE DES PROPRIÉTÉS FTPL

Nous proposons de mesurer le taux de couverture des propriétés temporelles en utilisant les chemins de reconfiguration. Ces mesures ont pour objectif de donner une indication de la pertinence des cas de tests. Nous proposons de définir les critères de couverture permettant de satisfaire les buts suivants :

— Ces critères de couverture peuvent servir de condition d'arrêt pour la phase de génération de tests. Par exemple, en générant un nombre d'évènements contrôlables suffisant pour atteindre un certain taux de couverture. Ces critères de couverture se mesurent en observant les configurations successives du système, qui est mis à jour à chaque changement de configuration.

— Ces critères servent de moyen de mesure et de comparaison entre plusieurs suites de tests. En effet, il est possible de mesurer quel taux de couverture est obtenu pour une suite de tests (ou chaque cas de test séparément) vis-à-vis des propriétés temporelles et des politiques d'adaptation.

Les propriétés FTPL sont utilisées pour décrire des propriétés de sûreté ou de vivacité qui peuvent être vérifiées lors de l'exécution du système. Nous proposons de construire un automate de propriétés en utilisant les évènements concernés comme étiquettes de transitions, comme dans les travaux de [143].

Définition 34 : Automate de propriété FTPL

Soit φ une propriété FTPL, et soit Ev_φ l'ensemble des évènements se produisant dans φ. L'automate associé avec φ, noté $\mathcal{A}_\varphi$, est défini comme un quadruplet $\langle Q, q_0, Q_f, T\rangle$ où Q est un ensemble d'états, $q_0 \in Q$ est l'état initial, $Q_f \subset Q$ est l'ensemble des états finaux, et $T \in Q \times Ev_\varphi \times Q$ est la fonction de transition.

Nous présentons un exemple de la création de l'automate visible dans les figures 6.2 et 6.1.

Exemple 16 : *L'automate que nous souhaitons construire provient de la propriété (after acceptJoin until acceptQuit) (and) (always VehicleId.battery < 33). Cette propriété est vraie lorsqu'un véhicule est dans un convoi (after acceptJoin until acceptQuit), et que sa batterie est strictement inférieure à 33% (VehicleId.battery < 33). Cette propriété est constituée par des mots-clés temporels et leurs évènements associés. Notre automate*

FIGURE 6.1 – Automates **after** *ev temp* (gauche), et **before** *ev trace* ou *trace* **until** *ev* (droite)

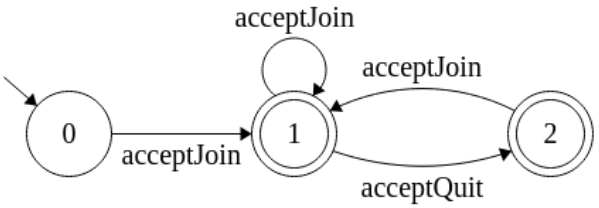

FIGURE 6.2 – automate pour la composition **after**-**until**

est donc constitué des transitions $acceptJoin$ *et* $acceptQuit$ *qui correspondent aux transitions des modèles d'usage des véhicules. L'état final est atteint lorsque le scénario de la propriété a été complété, dans notre exemple lorsqu'un véhicule est entré et sorti du convoi.*

Comme nous pouvons le voir dans cet exemple, les états 0 et 2 sont équivalents d'un point de vue de la configuration du système. Cependant, ces états sont distincts pour attester que la propriété a été parcourue au moins une fois et dissocier l'état initial de l'état final.

L'automate associé à la propriété (after acceptJoin until acceptQuit) (and) (VehicleId.battery < 33) est obtenu en composant deux autres automates. Un automate $after$ *décrivant les états pour l'évènement* $acceptJoin$ *et un automate* $until$ *décrivant les états pour l'évènement* $acceptQuit$*. Dans la figure 6.1, l'état représenté sous forme de carré représente un état hiérarchique dans lequel un autre automate peut être inséré et* $\sum$ *représente l'ensemble des étiquettes des transitions possibles.*

Dans notre cas, les états hiérarchiques se situent dans les automates ***after*** *conformément à la sémantique donnée en section 3.4.2. En effet, cette définition stipule que la transition* ***after*** *est suivie d'un évènement et d'un autre marqueur temporel (****after****,* ***before*** *ou* ***until****). Les états représentés sous la forme d'un double cercle sont les état finaux de leurs automates respectifs comme cela est expliqué dans [143].*

Les états hiérarchiques permettent de combiner plusieurs automates. Par exemple, l'automate associé à la propriété ***after*** $acceptJoin$ ***until*** $acceptQuit$ *est obtenu en combinant l'automate* $after$ *décrivant les états pour l'évènement* $acceptJoin$ *et l'automate* ***until*** *décrivant les états pour l'évènement* $acceptQuit$*.*

Cet exemple de combinaison est visible en figure 6.2 qui est le fruit de la composition des automates visibles en figure 6.1. Il est important de souligner que l'automate pour la

propriété φ se concentre uniquement sur les évènements pertinents *selon φ, et les autres évènements ne sont pas représentés dans les transitions de l'automate.*

Afin de mesurer la couverture de la propriété d'un automate, chaque configuration ($\sigma(i)$) est associée à un état de l'automate.

Définition 35 : Association d'un état

Soit $\mathcal{A}_\varphi$ un automate de la propriété φ, soit σ un chemin de reconfiguration et soit Θ un ensemble d'évènements de test. L'état associant un état dans $\mathcal{A}_\varphi$ avec chaque configuration de σ, est défini récursivement par :

$$\begin{aligned} state(\sigma(0)) &= q_0 \\ state(\sigma(i)) &= \begin{cases} q_i \ if \ \exists ev \in \Theta . \sigma(i-1) \xrightarrow{ev} \sigma(i) \wedge state(\sigma(i-1)) \xrightarrow{ev} q_i \in T_{\mathcal{A}_\varphi} \\ q_{i-1} \ else \end{cases} \end{aligned}$$

Littéralement, si on est au début du chemin de reconfiguration alors l'état est $q0$. Dans le cas général, soit nous sommes à l'état q_i s'il existe un évènement faisant la transition entre $\sigma(i-1)$ et $\sigma(i)$ et que l'automate contienne une transition correspondant à cet évènement ; sinon, on reste dans le même l'état : q_{i-1}. Nous définissons à présent le critère de couverture des propriétés FTPL pour le test.

Définition 36 : Transitions couvertes dans $\mathcal{A}_\varphi$

Soit ev un évènement de test, tc un cas de test basé sur un chemin de reconfiguration σ et soit φ une propriété FTPL. L'ensemble des transitions *couvertes* dans $\mathcal{A}_\varphi = \langle Q, q_0, Q_f, T \rangle$, noté $covered(tc, \varphi)$, est défini comme suit :

$$\begin{aligned} \{q \xrightarrow{ev} q' \in T \quad | \exists j \geq 0. \quad & (state(\sigma(j)) = q \wedge state(\sigma(j+1)) = q' \wedge \\ & \sigma(j) \xrightarrow{ev} \sigma(j+1) \wedge \exists k \geq j . state(\sigma(k)) \in Q_f)\} \end{aligned}$$

Littéralement, cela signifie que s'il existe un évènement qui sert de transition de l'état q à l'état q' de $\mathcal{A}_\varphi$ alors cette transition existe sous forme de reconfiguration et les indices de configuration se succèdent directement et qu'il existe un entier k supérieur ou égal à j tel que l'état de la configuration à l'indice k soit l'état final de $\mathcal{A}_\varphi$.
Cette définition fait le lien entre les transitions de l'automate de propriété et les transitions entre chaque configuration. Notons que cette définition requiert, pour qu'une transition soit considérée comme couverte que l'exécution atteigne ensuite un état associé à l'état final de l'automate. Comme dans un état final, la décision peut être faite sur la validité de la propriété. C'est pour cela qu'il est obligatoire que le cas de test atteigne l'état final, dans le cas contraire, la propriété ne peut être évaluée.
Nous définissons à présent notre critère de couverture pour les automates de propriétés FTPL inspiré des travaux de [22].

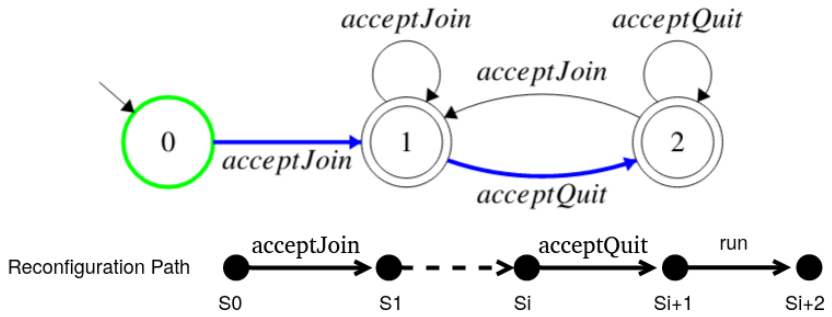

FIGURE 6.3 – Couverture de la propriété **after until**

Soit une propriété φ et A_φ son automate associé. Pour qu'un couple de transitions soit couvert, il faut que ces transitions entrantes et sortantes soient déclenchées par le générateur de test comme évènement externe ou par le système comme opération de reconfiguration. Les transitions réflexives, notées $\sum$, servent à capturer toutes les exécutions possibles du modèle et ne rentrent pas en compte pour la mesure de couverture. Ceci nous amène à la mise en place des critères de couverture de propriétés grâce aux suites de tests.

Définition 37 : Couverture de propriété

Une propriété FTPL φ est dite couverte (noté *covered*) par une suite de tests TS, lorsque chaque couple de transitions de la propriété φ de l'automate est couverte (critère 'all-transition-pairs coverage' rappelé en sous-section 2.3.1.1) par au moins un cas de test dans TS :

$$\bigcup_{tc \in TS} covered(tc, \varphi) = T_{\mathcal{A}_\varphi}$$

Une propriété est alors dite *couverte* lorsque tous les couples de transitions de son automate associé ont été couverts.

Nous présentons un exemple de couverture de propriété FTPL à partir de la figure 6.3.

Exemple 17 (*Couverture de propriété sur une trace d'exécution*) : *Soit la propriété* ***after*** $acceptJoin$ ***always*** $battery > 33$ ***until*** $acceptQuit$ *dont l'automate et le chemin de reconfiguration sont représentés dans la figure 6.3. Nous représentons les transitions parcourues en bleu et les états couverts en vert.*

Pour un cas de test et le chemin de reconfiguration associé, imaginons que les parties incomplètes du chemin ne contiennent que des opérations de type run*. Dans ce cas, les transitions de la propriété qui sont couvertes par ce cas de test sont* $0 \overset{V1.acceptJoin}{\rightarrow} 1$ *et* $1 \overset{V1.acceptQuit}{\rightarrow} 2$*.*

Par conséquent, la propriété n'est pas couverte. Par exemple, la transition $2 \overset{V1.acceptJoin}{\rightarrow} 1$ *n'est pas couverte et soit nous pouvons compléter la trace en continuant l'exécution*

du système à partir de l'état actuel de l'automate de la propriété soit en relançant une exécution avec un nouveau cas de test et en partant de l'état initial de l'automate de la propriété.

6.1.2/ VERDICT DE TEST EN RAPPORT AVEC LES PROPRIÉTÉS FTPL

Le verdict de test que nous proposons d'établir dans cette sous-section est en rapport avec les propriétés temporelles qui peuvent être évaluées à l'exécution.
Nous nous appuyons sur les travaux de [96] sur la vérification des propriétés FTPL. La sémantique du langage FTPL est rappelée en section 3.3.2.
Pour chaque évènement externe ext qui peut se produire sur un chemin de reconfiguration σ, nous définissons : *a*) une garde cp_{ext}, qui est une logique du premier ordre utilisant les évènements externes ext, et *b*) une assertion $eval_\sigma$, dont les valeurs sont dans $\mathbb{B}_2$. Intuitivement, si avant ou lors du $i_{ième}$ état (ou, si $i = 0$, à l'état initial) d'un chemin d'exécution σ, il y a au moins une occurrence de ext tel que $cp_{ext} = \top$ alors $eval_\sigma(cp_{ext}, i) = \top$. Sinon $eval_\sigma(cp_{ext}, i) = \bot$.
Avec l'exécution d'un cas de test sur le système, nous pouvons observer les éléments du chemin de reconfiguration σ vérifiables par les propriétés FTPL. Cependant, si le cas de test ne couvre pas certaines propriétés, il est alors impossible de conclure sur leur satisfaisabilité.
Par exemple, considérons une propriété **after** e_1 **before** e_2 **eventually** cp. Si la trace s'arrête après avoir observé e_1 mais avant avoir observé e_2, il est impossible de conclure sur l'occurrence de cp dans l'intervalle. Dans ce cas, la propriété devrait être déclarée comme 'potentiellement fausse' ($\bot^p$ conformément à la sémantique utilisée dans [96]).
Le cas de test à l'origine de la trace ne permet alors pas de conclure vis-à-vis de cette propriété celle-ci sera déclarée 'inconclusive'. De ce fait, notre processus d'établissement de verdict de test se base sur un préfixe de la trace de reconfiguration qui s'arrête si aucune configuration n'est associée à l'état final de la propriété de l'automate. Nous définissons maintenant le verdict de test pour l'exécution d'un cas de test vis-à-vis d'une propriété donnée.

Définition 38 : Verdict de test pour une propriété FTPL

Basé sur un chemin de reconfiguration σ, le verdict de l'exécution d'un cas de test tc, par rapport à la propriété φ est défini par :

$$verdict(tc, \varphi) = \begin{cases} \textit{pass if } \exists i.state(\sigma(i)) \in Q_{f_{A_\varphi}} \wedge \sigma_0^i \models \varphi \wedge \forall k > i \wedge state(\sigma(k)) \notin Q_{f_{A_\varphi}} \\ \textit{fail if } \exists i.state(\sigma(i)) \in Q_{f_{A_\varphi}} \wedge \sigma_0^i \not\models \varphi \\ \textit{inconclusive if } \forall i.state(\sigma(i)) \notin Q_{f_{A_\varphi}} \end{cases}$$

Il est important de rappeler que chaque cas de test de chaque suite de test doit être

évalué par chaque propriété du système. Ce verdict permet de s'assurer que le système ne viole pas de propriété. Cependant, il est possible que le système ne lève pas d'erreur mais ne respecte pas les règles de reconfiguration définies par les politiques d'adaptation.
C'est pourquoi nous proposons en section 6.2.1 un verdict basé cette fois sur les politiques d'adaptation.

6.2/ VALIDITÉ DU SYSTÈME VIS À VIS DES POLITIQUES D'ADAPTATION

Dans cette section, nous abordons l'évaluation du comportement du système des politiques d'adaptation. Nous commençons par définir des critères de couverture pour ensuite établir des verdicts de tests sur le respect des politiques d'adaptation ainsi que des outils permettant de valider l'implémentation des politiques d'adaptation.

6.2.1/ COUVERTURE DE POLITIQUE D'ADAPTATION

Nous utilisons les politiques d'adaptation selon la même vision que [46, 95], dans laquelle les politiques d'adaptation utilisent des schémas temporels prenant en compte des évènements contrôlables.

Les politiques d'adaptation indiquent au système comment se reconfigurer, il est donc important de s'assurer que les reconfigurations qui découlent de ces politiques d'adaptation soient bien déclenchées par le système. Les politiques d'adaptation sont définies comme un ensemble de règles qui s'appliquent au système et en particulier à ses composants.

Définition 39 : Couverture d'une règle d'adaptation

Soit tc un cas de test, σ un chemin de reconfiguration lié à tc, et $A = \langle R_N, R_R \rangle$ (cf définition 25) une politique d'adaptation. Une règle $r = \langle F, B, G, I \rangle \in R_R$ est dite couverte (*covered*) par tc, noté tc *covers* r, si :

$$\forall ope.ope \in dom(I) \Rightarrow \exists c, c' \in \sigma.c \xrightarrow{ope} c' \wedge B_c \wedge G_c$$

Intuitivement, cette définition correspond à l'activation de l'opération de reconfiguration pour un pas spécifique du chemin de reconfiguration. L'opération de reconfiguration est décrite dans la règle d'adaptation dans laquelle la garde et la règle de déclenchement sont évaluées à vrai. Ce critère de couverture nécessite que la règle soit déclenchée au moins une fois sans tenir compte de son utilité.

À présent, nous élevons cette notion de couverture à l'ensemble des règles. La couver-

ture d'une politique d'adaptation mesure le déclenchement de toutes les règles qu'elle contient. Cette couverture est évaluée selon un ensemble de chemins de reconfiguration obtenu par l'exécution des cas de tests. Nous présentons un exemple de couverture de règle d'adaptation ci-dessous.

WHEN (after getRelay before passRelay) = TRUE

IF (battery < 30) = TRUE

THEN utility of passRelay is high

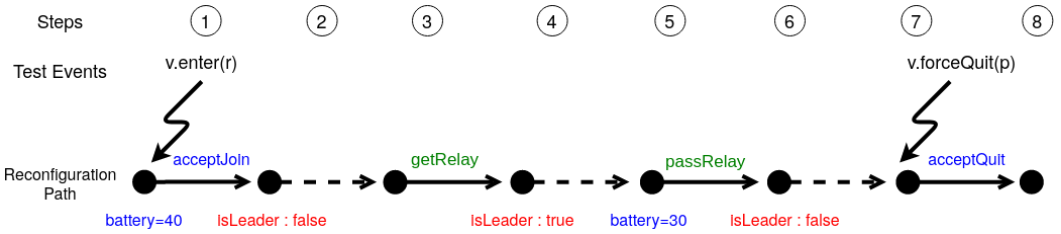

FIGURE 6.4 – Couverture de la règle passRelay

Exemple 18 (*Couverture d'une règle d'adaptation*) : *Cet exemple visible en figure 6.4 où la règle est visible en haut de la figure où sont représentés en bleu (**WHEN**) les éléments de la condition temporelle de déclenchement de la reconfiguration en rouge (**IF**) les éléments de la garde et en vert (**THEN**) les éléments de la reconfiguration. Le code couleur est conservé dans le chemin de reconfiguration visible en bas de la figure.*
*Dans cet exemple, la condition **WHEN** est vraie lorsque le véhicule est dans un convoi. La condition **IF** est vraie lorsque la batterie du véhicule est inférieure à 30%. Si les conditions **WHEN** et **IF** sont réunies alors (**THEN**) la règle $passRelay$ est éligible pour être déclenchée avec une utilité haute (high).*
*Nous pouvons voir qu'à l'état 1 de notre exemple le véhicule n'est pas encore dans un convoi et a un niveau de batterie supérieur à 30%. À l'état 2, le véhicule est dans un convoi. Plus tard à l'état 4, le véhicule est élu $leader$ du convoi et valide ainsi la condition **WHEN**. Plus tard, à l'état 5, le véhicule a un niveau de batterie inférieur à 30% ce qui valide la condition **IF** et rend la règle d'adaptation éligible pour être déclenchée. À L'état 6, la reconfiguration a été déclenchée et la règle est donc couverte.*

Maintenant que nous avons définit le critère de couverture pour une règle d'adaptation, nous allons définir le critère de couverture pour une politique d'adaptation.

Définition 40 : Couverture de politique d'adaptation

Soit R_R un ensemble de règles de reconfiguration d'une politique d'adaptation A, et soit TS une suite de tests. La politique d'adaptation A est dite couverte (*covered*) si chaque règle d'adaptation $r \in R_R$ est couverte par un cas de test tc : $\forall r.r \in R_R \Rightarrow \exists tc \in TS.\, tc$ *covers* r

Maintenant que les critères de couverture ont été définis, nous pouvons passer à l'établissement des verdicts de tests.

6.2.2/ VERDICT DE TEST BASÉ SUR UNE POLITIQUE D'ADAPTATION

Dans cette section, nous définissons comment établir un verdict de test cette fois sur les politiques d'adaptation $A = \langle R_N, R_R \rangle$. Intuitivement, le verdict sert à détecter les occurrences de reconfigurations non attendues en se basant sur les règles de reconfiguration décrites dans les politiques d'adaptation.

Pour chaque configuration $\sigma(i)$ de σ liée à l'exécution d'un cas de test, nous classons chaque règle d'opération de reconfiguration selon 3 ensembles définis de la manière suivante :

— $trig_{\sigma(i)}$ est l'ensemble des politiques d'adaptation déclenchables dans $\sigma(i)$:

$$trig_{\sigma(i)} = \{r \in R_R | B^r_{\sigma(i)} = true\}$$

— $elig_{\sigma(i)}$ est l'ensemble des règles d'adaptation activables :

$$elig_{\sigma(i)} = \{r \in R_R | B^r_{\sigma(i)} = true \wedge G_{\sigma(i)} = true\}$$

— $actual_{\sigma(i)} \in \mathcal{R}_{run}$ est l'opération de reconfiguration, déclenchée à l'étape i de σ.

Ces ensembles permettent d'identifier l'ensemble des reconfigurations qui pourraient être exécutées pour une configuration donnée lors de l'exécution du système. En supplément, soit $R_{Ntrig_{\sigma(i)}}$ (resp. $R_{Nelig_{\sigma(i)}}$) dénote l'ensemble des noms d'opérations de reconfiguration dans R_N utilisés dans la partie utilité I pour les règles dans $trig_{\sigma(i)}$ (resp. $elig_{\sigma(i)}$). Nous expliquons maintenant la façon dont nous utilisons ces ensembles pour établir le verdict de test et produire des mesures de couverture sur les politiques d'adaptation. Pour ce faire, nous définissons deux types de comportements illégitimes sur le déclenchement des reconfigurations

Définition 41 : Reconfiguration inappropriée

Une reconfiguration est dite inappropriée à un certain pas $\sigma(i) \overset{ope}{\rightarrow} \sigma(i+1)$ d'un chemin de reconfiguration σ si la reconfiguration exécutée ($actual$) n'appartient pas à l'ensemble des reconfigurations éligibles ($elig$) :

$$actual_{\sigma(i)} \in \mathcal{R} \wedge actual_{\sigma(i)} \notin R_{Nelig_{\sigma(i)}}$$

Nous distinguons le cas où une reconfiguration inappropriée est choisie alors que l'ensemble des reconfigurations éligibles est non vide, et le cas où une reconfiguration inattendue est exécutée alors que l'ensemble des reconfigurations éligibles est vide.

Définition 42 : Reconfiguration inattendue

Une reconfiguration est dite inattendue si le système déclenche une opération de reconfiguration alors qu'aucune règle de reconfiguration n'est éligible :

$$actual_{\sigma(i)} \in \mathcal{R} \wedge elig_{\sigma(i)} = \emptyset$$

La reconfiguration inattendue est donc un cas particulier de la reconfiguration inappropriée.
Le verdict de test est établi en déterminant si une reconfiguration inappropriée ou une reconfiguration inattendue intervient. Dans ce cas le test échoue ($fail$), sinon il réussit ($pass$).

Définition 43 : Verdict de test pour les politiques d'adaptation

Soit tc un cas de test, et AP une politique d'adaptation. Le verdict de test associé à AP est établi par la fonction de verdict ($verdict$) définie par :

$$verdict(tc, A) = \begin{cases} \textit{fail} \text{ if } \exists \sigma(i) \in exec(tc).\ actual_{\sigma(i)} \neq run \wedge actual_{\sigma(i)} \notin R_{Nelig_{\sigma(i)}} \\ \textit{pass} \text{ otherwise} \end{cases}$$

Grâce à ce mécanisme, nous garantissons que le système déclenche les reconfigurations attendues et ne déclenche pas de reconfiguration inattendue. Nous souhaitons également définir une mesure de couverture sur les occurrences de reconfiguration afin de fournir des données sur les règles des politiques d'adaptation.

6.2.3/ MESURE DE COUVERTURE POUR LES POLITIQUES D'ADAPTATION

En supplément des verdicts de tests, nous proposons de nous intéresser à la question de la couverture des différentes règles pour un cas de test donné (ou une suite de tests). Le but de cette mesure est de fournir un retour sur la pertinence des tests examinés en informant sur les règles qui ont été déclenchées et à quelle fréquence [34]. Ce mécanisme permet également de détecter des erreurs dans la conception des politiques d'adaptation. Par exemple, c'est la cas quand une règle est trop restrictive et n'est jamais déclenchée.
Il est également possible que des erreurs soient présentes au moment de l'implémentation et particulièrement au moment du choix des valeurs floues utilisées pour déterminer l'utilité des règles.

STE en présence des politiques d'adaptation Avant de définir notre mesure sur le nombre de règles déclenchées, nous définissons de quelle manière les politiques d'adaptation affectent le comportement du modèle de reconfiguration $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ (cf

définition 16 du modèle de reconfiguration). Soit S un STE (système de transition d'états) et un ensemble fini de politiques d'adaptation AP, nous utilisons $S \triangleleft AP$ pour dénoter le modèle S restreint aux politiques dans AP. Soit $\mathcal{R}_{run} = \mathcal{R} \cup \Theta \cup \{run\}$ un ensemble d'opérations d'évolution, où $\mathcal{R}$ est un ensemble d'opérations de reconfiguration, Θ est l'ensemble des évènements contrôlables, et run est le nom de l'action générique permettant de représenter la mise à jour des variables internes au système à composants. Dans cette définition, nous supposons comme dans les travaux de [92], que les évènements externes sont capturés par le système et traités par le déclenchement de méthodes internes par le biais de transitions dans Θ.

Définition 44 : LTS resteint au politiques d'adaptation

La restriction du modèle S par les politiques d'adaptation dans AP est définie de la façon suivante $S \triangleleft AP = \langle C_{\triangleleft AP}, C^0_{\triangleleft AP}, \mathcal{R}_{run}, \rightarrow, l \rangle$, dans lequel $C_{\triangleleft AP}$ est l'ensemble minimal tel que si $c \in C$ et $A \in AP$ alors $c_{\triangleleft A} \in C_{\triangleleft AP}$, $\mathcal{R}_{run} \cap (\cup_{A \in AP} R_N) \neq \emptyset$, $l : C_{\triangleleft AP} \rightarrow CP$ est une fonction d'interprétation, (rappelée en section 3.3) et pour chaque $ope \in \mathcal{R}_{run}$, la relation de transition $\rightarrow \in C_{\triangleleft AP} \times \mathcal{R}_{run} \times C_{\triangleleft AP}$ est le plus petit ensemble de triplets $(c_{\triangleleft A}, ope, c'_{\triangleleft A})$ satisfaisant les règles suivantes :

$$\text{[ACT1]} \quad \frac{c \xrightarrow{ope} c'}{c_{\triangleleft A} \xrightarrow{ope} c'_{\triangleleft A}} \quad (ope \in \bigcup_{A \in AP} R_N) \wedge B \wedge G$$

$$\text{[ACT2]} \quad \frac{c \xrightarrow{ope} c'}{c_{\triangleleft A} \xrightarrow{ope} c'_{\triangleleft A}} \quad ope \notin \bigcup_{A \in AP} R_N$$

Cette définition signifie que les transitions de S sous AP résultent soit d'opérations de reconfiguration obéissant aux politiques d'adaptation (Règle *[ACT1]*), ou de reconfigurations qui ne sont pas impliquées dans les politiques d'adaptation (Règle *[ACT2]*)).

Concernant l'évaluation de la règle *[ACT1]*, il est possible d'évaluer la condition temporelle de déclenchement b, de manière décentralisée comme dans [11, 96].

6.2.3.1/ NOMBRE D'EXÉCUTIONS D'UNE RÈGLE

Le but de la mesure du nombre d'exécutions de règles est d'évaluer si les différentes règles des politiques d'adaptation ont été activées pour les cas de tests considérés.
Pour cela, nous définissons une fonction qui compte le nombre d'exécutions d'une règle pour une trace d'exécution.

Définition 45 : Nombre d'exécutions d'une règle

Soit σ un chemin de reconfiguration, le nombre d'exécutions d'une règle $r \in R_R$ dans σ est défini par :

$$card(actual^r(\sigma)) = \sum_i f_a^r(\sigma(i))$$

dans lequel $f_a^r(\sigma(i))$ est la fonction caractéristique du prédicat $actual_{\sigma(i)} \in dom(I_r)$ qui vaut 1 si le prédicat est validé ou 0 sinon.

Une reconfiguration inappropriée ne peut survenir que lorsqu'une reconfiguration est effectuée ($actual_{\sigma(i)} \in \mathcal{R}$). Mais il est également possible qu'une reconfiguration soit délibérément ignorée alors qu'elle était légitime. En effet, il est possible que l'implémentation du système ignore une potentielle reconfiguration. Ce cas n'est pas nécessairement considéré comme une erreur et peut être décelé grâce aux critères de couverture.

Cette information permet d'évaluer si une règle est couverte par un cas de test (ou une suite de tests). Si une règle n'est jamais couverte, plusieurs causes sont possibles. La première cause est que les cas de tests n'atteignent pas la configuration où la reconfiguration est applicable. Dans ce cas, la suite de tests doit être redéfinie afin de couvrir cette règle. La seconde cause est que des parties des règles sont mal implémentées et sont trop restrictives (condition temporelle de déclenchement et/ou la garde inatteignables). Nous proposons de mesurer le nombre de fois où ces parties de la règle sont satisfaites.

6.2.3.2/ ÉLIGIBILITÉ D'UNE RÈGLE

Nous définissons le nombre de déclenchement d'une règle qui est, pour une règle d'adaptation donnée, le nombre de configurations dans lesquelles la propriété de déclenchement est évaluée à vraie.

Définition 46 : Nombre de déclenchements d'une règle

Soit σ un chemin de reconfiguration, le nombre de déclenchement d'une règle $r \in \mathcal{R}$ sur σ est défini par :

$$\#trig^r(\sigma) = \sum_i f_t^r(\sigma(i))$$

dans lequel $f_t^r(\sigma(i))$ est la fonction caractéristique pour le prédicat

$$r \in trig_{\sigma(i)} \wedge r \notin trig_{\sigma(i-1)} \wedge actual_{\sigma(i-1)} \notin dom(I_r)$$

À présent, nous définissons le nombre d'éligibilités d'une règle pour un chemin donné σ. Le nombre d'éligibilités d'une règle est le nombre de configurations de σ où la règle est

éligible.

Définition 47 : Nombre d'éligibilité d'une règle

Soit σ un chemin de reconfiguration, le nombre d'éligibilité d'une règle $r \in \mathcal{R}$ est défini par :

$$\#elig^r(\sigma) = \sum_i f_e^r(\sigma(i))$$

dans lequel $f_e^r(\sigma(i))$ est la fonction caractéristique du prédicat

$$r \in elig_{\sigma(i)} \wedge r \notin elig_{\sigma(i-1)} \wedge actual_{\sigma(i-1)} \notin dom(I_r)$$

Ces mesures sur chaque règle permettent de déterminer quelle partie de la règle n'est pas satisfaite lors de l'exécution des cas de tests.
Cependant, dans certains cas, il est possible que la règle soit éligible mais que l'implémentation l'ignore. Pour cela nous complétons nos mesures avec la fréquence de déclenchement des règles d'adaptation.

6.2.3.3/ FRÉQUENCE D'UNE RÈGLE

La fréquence d'activation d'une règle est obtenue par le nombre de fois où cette règle a été déclenchée par rapport au nombre de fois où la règle a été éligible.

Définition 48 : Fréquence d'une règle

Nous définissons la fréquence d'une règle $r \in R_R$ pour un cas de test donné tc comme suit :

$$freq^r(tc) = \frac{\#actual^r(exec(tc))}{\#elig^r(exec(tc))}$$

Cette mesure est utile pour évaluer si une règle est souvent activée ou non. Une fois mesurée, la fréquence peut être comparée à la valeur floue de la règle étudiée et déceler une potentielle incohérence dans l'implémentation de la politique d'adaptation. Par exemple, il est possible de détecter qu'une règle avec un haut niveau d'utilité est activée avec une fréquence inférieure à une règle avec un faible niveau d'utilité.

6.2.4/ CONFORMITÉ AVEC LES POLITIQUES D'ADAPTATION

La conformité de l'implémentation vis-à-vis d'une politique d'adaptation peut être effectuée grâce aux valeurs d'utilité dans les règles d'adaptation. Soit une règle d'adaptation $r = (f, b, g, ir)$. À partir de la définition 25, la relation $ir = (ope, f)$ associe dans r l'utilité f avec la reconfiguration ope. En supposant que les valeurs floues soient ordonnées, nous

nous attendons à ce que chaque règle avec une utilité N soit appliquée avec une fréquence plus élevée qu'une règle avec une utilité $N-1$. Formellement, pour une politique d'adaptation donnée A, et une suite de tests donnée TS, nous signifions qu'une politique d'adaptation est fidèlement implémentée si :

$$\forall r, r' \in A, f^r > f^{r'} \Rightarrow freq^r(TS) > freq^{r'}(TS)$$

Soit $\sqsubseteq$ la relation d'intégration de sous-mots. Étant donné l'ensemble $\Theta \cup O$ d'évènements contrôlables et non-contrôlables (cf section 5.3.1), nous utilisons $\sqsubseteq$ modulo δ, que nous écrivons $\sqsubseteq_\delta$, après avoir retiré la quiescence δ des mots considérés.

Proposition 3 (*Atteignabilité du chemin de reconfiguration*) : *Si A est fidèlement implémentée, alors $\forall tc \in TS$, $\exists \sigma \in \Sigma_{S \lhd A}$ t.q. $tc \sqsubseteq_\delta tr(\sigma)$. De plus, $\forall i \geq 0$, $\sigma(i)$ est atteignable, avec les opérations de reconfiguration données par l'ensemble $elig_{\sigma(i)}$ des règles éligibles.*

Cette proposition indique que pour une politique d'adaptation fidèlement implémentée, il existe un chemin de reconfiguration basé sur un cas de test, qui met en relation une trace d'exécution avec ce cas de test.

Preuve. Par construction. À partir des configurations initiales, chaque tc est généré en appliquant l'algorithme 3 et en utilisant la définition 32. À partir des règles de $S \lhd A$, (cf. définition 44), chaque état $\sigma(i)$ du chemin σ correspondant à tc peut être obtenu en appliquant à chaque $\sigma(i)$ soit *[ACT1]* pour une des opérations de reconfiguration, présente dans l'ensemble $elig_{\sigma(i)}$ des règles éligibles à cette configuration, ou *[ACT2]* dans le cas d'une reconfiguration observable.

En partant des configurations initiales, dans le cas de l'application de la règle *[ACT1]* à l'état *i*, l'état cible ($actual_{\sigma(i)}$) est le résultat de la reconfiguration choisie en fonction de sa valeur d'utilité dans l'ensemble des règles de reconfiguration éligibles ($elig_{\sigma(i)}$) au moment de cette configuration. Ce choix de reconfiguration pendant l'exécution du système augmente la fréquence de reconfiguration et permet à la politique d'adaptation d'être fidèlement implémentée.

L'atteignabilité des configurations $\sigma(i)$ sur σ est garantie par construction. La relation $tc \sqsubseteq_\delta tr(\sigma)$ se base sur les définitions 32, 16 et 44.

6.3/ CONCLUSION

Dans ce chapitre, nous avons dans un premier temps explicité comment nous mettons en place la validation du système vis à vis des propriétés FTPL grâce à des critères de couverture ainsi que des verdicts de tests. Dans un second temps, nous avons procédé de même façon pour les politiques d'adaptation avec en plus la mesure de couverture

des politiques d'adaptation ainsi que l'évaluation de conformité vis à vis de celles-ci. Nous abordons dans la prochaine partie les chapitres décrivant notre approche expérimentale permettant de valider les contributions présentées.

7

EXPÉRIMENTATIONS

Sommaire

7.1 Implémentation 118
7.1.1 Système sous test 118
7.1.2 Propriétés FTPL 119
7.1.3 Politiques d'adaptation 120
7.1.4 Modèle d'usage 122
7.2 Expérimentation sur les critères de couverture 124
7.2.1 Protocole d'expérimentation 124
7.2.2 Résultats 125
7.2.3 Discussion 129
7.3 Évaluation de la pertinence des configurations initiales 129
7.4 Expérimentations sur les fréquences de déclenchement 133
7.4.1 Protocole d'expérimentation 133
7.4.2 Discussion 135
7.5 Conclusion 136

Dans les chapitres précédents, nous avons décrit nos contributions pour le test et la vérification des systèmes adaptatifs. Dans ce chapitre, nous présentons les expérimentations que nous avons mises en place et les résultats obtenus pour évaluer ces contributions. Dans un premier temps, dans la section 1, nous décrivons l'environnement mis en place pour effectuer ces expérimentations. Ces expérimentations s'appuient sur l'exemple VA-Net utilisé tout au long de la thèse pour illustrer nos propositions et ont pour but de valider plusieurs aspects ; dans la section 2, nous évaluons le premier aspect de nos travaux qui est la pertinence des critères de couverture des propriétés et des politiques d'adaptation ainsi que la méthode de génération d'évènements de tests à partir d'un modèle d'usage probabiliste ; dans la section 3, nous évaluons le deuxième aspect à savoir l'utilisation de notre modèle à composants et sa méthode de génération de configurations initiales ; dans la section 4, nous évaluons le troisième aspect à savoir la cohérence entre le comportement du système et l'implémentation des politiques d'adaptation. Enfin, nous terminons

dans la section 5 avec un bilan et une discussion sur ce qui ressort de ces expérimentations.

7.1/ IMPLÉMENTATION

Pour l'expérimentation, un simulateur de l'exemple VANet qui implémente les fonctions décrites dans le cas d'étude a été développé en langage Java (presque 6000 lignes de code).

Les entités considérées sont la route et ses véhicules, ainsi que les convois qui sont des entités virtuelles existant uniquement pour regrouper des véhicules. Les données du convoi comportent le nombre de véhicules qu'il contient ainsi qu'une référence vers le véhicule *leader*. Les véhicules sont caractérisés par leur niveau de batterie, leur distance à la destination et leur état qui indique, pour chaque véhicule, s'il est seul (*solo*), dans un convoi (*platooned* ou *leader*) ou arrêté (*stopped*) pour se recharger. Bien que ce système soit une simplification d'un système réel, dans lequel des abstractions ont été faites (par exemple la distance entre véhicules), la nature adaptative du système le rend suffisamment complexe pour effectuer des expérimentations. La simulation peut être changée facilement pour, par exemple, paramétrer les configurations initiales et les séquences d'évènements. Il est également possible pour l'ingénieur de validation de modifier l'implémentation des politiques d'adaptation qui guident les reconfigurations du système. Pour que notre système simule un cas d'étude en temps réel, nous utilisons des tics d'horloge qui représentent l'évolution du temps. À noter que l'implémentation des politiques d'adaptation peut dépendre des utilités de reconfiguration et des stratégies pour la prise en compte des reconfigurations avec le même niveau d'utilité.

7.1.1/ SYSTÈME SOUS TEST

Dans notre implémentation, les évènements externes et opérations de reconfiguration sont liés à des méthodes dans des objets spécifiques. Nous considérons les évènements externes suivants qui représentent des points de contrôle sur le système :

1. *createVehicle* ajoute un nouveau véhicule sur la route.
2. *requestJoin* qui représente un véhicule faisant une requête pour joindre un convoi.
3. *forceQuitPlatoon* survient quand un véhicule doit quitter le convoi.

Ces évènements sont utilisés dans les modèles d'usage (cf figure 5.12) considérés. Une probabilité est associée à chacun de ces évènements.

Les reconfigurations suivantes peuvent être effectuées dans le contexte d'un convoi :

1. *createPlatoon* ajoute un nouveau convoi sur la route.

2. *deletePlatoon* supprime un convoi de la route.

Dans le contexte d'un véhicule, on considère :

1. *getRelay* élit un véhicule pour qu'il devienne le leader du convoi.
2. *passRelay* remplace le véhicule leader avec un autre véhicule du convoi.
3. *acceptJoin* ajoute le véhicule au convoi, en réponse à un *requestJoin* qui s'est correctement déroulé.
4. acceptQuit survient lorsque le véhicule quitte le convoi.
5. addVehicle indique qu'un nouveau véhicule est sur la route.

Au moment de l'exécution, le système produit des traces fournissant les informations sur les configurations successives résultant des différentes opérations déclenchées. Ces traces peuvent être analysées pour établir le verdict de test.

7.1.2/ PROPRIÉTÉS FTPL

Pour les expérimentations, 5 propriétés temporelles FTPL ont été conçues. Les deux premières propriétés sont exprimées dans le contexte d'un convoi :

1. $\varphi 1$ **: after** createPlatoon **before** deletePlatoon **always** $Leader \neq$ null
   spécifie que, quand un convoi existe il doit toujours avoir un véhicule leader.
2. $\varphi 2$ **: after** createPlatoon **before** deletePlatoon **always** $VehicleNumber > 2$
   spécifie qu'un convoi doit toujours contenir au moins deux véhicules.

Les conditions $Leader \neq null$ et $VehicleNumber > 2$ concernent le modèle architectural. En effet, si un convoi n'a pas de véhicule $Leader$ le problème sera facilement détectable au niveau de son interface de délégation. De façon similaire, le nombre de véhicules présents dans un convoi est calculable en comptant le nombre de liens entre sous-composants du composant convoi.

Les 3 propriétés suivantes sont exprimées dans le contexte d'un véhicule :

1. $\varphi 3$ **: always** $Battery > 0$ and $Distance > 0$
   spécifie qu'un véhicule doit toujours avoir une distance restante et un niveau de batterie strictement positif.
2. $\varphi 4$ **: after** getRelay **before** passRelay **always** $Battery > 15$
   spécifie que, quand un véhicule est leader, il doit toujours avoir un niveau de batterie strictement supérieur à 15%.
3. $\varphi 5$ **: after** acceptJoin (**always not** docks) **until** acceptQuit spécifie qu'un véhicule dans un convoi ne peut pas être connecté à une station pour recharger sa batterie. Pour identifier si un véhicule est connecté à une station, nous regardons si l'interface correspondante $docks$ est active.

Les 4 propriétés suivantes ne font pas partie des expérimentations pour l'évaluation de la qualité des tests mais existent dans l'implémentation :

1. $\varphi 6$ **: after** acceptJoin **before** acceptQuit **always** $state \neq stopped$
   spécifie que, quand un véhicule est à l'intérieur d'un convoi, il ne peut être à l'arrêt.
2. $\varphi 7$ **: after** getRelay **before** acceptQuit **eventually** state ≠ leader spécifie qu'après avoir été élu leader, le véhicule doit passer le relai avant de quitter le convoi.
3. $\varphi 8$ **: after** getRelay **always** state = leader **until** passRelay spécifie qu'un véhicule élu leader garde ce status jusqu'au moment où il passe le relai.
4. $\varphi 9$ **: before** acceptJoin **always** state = solo spécifie que tant qu'un véhicule n'a pas été accepté pas un convoi son état est $solo$.

7.1.3/ POLITIQUES D'ADAPTATION

Les politiques d'adaptation que nous avons conçues décrivent le comportement des véhicules pour le système dans lequel les reconfigurations considérées consistent, pour un véhicule donné, à créer ou rejoindre un convoi, remplacer le leader du convoi ou quitter le convoi. Le cas d'étude VANet a 8 règles de reconfiguration de la politique d'adaptation visibles en figure 7.1, parmi lesquelles on retrouve $GetRelay$, $PassRelay$ et $Quit$ qui sont nommées $R1$ à $R8$. Les règles $R1$-$R4$ ont une utilité haute ($high$), les règles $R5$-$R6$ ont une utilité moyenne ($med.$), et les règles $R7$-$R8$ ont une utilité faible (low).

Les règles $R2$, $R5$ et $R8$ utilisent la fonction isTakingNextStation() qui renvoie vrai si le niveau de batterie du véhicule l'oblige à s'arrêter à la prochaine station, sinon la fonction renvoie faux.

Les règles $R1$ et $R6$ concernent des changements de véhicules $leader$ dans le convoi. Ainsi, la règle $R1$ indique que si le véhicule leader a un niveau de batterie inférieur à 30% alors un passage de relais est nécessaire. Et pour un véhicule dans un convoi qui n'est pas $leader$, la règle $R6$ indique que si son niveau de batterie est supérieur à celui du véhicule $leader$ alors un passage de relais est utile.

Les règles $R2$, $R3$, $R4$, $R5$, $R7$ et $R8$ définissent dans quelles conditions il est utile qu'un véhicule quitte le convoi :

— Soit parce que le véhicule se rapproche d'une station et son niveau d'énergie est faible ($R2$,$R5$ et $R8$),

— soit parce que le véhicule a un niveau de batterie trop faible pour pouvoir rester dans le convoi ($R4$),

— soit parce que le véhicule se rapproche de sa destination d'arrivée ($R3$ et $R7$).

Nos règles d'adaptation ont pour but de guider le comportement du système VANet en fonction des attributs des véhicules. Lors de la spécification, il est supposé que, plus

```
R1 : when (after getRelay before passRelay) = TRUE
  if (battery < 30) = TRUE then
       utility of passRelay is high

R2 : when (after acceptJoin before acceptQuit) = TRUE
  if (isTakingNextStation() and nextStation < 70) = TRUE then
       utility of acceptQuit is high

R3 : when (distance < 50) = TRUE
   if (state = platooned) = TRUE then
        utility of acceptQuit is high

R4 : when (after acceptJoin before acceptQuit) = TRUE
  if (battery < 20) = TRUE then
       utility of acceptQuit is high

R5 : when (after acceptJoin before acceptQuit) = TRUE
  if (isTakingNextStation() and nextStation < 85) = TRUE then
       utility of acceptQuit is medium

R6 : when (after acceptJoin before acceptQuit) = TRUE
  if (battery > Leader.battery) = TRUE then
       utility of getRelay is medium

R7 : when (distance < 100) = TRUE
   if (state = platooned) = TRUE then
        utility of acceptQuit is low

R8 : when (after acceptJoin before acceptQuit) = TRUE
  if (isTakingNextStation() and nextStation < 100) = TRUE then
       utility of acceptQuit is low
```

FIGURE 7.1 – Les règles de reconfiguration de la politique d'adaptation VANet

l'utilité d'une reconfiguration est élevée, plus il y a de chances de déclencher l'opération de reconfiguration correspondante. Par exemple, quand un véhicule souhaite quitter le convoi au même moment qu'un autre véhicule est éligible pour un passage de relai, la reconfiguration avec la priorité la plus élevée est choisie. L'utilité est déterminée par plusieurs critères, comme le niveau de batterie ou la distance par rapport à la destination. Des travaux similaires pour la classification des opérations de reconfiguration existent dans la littérature, par exemple dans [47], les auteurs conçoivent un classement des valeurs floues, l'utilité d'appliquer des opérations est garantie par une valeur de déclenchement. Les politiques d'adaptation implémentées dans le logiciel Tangram4Fractal [25] sont prises dans leur ordre d'apparition, ce qui signifie que la première règle applicable est sélectionnée. Dans les travaux de [130], les auteurs classent les reconfigurations en

fonction de leurs priorités et si deux reconfigurations avec la même priorité sont éligibles, celle apparue en premier est sélectionnée.

Un exemple de l'exécution du système est visible en figure 7.2. Cette figure montre un graphique par véhicule contenant :

1. La distance restante (courbe en tirets) et le niveau de batterie (courbe pleine) en pourcentage ;
2. La taille du convoi dans lequel se situe le véhicule (courbe en pointillés). Si le véhicule est en mode solo, afin de correspondre à l'échelle du graphique, il est considéré comme étant dans un groupe de véhicule de taille 1. Les points marqués en gras symbolisent que le véhicule est leader du convoi ;
3. La station apparaît sous forme de gros points (rouges) sur l'axe des abscisses. ;
4. Les évènements externes sont affichés par des lignes verticales. Les noms des évènements sont indiqués.

Le comportement des véhicules illustré dans ces graphiques est guidé par les politiques d'adaptation. Considérons le véhicule A dans la figure 7.2, quand son niveau de batterie descend en dessous de 20% au tic numéro 80, ce véhicule doit quitter le convoi pour recharger ses batteries à la prochaine station. Comme le véhicule A est leader au moment de quitter le convoi, un nouveau $leader$ doit être élu. Nous pouvons voir que le véhicule C devient $leader$. Quelques étapes plus tard, le véhicule C devient faible en énergie et quitte à son tour le convoi. Comme il n'y a pas de nouveau véhicule disponible pour devenir le nouveau leader, le convoi est supprimé. Parfois, les évènements externes causent une reconfiguration, c'est le cas de l'évènement $requestJoin$ au tic numéro 200, après lequel les véhicules B et C sont regroupés pour former un nouveau convoi. Cet évènement correspond, dans le monde réel, à une situation dans laquelle un véhicule se rapproche d'un autre véhicule et fait une requête de regroupement.

7.1.4/ MODÈLE D'USAGE

Le modèle probabiliste est implémenté avec ModelJUnit [106], une librairie qui permet de concevoir et exploiter les machines à états finis en Java. Contrairement à d'autres outils fonctionnant sur des automates explicitement définis, ModelJUnit offre la possibilité d'encoder les machines à états en utilisant des variables et de les abstraire en définissant une fonction d'identification d'état. Ainsi, le modèle d'environnement du VANet, qui dépend fortement de la création dynamique d'instances de véhicules, n'a pas à être défini explicitement. De plus, ModelJUnit peut être utilisé pour générer des cas de test hors ligne, ou en ligne en étant directement connecté au système à tester. Dans nos expérimentations, nous utilisons cette dernière possibilité pour générer les traces de test. Cela permet de connaître l'état du composant concerné, par exemple, afin d'être en mesure de

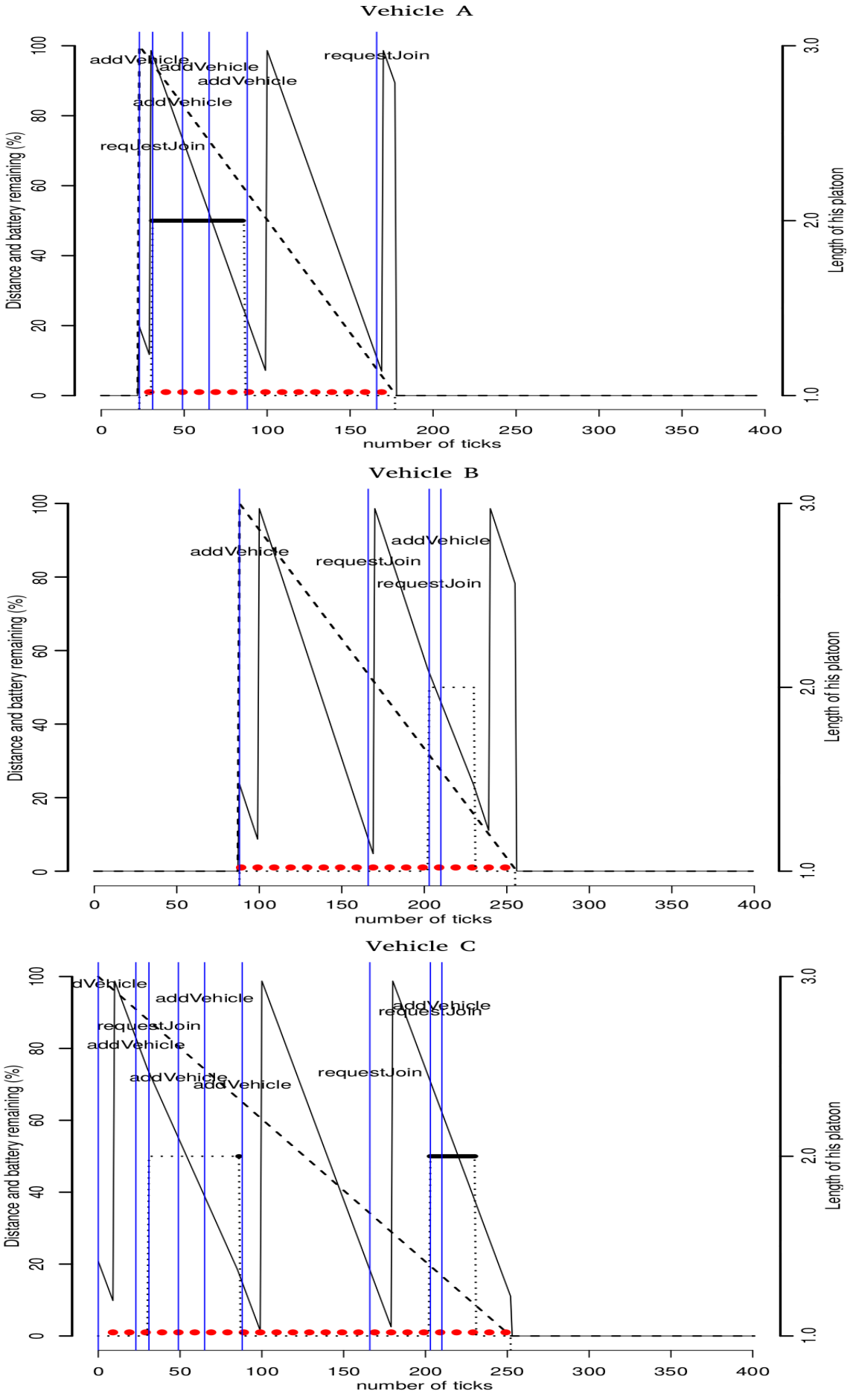

FIGURE 7.2 – Un extrait de l'exécution du cas d'étude

décider si un véhicule est en mode *solo* ou *platooned*. En effet, le seul événement externe consiste, pour un véhicule *solo*, à demander l'adhésion à un convoi, il est obligatoire de savoir si le véhicule est déjà dans un convoi ou non avant d'envoyer une telle demande. Comme il n'y a pas de contrôle sur cet événement, nous devons examiner l'état actuel de l'exécution pour savoir si la demande est pertinente ou non.

Pour le besoin de nos expérimentations, ModelJUnit a été étendu pour accepter les probabilités sur les transitions et un nouvel algorithme pour effectuer une marche aléatoire qui est décrite en section 5.3.2 a été implémenté. Chaque cas de test généré est enregistré pour être ré-exécuté plus tard. L'objectif de nos expérimentations est de valider les capacités de détection offertes par notre approche sur les politiques d'adaptation.

7.2/ EXPÉRIMENTATION SUR LES CRITÈRES DE COUVERTURE

Cette section relate des expérimentations effectuées pour évaluer les contributions présentées dans le chapitre 6.

7.2.1/ PROTOCOLE D'EXPÉRIMENTATION

Ces expérimentations ont pour but de répondre à trois questions de recherche en s'appuyant sur notre exemple de convoi de véhicules autonomes.

RQ1. Dans quelle mesure le critère de couverture des propriétés est intéressant pour détecter les erreurs ?

RQ2. Dans quelle mesure le critère de couverture des règles d'adaptation est complémentaire au critère de couverture des propriétés ?

RQ3. Dans quelle mesure les modèles probabilistes sont appropriés pour générer automatiquement des cas de tests qui satisfont les critères de couverture ?

Afin de valider ces questions de recherche, nous avons mis au point une procédure d'expérimentation que nous avons appliqué au cas de test VANet.

Dans un premier temps, nous avons conçu un modèle de test pour l'environnement VANet décrivant les occurrences d'interactions qui peuvent arriver entre les véhicules. Ensuite, pour répondre à la RQ3, nous avons utilisé notre générateur de test pour produire aléatoirement des suites de tests, et nous avons noté la taille moyenne des suites de test qu'il est nécessaire de générer pour satisfaire les critères de couverture considérés.
Nous espérons que, premièrement, le processus de génération de test sera capable de générer des suites de tests qui satisferont les critères de couvertures, et, deuxièmement, que cela ne demandera pas une extension des suites de tests pour atteindre ce taux de couverture. Cela fait référence à la section 6.1.1 définissant les critères de couverture

des propriétés FTPL (RQ1) et à la section 6.2.3 définissant les critères de couverture des politiques d'adaptation (RQ2). Nous évaluons la capacité d'une approche aléatoire à accomplir ces critères de couverture proposés.

Dans un second temps, nous avons conçu un ensemble de mutants afin d'évaluer les capacités des cas de tests à détecter des fautes en détectant ces mutants. Ces derniers sont des variants du système original dans lequel une faute a été introduite. La faute en question doit être réaliste et représenter une faute d'implémentation (ou un mauvais choix) possible que le programmeur pourrait faire pendant le développement du système. Les mutants sont produits à la main et les mutations considérées se basent sur un modèle simple qui consiste à changer, dans le code du logiciel, les conditions utilisées pour décider quand une reconfiguration doit avoir lieu ou non. Les mutations vont donc renforcer ou affaiblir les conditions où l'implémentation permet de déclencher une reconfiguration.

Les cas de tests sont lancés sur le système contenant les mutants et nous établissons deux verdicts (basés sur les propriétés et les politiques d'adaptation) pour décider si chaque test a réussi ou échoué. Quand un mutant est tué, la propriété ou la règle responsable de la détection de la faute est notée. Cette approche a pour but d'évaluer les sections sur la validité du système vis-à-vis des propriétés FTPL 6.1 et des politiques d'adaptation 6.2 liées à la pertinence des critères de couverture en terme de capacité de détection de fautes. Nous nous attendons à ce qu'un ou plusieurs mutants soient tués par chaque verdict de test proposé.

Dans un troisième et dernier temps, comme la conception d'un modèle d'usage est une tâche non-triviale susceptible de créer des erreurs, et particulièrement au moment de définir les probabilités du système, nous étudions l'impact des différents choix de probabilités dans ce processus. Nous considérons donc, pour chaque type de composant, deux modèles d'usage supplémentaires : un modèle dans lequel les probabilités de quiescences sont augmentées et un autre modèle dans lequel les probabilités de quiescences sont réduites par rapport au modèle original. Nous comparons quels impacts ces changements ont sur la taille des suites de tests, la longueur des cas de tests et nous évaluons si les capacités de détection sont impactées en lançant ces nouveaux tests avec les mutants décris précédemment.

7.2.2/ RÉSULTATS

Génération de tests aléatoire. Afin de lisser les résultats de notre approche aléatoire, nous avons généré 15 cas de tests de 4000 pas. Tous les cas de tests ont le même état initial. Les résultats sont visibles dans le tableau 7.1 qui affiche la moyenne, le nombre minimum et maximum de pas nécessaires pour atteindre le taux de couverture souhaité (c'est-à-dire 100% pour les propriétés et règles).

	#steps (avg)	#steps (min)	#steps (max)
100% couverture des propriétés	1486	307	3475
100% couverture des règles	1269	322	2991

TABLE 7.1 – Nombre nécessaire de pas pour atteindre un taux de couverture de 100%

Nous pouvons voir que, pour une longueur supérieure à 1500 pas, un taux de couverture de 100% des règles et propriétés peut être atteint. Le nombre de pas de test est raisonnable, car la génération de test demande peu de temps (chaque génération de test sur le simulateur demande moins de 30 secondes).

Dans la première expérimentation, nous avons utilisé un seul long test. Afin d'établir s'il est préférable d'utiliser plusieurs petits tests qu'un seul gros test, nous avons également fait de même avec plusieurs tests de longueur inférieure. Les résultats de ces générations de tests sont visibles dans le tableau 7.2. Nous présentons pour chaque suite de test, ses caractéristiques en termes de nombre de tests (colonne #tests) et de longueur de tests (colonne |test|). Nous faisons ensuite un rapport sur la couverture moyenne des propriétés et règles.

Les résultats montrent qu'il faut entre 322 et 2991 pas de test pour déclencher toutes les règles de reconfiguration.
Utiliser plusieurs petits tests peut être intéressant pour agréger les résultats de plusieurs simulations dont le temps de calcul est long. Néanmoins, les règles de sortie de véhicule d'un convoi requiert un nombre de pas suffisant pour que leur niveau de batterie ou que leur distance de leur destination soient faibles pour être déclenchées. C'est pourquoi la taille de chaque cas de test doit rester supérieure à un certain seuil qui est 400 dans notre cas. Ces résultats montrent également que l'utilisation des politiques d'adaptation comme critère de couverture est complémentaire au critère de couverture des propriétés temporelles et répond à la RQ2. L'influence de la configuration initiale sur les taux de couverture sera évaluée en section 7.3. À présent, nous définissons notre approche de détection d'erreurs à partir de mutants.

#test	|test|	% Prop.	% Rules
5	2000	100%	100%
10	1000	100%	100%
10	400	100%	100%
20	200	100%	87.5%
10	200	98.5%	87.5%
10	100	96.7%	62.5%
500	100	100%	87.5%

TABLE 7.2 – Le taux de couverture agrégé obtenu en fonction de chaque suite de test

	M1	M2	M3	M4	M5	M6	M7	M8	M9
$\varphi 1$									#108
$\varphi 2$						#10			
$\varphi 3$	#268	#114	#23				#429		
$\varphi 4$				#73	#53				
$\varphi 5$								#338	

TABLE 7.3 – Détection des mutants

Détection de mutants. Nous avons conçu un ensemble de mutants pour : (i) renforcer les conditions dans le code pour déclencher les reconfigurations (mutants M1, M4) ce qui mène à des reconfigurations qui ne sont pas déclenchées alors qu'elles le devraient, (ii) un affaiblissement des conditions de reconfiguration (mutants M12, M13) qui mènent à des déclenchements de reconfigurations inattendus, (iii) une implémentation différente dans la façon de prendre en charge les évènements (mutants M2, M3), (iv) un changement dans les priorités des reconfigurations (mutants M5, M7, M10, M11), et (v) des erreurs fonctionnelles dans l'implémentation (mutant M6, M8, M9).

Nous avons testé ces mutants sur un unique cas de test de 4000 pas avec les propriétés FTPL définies en section 3.4. Ce qui répond à la RQ3 sur le fait que les modèles probabilistes sont appropriés pour générer automatiquement des cas de tests capables de satisfaire les critères de couverture avec un nombre de pas suffisants. Le même cas de test à été rejoué sur tous les mutants. L'exécution stoppe dès qu'une violation de propriété est détectée. Les résultats sont visibles dans le tableau 7.3, pour les mutants M1 à M9 et indiquent, pour chaque mutant détecté quelle propriété à été violée et à quel pas de test. Les mutants M1-M9 sont détectés par le verdict donné en définition 38 qui s'appuie sur la vérification de propriétés temporelles pour détecter une violation de propriété. Les mutants M12 et M13 sont détectés en utilisant le verdict de test donné dans la définition 43 qui détecte des reconfigurations inattendues (respectivement aux pas de test 398 et 3026). Enfin, les mutants M10 et M11 ne sont pas tués par ces verdicts, mais ils sont détectés par l'analyse des fréquences présentée dans la définition 48.

En effet, la modification introduite dans la troisième catégorie de mutant a changé le comportement du système vis-à-vis de l'implémentation initiale sans que l'effet soit observable pour autant. Toutefois, nous avons observé dans ces cas que la fréquence de déclenchement des règles par rapport à l'implémentation originale est étrange par rapport à l'utilité de la règle décrite dans la politique d'adaptation. Pour le mutant M10 une règle avec une utilité faible est systématiquement déclenchée. Inversement, avec le mutant M11, une règle d'utilité moyenne n'est jamais déclenchée sur un cas de test de 4000 pas, qui est suffisamment long.

Un approfondissement sur l'analyse des fréquences de déclenchement est proposé en section 7.4 permettant de déceler ces mutants.

Implementation	Property cov.% (avg)	Rules cov. % (avg)
Reference	100%	100%
Variant 1	100%	92.5%
Variant 2	100%	86.1%
Variant 3	100%	100%

TABLE 7.4 – Résultats de la génération de tests pour les différentes variantes

La conception des mutants nous a permis d'évaluer si le système est capable de les détecter. Il en résulte que tous nos mutant impliquant les propriétés du système ont été détectés tant que les critères de couverture ont été respectés. Ce qui répond à la RQ1 sur l'intérêt des critères de couverture pour détecter des erreurs.

Nous nous intéressons maintenant à l'évaluation des probabilités utilisées dans nos modèles d'usage.

Choix des probabilités des transitions des modèles d'usage. La tâche de conception d'un modèle approprié est fastidieuse et sujette aux erreurs, spécialement quand il faut définir les probabilités sur les différents évènements. Afin d'évaluer les risques de la définition de probabilités incorrectes sur le modèle d'usage, nous avons conçu 3 variantes du modèle d'usage, qui correspondent à une modification pour les évènements externes à leur probabilité d'apparition.

Dans la première variante, nous avons réduit la probabilité des évènements *forceQuitPlatoon*, *createVehicle* et *requestJoin*, et augmenté en conséquence la probabilité de l'évènement δ (quiescence). Dans la deuxième variante, nous avons augmenté la probabilité des évènements externes *forceQuitPlatoon*, *createVehicle* et *requestJoin*, et nous avons réduit la probabilité de l'évènement δ. Dans la troisième et dernière variante, nous avons augmenté uniquement la probabilité des évènements *createVehicle* et *requestJoin* et réduit la probabilité de l'évènement *forceQuitPlatoon*. Ces variantes illustrent les différents possibles, notamment au moment de définir l'équilibre entre l'apparition des quiescences et évènements externes.

Les résultats obtenus sont visibles dans le tableau 7.4. Nous avons lancé le processus de génération de test sur un unique test de 4000 pas. Nous avons mesuré les taux de couverture de chaque variante vis-à-vis des propriétés et règles pour les comparer au taux obtenu du modèle d'usage de référence.

Nous pouvons voir que l'augmentation de la probabilité d'apparition pour certaines opérations critiques telles que *forceQuitPlatoon*, peuvent mener à la réduction du déclenchement de certaines règles. En l'occurrence, le système force un véhicule à quitter le convoi plus fréquemment, ce qui empêche certaines configurations, d'être atteintes et donc des reconfigurations de se produire.

Il est donc facile de comprendre pourquoi il est primordial, au moment de la conception d'un modèle d'usage, de bien choisir les probabilités associées aux évènements externes et particulièrement aux évènements qui font désactiver d'autres composants.

7.2.3/ DISCUSSION

Nous avons présenté dans cette section, une méthode d'analyse de la pertinence des critères de couverture pour la détection d'erreurs.
Un premier obstacle à la validité de l'expérimentation est lié à la définition des mutants, il est en effet facile de penser que ces mutants ont été conçus pour être détectés par notre approche. Toutefois, ces mutants ont été conçus de manière systématique (en réduisant ou augmentant des valeurs des conditions de décision), et c'est pour cela que nous pensons que cet obstacle est limité.

Un deuxième obstacle à la validité pourrait être lié à la définition du modèle d'usage. En effet, un modèle d'usage peut empêcher ou favoriser de manière abusive des déclenchements de reconfigurations et influencer les taux de couvertures obtenus. Afin d'atténuer cela, nous avons effectué des expérimentations sur différents modèles qui présentent des probabilités différentes sur leurs transitions pour valider nos choix de probabilités.

Maintenant que les expérimentations ont été présentées pour les critères de couverture, nous passons à l'évaluation de l'algorithme de génération de configurations initiales.

7.3/ ÉVALUATION DE LA PERTINENCE DES CONFIGURATIONS INITIALES GÉNÉRÉES AUTOMATIQUEMENT

Cette expérimentation a pour but de répondre à trois questions de recherche en s'appuyant sur notre exemple de convoi de véhicules autonomes.

RQ4 Dans quelle mesure est-il possible d'automatiser un tel processus ?

RQ5 Dans quelle mesure est-il possible d'évaluer le système avec un tel processus ?

RQ6 Dans quelle mesure la suppression de configurations équivalentes est efficace pour réduire le nombre de configurations générées.

Cette section décrit les expérimentations faites pour valider la contribution présentée en section 5.2 sur la génération automatique de configurations initiales qui repose sur la conception du modèle à composants défini dans le chapitre 4. Cette expérimentation utilise l'algorithme 1 et l'algorithme 2 pour la génération automatique d'un ensemble de configurations initiales et de son échantillonnage.

Nous décrivons ici le protocole d'expérimentations. Une fois que l'ensemble de configurations initiales brutes $Inst$ de cardinalité 1200 est généré en appliquant l'algorithme 1 sans la suppression de solutions symétriquement équivalentes, la matrice $Scores$ des différences de scores est calculée. Ensuite, les 10 configurations les plus proches (différences faibles) et les 10 configurations les plus éloignées (différences élevées) sont obtenues en utilisant l'algorithme 2 sur la matrice $Score$. Dans ces expérimentations, les évènements de tests sont générés à chaque simulation. Comme le générateur de test procède à une marche aléatoire (Markovian walk) sur la sélection des modèles d'usage, cela signifie que pour la même configuration initiale, les cas de tests diffèrent au niveau de leurs ensembles de tests. Afin d'avoir une meilleure confiance dans les résultats, les expérimentations sont rejouées 170 fois pour chaque ensemble de configurations une par une (soit $2 \times 10 \times 170$ simulations lancées). Cela permet d'observer les traces produites avec les reconfigurations déclenchées et de mesurer les taux de couverture. Toute cette procédure de génération de configurations initiales et de séquences de tests est effectuée de manière automatique permettant de répondre à la RQ4.

Les expérimentations consistent à partir d'une configuration initiale et à laisser le générateur de test gérer les modèles d'usage des composants afin d'envoyer des évènements à une fréquence donnée au système sous test. Lors des expérimentations constituées de 3000 pas, des reconfigurations sont déclenchées (et des traces produites) faisant évoluer l'architecture du système. Le système et ses artefacts ont également été modifiés sur plusieurs aspects par rapport au protocole de la section 7.2.1. Dans un premier temps, nous avons ajouté une règle de reconfiguration. Cette règle de reconfiguration est issue de l'opération de reconfiguration pour que le véhicule puisse quitter le convoi s'il approche de sa destination. Il existait déjà une règle avec une utilité haute ($high$) et nous avons ajouté une règle avec une utilité faible (low). Nous avons également des configurations initiales différentes et donc un taux de couverture différent. Pour finir, le modèle d'usage a également changé. En effet, en section 7.2.1, les composants étaient créés en ligne avec le générateur de test qui utilisait les modèles d'usage alors que dans ce protocole, les composants sont créés dans la configuration initiale. Cela a eu pour effet de changer les probabilités de certains évènements contrôlables et c'est pourquoi nous avons obtenu des résultats différents pour les taux de couverture.

Les résultats. Le taux de couverture est agrégé séparément des propriétés et des règles d'adaptation en appliquant les critères de couverture.
Soit un ensemble de configurations initiales, pour chaque expérimentation, le taux de couverture des règles est obtenu en calculant le ratio entre le nombre de règles d'adaptation qui sont couvertes par au moins un cas de test et le nombre de règles prises en compte dans le système. Chaque cas de test est constitué au départ d'une des configurations initiales de l'ensemble.

Un extrait des résultats des expérimentations est reporté dans le tableau 7.5 [1]. Dans ce tableau, les lignes secondaires correspondent aux taux de couverture atteints pour les règles et les propriétés classées en fonction de l'ensemble de configurations choisi dans les lignes principales, à savoir entre l'ensemble avec des petites valeurs de $difScore$ et l'ensemble avec des grandes valeurs de $difScore$. Les colonnes représentent le numéro de la simulation effectuée (de 1 à 170) dont un extrait de 9 valeurs est visible dans la table. Par exemple, pour la simulation dans la colonne $n+1$, la première ligne principale (Small dif. score), indique 94% de couverture pour les propriétés et 75% pour les règles, la seconde ligne principale (Big dif. score) quant à elle, indique 100% de couverture pour les propriétés et 75% pour les règles d'adaptation. La colonne nommée Av. (contraction de average) indique le taux de couverture moyen de sa ligne secondaire. Par exemple, dans la première ligne principale (Small dif. score), la première ligne secondaire indique un taux de couverture moyen de 86.5% pour les propriété sur les 170 simulations effectuées. Lorsqu'il est indiqué un taux de couverture de 0% pour les règles d'adaptation, cela signifie qu'aucune règle n'a été déclenchée pendant les 3000 pas de la simulation incluant en moyenne 300 évènements externes envoyés au système [2]. Inversement, un score de 100% indique que toutes les règles d'adaptation ont été déclenchées. Nous pouvons donc en conclure que les critères de couverture sont satisfaits avec cette approche permettant de garantir les mécanismes d'évaluation du système présentés en section 7.2 ce qui répond à la RQ5. La colonne nommée M.fr. (contraction de max frequency) indique la valeur la plus fréquente parmi les 170 simulations effectuées pour chaque ensemble de configurations. Lorsqu'il est indiqué 75, cela signifie que 75% est le taux de couverture le plus fréquent.

À noter que la valeur la plus fréquente de taux de couverture pour les règles d'adaptation est 75% ce qui est dû aux règles de reconfiguration $QuitDistance$ qui ne sont pas déclenchées à cause du paramètre $distance$. Pour ces règles, les simulations sur 3000 pas peuvent être trop courtes pour que la valeur du paramètre $distance$ puisse suffisamment décroître empêchant ainsi de satisfaire la garde des règles concernées.

Run number			n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8		Av.	M.fr.
Small dif. score	Prop.Cov.(%)	...	56	94	17	94	56	61	94	100	94	...	86.5	94
	Rule Cov.(%)	...	0	75	0	75	0	13	75	75	75	...	58.5	75
Big dif. score	Prop.Cov.(%)	...	100	100	100	100	100	100	94	100	100	...	98.1	100
	Rule Cov.(%)	...	94	75	75	75	75	100	75	75	75	...	80.2	75

TABLE 7.5 – Extrait des résultats expérimentaux et taux de couverture

Suppression des équivalences symétriques. Afin d'évaluer l'effet de la suppression des équivalences symétriques, nous avons fait des expérimentations qui consistent à

1. Le tableau complet est visible à cette adresse : https://fdadeau.github.io/CSConfigGen/table.html
2. Durant une simulation, à chaque tic d'horloge, il y a 10% de chances que le pas de test corresponde à un évènement d'un modèle d'usage envoyé au système alors que les quiescences (δ) se produisent 90% du temps.

compter le nombre de configurations qui sont générées, avec et sans la détection de symétrie et de comparer les résultats.

Pour l'exemple VANet, nous avons conçu 5 paramétrages qui sont différents pour le nombre minimal et maximal de composants de chaque type à générer. L'invariant spécifie que les véhicules qui ne sont pas dans un convoi ne sont pas connectés entre eux. Les paramétrages sont les suivants :
Paramétrage#1 : 1 Route, 1 à 5 Véhicules, 0 à 2 Convois, 0 à 1 Station.
Paramétrage#2 : 1 Route, 5 Véhicules, 0 à 2 Convois, 0 à 1 Station.
Paramétrage#3 : 1 Route, 6 Véhicules, 0 à 2 Convois, 0 à 1 Station.
Paramétrage#4 : 1 Route, 1 à 5 Véhicules, 1 Convois, 0 à 1 Station.
Paramétrage#5 : 1 Route, 1 à 5 Véhicules, 0 à 2 Convois, 1 Station.

Setup	#1	#2	#3	#4	#5
P, D, B, I	62	26	39	30	44
P, D, B, $\overline{I}$	181	93	166	54	149
$\overline{P}$, D, B, I	325	244	1098	158	222
$\overline{P}$, D, B, $\overline{I}$	640	482	2398	378	491
$\overline{P}$, $\overline{D}$, B, I	1083	897	5270	410	700
$\overline{P}$, $\overline{D}$, $\overline{B}$, I	2249	1953	17495	1294	1466
$\overline{P}$, $\overline{D}$, $\overline{B}$, $\overline{I}$	22971	21213	337625	4174	21218

TABLE 7.6 – Nombre de configurations pour chaque paramétrage

En activant ou désactivant (les symétries désactivées sont notées avec une ligne au-dessus de leur symbole correspondant) certaines éliminations de symétries (parenté P, délégation D ou liens B) et le filtrage d'invariant (suppression des solutions erronées par construction I), nous obtenons les résultats visibles dans le tableau 7.6.

Les paramétrages #1, #3 et #5 ont pris environ 25 secondes sur un ordinateur ordinateur portable standard (Dual-core i5 1.6GHz avec 8Go de RAM) pour générer 21.000–23.000 configurations. En raison de l'explosion combinatoire du nombre de configurations, il faut environ 20 minutes pour générer les 337.625 configurations du paramétrage 3, ce qui a été réduit à 16 minutes en activant les réductions de symétrie. Toutes les réductions de symétrie sont pertinentes pour gérer l'explosion combinatoire. En effet, lorsque toutes les éliminations de symétries sont actives, entre 26 et 62 configurations est généré même pour des paramétrages hautement combinatoires.

Les expérimentations montrent également qu'un très grand nombre de configurations non pertinentes (entre 21213 et 337625) peuvent être générées en se basant uniquement sur la description du modèle à composant (sans considérer l'invariant). Ces résultats montrent que la suppression de configurations équivalentes est efficace pour réduire le nombre de configurations générées et répondent à la RQ6. Nous précisons que ces

configurations symétriques ont un score de 0 de différence entre elles, et donc, une seule d'entre elle doit être gardée dans le processus de sélection. En conséquence, retirer ces configurations le plus tôt possible évite des calculs inutiles d'être faits.

Les résultats d'expérimentations obtenus permettent de valider l'utilisation de configurations initiales générées automatiquement ainsi que leur échantillonnage. En effet, les résultats montrent que les ensembles de configurations générés ayant une différence significative permettent d'atteindre un taux de couverture plus élevé à la fois sur les politiques d'adaptation ($guards$ et $triggers$ couvertes pour chaque règle) et sur les propriétés temporelles que les ensembles ayant des différences faibles.

Maintenant que notre algorithme de génération de configurations initiales est validé expérimentalement, nous passons à l'évaluation de la cohérence entre le comportement du système spécifié ou attendu et l'implémentation des politiques d'adaptation.

7.4/ EXPÉRIMENTATIONS SUR LES FRÉQUENCES DE DÉCLENCHEMENT DES RÈGLES DE RECONFIGURATION

Dans cette section, nous souhaitons évaluer que le comportement du système est cohérent avec l'implémentation des politiques d'adaptation en utilisant les mesures de fréquences décrites en section 6.2.3.

7.4.1/ PROTOCOLE D'EXPÉRIMENTATION

Cette expérimentation a pour but de répondre à deux questions de recherche en s'appuyant sur notre exemple de convoi de véhicules autonomes.

RQ7. Dans quelle mesure l'analyse de fréquences permet d'évaluer les valeurs floues d'utilité des règles ?

RQ8. Dans quelle mesure les résultats suspicieux peuvent être détectés ?

Pour ces expérimentations, nous avons fait en sorte que les véhicules soient tous créés à l'état initial et soient retirés une fois qu'ils atteignent leur destination. Cette vision est différente des précédentes expérimentations où les véhicules pouvaient apparaître dynamiquement sur la route. À l'état initial, le nombre de véhicules sur la route oscille entre 100 et 250 et sont présents entre 4 et 25 convois avec une taille maximale de 8 véhicules. Ce mécanisme assure de maîtriser le nombre de véhicules sur la route et permet d'augmenter les chances d'atteindre des configurations qui peuvent déclencher des reconfigurations, par exemple, en augmentant le nombre de véhicules ou en modifiant la taille maximale des convois.

Notre processus de génération de test est utilisé pour produire et lancer des cas de test pour stimuler le système qui va se reconfigurer et produire des traces de reconfiguration. En analysant les traces de reconfiguration, nous mesurons les fréquences de déclenchement des règles conformément aux formules fournies dans la section 6.2.3. Nous comparons ensuite ces fréquences avec les valeurs floues spécifiées dans les politiques d'adaptation. Dans ces expérimentations, l'analyse est faite sur des cas de tests de 100.000 pas. Nous avons choisi de ne pas augmenter davantage la taille des cas de tests, car une simulation est 100.000 prend environ une demi heure et les résultats obtenus étaient suffisamment stables pour ne pas chercher à alourdir cette partie de l'expérimentation. Pour s'assurer que les suites de tests activent au moins une fois chaque reconfiguration, nous utilisons nos critères de couverture définis en sections 6.1.1 et 6.2.3.

Les résultats En mesurant les fréquences de déclenchement à chaque pas, nous pouvons dessiner le graphique montrant l'évolution des fréquences au fur et à mesure de l'exécution du système. Ces graphiques peuvent être utilisés pour comparer les fréquences effectives par rapport aux utilités des politiques d'adaptation. À la fin de l'exécution, il est possible de comparer les fréquences entre elles pour vérifier leurs taux de déclenchements. Comme cela est montré dans la figure 7.3, les fréquences se stabilisent avec l'augmentation du nombre de pas de tests, ce qui montre que le nombre de tests que nous générons rend possible une comparaison fiable de ces fréquences définies en section 6.2.2).

L'utilisation de graphiques permet d'analyser visuellement les fréquences et de détecter facilement les comportements suspicieux. Par exemple, il pourrait s'agir d'une reconfiguration qui serait très souvent sélectionnée malgré une utilité faible ou une instabilité dans la fréquence obtenue.

Afin d'évaluer les capacités de l'approche pour détecter des potentiels problèmes dans l'implémentation d'une politique d'adaptation, plusieurs implémentations (E1-E6) ont été simulées, qui diffèrent dans leur sélection des reconfigurations. La première implémentation E1 sélectionne les reconfigurations par rapport à leur niveau de priorité. De plus, si une reconfiguration éligible n'a pas été déclenchée, sa priorité est augmentée pour les prochains pas de test. Les résultats de l'exécution de E1 peuvent être trouvés dans la partie gauche de la figure 7.3 et dans le tableau 7.7. En examinant le graphique, il est possible de s'apercevoir que la courbe de reconfiguration R5 est sous les courbes des reconfigurations R7 et R8. Ces courbes montrent une incohérence car R5 est de priorité moyenne et devrait avoir une fréquence de déclenchement supérieure à R7 et R8 sur les cas de tests longs.

Pour aller plus loin dans les expérimentations et répondre à la RQ7, nous avons simulé d'autres choix d'implémentation à partir des niveaux de priorités ajustés grâce à une

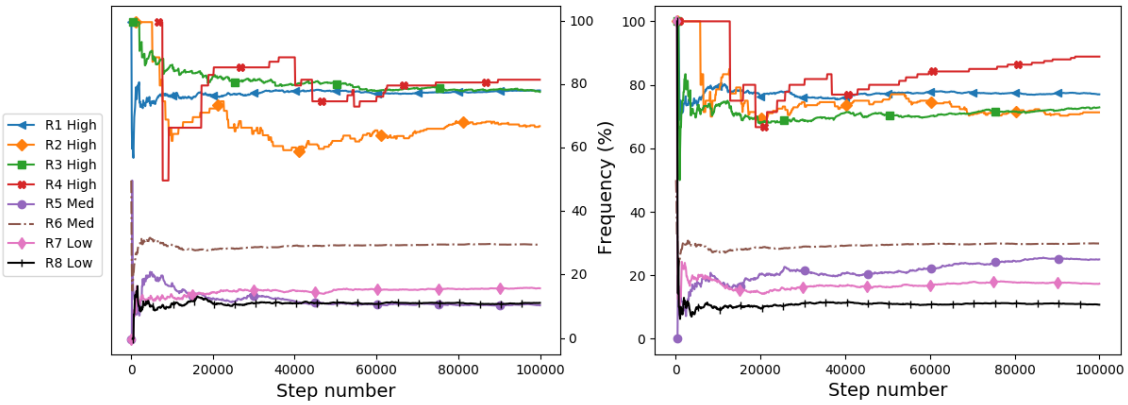

FIGURE 7.3 – Analyse des fréquences de l'implémentation (à gauche) E1 comparé à l'implémentation E2 (à droite)

valeur numérique. Ainsi plus la valeur est élevée et plus la priorité est élevée et la règle a de chances d'être déclenchée. Les résultats sont visibles dans le tableau 7.7. Dans les simulations pour E3, nous avons fait une supposition d'équité, par exemple, si deux reconfigurations avec la même priorité sont éligibles, la reconfiguration qui n'a pas été sélectionnée le sera la prochaine fois que ce cas de figure se présente. Dans la simulation avec E4, les reconfigurations sont choisies sur la base de leur niveau d'utilité comme dans [130]. L'implémentation E5 sélectionne une règle avec une utilité faible en priorité dans 20% des cas. Pour finir, l'implémentation E6 sélectionne la première reconfiguration de la politique d'adaptation qui peut être déclenchée comme dans [25].

Les résultats des simulations avec E2 et E3 montrent que les choix d'implémentations concordent avec les utilités spécifiées. L'implémentation E4 montre des résultats acceptables, mais la reconfiguration R4 est déclenchée trop souvent en comparaison des autres reconfigurations avec la même utilité. L'implémentation E5 n'est pas cohérente, la reconfiguration R5 de priorité moyenne est sous les reconfigurations R7 et R8 de faible priorité. Pour finir, la simulation avec E6 montre plusieurs incohérences dans les fréquences. Ces expérimentations montrent l'intérêt d'une approche permettant à l'utilisateur d'identifier à la fois les comportements suspicieux qui seraient à explorer davantage et concevoir une implémentation qui est en accord avec la politique d'adaptation.

7.4.2/ DISCUSSION

Nous avons présenté dans cette section une méthode d'analyse de la cohérence du comportement du système avec les politiques d'adaptation en analysant les fréquences de déclenchement des règles de reconfigurations.
Dans ces expérimentations, il existe un obstacle à la validité qui est lié à l'analyse de la fréquence, si le nombre d'occurrences de déclenchement de règles est trop faible, la

	R1(%)	R2(%)	R3(%)	R4(%)	R5(%)	R6(%)	R7(%)	R8(%)
	high	high	high	high	med.	med.	low	low
E1	78.4	67.2	78	82.8	10.6	29.7	16	11.5
E2	77	71.3	72.9	87.9	25.5	30	17.3	10.9
E3	68.7	64.7	70.8	76	23.1	31.4	14.9	9.4
E4	79.2	72.3	74.1	92.9	25	30.2	17.2	11.4
E5	64.4	66	56.6	68.2	17	29.7	19.1	24.4
E6	22.9	14.9	0.7	0	4.2	26.2	10.4	23.3

TABLE 7.7 – Moyenne des résultats obtenus pour des simulations de 100.000 pas

classification de règles pourrait être imprécise. Toutefois, nous pouvons remarquer que durant nos expérimentations, chaque reconfiguration a été déclenchée entre plusieurs dizaines de fois et plusieurs milliers de fois sur un cas de test de 100.000 pas, ce qui assure un nombre significatif d'éligibilités de chaque règle, assurant ainsi la pertinence des fréquences calculées.

7.5/ CONCLUSION

Dans ce chapitre, nous avons défini un protocole expérimental pour comparer notre approche sur les systèmes adaptatifs et les résultats obtenus de notre logiciel VANet. Ce protocole est divisé en trois parties. Dans la première partie, nous avons évalué la pertinence des critères de couverture et du modèle d'usage ; dans la deuxième partie, nous avons évalué notre génération de configurations initiales et ; dans la troisième partie, nous avons évalué la cohérence entre le comportement du système et l'implémentation des politiques d'adaptation. Nous avons séparé ce protocole en trois parties pour nous focaliser sur chaque élément de notre approche. Ces parties peuvent néanmoins fonctionner ensemble, et cela, à notamment été le cas pour les parties 2 et 3 où la vérification des propriétés temporelles était toujours active.

8

CONCLUSIONS ET PERSPECTIVES

Nous présentons dans ce chapitre les conclusions sur le travail effectué au cours de cette thèse et les pistes de recherche qui en découlent. Les travaux de recherche effectués et présentés dans cette thèse ont pour but d'expliciter nos contributions sur la validation de systèmes à composants adaptatifs par génération de tests.

8.1/ SYNTHÈSE ET BILAN

Nous commençons par dresser le bilan de nos travaux en répondant aux questions de recherche posées en section 1.2.1 :

RQ 1. Quels objectifs de test définir pour valider les systèmes adaptatifs ?

Une fois la suite de tests en place, nous avons défini dans le chapitre 6 des critères permettant d'évaluer la qualité de cette suite de tests.

Pour cela, nous avons tout d'abord présenté un critère de couverture sur les propriétés FTPL du système similaire aux travaux de [22].

Ensuite, nous avons défini un critère de couverture sur les règles de reconfiguration présentes dans la politique d'adaptation.

RQ 2. Comment construire les tests ?

Nous avons dans le chapitre 5 détaillé notre approche pour la construction de cas de tests.

Dans un premier temps, nous avons présenté notre méthode de génération automatique de configurations initiales. La génération de configurations initiales s'appuie sur le modèle architectural défini dans le chapitre 4. Ce modèle architectural se base sur les travaux de [97] et pour lequel nous avons apporté des contributions en termes d'instanciation du modèle.

En effet, nous avons modifié la description de configuration du modèle en partant de $\langle Elem, Rel \rangle$ pour arriver à $\langle Elem, Rel, Inst \rangle$ afin de permettre les instances de composants. Ce changement a demandé d'harmoniser le modèle autour de cette nouvelle définition de configuration, notamment au niveau des contraintes architecturales.

La méthode de génération de configurations initiales utilise un algorithme qui se base sur les contraintes architecturales définies par le modèle à composants défini dans le chapitre 4.
Dans un second temps, nous avons présenté notre méthode de génération d'évènements de tests. Ces évènements de tests sont constitués par des évènements contrôlables qui sont décrits par les modèles d'usages. Un algorithme de génération de tests sélectionne les évènements de tests à envoyer au système.
Ces deux méthodes nous permettent de générer plusieurs cas de test qui forment une suite de tests utilisée comme base dans nos simulations sur notre système adaptatif.
RQ 3. Comment établir le verdict de test ?
L'établissement d'un verdict de test permet de décider si le système se comporte correctement par rapport à sa spécification.
Pour cela, nous avons mis en place plusieurs mécanismes d'évaluation.
En effet, en plus des propriétés temporelles proposées par [96], nous avons défini un mécanisme d'évaluation de cohérence entre une politique d'adaptation et le comportement du système. De plus, nous avons proposé un mécanisme de mesure de fréquence de déclenchement de règles de reconfiguration permettant de comparer avec l'utilité de ces règles définies par la politique d'adaptation et soulever de potentielles erreurs de spécification de l'implémentation.

Expérimentations Nous avons conduit une expérimentation sur notre modèle fil rouge VANet implémenté en JAVA afin d'évaluer la pertinence de notre approche sur trois aspects : la mesure de couverture des propriétés et politiques d'adaptation, l'évaluation de la pertinence des configurations initiales et la détection d'anomalies dans les fréquences de déclenchement des règles.
Dans un premier temps, nous avons généré de manière automatique des configurations initiales servant d'étape de départ aux simulations du système. Les simulations se sont déroulées en envoyant des évènements externes au système à l'aide d'un algorithme utilisant une machine à états.
L'évaluation des configurations initiales générées nous a permis de valider notre méthode de génération automatique.
Le système sous test génère des logs à l'exécution pour permettre d'effectuer son évaluation.
Dans un second temps, afin d'évaluer le comportement du système, nous avons implémenté le fait que le système sous test génère des logs à l'exécution. En effet, nous avons implémenté des machines à états pour pouvoir évaluer le respect des propriétés temporelles. Ces machines à états ont des transitions correspondant à des (re)configurations particulières que le système ajoute dans les logs au fur et à mesure de son exécution. La mesure de couverture que nous avons appliquée sur ces machines à états nous a permis

de nous assurer du comportement du système vis-à-vis des propriétés temporelles.
Pour finir, les règles de reconfiguration ont été implémentées pour guider le système et les logs ce qui nous a permis d'établir la conformité entre la définition de ces règles et leur application par le système. En effet, l'analyse des logs nous a permis d'établir la mesure de couverture de ces règles. De plus, la détection d'anomalies sur les fréquences de déclenchements nous a permis de remonter un problème dans l'implémentation des utilités dans les règles de reconfiguration. Une fois cette erreur corrigée, cette détection nous donne une bonne confiance dans la prise en compte des utilités par le système et le déclenchement des règles associées.

8.2/ PERSPECTIVES

Cette section discute des perspectives envisagées qui sont organisées ci-dessous en quatre parties. Tout d'abord, nous présentons celles liées aux critères de couverture pour les modèles d'usage. Dans un second temps, nous indiquons les perspectives sur la génération de configurations initiales et pour l'initialisation des paramètres. Enfin, nous mentionnons une idée pour la mesure de conformité des propriétés extra-fonctionnelles.

Critères de couverture des modèles d'usage Les modèles d'usage (cf définition 32) sont utilisés pour décrire les évènements externes que le système est amené à rencontrer.
Une perspective de recherche consiste à utiliser des critères de couverture sur ces modèles d'usage. Cela permettrait de déceler si certains évènements contrôlables sont trop peu activés ou jamais activés. En fonction des résultats obtenus il serait possible d'assister la génération aléatoire en forçant certains évènements à avoir lieu. Il serait par exemple possible d'éviter les goulots d'étranglement et ainsi optimiser la génération des cas de tests en ciblant précisément quels états du modèle d'usage atteindre.
Pour ce faire, il est possible d'utiliser les mêmes critères de couverture que pour les propriétés FTPL (cf section 6.1.1).
Nous pensons qu'il serait intéressant de comparer l'utilisation d'un critère de couverture des modèles d'usages avec d'autres techniques telles que les travaux de [51] utilisant des algorithmes génétiques. Cette comparaison peut se faire sur la capacité de détection d'erreurs, la complexité de mise en place des méthodes ainsi que les ressources nécessaires pour faire fonctionner chaque méthode.

Génération de configurations initiales Dans nos expérimentations, au moment de la génération de configurations initiales, nos différents paramétrages (cf section 7.3) ont un nombre de composants limité. En prenant un paramétrage contenant un nombre élevé

de composants à générer, notre technique de génération de configurations initiales montrerait ses limites en matière de temps de génération.
Deux solutions sont possibles. La première solution consisterait à utiliser un meta-modèle pour la conception du modèle afin de réduire le temps de génération en utilisant un algorithme de génération aléatoire. Même si l'utilisation d'un tel algorithme donnerait de bonnes chances d'avoir une configuration initiale cohérente produite à la fin de la génération il ne serait cependant pas possible d'en être certain.
La deuxième solution serait de coupler à l'élimination de solutions symétriquement équivalentes une technique d'élimination des solutions incohérentes vis-à-vis des contraintes de cohérence architecturales. Nous pensons qu'en retirant les solutions incohérentes au plus tôt cela réduirait la complexité de l'algorithme 1.
Il serait donc intéressant de développer ces deux techniques sur des paramétrages de configurations complexes et voir laquelle donne les meilleurs résultats.

Initialisation des paramètres Notre solution d'initialisation des paramètres par tirage probabiliste selon la Beta-distribution sur un unique domaine fonctionne dans nos expérimentations, cependant elle est améliorable. En effet, des contraintes de cohérence peuvent rendre les domaines discontinus. Une amélioration possible consisterait à utiliser un solveur numérique permettant de prendre en compte les contraintes des domaines et de générer les valeurs en fonction de ces domaines réévalués et ainsi éviter une stratégie de génération contenant des rejets. Actuellement, si une discontinuité de domaine est décelée, afin de réduire le nombre de rejets lors de la génération, il est possible d'affiner à la main le domaine en une union de domaines valides et la transformation du domaine [0,1] s'effectue proportionnellement vers l'union des domaines valides.

Mesure de conformité des propriétés extra-fonctionnelles Dans nos travaux, nous avons proposé une technique basée sur des tests qui peut être utilisée pour valider que la politique d'adaptation a été fidèlement mise en œuvre par le système [32]. Bien que cette technique valide également que le système ne viole aucune propriété fonctionnelle, elle n'a fourni aucune évaluation concernant les propriétés extra-fonctionnelles (PEF) qui sont visées par la politique d'adaptation.
Intuitivement, les propriétés extra-fonctionnelles peuvent être vues comme des critères basés sur la configuration du système qu'il est possible d'extraire sous forme de valeurs numériques. Ces critères peuvent être variés, par exemple le temps d'exécution, la consommation d'énergie ou être plus spécifiques au système étudié. Ainsi, chaque propriété non-fonctionnelle peut être définie par rapport à un critère et chaque critère peut être associé à une évaluation (si la propriété extra-fonctionnelle est satisfaite ou non).
Ainsi, nous pensons qu'il serait pertinent d'étendre cette technique afin de comparer dif-

férentes politiques d'adaptation avec d'autres. Cette technique consisterait donc à comparer les politiques d'adaptation en ce qui concerne l'optimisation des propriétés extra-fonctionnelles, en supposant que ces dernières puissent être mesurées.
Cette technique utiliserait nos travaux sur l'instanciation de configurations initiales du système, ensuite, la mesure de la propriété extra-fonctionnelle considérée se ferait par l'exécution du système, régi par une politique d'adaptation donnée. Ainsi, il serait possible d'évaluer si une politique d'adaptation donnée satisfait les propriétés extra-fonctionnelles considérées, ou de les comparer entre elles afin d'établir laquelle est la plus optimale par rapport aux propriétés extra-fonctionnelles considérées.

BIBLIOGRAPHIE

[1] B. Alkan and R. Harrison. A virtual engineering based approach to verify structural complexity of component-based automation systems in early design phase. *Journal of Manufacturing Systems*, 53 :18–31, 2019.

[2] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, and N. Vacelet. Projet bz-testing-tools-génération de tests aux limites à partir d'un modèle formel b ou z-annexes techniques. *Compte rendu d'avancement au*, 30, 2002.

[3] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, pages 27–47, 2009.

[4] P. André, C. Attiogbé, and A. Lanoix. A tool-assisted method for the systematic construction of critical embedded systems using event-b. *Computer Science and Information Systems*, 17(1) :315–338, 2020.

[5] C. Attiogbé and J. Rocheteau. Architectural invariants and correctness of iot-based systems. *arXiv preprint arXiv :1912.08912*, 2019.

[6] I. Babuska and T. Oden. Verification and validation in computational engineering and science : basic concepts. *Computer methods in applied mechanics and engineering*, 193(36) :4057–4066, 2004.

[7] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 3–12. IEEE Computer Society, 2006.

[8] F. Baude, D. Caromel, C Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM : a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2) :5–24, 2009.

[9] F. Baude, L. Henrio, and P Naoumenko. A component platform for experimenting with autonomic composition. In Fabrizio Davide, editor, *Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems, Autonomics 2007, 28-30 October 2007, Rome, Italy*, volume 302 of *ACM International Conference Proceeding Series*, page 8. ACM, 2007.

[10] F. Baude, L. Henrio, and C. Ruz. Programming distributed and adaptable autonomous components - the gcm/proactive framework. *Softw. Pract. Exp.*, 45(9) :1189–1227, 2015.

[11] A. Bauer and Y. Falcone. Decentralised LTL monitoring. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012 : Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[12] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, J. Marino, et al. Service component architecture, assembly model specification. *SCA Version*, 1 :15, 2007.

[13] K. Bergner, A. Rausch, and M. Sihling. Componentware—the big picture. In *20th ICSE Workshop on Component-based Software Engineering*, 1998.

[14] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications : Gsm 11-11 standard case study. *Software : Practice and Experience*, 34(10) :915–948, 2004.

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

[16] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from jml specifications. In *International Symposium on Formal Methods*, pages 428–443. Springer, 2006.

[17] K. Bowers, E. Fredericks, and B. Cheng. Automated optimization of weighted non-functional objectives in self-adaptive systems. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, pages 182–197, 2018.

[18] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, pages 48–70, 2009.

[19] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In *International Symposium on Component-based Software Engineering*, pages 7–22. Springer, 2004.

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B Stefani. The fractal component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006.

[21] M. Camilli, C. Bellettini, A. Gargantini, and P. Scandurra. Online model-based testing under uncertainty. In *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018*, pages 36–46, 2018.

[22] K. Castillos, F. Dadeau, and J. Julliand. Coverage criteria for model-based testing using property patterns. In Holger Schlingloff and Alexander K. Petrenko, editors, *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014*, volume 141 of *EPTCS*, pages 29–43, 2014.

[23] K. Cabrera Castillos. *Ge'ne'ration automatique de sce'narios de tests a partir de proprie'te's temporelles et de mode les comportementaux. (Automated test scenario generation from termporal properties and behavioural models)*. PhD thesis, University of Franche-Comté, Besançon, France, 2013.

[24] A. Cavarra, C. Crichton, J. Davies, A. Hartman, and L. Mounier. Using uml for automatic test generation. In *Proceedings of ISSTA*, volume 15. Citeseer, Springer-Verlag, 2002.

[25] F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Expression qualitative de politiques d'adaptation pour Fractal. In Y. Aït Ameur, editor, *2ème Conf. sur les Architectures Logicielles (CAL 2008), 3-7 Mars 2008, Montréal, Québec, Canada*, volume RNTI-L-2 of *Revue des Nouvelles Technologies de l'Information*, page 119. Cépaduès-Éditions, 2008.

[26] F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Composition et expression qualitative de politiques d'adaptation pour les composants fractal. In *Actes des Journées nationales du GDR GPL 2009*, 2009.

[27] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems : A research roadmap. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[28] S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE, 2009.

[29] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. *Formal Methods in System Design*, 28(3) :189–212, 2006.

[30] E. Clarke, O. Grumberg, and D. Peled. Model checking the mit press. *Cambridge, Massachusetts, London, UK*, 1999.

[31] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006) :1–6, 2006.

[32] F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Testing adaptation policies for software components. *Software Quality Journal*, 2020. https ://doi.org/10.1007/s11219-019-09487-w.

[33] F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Automated generation of initial configurations for testing component systems. In Gwen Salaün and Anton Wijs, editors, *Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28-29, 2021, Proceedings*, volume 13077 of *Lecture Notes in Computer Science*, pages 134–152. Springer, 2021.

[34] F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Online testing of dynamic reconfigurations w.r.t. adaptation policies. In *Modeling and Analysis of information Systems*, volume 28 (1), pages 52–73, 2.21.

[35] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1-2) :3–50, 1993.

[36] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures. *annals of telecommunications-annales des télécommunications*, 64(1) :45–63, 2009.

[37] R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, and N. Bencomo. Software engineering for self-adaptive systems : Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 3–30. Springer, 2017.

[38] R. de Lemos, H. Giese, H. Müller, and M. Shaw, editors. *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*. Springer, 2013.

[39] R. De Lemos, H. Giese, H. A Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M Villegas, and T. Vogel. Software engineering for self-adaptive systems : A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[40] R. De Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems : A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[41] D. De Niz, G. Bhatia, and R. Rajkumar. Model-based development of embedded systems : The sysweaver approach. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 231–242. IEEE, 2006.

[42] L. DeMichiel and M. Keith. Enterprise java beans, version 3.0 : Ejb core contracts and requirements, release 8 may 2006, 2006.

[43] E. W. Dijkstra et al. Notes on structured programming, 1970.

[44] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications.

ACM Transactions on Autonomous and Adaptive Systems (TAAS), 1(2) :223–259, 2006.

[45] J. Dormoy and O. Kouchnarenko. Event-based adaptation policies for Fractal components. In *AICCSA 2010, ACS/IEEE Int. Conf. on Computer Systems and Applications*, pages 1–8, Hammamet, Tunisia, may 2010.

[46] J. Dormoy and O. Kouchnarenko. Event-based adaptation policies for fractal components. In *ACS/IEEE International Conference on Computer Systems and Applications-AICCSA 2010*, pages 1–8. IEEE, 2010.

[47] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In Luís Soares Barbosa and Markus Lumpe, editors, *Formal Aspects of Component Software - 7th International Workshop, FACS 2010, Guimarães, Portugal, October 14-16, 2010, Revised Selected Papers*, volume 6921 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2010.

[48] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Runtime verification of temporal patterns for dynamic reconfigurations of components. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, volume 7253 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2011.

[49] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420, 1999.

[50] B. Eberhardinger, H. Seebach, D. Klumpp, and W. Reif. Test Case Selection Strategy for Self-Organization Mechanisms. In Mario Winter, Andreas Spillner, and Andrej Pietschker, editors, *Test, Analyse und Verifikation von Software – gestern, heute, morgen*. dpunkt Verlag, 2017.

[51] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. Towards testing self-organizing, adaptive systems. In M. G. Merayo and E. M. de Oca, editors, *Testing Software and Systems*, pages 180–185, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[52] R El Ballouli, S Bensalem, M Bozga, and J Sifakis. Dr-bip-programming dynamic reconfigurable systems. Technical report, Technical Report TR-2018-3, Verimag Research Report, 2018.

[53] A. El-Hokayem, S. Bensalem, M. Bozga, and J. Sifakis. A layered implementation of dr-bip supporting run-time monitoring and analysis. In *International Conference on Software Engineering and Formal Methods*, pages 284–302. Springer, 2020.

[54] E.A. Emerson and J. Halpern. "sometimes" and "not never" revisited : on branching versus linear time temporal logic. *J. ACM*, 33(1) :151–178, 1986.

[55] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *International Conference on Software Engineering and Formal Methods*, pages 204–220. Springer, 2011.

[56] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *2002 {USENIX} Annual Technical Conference ({USENIX}{ATC} 02)*, 2002.

[57] M. Felderer, B. Marculescu, F. Gomes de Oliveira Neto, R. Feldt, and R. Torkar. A testability analysis framework for non-functional properties. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 54–58, 2018.

[58] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software : continuous assurance of non-functional requirements. *Formal Asp. Comput.*, 24(2) :163–186, 2012.

[59] A. Filieri, G. Tamburrelli, and C. Ghezzi. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Software Eng.*, 42(1) :75–99, 2016.

[60] H. Fouchal, A. Rollet, and A. Tarhini. Robustness testing of composed real-time systems. *Journal of Computational Methods in Sciences and Engineering*, 10(3-6) :135–148, 2010.

[61] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144, 2012.

[62] M. Fowler. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[63] G. Fraser and F. Wotawa. Property relevant software testing with model-checkers. *ACM SIGSOFT Software Engineering Notes*, 31(6) :1–10, 2006.

[64] E. Fredericks, A. Ramirez, and B. Cheng. Towards run-time testing of dynamic adaptive systems. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 169–174. IEEE, 2013.

[65] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng. Towards run-time testing of dynamic adaptive systems. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 169–174, May 2013.

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

[67] M. García-Valls, D. Perez-Palacin, and R. Mirandola. Time-sensitive adaptation in cps through run-time configuration generation and verification. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 332–337. IEEE, 2014.

[68] J.-M. Gauthier, F. Bouquet, A. Hammad, and F. Peureux. Tooled process for early validation of sysml models using modelica simulation. In *International Conference on Fundamentals of Software Engineering*, pages 230–237. Springer, 2015.

[69] S. Ghahremani, H. Giese, and T. Vogel. Towards linking adaptation rules to the utility function for dynamic architectures. *CoRR*, abs/1805.03599, 2018.

[70] S. Ghahremani, H. Giese, and T. Vogel. Improving scalability and reward of utility-driven self-healing for large dynamic architectures. *ACM Trans. Auton. Adapt. Syst.*, 14(3) :12 :1–12 :41, 2020.

[71] Carlo Ghezzi. Adaptive software needs continuous verification. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–4. IEEE, 2010.

[72] A. González, É. Piel, and H.-G. Groß. Architecture support for runtime integration and verification of component-based systems of systems. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshop Proceedings (ASE Workshops 2008), Sept. 2008, L'Aquila, Italy*, pages 41–48, 2008.

[73] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on software Engineering*, (2) :156–173, 1975.

[74] M. Greiler, H.-G. Gross, and A. van Deursen. Evaluation of online testing for services : a case study. In *Proceedings of the 2nd Int. Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2010*, pages 36–42, 2010.

[75] A. Gupta and S. Nadarajah. *Handbook of beta distribution and its applications*. CRC press, 2004.

[76] S. Gupta, A. Ansari, S. Feng, and S. A. Mahlke. Adaptive online testing for efficient hard fault detection. In *27th Int. Conf. on Computer Design*, pages 343–349, 2009.

[77] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *JSTOR : Applied Statistics*, 28(1) :100–108, 1979.

[78] G. Heineman and W. Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, 5, 2001.

[79] M. Helvensteijn. Dynamic delta modeling. In *16th International Software Product Line Conference, SPLC'12*, pages 127–134, 2012.

[80] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[81] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[82] J. Holland. Studying complex adaptive systems. *Journal of systems science and complexity*, 19(1) :1–8, 2006.

[83] H. Hong, S. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 232–242. IEEE, 2003.

[84] R. Hoogendoorn, B. van Arerm, and S. Hoogendoom. Automated driving, traffic flow efficiency, and human factors : Literature review. *Transportation Research Record*, 2422(1) :113–120, 2014.

[85] W. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, (3) :208–215, 1976.

[86] A. Hussain, S. Tiwari, J. Suryadevara, and E. Enoiu. From modeling to test case generation in the industrial embedded system domain. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies : Applications and Foundations*, volume 11176 of *Lecture Notes in Computer Science*, pages 499–505. Springer, 2018.

[87] D. Jia, K. Lu, J. Wang, X. Zhang, and X. Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE communications surveys & tutorials*, 18(1) :263–284, 2015.

[88] D. Jia, K. Lu, J. Wang, X. Zhang, and X. Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE Communications Surveys Tutorials*, 18(1) :263–284, 2016.

[89] S. Karris. *Introduction to Simulink with engineering applications*. Orchard Publications, 2006.

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

[91] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), 7-9 June 2004, Yorktown Heights, NY, USA*, pages 3–12, 2004.

[92] M. Kim, I. Lee, J. Shin, O. Sokolsky, et al. Monitoring, checking, and steering of real-time systems. *ENTCS*, 70(4) :95–111, 2002.

[93] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil. An architecture framework for experimentations with self-adaptive cyber-physical systems. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 93–96. IEEE, 2015.

[94] H. Koskinen. On the coverage of a random sensor network in a bounded domain. In *Proceedings of 16th ITC specialist seminar*, pages 11–18. Citeseer, 2004.

[95] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS, 10th Int. Symp. on Formal Aspects of Component Software*, volume 8348 of *LNCS*, pages 234–253. Springer, 2014.

[96] O. Kouchnarenko and J.-F. Weber. Decentralised evaluation of temporal patterns over component-based systems at runtime. In I. Lanese and E. Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *LNCS*, pages 108 – 126, Bertinoro, Italy, sep 2015. Springer.

[97] O. Kouchnarenko and J-F. Weber. Practical analysis framework for component systems with dynamic reconfigurations. In Michael J. Butler, S. Conchon, and F. Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2015.

[98] J. R. Koza. *Genetic Programming : On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[99] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, pages 7–es, 2006.

[100] K. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In *International Workshop on Formal Approaches to Software Testing*, pages 79–94. Springer, 2004.

[101] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002 : Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[102] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In Lars Grunske, Ralf H. Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering, 13th International Symposium, CBSE 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings*, volume 6092 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2010.

[103] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6) :382–403, 2006.

[104] E. Letier. *Reasoning about agents in goal-oriented requirements engineering*. PhD thesis, PhD thesis, Université catholique de Louvain, 2001.

[105] Z. Lotfi A. Fuzzy logic. *Computer*, 21(4) :83–93, 1988.

[106] M. Utting. How to design extended finite state machine test models in Java. In *Model-Based Testing for Embedded Systems*, Series on Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 147–170. CRC Press, 2011.

[107] R. Mazo, D. Diaz, and C. Salinesi. Constraint programming as a means to manage configurations in self-adaptive systems.

[108] K. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[109] A. Memon, M. Soffa, and M. Pollack. Coverage criteria for gui testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, 2001.

[110] P. Merle and J.-B. Stefani. *A formal specification of the Fractal component model in Alloy*. PhD thesis, INRIA, 2008.

[111] A. Mili, B. Cukic, Y. Liu, and R. Ayed. Towards the verification and validation of online learning adaptive systems. In *Software Engineering with Computational Intelligence*, pages 173–203. Springer, 2003.

[112] H. Muller. The rise of intelligent cyber-physical systems. 2017.

[113] Masoud Najafi. Selection of variables in initialization of modelica models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, number 029, pages 111–119. Citeseer, 2008.

[114] H. Nakagawa, A. Ohsuga, and S. Honiden. gocc : A configuration compiler for self-adaptive systems using goal-oriented requirements description. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 40–49, 2011.

[115] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel. Automatic test generation : A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3) :140–155, 2006.

[116] K. Ng, G.-L. Tian, and M.-L. Tang. Dirichlet and related distributions : Theory, methods and applications. 2011.

[117] C. D. Nguyen, A. Perini, P. Tonella, and B. Kessler. Automated continuous testing of multi-agent systems. In *The fifth European workshop on Multi-agent systems*. Citeseer, 2007.

[118] P. Oreizy et al. Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the international workshop on the role of software architecture in testing and analysis*, pages 54–57. Citeseer, 1998.

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

[120] A. Orso and G. Rothermel. Software testing : a research travelogue (2000–2014). In *Future of Software Engineering Proceedings*, pages 117–132. 2014.

[121] M. Oussalah. *Ingénierie des composants : concepts, techniques et outils*. Vuibert informatique, 2005.

[122] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[123] W. Pedrycz. *Fuzzy control and fuzzy systems*. Research Studies Press Ltd., 1993.

[124] V. Poladian, D. Garlan, M. Shaw, M. Satyanarayanan, B. R. Schmerl, and J. Pedro Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007*, pages 214–223, 2007.

[125] A. Pretschner and J. Philipps. Methodological issues in model-based testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.

[126] R. Rajkumar, I. Lee, L. Sha, and J. A. Stankovic. Cyber-physical systems : the next computing revolution. In *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 731–736, 2010.

[127] A. Ramirez and B. Cheng. Verifying and analyzing adaptive logic through uml state models. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 529–532. IEEE, 2008.

[128] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *European Conference on Object-Oriented Programming*, pages 205–230. Springer, 2002.

[129] G. Reger and K. Havelund. What is a trace ? a runtime verification perspective. In *International Symposium on Leveraging Applications of Formal Methods*, pages 339–355. Springer, 2016.

[130] D. Romero, C. Quinton, L. Duchien, L. Seinturier, and C. Valdez. Smartyco : Managing cyber-physical systems for smart environments. In *Software Architecture - 9th European Conference, ECSA 2015*, pages 294–302, 2015.

[131] C. Rouff, R. Buskens, L. Pullum, X. Cui, and M. Hinchey. The adaptiv approach to verification of adaptive systems. In *Proceedings of the fifth international c* conference on computer science and software engineering*, pages 118–122, 2012.

[132] A. Abdo A. Saeed and S.-W. Lee. Non-functional requirements trade-off in self-adaptive systems. In *4th International Workshop on Requirements Engineering for Self-Adaptive, Collaborative, and Cyber Physical Systems, RESACS@RE 2018, Banff, AB, Canada, August 20, 2018*, pages 9–15, 2018.

[133] L. Sakizloglou, S. Ghahremani, T. Brand, M. Barkowsky, and H. Giese. Towards highly scalable runtime models with history. *CoRR*, abs/2004.03727, 2020.

[134] R. Sargent. Verification and validation of simulation models. In *2008 Winter Simulation Conference*, pages 157–169. IEEE, 2008.

[135] W. Schamai. *Modelica modeling language (ModelicaML) : A UML profile for Modelica*. Linköping University Electronic Press, 2009.

[136] C. Schulte and P. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1) :1–43, 2008.

[137] R. Seiger, S. Huber, P. Heisig, and U. Aßmann. Toward a framework for self-adaptive workflows in cyber-physical systems. *Software & Systems Modeling*, 18(2) :1117–1134, 2019.

[138] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable sca applications with the frascati platform. In *2009 IEEE International Conference on Services Computing*, pages 268–275. IEEE, 2009.

[139] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5) :559–583, 2012.

[140] G. Settanni, F. Skopik, A. Karaj, M. Wurzenberger, and R. Fiedler. Protecting cyber physical production systems using anomaly detection to enable self-adaptation. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 173–180. IEEE, 2018.

[141] A. Sinclair. *Algorithms for Random Generation and Counting : A Markov Chain Approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.

[142] G. Steinbauer and F. Wotawa. Model-based reasoning for self-adaptive systems - theory and practice. In *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, pages 187–213. 2013.

[143] S. Taha, J. Julliand, F. Dadeau, K. Cabrera Castillos, and B. Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Aspects of Computing*, 27(4) :641–664, dec 2015.

[144] M.B. Tahoori and S. Mitra. Automatic configuration generation for fpga interconnect testing. In *Proceedings. 21st VLSI Test Symposium, 2003.*, pages 134–139. IEEE, 2003.

[145] G. Tamura. *QoS-CARE : A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration. (QoS-CARE : Un Système Fiable pour la Préservation*

de Contrats de Qualité de Service à travers de la Reconfiguration Dynamique). PhD thesis, Lille University of Science and Technology, France, 2012.

[146] Y. Tan, M. C Vuran, and S. Goddard. Spatio-temporal event model for cyber-physical systems. In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 44–50. IEEE, 2009.

[147] C. Thorpe, T. Jochem, and D. Pomerleau. The 1997 automated highway free agent demonstration. In *Proceedings of conference on intelligent transportation systems*, pages 496–501. IEEE, 1997.

[148] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3) :103–120, 1996.

[149] F. Trollman, J. Fähndrich, and S. Albayrak. Hybrid adaptation policies : towards a framework for classification and modelling of different combinations of adaptation policies. In *Proc. Int. Conf. SEAMS@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 76–86, 2018.

[150] S. Tsugawa, S. Kato, and K. Aoki. An automated truck platoon for energy saving. In *2011 IEEE/RSJ international conference on intelligent robots and systems*, pages 4109–4114. IEEE, 2011.

[151] L. Van Aertryck, M. Benveniste, and D. Le Métayer. Casting : A formally based software test generation method. In *First IEEE International Conference on Formal Engineering Methods*, pages 101–110. IEEE, 1997.

[152] N.M. Villegas, H.A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 80–89, 2011.

[153] J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component-based software systems. *Software Quality Journal*, 10(2) :115–133, 2002.

[154] G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Software : Practice and Experience*, 25(1) :97–108, 1995.

[155] N. Wang, D. Schmidt, and C. O'Ryan. Overview of the corba component model. In *Component-based software engineering : putting the pieces together*, pages 557–571. 2001.

[156] J.-F. Weber. Tool support for fuzz testing of component-based system adaptation policies. In *13th International Conference on Formal Aspects of Component Software*, volume 10231 of *LNCS*, pages 231 – 237, 2016.

[157] Jean-François Weber. *Guider et contrôler les reconfigurations de systèmes à composants*. PhD thesis, Université de Franche-Comté (UFC).

[158] E. Weyuker. Testing component-based software : A cautionary tale. *IEEE software*, 15(5) :54–59, 1998.

[159] J. Whittaker and M. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software engineering*, 20(10) :812–824, 1994.

[160] J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Trans. on Software Engineering*, 20(10) :812–824, 1994.

[161] C. Wilke, C. Piechnick, and U. Aßmann. Testing self-adaptive software : Requirement analysis and solution scheme, 2014.

[162] F. Wotawa. Testing self-adaptive systems using fault injection and combinatorial testing. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Companion, Vienna, Austria, August 1-3, 2016*, pages 305–310, 2016.

[163] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. Leite. From goals to high-variability software design. In *International Symposium on Methodologies for Intelligent Systems*, pages 1–16. Springer, 2008.

[164] J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380, 2006.

TABLE DES FIGURES

1.1 Processus de génération de tests en ligne 9

2.1 Activités d'une boucle de contrôle MAPE 14
2.2 Activités d'une boucle de contrôle MAPE-K 15
2.3 Exemple de composant composite 17
2.4 Interfaces fournie et requise d'un composant 17
2.5 Assemblage de composants 19
2.6 Composant *Helloworld* en Fractal 23
2.7 Spécification de liens entre interfaces en utilisant Fractal ADL 24
2.8 Interfaces multicast et gathercast 26

3.1 Exemple d'un système de transitions étiqueté 41
3.2 Exemple d'un système de transitions étiqueté interprété 44
3.3 Un exemple configuration VANet 47
3.4 Évolution de la configuration d'un système adaptatif 50
3.5 Syntaxe FTPL 52
3.6 Deux règles de reconfiguration du convoi de véhicules autonomes 56

4.1 Partie du processus étudiée dans cette section 60
4.2 Éléments de configuration 62
4.3 Relations architecturales 66
4.4 Une instance d'architecture à composants de l'exemple VANet 68
4.5 Instance de paramètres dans *Params* et *Definer* 71

5.1 Processus de génération de tests en ligne 78
5.2 Symétries dans les relations de parenté 80
5.3 Symétries dans les relations de délégation 80

5.4 Symétries dans les relation de lien 81
5.5 Illustration des configurations A et B 82
5.6 Densité de probabilité 86
5.7 Beta-distribution aux extrémités 87
5.8 Beta-distribution au minimum 87
5.9 Beta-distribution au maximum 87
5.10 Beta-distribution aléatoire 88
5.11 Processus de génération d'évènements contrôlables à l'aide d'un générateur de test 90
5.12 Modèle d'usage pour un véhicule 92
5.13 Processus de génération d'évènements de test en ligne 96
5.14 Cas de test pour l'exemple VANet 98

6.1 Automates **after** *ev temp* (gauche), et **before** *ev trace* ou *trace* **until** *ev* (droite)103
6.2 automate pour la composition **after**-**until** 103
6.3 Couverture de la propriété **after until** 105
6.4 Couverture de la règle passRelay 108

7.1 Les règles de reconfiguration de la politique d'adaptation VANet 121
7.2 Un extrait de l'exécution du cas d'étude 123
7.3 Analyse des fréquences de l'implémentation (à gauche) E1 comparé à l'implémentation E2 (à droite) 135

LISTE DES TABLES

5.1 Résultats de la génération des instances de la Beta-distribution 88

7.1 Nombre nécessaire de pas pour atteindre un taux de couverture de 100% . 126
7.2 Le taux de couverture agrégé obtenu en fonction de chaque suite de test . 126
7.3 Détection des mutants 127
7.4 Résultats de la génération de tests pour les différentes variantes 128
7.5 Extrait des résultats expérimentaux et taux de couverture 131
7.6 Nombre de configurations pour chaque paramétrage 132
7.7 Moyenne des résultats obtenus pour des simulations de 100.000 pas . . . 136

LISTE DES DÉFINITIONS

1 Définition : Domaine et codomaine d'une relation . . . 38
2 Définition : Relation réflexive, symétrique, antisymétrique et transitive . . . 38
3 Définition : Fermeture transitive d'une relation et relation acyclique . . . 39
4 Définition : Relation fonctionnelle . . . 39
5 Définition : Ensemble des parties . . . 39
6 Définition : Filtrage de fonction multivaluées . . . 40
7 Définition : Fonction injective, surjective et bijective . . . 40
8 Définition : Système de transitions ST . . . 40
9 Définition : Système de transitions étiqueté ST_E . . . 41
10 Définition : Langage du premier ordre . . . 42
11 Définition : Termes . . . 42
12 Définition : Formules . . . 43
13 Définition : structure d'interprétation . . . 43
14 Définition : Affectation . . . 44
15 Définition : Système de transitions étiqueté interprété $STEI$ . . . 44
16 Définition : Modèle de reconfiguration . . . 48
17 Définition : Chemin de reconfiguration . . . 48
18 Définition : Trace d'exécution . . . 48
19 Définition : Exécution . . . 49
20 Définition : Atteignabilité d'une configuration . . . 50
21 Définition : Satisfaction d'une propriété de configuration . . . 52
22 Définition : Satisfaction d'un évènement sur une configuration . . . 53
23 Définition : Satisfaction d'une propriété de trace sur un chemin de reconfiguration . . . 53

24 Définition : Satisfaction d'une propriété temporelle sur un chemin de reconfiguration 54
25 Définition : Politiques d'adaptation 55

26 Définition : Configuration 60
27 Définition : Éléments architecturaux 61
28 Définition : Relations architecturales 63
29 Définition : Instances architecturales 66

30 Définition : Évènements contrôlables 90
31 Définition : Évènements de test 91
32 Définition : Modèle d'usage à partir d'un automate probabiliste 91
33 Définition : Cas de test 98

34 Définition : Automate de propriété FTPL 102
35 Définition : Association d'un état 104
36 Définition : Transitions couvertes dans $\mathcal{A}_\varphi$ 104
37 Définition : Couverture de propriété 105
38 Définition : Verdict de test pour une propriété FTPL 106
39 Définition : Couverture d'une règle d'adaptation 107
40 Définition : Couverture de politique d'adaptation 108
41 Définition : Reconfiguration inappropriée 109
42 Définition : Reconfiguration inattendue 110
43 Définition : Verdict de test pour les politiques d'adaptation 110
44 Définition : LTS resteint au politiques d'adaptation 111
45 Définition : Nombre d'exécutions d'une règle 112
46 Définition : Nombre de déclenchements d'une règle 112
47 Définition : Nombre d'éligibilité d'une règle 113
48 Définition : Fréquence d'une règle 113

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets `tex-upmethodology` — http://www.arakhne.org/tex-upmethodology/

école doctorale **sciences pour l'ingénieur et microtechniques**

Titre : Contributions à la validation de systèmes à composants adaptatifs par génération de tests

Mots-clés : Systèmes adaptatifs, Systèmes à composants, Validation

Résumé :

Les systèmes adaptatifs peuvent se reconfigurer en fonction de politiques d'adaptation qui sont vues comme des artefacts qui décrivent les comportements souhaitables du système sans les imposer. Une politique d'adaptation est conçue comme un ensemble de règles qui indiquent, pour un ensemble donné de configurations, quelles opérations de reconfiguration peuvent être déclenchées, dont la priorité est donnée avec des valeurs floues représentant leurs utilités respectives. La politique d'adaptation doit être fidèlement implémentée par le système et spécialement vis-à-vis du respect des priorités des règles. Nous proposons une approche originale de validation d'un système adaptatif à composants, basée sur du test boîte noire, et visant à assurer le respect d'une politique d'adaptation par le système. Pour ce faire, nous présentons notre approche qui vise à générer de grandes suites de tests afin de mesurer les occurrences de reconfigurations et de les comparer à leurs valeurs d'utilité spécifiées dans les règles d'adaptation. Tout d'abord, nous présentons la génération automatisée d'états initiaux, afin de produire des configurations structurées à partir desquelles un système adaptatif démarre et qui visent à provoquer des reconfigurations significatives. Ensuite, nous présentons un générateur de tests en ligne basé sur un modèle d'utilisation du système utilisé pour stimuler le système et provoquer des reconfigurations. Comme le système peut se reconfigurer dynamiquement, ce générateur de test en ligne observe les réponses et l'évolution du système afin de décider de la prochaine étape de test appropriée à effectuer. Des mesures de couverture de la politique d'adaptation et des propriétés du système sont établies pour s'assurer que tous les états pertinents du système sont parcourus. Au final, les fréquences relatives des reconfigurations sont mesurées afin de déterminer si la politique d'adaptation est fidèlement mise en œuvre. Les expériences sont menées sur le cas d'un peloton de véhicules autonomes.

Title: Contributions to the validation of adaptive component systems by test generation

Keywords: Adaptive systems, Component systems, Validation

Abstract:

Adaptive systems can reconfigure themselves according to adaptation policies which are seen as artefacts that describe desirable system behaviours without imposing them. An adaptation policy is conceived as a set of rules that indicate, for a given set of configurations, which reconfiguration operations can be triggered, prioritised with fuzzy values representing their respective utility. The adaptation policy must be faithfully implemented by the system and especially with respect to the rules' priorities.

We propose an original approach to validate an adaptive system with components, based on black box testing, and aiming to ensure the respect of an adaptation policy by the system.

To do so, we present our approach which aims at generating large test suites in order to measure the occurrences of reconfigurations and to compare them to their utility values specified in the adaptation rules. First, we present the automated generation of initial states, in order to produce structured configurations from which an adaptive system starts and which aim at provoking significant reconfigurations. Secondly, we present an online test generator based on a system usage model used to stimulate the system and cause reconfigurations. As the system can reconfigure itself dynamically, this online test generator observes the responses and evolution of the system in order to decide on the next appropriate test step to perform. Coverage measures of the adaptation policy and system properties are established to ensure that all relevant system states are covered.

Finally, the relative frequencies of reconfigurations are measured to determine whether the adaptation policy is being implemented accurately.

The experiments are conducted in the case of a platoon of autonomous vehicles.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon, France